TU WIEN Informatics

Parameter Optimization for Surface Reconstruction

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Masterstudium Visual Computing

by

Florian Steinschorn, Bsc.
Registration Number 01227109

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Philipp Erler, B.Eng.

Vienna, 18th January, 2025

Florian Steinschorn Michael Wimmer

Erklärung zur Verfassung der Arbeit

Florian Steinschorn, Bsc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Jänner 2025

Florian Steinschorn

Danksagung

I möchte meinem Betreuer Michael Wimmer danken, und besonders Philipp Erler, der mich beim Schreiben dieser Arbeit sehr unterstützt hat. Ich möchte mich außerdem bei meiner Frau und meiner Familie bedanken, die es nicht versäumt haben mich immer wieder zu motivieren wenn ich es nötig hatte.

Acknowledgements

I want to thank my advisor Michael Wimmer and especially Philipp Erler, who assisted me heavily in writing this thesis. I also want to thank my wife and family, who kept motivating me when I had trouble focusing on my work.

Kurzfassung

In dieser Arbeit vergleichen wir unterschiedliche Algorithmen zur Parameter-Optimierung am Beispiel von Screened Poisson Surface Reconstruction. Um dies zu erreichen, implementierten wir zuerst fünf aktuelle Algorithmen. GEIST ist ein Graphen-basierter Algorithmus, der den Parameter-Raum in einen ‘optimalen’ und einen ‘nicht-optimalen’ Bereich aufteilt um die nächsten Konfigurationen auszuwählen. Iterated F-Race platziert eine Normalverteilung für die Auswahl der nächsten Konfigurationen über die besten Konfigurationen der letzten Iteration. ParamILS verwendet iterative lokale Suche um einen besseren Nachbarn zu finden und damit schlussendlich ein Optimum. PostSelection verwendet eine verkürzte Version eines Algorithmus um vielversprechende Kandidaten zu finden, und eine detailliertere Version um diese auszuwerten. Als simplen Vergleich implementierten wir außerdem eine Version von Brute-Force.
Für alle diese Algorithmen führen wir zuerst eine Reihe an Tests durch um eine gute Konfiguration für deren Ausführung zu finden. Danach testen wir sie an Punktwolken aus zwei Datensets. Jedes Datenset enthält alle Wolken in unterschiedlichen Qualitäten, wir sind also in der Lage unterschiedliche Input Qualitäten als auch Typen zu testen. Wir zeigen, dass jeder der implementierten Algorithmen in der Lage ist bessere Parameter-Konfigurationen zu finden als den Screened Poisson Surface Reconstruction Standard. In den meisten Fällen erreichen GEIST und PostSelection die besten Resultate, brauchen dafür aber auch am längsten. ParamILS und Iterated F-Race führen zu guten Resultaten in einer deutlich kürzeren Zeit. Brute-Force ist nicht konkurrenzfähig wenn es um hochqualitative Konfigurationen geht, verbessert aber noch immer den Standard in den meisten Fällen.
Um unsere Resultate über verschiedene Typen und Qualitäten zusammenzufassen kann man sagen, dass die Standardkonfiguration akzeptable, aber nicht ideale, Ergebnisse für Punktwolken von glatten Objekten mit wenigen Störungen erzielt. Wir schlagen eine Alternative hierführ vor. Ist die Oberfläche rauer so muss idealerweise das Gewicht stärker auf die Position der Punkte gelegt werden. Gibt es sehr viele Störungen in den Daten, so muss dieses Gewicht und die Octree Tiefe reduziert werden.
Wir diskutieren die Vorteile und Nachteile von jedem implementierten Algorithmus und vergleichen ihre Resultate um empfehlen zu können welchen man verwenden sollte. Wir beschreiben unsere Implementierung von jedem und beschreiben kurz was an weiterer Arbeit verrichtet werden könnte um diese Arbeit zu erweitern. Zum Schluss geben wir Empfehlungen ab, welche Konfiguration für welche Typen und Punktwolken

verwendet werden sollten. Für sehr genaue Daten sollten *depth* und *pointWeight* höher gewählt werden, als für ungenaue. Ist die Topologie des Objekts sehr komplex, dann sollte *pointWeight* sehr hoch eingestellt werden. Wir kommen zu dem Schluss, dass in den meisten Fällen IF-Race der beste Kompromiss zwischen Geschwindigkeit und Rekonstruktionsqualität ist. Wenn die Laufzeit kein relevanter Faktor ist, dann stellt GEIST eine Alternative dar, die sehr hoch qualitative Ergebnisse liefert.

Abstract

In this thesis, we compare different parameter-optimization algorithms on the example of Screened Poisson Surface Reconstruction. To do this, we first implemented five state-of-the-art algorithms. GEIST is a graph-based algorithm that splits the parameter space into an 'optimal' and a 'non-optimal' set to select new configurations. Iterated F-Race places a normal distribution of selection probabilities on the best configurations of the last iteration and uses that to choose the next configurations. ParamILS uses iterative local search to select a better neighbor and find an optimum this way. PostSelection uses a shortened version of an algorithm to find promising candidates and a second, more detailed one to evaluate these. As a simple baseline we also implemented Brute-Force.
For all of these algorithms, we first conduct several tests to find a good configuration to run them with. After that, we test them on point clouds from two datasets. Each dataset contains each cloud in different qualities, so we are able to test varying input qualities as well as types. We show that each of the implemented algorithms is able to find better parameter configurations than the default Screened Poisson Surface Reconstruction configuration. In most cases, GEIST and PostSelection lead to the best results but also have the longest run times, while ParamILS and Iterated F-Race lead to good results in a far shorter time period. Brute-Force is not competitive when it comes to high-quality configurations, but still leads to an improvement over the default in most cases.
To summarize the results over different types and qualities, the default configuration yields acceptable but not ideal results for point clouds of smooth meshes with little noise and we suggest an alternative. If the surface is rougher, the importance weight of the points should ideally be set higher. If there is a lot of noise, this weight as well as the Octree depth should be reduced.
We discuss the advantages and disadvantages of each implemented algorithm and compare their results to recommend which one to use. We describe our implementations of each and quickly mention what work could be done to expand on this thesis. Finally, we give recommendations as to which configurations to use for different types of point clouds. For data with higher accuracy, $depth$ and $pointWeight$ should be higher than for data with lower accuracy. If the topology of the object is very complex, $pointWeight$ is best set very high in comparison to simpler point clouds. We find that for most cases, IF-Race is the best compromise to use between speed and resulting quality of reconstruction. If time is of no concern, GEIST is an alternative that yields high-quality results.

Contents

CHAPTER 1

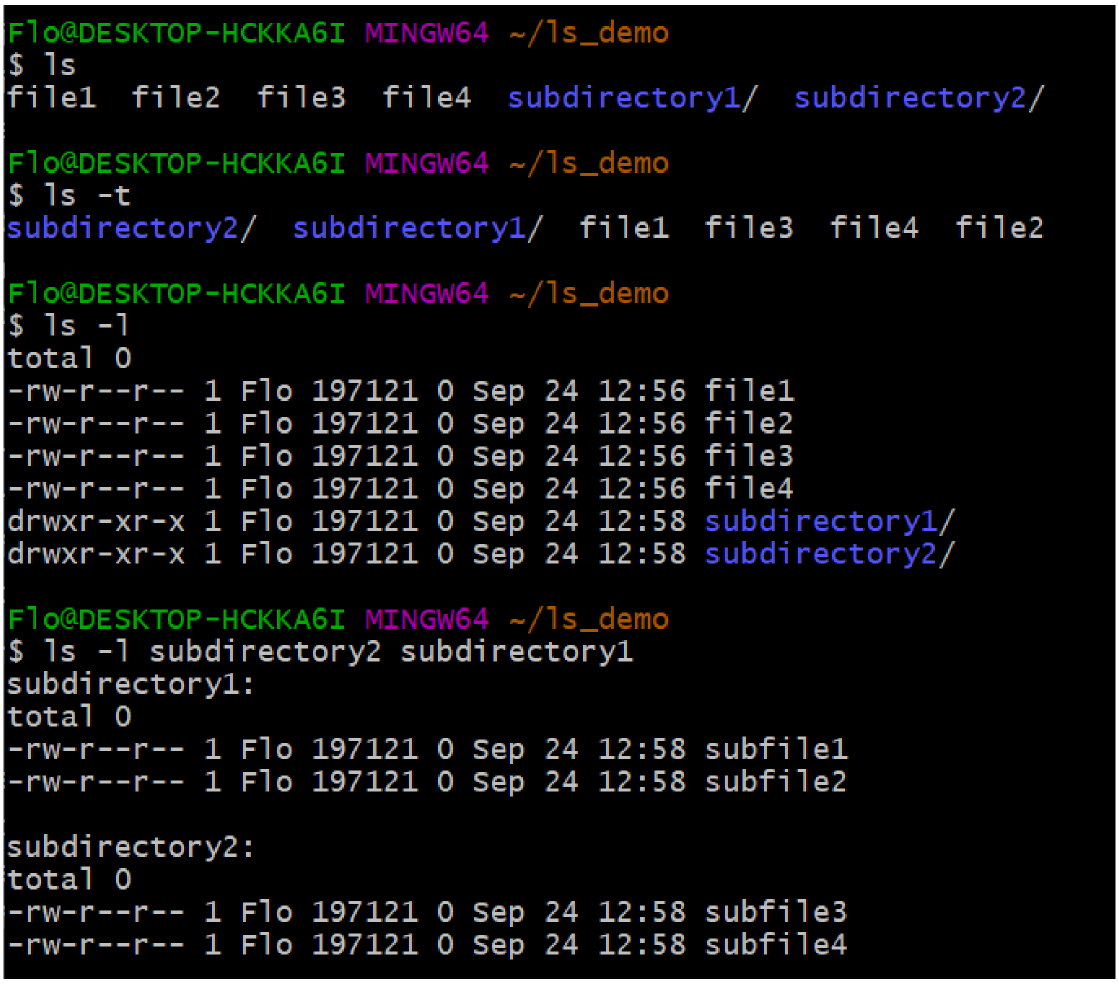
Introduction

Many operations in the field of computer science, or using a computer in general, require the selection of parameters or configurations with which to run. The outcome of the operation often heavily depends on the quality of the selected parameters. In this thesis, we use Screened Poisson Surface Reconstruction (SPSR) [KH13], which is an algorithm to generate 3D meshes from point clouds, to demonstrate the effectiveness of different algorithms in optimizing these parameters. This is a necessary step in, for example, generating 3D models from a series of images or 3D scanning an object.

1.1 Motivation

From the simplest command line functions to the most complex programs, nearly all computer operations require a selection of parameters to run with. These parameters can have various effects on the execution of the function or program. They might enable different features, select targets for the operation or set other values that are necessary in some way for the program to function.
Even simple commands, like the Unix ls (**lis**t) function, often require parameters to run. The ls function, for example, expects a number of options as well as a number of paths. The options alter the behavior of the function, for example printing out one element per line instead of multiple per line, or sorting the output by different properties. The expected paths are the paths whose content should be listed. In the case of a path pointing to a file, only that file is listed. Some examples can be seen in Figure 1.1.

Of course, the ls function can be called without any parameters. This is due to the fact that it, as well as many other operations, has default parameters defined. In the case of ls, there is a default value for each possible option, as well as '. ' (current directory) as the default for the paths. For this function, this is a completely adequate setup. Through available documentation, the effects of each parameter are easy to understand, and users



```
Flo@DESKTOP-HCKKA6I MINGW64 ~/ls_demo
$ ls
file1  file2  file3  file4  subdirectory1/  subdirectory2/

Flo@DESKTOP-HCKKA6I MINGW64 ~/ls_demo
$ ls -t
subdirectory2/  subdirectory1/  file1  file3  file4  file2

Flo@DESKTOP-HCKKA6I MINGW64 ~/ls_demo
$ ls -l
total 0
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:56 file1
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:56 file2
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:56 file3
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:56 file4
drwxr-xr-x 1 Flo 197121 0 Sep 24 12:58 subdirectory1/
drwxr-xr-x 1 Flo 197121 0 Sep 24 12:58 subdirectory2/

Flo@DESKTOP-HCKKA6I MINGW64 ~/ls_demo
$ ls -l subdirectory2 subdirectory1
subdirectory1:
total 0
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:58 subfile1
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:58 subfile2

subdirectory2:
total 0
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:58 subfile3
-rw-r--r-- 1 Flo 197121 0 Sep 24 12:58 subfile4
```

Figure 1.1: Calling ls with different parameters. Top to Bottom: no parameters, sort by creation time, print one per line, print one per line, and selecting both subdirectories.

can determine which options and paths they want to choose. The same is not true, if the result of the operation can be of different quality and it is not straightforward to determine what effect each parameter will have on the outcome.
An example of such an operation is SPRS, a complex surface reconstruction algorithm. One of its parameters is *pointWeight*, which is the importance of interpolating point values over gradients. This explanation might be useful for someone who knows how the algorithm works, but even then, selecting the best value for the parameter is difficult. Most users will not know what it does at all and will have no chance of setting it correctly. Another of SPSR's parameters is *depth*, which sets the maximal depth of the octree used in the reconstruction. This is an easier concept to understand, a higher octree depth leads to a finer reconstruction, but still, it is not easy to choose the appropriate value for this parameter. Setting *depth* too high will lead to far higher runtimes without leading

to any significant improvement in the result.
In most cases, sensible default values still exist or can be selected by an expert, but finding an optimum is difficult. For small parameter spaces or fast operations, it might be possible to fully map all parameter combinations to their output, but more complex problems and higher numbers of possible parameters make such an approach infeasible. In these cases, optimization algorithms are necessary in order to find the parameter configurations required to run the operation optimally. A large number of such optimization algorithms exist, so one must decide on which one to use. To help with this step, in this thesis, we test multiple different optimization algorithms on the example of SPSR. We evaluate them against each other and give some information on which parameters to use for SPSR for different inputs.

1.2 Screened Poisson Surface Reconstruction

Screened Poisson Surface Reconstruction (SPSR) by Kazhdan and Hoppe [KH13] is an algorithm to construct 3D meshes from point clouds. This is a necessary step in most recreations of physical objects as computer models. A typical 3D scanning workflow often consists of a step that creates a point cloud of a physical model and a second step that reconstructs a 3D model from that point cloud. The first step can be solved by a number of different methods. The point cloud can be determined by using direct measurements like from a laser distance scanner (Lidar), specialized hardware like a 3D scanner or calculated from existing data. A prominent example of the latter would be 3D reconstructions from a collection of images. This is used to generate meshes from small objects up to reconstructions of entire cities from satellite images. The second step in this 3D scanning workflow is what SPSR can be used for, creating the actual mesh from the extracted points.
3D scanning in general is an important tool for many different tasks. It is very useful if a 3D model should be as close as possible to reality. This can be the case for 3D animations in movies or games, for example, to get a real actor into a virtual world. Another area where an accurate representation of the physical world is important is navigation, where a 3D model of the surroundings can be more helpful than a 2D map. Additionally, to being very close to reality, a 3D scanned model can in many cases be created far faster than if it were modeled manually. While there is still some work to be done on the scanning result, this can help speed up the development of 3D scenes.
We chose SPSR as our test subject not only because it is the gold standard for mesh reconstruction at the moment, but also because of the properties of its parameter. SPSR, as implemented by Meshlab[1], can use, among others, different depths, point weights, or samples per node, which all heavily change the resulting mesh. A bad selection of these parameters leads to a bad or failed reconstruction or can cause the process to take far more time than for a similar result with different parameters. Additionally, there is a good variety of parameter types to run SPSR with, numerical as well as categorical.

[1]http://www.meshlab.net/

1.3 Contributions

For this thesis, we implemented five different parameter optimization algorithms and tested them against each other on multiple point clouds from the 'ABC' and 'famous' datasets provided in [EGO$^+$20]. The focus of these tests lies on SPSR because it has a parameter space with desirable properties for our tests, further discussed in Section 3.2. Other than SPSR, all algorithms can be applied to any other parameterized operation, as we describe in Section 5.1.
One of the main contributions of this thesis is near optimal parameter configurations for SPSR for different point clouds and point-cloud types. These have been determined by optimizing the Chamfer Distance of the resulting reconstructed meshed to the ground truths. We give parameter configurations for an 'average' point cloud, as well as for topologically difficult ones and different noise levels. We find that the main parameters to adjust are *depth* and $pointWeight$, with both being required to be higher for more accurate data and lower for noisier or more inaccurate point clouds. To fit the reconstruction to holes or other more complex structures, $pointWeight$ should be chosen very high. For less complex clouds, this is not advisable, since it could lead to overfitting.
Another contribution of this thesis is a number of adjustments to the tested optimization algorithms. These are mostly necessary for the algorithms to run on parameter spaces as large as the one we use for SPSR, where some time optimizations are necessary. Another case where we had to adjust the original algorithms is for them to run on a single point cloud instead of requiring multiple tests to come to a result.
Like SPSR, the tested algorithms also require parameters to run. We test each algorithm against different point clouds and present the optimal parameters.
Additionally, we provide a framework to run all the necessary optimizations. Our framework can apply different optimization algorithms to operations that need good parameters selected to run efficiently. It also includes several implemented visualizations to help better understand each algorithm and evaluate the results (Section 3.3).

1.4 Structure

This thesis is structured into four main parts and a chapter containing all our results in tabular form. The first chapter is Related Work (Section 2), which describes some general parameter types and optimization algorithms (Sections 2.1-2.2), before looking at several available algorithms sorted by their basic types (Sections 2.3-2.5).
The next chapter is Methodology (Section 3), where we go into detail on all implemented parts of the framework. First, we present the five optimization algorithms we test (Section 3.1). For each algorithm we explain how it works and if there are any deviations from the literature that we added for our own implementation. Next, we explain SPSR in Section 3.2, as well as its parameters (Section 3.2.1) and the Quality Metrics we use (Section 3.2.2). As the final part of the Methodology chapter we describe our three implemented visualizations in Section 3.3.

A big chapter is Experiments (Section 4), where we give an overview of all tests we conducted and their results. The first tests were conducted on our test equation and are described in Section 4.1. The next round of tests was to find good algorithm configurations to run our optimization algorithms with. These are described in Section 4.2. With these algorithm configurations established, the next Section (4.3) contains tests on the same point cloud but with different data qualities. Finally, Section 4.4 contains descriptions and results of our tests on different point clouds with the same data quality.
The last main section is Discussion and Future Work (Section 5). It is made up of smaller chapters that each either summarize some aspect of our test results (Sections 5.1-5.3), presents some findings that result from our work (Section 5.4), or discuss what further work could be done on this topic (Sections 5.5-5.7).
At the end of this thesis, there is a chapter of tables containing all our test results in detail (Section 6).

CHAPTER 2

Related Work

There are many different optimization algorithms tuned to different problems. They differ in the amount and type of parameters they can work with as well as in the underlying principle. Some work on local data, such as ParamILS [HHLBS09], while others use global data to find new candidates to test, such as GEIST [TJA$^+$18] or IF-Race [BBS07].

In this chapter, we will first look at some general optimization algorithms that are used in Parameter-Space Optimization, then some specialized techniques will be discussed. Huang et al. give a good overview of existing specialized Parameter-Space Optimization techniques without evaluating their performance [HLY20]. They split the presented algorithms into three categories, which will be presented in Sections 2.3-2.5.

2.1 Parameter Types

An important concept when talking about optimization algorithms is parameter types. Not only can some algorithms just work with certain amounts of parameters, but also only certain types of them. At a basic level, there are discrete and continuous parameters. Discrete parameters can only take on certain fixed values (there can be an infinite number of them), an example would be Integers. On the other hand, continuous parameters might be restricted to a certain range, but they can take any value in that range and are not restricted to a subset. Such a parameter would be represented by for example a float value. An interesting subset of discrete parameters are categorical parameters, which do not represent a number, but a selection from a number of possible values. The most common example of these is boolean values, that are either true or false.

2.2 General Optimization Algorithms

An algorithm that is used often in Parameter-Space Optimization, whether as a tool or to compare another system too, is Gaussian Processes (GP) [WR06]. GPs use multidi-

mensional normal-distributed variables to model unknown functions based on empirical data. This can be used to predict the results of untested parameter configurations to efficiently select promising candidates to evaluate.
Another widespread optimization technique is Gradient Descent (GD) and its variations. Basic GD works by starting from a random point in the parameter space, computing the gradient at that point, and moving in the resulting direction for the next sample. In high-dimensional spaces this can lead to a lot of calculations that slow the process down, so some simplifications have to be found. A possibility to speed the algorithm up is Stochastic Gradient Descent (SGD), which only samples one dimension of the gradient each step [Bot98]. Of course, any compromise between SGD and standard GD is possible, considering just a subset of parameters in each iteration. The main disadvantage of GD is that the function to be sampled has to be derivable.

2.3 Simple Generate-Evaluate Methods

The Simple Generate-Evaluate Methods are the simplest and most straightforward methods. They first create a set of parameter configurations and then evaluate each of them to find the best candidate. The simplest algorithm in this category is brute-force, where candidates are chosen completely at random. For this to yield any good result, a lot of samples have to be taken in the evaluation phase, and valuable computing power gets wasted on unpromising candidates that could have been discarded using a more intelligent approach.
A more refined approach is F-race [BSPV02]. The idea of this algorithm is to iteratively test the generated candidates and exclude any future candidates where evidence can be gathered from the previous results suggesting that they won't be good solutions. This way less computing power is wasted on unnecessary parameter configurations and the algorithm can terminate sooner than a brute-force algorithm.

2.4 Iterative Generate-Evaluate Methods

Iterative Generate-Evaluate Methods repeat the two steps already present in Simple Generate-Evaluate Methods. They generate a set of candidates and evaluate them. Other than the simpler version, the results are used to again generate a new set of parameter candidates. This way, more interesting regions of the parameter space can be identified and the search conducted in a more controlled and directed fashion.
This category is the largest in Huang et al.'s overview [HLY20] and they split it up into four subcategories.
The first subcategory is **Experimental Design Based Tuning**. In this category falls for example CALIBRA by Adenso-Díaz and Laguna [ADL06]. CALIBRA iteratively narrows down the search space for each parameter to generate new candidates. The two major drawbacks of this approach are that it only works with up to 5 parameters and that the correlation of parameters is not considered at all.
The second subcategory is **Numerical Optimization Based Tuning**. Algorithms

from this category are only applicable to real or integer values, but not to categorical ones. They use numerical optimization methods to generate new parameter configurations for the next evaluation step.
The third subcategory is **Heuristic Search Based Tuning** and encompasses the most solutions available at the moment. They use some heuristic rule to generate new candidates each increment. One possibility to do this is modifying F-race to be executed iteratively. Balaprakash et al. present iterated F-race, which uses the surviving candidates of the previous iteration to generate new configurations similar to them [BBS07]. One disadvantage of iterated F-race is that it was not primarily designed to reduce computational cost and a large number of samples might be needed to reach acceptable results.
A quite simple but promising algorithm is GEIST by Thiagarajan et al. [TJA+18]. They build a graph of all possible parameter configurations and label each tested as optimal or not. These labels are then propagated through the whole graph and a new test-set is selected from the optimal-labeled nodes. This algorithm can deal with categorical parameters but does not support real numbers without discretizing them.
Another group in this subcategory are Meta-Evolutionary Algorithms as introduced by Mercer and Sampson [MS78]. There are many different approaches in this category, such as CMA-ES by Hansen and Lozano [Han06]. Although these evolutionary algorithms seem to reach very good solutions, they usually need a lot of iterations and evaluations to get there. Another problem is that many of them do not work with categorical parameters. Bäck et al. look back at 30 years of evolutionary algorithms for parameter optimization [BKvS+23]. They discuss the development of the field, and its major changes, and try to give a short outlook on what to expect in the future.
Hutter et al. introduced ParamILS [HHLBS09], an iterative generate-evaluate method that uses Iterated Local Search (ILS) [LMS03] to find new candidates in the generate step. Since ILS uses local neighborhoods, it requires the discretization of parameters and can not be applied to real numbers.
The fourth subcategory is **Model-Based Optimization Approaches**. This category encompasses algorithms that build a surrogate model of the problem to optimize, which is refined with each new test and used to generate new test cases. An example for this category is SMAC by Hutter et al. [HHLB11]. It uses a random forest model to fit even categorical parameters.
Another approach that uses forests was presented by Balaprakash et al. [BGW13]. They use dynamic trees that are iteratively updated with new experimental results and used to generate new candidates at promising regions. Gramacy et al. use a similar approach of Dynamic Gaussian Process Trees [GLM04], where they fit GP to the obtained results to determine new promising candidates.
It is also possible, to train a neural network on parameter configurations and the resulting performance [MAJ+17]. This neural network, again, can be used to find new candidates of parameter configurations. This is especially prevalent for using Bayesian optimization [SLA12], where a neural network learns a probability model of an objective function that needs to be optimized. It is then possible to get an estimated value for any point

in the parameter space, as well as its standard deviation. Recently, a lot of work has been done using this in tuning parameters specifically for runtime performance in the space of High-Performance Computing and AI. Wu et al. introduce 'ytopt' [WKB$^+$20], an auto-tuning framework that uses Bayesian optimization to tune Polly [GGL12] LLVM [Lat02] pragmas to achieve the fastest compiled code. Similarly using Bayesian optimization, Menon et al. introduce 'HiPerBOt' [MBG20] to tune platform and application level parameters, like compiler flags or runtime settings.

2.5 High-Level Generate-Evaluate Methods

High-Level Generate-Evaluate Methods are similar to Simple Generate-Evaluate Methods in that they only employ one generate and one evaluate step. Other than the simple version, they use some pre-computation step to generate promising candidates that are then evaluated as usual. This way, cheaper versions of some calculations can be used to find a rough estimate of good solutions before finding the actual best candidate. An example in this category is post-selection [YSMdO$^+$13]. It uses a shorter version of another algorithm to generate promising candidates before running the full algorithm. This is done for example with ParamILS [HHLBS09], which is aborted after a set number of iterations.

CHAPTER 3

Methodology

To create the optimization framework and find the best parameter configurations, we implemented multiple state-of-the-art parameter-space optimization algorithms, which will be described in Section 3.1. In the scope of our framework, we call these algorithms **strategies**. A strategy's task is to select parameter configurations for the **target** to test next. This is usually done by an algorithm, but could in the scope of the framework also be for example user input or read from any other source. A target is any parameterized operation that should be optimized. In our case, this is mainly SPSR, but as an additional aid in the development and debugging of the strategies, a simple test equation $(a - b + c - d + e)$ is available as a second target. To get better insight into the inner workings of an algorithm, and to find any problems or weaknesses, we also implemented the possibility to visualize the current state of the optimization using **visualizations**. The different visualizations available can be found in Section 3.3. One target, one strategy, and an arbitrary number of visualizations are required to start our framework.
Another important concept in the framework and the rest of this thesis is a **parameter configuration**. A parameter configuration is a specific selection of parameters to run a target with, so it is one single point of the parameter space. As will be described in the following section, not only targets but also strategies can require a selection of parameters to run. To avoid confusion between the two, we will call these **algorithm configurations**.
For each algorithm, we first ran a number of tests, manually looking for the best algorithm configuration to start it with if targeting SPSR. It would be interesting to run the algorithms recursively on themselves to find the optimal algorithm configurations, using the whole framework call as a new target and the algorithm configurations as parameter configurations. This, however, would be far too resource-intensive, as will be further discussed in Section 3.1. Instead, promising algorithm configurations were selected and tested against each other. After finding a good algorithm configuration for each algorithm, we tested them against each other on SPSR for different point clouds

and compared the results to find out which one yields the best results for this scenario. To do this, we looked at both the runtime of the optimization as well as the quality of the resulting reconstruction, indicated by the calculated Chamfer Distances described in Section 3.2.2.

3.1 Optimization Algorithms

In the following sections, we describe all five algorithms we implemented in detail. First, we discuss their original paper, as well as how exactly the algorithm works. Secondly, we describe any changes or additions we had to make for our implementation and any noteworthy uniqueness in our version. These changes are mostly due to the much larger parameter space we have to work with over what most authors tested with. Furthermore, some algorithms had to be adjusted for the framework or the target they were applied to. SPSR requires some discrete parameters and our framework only runs one test for each parameter configuration.

3.1.1 Brute-Force

Brute-Force is a well-known principle for algorithms in most fields of computer science. It works by solving a problem not by some sophisticated strategy, but by using more resources, such as computing power. The main advantage of this approach is that it is usually very easy to implement and can be used when the achieved improvement by an algorithm is not greater than the cost of implementing the algorithm in the first place.

Implementation

In our case, Brute-Force works by randomly selecting new parameter configurations to test until either a certain value has been reached, or for a set number of iterations. The number of tested configurations in each iteration can be set and should normally be equal to the number of CPU cores available. Fewer tests would waste processing time and more tests would lead to the possibility of reaching a termination criterion after the first batch of tests and doing unnecessary work. This separation into batches is not ideal and will lead to longer run times, but it fits better with the rest of the framework, making it possible to use all the visualizations without modifications.

3.1.2 GEIST

The first more complex algorithm we implemented is *Good Enough Iterative Sampling for Tuning* (GEIST) by Thiagarajan et al. [TJA+18]. It is a Heuristic Search Based Tuning algorithm (Section 2.4) that uses a graph of the parameter space that is split into an optimal and a non-optimal region to select parameter configurations for the next iteration.

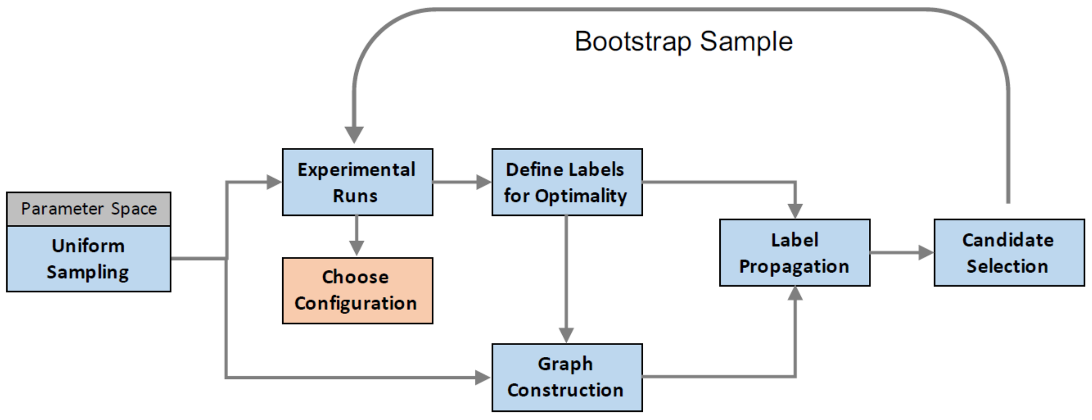


Figure 3.1: Visual representation of the GEIST algorithm. Reprinted from [TJA+18]

Algorithm

GEIST works by first creating a graph of the entire parameter space. This means that this algorithm only works for discrete data and continuous parameters have to be separated into intervals to create discrete data. To initialize GEIST, a number of random parameter configurations are selected and tested. In the graph, these configurations are then labeled as 'optimal' or 'non-optimal' depending on some heuristic. The heuristic we selected, as well as any other implementation-specific details, can be found in the next Section. These few node labels are now propagated through the graph, creating a not necessarily connected 'optimal' as well as a 'non-optimal' section. For each following iteration, random parameter configurations from the 'optimal' section are selected, and their results as well as the previous results used to initialize the label propagation. This is done until some termination criteria is met. A visualization of the whole algorithm can be found in Figure 3.1. GEIST is described in pseudo-code in Algorithm 3.1 as published by Thiagarajan et al. [TJA+18].

Implementation

Our implementation follows the paper by Thiagarajan et al. in most points. One difference is the strategy of propagating labels in the graph after testing each set of parameter configurations. In the original paper, the authors use Confidence Aware Modulated Label Propagation (CAMLP) [YFK16], which assigns a probability for each possible label to each node, in the case of GEIST that is optimal or non-optimal. To do this, Equation 3.1 is evaluated repeatedly, until it converges to a stable assignment.

$$p_{ik} = \frac{1}{Z_i}\left(b_{ik} + \beta \sum_{j \in N(i)} W_{ij} p_{jk}\right) \tag{3.1}$$

In this equation, b_{ik} is the prior probability of label k for node i. $N(i)$ denotes the set of neighbors of node i, p_{jk} is the probability of node j having label k. W_{ij} refers to the

Algorithm 3.1: GEIST, adapted from [TJA+18]

```
   Input: Parameter space $\mathcal{S}$, initial sample size $N_0$, theshold $\Delta_l$, number of
          iterations $T$, size of sample added in each iteration $N_+$.
 1 Initialize bootstrap set $\mathcal{B} = \{\}$.
 2 Initialize unseen test set $\mathcal{U} = \mathcal{S}$.
 3 Generate a uniform random sample $\mathcal{S}_0$ of size $N_0$ from $\mathcal{S}$.
 4 Update $\mathcal{B} = \mathcal{B} \cup \mathcal{S}_0$.
 5 Construct neighbourhood graph $G$ for $S$.
 6 loop for T iterations:
 7     Run experiments for samples in $\mathcal{B}$ and build $\{(c_i, y_i)\}_{i \in \mathcal{B}}$.
 8     Update $\mathcal{U} = \mathcal{U} \backslash \mathcal{B}$.
 9     Compute categorical label $\mathcal{L}(x_i)$, $\forall i \in \mathcal{B}$.
10     Predict the labels for all configurations in $\mathcal{U}$ using Equation 3.1.
11     Randomly select $N_+ optimal$ cases from $\mathcal{U}$ to build $\mathcal{S}_+$.
12     Update $\mathcal{B} = \mathcal{B} \cup \mathcal{S}_+$.
```

edge strength between nodes i and j, β controls the influence of neighboring nodes. As a normalization constant, Z_i scales the sum of all p_{ik} for a node to 1.0.
For our implementation, we realized that label propagation needs to be simplified to accommodate larger parameter spaces. While the largest number of nodes the original authors tested GEIST with is 25,920 [TJA+18], for SPSR we get parameter spaces with multiple millions of nodes, depending on the configuration. This leads to label propagation taking several minutes to complete even with our simplified implementation, which is a simple nearest-neighbor algorithm.
To determine the label of a tested parameter configuration, we use an 'optimal threshold', where any node with a value over that threshold is classified as optimal, while the rest are not optimal.
Instead of the fixed number of iterations that Thiagarajan et al. use, we developed a more dynamic termination criteria. We chose to lower the threshold dividing optimal and non-optimal nodes each turn by a static amount, as well as additionally if the number of optimal nodes does not decrease between iterations. If the threshold reaches zero percent, the algorithm terminates. This way the number of iterations that will be necessary does not have to be defined prior to the optimization run and we get the effect of simulated annealing.

3.1.3 Iterative F-Race

Algorithm

Iterative F-Race, as described by Balaprakash et al. [BBS07], improves on F-Race [BSPV02] and is also a Heuristic Search algorithm. The basic F-Race algorithm works similarly to Brute-Force (Section 3.1.1) in that it first selects a number of parameter configurations and then tests them. Different to Brute-Force, F-Race gathers evidence against configurations while testing them and skips them if enough evidence against them can be found. Iterated F-Race expands upon this idea by repeating this process multiple times. In each iteration, a number of elite configurations from the previous iteration are used to generate a probability value for each possible parameter configuration to be selected. Using these values, the configurations for the next iteration are selected.

Implementation

There is one big difference between the version of the algorithm described in the paper [BBS07] and our implementation. To gather evidence against certain parameter configurations, without prior knowledge of the parameter space, it would be necessary to test each of them against multiple point clouds. This way, if the first results are worse than any other configuration, the current one could be skipped. While this would be possible, this work aims to find an algorithm that works on a single point cloud. Additionally, during testing we found out that small numbers of parameter configurations per iteration lead to better results, meaning that with a CPU with 16 cores, all configurations can be tested in parallel, and skipping any would not lead to an increase in performance.
While the I/F-Race algorithm would work with continuous data, SPSR requires some discrete parameters, so our implementation uses a graph of the parameter space and calculates the probability of each node at each iteration. For continuous parameters, the interval between nodes can be selected arbitrarily small, mimicking continuous data at the expense of processing time.

3.1.4 ParamILS

ParamILS (Parameter Iterative Local Search) by Hutter et al. [HHLBS09] is an iterative generate-evaluate method for finding an optimum in a discrete parameter space.

Algorithm

ParamILS is based on an iterative first improvement procedure, where for each step single parameters are altered by one unit and then tested. If the result is an improvement, this changed configuration is taken as the new current configuration. Once no more better neighbors can be found, a local optimum has been reached and the algorithm moves in a random direction and repeats the process. The full process, as described by Hutter et al. can be found in Algorithm 3.2.

In this algorithm, Θ denotes the full parameter space, with θ being a single configuration from that space. r is the size of the random set of parameter configurations that is taken to initialize the algorithm, $p_{restart}$ the probability of restarting the whole search from a new random parameter configuration. The distance of movement in each perturbation step is determined by s. As the neighborhood $Nbh(\theta)$ we use the Manhattan neighborhood, so it is the set of configurations θ_i that differ in exactly one parameter to θ. The function $better(\theta_i, \theta_j)$ is a simple comparison of the estimated values resulting from the evaluations of θ_i and θ_j.

Algorithm 3.2: ParamILS, taken from [HHLBS09]

```
Input: Initial configuration $\theta_0 \in \Theta$, algorithm parameters $r, p_restart$, and $s$.
Output: Best parameter configuration $\theta$ found.
1  for $i = 1, ..., r$ do
2      $\theta \leftarrow$ random $\theta \in \Theta$;
3      if $better(\theta, \theta_0)$ then $\theta_0 \leftarrow \theta$;
4  end
5  $\theta_{ils} \leftarrow IterativeFirstImprovement(\theta_0)$;
6  while not $TerminationCriterion()$ do
7      $\theta \leftarrow \theta_{ils}$;
       // ===== Perturbation
8      for $i = 1, ..., s$ do $\theta \leftarrow$ random $\theta' \in Nbh(\theta)$;
       // ===== Basic local search
9      $\theta \leftarrow IterativeFirstImprovement(\theta)$;
       // ===== AcceptanceCriterion
10     if $better(\theta, \theta_{ils})$ then $\theta_{ils} \leftarrow \theta$;
11     with probability $p_{restart}$ do $\theta_{ils} \leftarrow$ random $\theta \in \Theta$;
12 end
13 return overall best $\theta_{inc}$ found;
14 Procedure $IterativeFirstImprovement(\theta)$
15 repeat
16     $\theta' \leftarrow \theta$;
17     foreach $\theta'' \in Nbh(\theta')$ in randomized order do
18         if $better(\theta'', \theta')$ then $\theta \leftarrow \theta''$;break;
19     end
20 until $\theta' = \theta$;
21 return $\theta$;
```

Implementation

For our implementation, we chose a fixed number, *max_repetition*, of times the same local optimum can be reached as the $TerminationCriterion()$. After reaching the same local optimum *max_repetition* times, it is returned as the final result. While many members of the neighborhood of a parameter configuration can be evaluated at the same time, we still chose a random configuration with a better value instead of selecting the best one. This has led to better results in some tests. We do not use $p_{restart}$ ($p_{restart} = 0$) in our implementation.
A larger difference between our implementation and the one by Hutter et al. [HHLBS09] is the perturbation step. In the original paper, a random neighbor of the current parameter configuration is selected s times. In larger parameter spaces this value s has to be too high for this strategy to work effectively. After many perturbation steps, their directions even out, leading to no significant change. To avoid this problem, for each parameter,

we select a random number m of steps to move from the interval $[-m_{max}, m_{max}]$. m_{max} is determined by multiplying the number of possible values of the parameter with the *perturbation_distance* parameter described in Section 4.2.3. This results in a move with a maximum length of *perturbation_distance* times the size of the parameter space in a random direction.

3.1.5 Post-Selection

Algorithm

The idea of the post-selection algorithm is to generate a set of high-quality candidate parameter configurations in a first step, and then determine the best of these configurations in a second step [YSMdO+13]. For the generation step, any number of strategies can be used, as long as they lead to good solutions quickly. To achieve this, it is possible to use faster convergences, limit the number of iterations the algorithm can run for, or accept solutions earlier in another way. The evaluation step's job is to determine which of the previously selected candidates is the best solution overall. This is achieved by running a more accurate strategy, either by choosing a different one than in the generation step or by tweaking its algorithm configuration. The amount of resources spent on each parameter configuration is usually much larger than in the first step.

Implementation

For our implementation, we use the other four implemented algorithms both as generation and evaluation algorithms. Since we want the results for one single point cloud, the values for each candidate parameter configuration are already known after the generate step. Because of this, and because some of the algorithms would not work on a disconnected set of candidates, we do not limit the evaluation step strictly to the generated candidates. They are used to initialize the evaluation algorithms, which then find the optimal parameter configuration from this initial selection.
Each evaluated algorithm receives the full set of tested parameter configurations to initialize, but they use different amounts of them. GEIST uses all previous results to initialize its optimal/non-optimal graph and starts selecting new configurations from there. Iterative F-Race uses the *nr_good_results* best configurations from the generate step and calculates the next probabilities from them. ParamILS starts from the best parameter configuration of the generate step and moves on from there. Brute-Force, as expected, ignores any previous results and was therefore not used in the evaluation step. To achieve a broader spread of possible candidates, it is possible to run the generate step multiple times and pool all results together. This is useful to start an algorithm configured to be very fast, but not very accurate, multiple times and get the best results, as opposed to relying on it getting the correct solution on the first try.

3.2 Screened Poisson Surface Reconstruction

Screened Poisson Surface Reconstruction (SPSR) by Kazhdan and Hoppe is an algorithm to construct watertight surfaces out of oriented point clouds [KH13]. It expands on Poisson Surface Reconstruction (PSR) [KBH06] by adding the points explicitly as interpolation constraints. Both algorithms assume an indicator function that is positive inside the model and zero (PSR) or negative (SPSR) outside. They start by converting the oriented point cloud into a vector field that should be equal to the indicator function's gradient $\nabla_{\mathcal{X}}$. This is achieved by minimizing the error function:

$$E(\mathcal{X}) = \int \|\nabla_{\mathcal{X}(p)} - \vec{V}_{(p)}\|^2 dp \tag{3.2}$$

In this equation, $\mathcal{X}$ is the indicator function and V is the vector describing point p. This minimization can be solved using Poisson equations. In practice, this is done using an octree and fitting a B-spline function into each node of the tree. After combining these B-splines, the resulting function's zero set should be close to the model's surface, although there can be some errors due to noisy data and some approximations in the calculation. To reduce this problem, PSR subtracts the average value of the function at the input samples. This is only a global offset, so the average error at all points is zero, but there are potentially errors that cannot be fixed by this process.
SPSR improves on PSR in this regard by explicitly interpolating the points. Kazhdan and Hoppe do this by extending Equation 3.2 with a term that discourages a deviation from zero at the samples, resulting in Equation 3.3.

$$E(\mathcal{X}) = \int \|\nabla_{\mathcal{X}(p)} - \vec{V}_{(p)}\|^2 dp + \frac{\alpha * Area(\mathcal{P})}{\sum_{p \in \mathcal{P}} \omega(p)} \sum_{p \in \mathcal{P}} \omega(p)\mathcal{X}^2(p) \tag{3.3}$$

Here, $\mathcal{P}$ is the set of input points, and $\omega(p)$ is the weight of one such point. $Area(\mathcal{P})$ is the estimated area of the reconstructed surface and α is used to balance between fitting the gradients and fitting the values.

We chose SPSR as the algorithm to test the optimization algorithms of this thesis because it is the gold standard for mesh reconstruction from point clouds at the moment and because there is a good variety of parameter types used for it. In particular, we use the implementation by Meshlab [1], which can be configured using a number of parameters and parameter types, continuous, discrete as well as categorical ($True/False$).

3.2.1 Parameters

Several parameters need to be set to run the Meshlab version of SPRS; these are:

- **Merge all visible layers** ($visibleLayer$): tells the program to use all visible layers instead of just the current.

[1] http://www.meshlab.net/

- **Reconstruction Depth** (*depth*): the maximum octree depth.
- **Adaptive Octree Depth** ($fullDepth$): specifies how far up the octree will be adapted, at coarser depths it will hold all $2^d * 2^d * 2^d$ nodes.
- **Conjugate Gradients Depth** (*cgDepth*): up to this depth a conjugate-gradients solver will be used, further on Gauss-Seidel relaxation will be used.
- **Scale Factor** (*scale*): the ratio of the diameter of the octree cube to the bounding cube of the point cloud.
- **Minimum Number of Samples** ($samplesPerNode$): the minimum number of points that need to fall into an octree node.
- **Interpolation Weight** ($pointWeight$): the importance of interpolating point values over gradients (α in Equation 3.3).
- **Gauss-Seidel Relaxations** ($iters$): the number of Gauss-Seidel iterations to be performed at each level of the octree.
- **Confidence Flag** ($confidence$): whether the length of normal vectors should be used as weights during the reconstruction.
- **Pre-Clean** (*preClean*): if enabled, a pre-clean step is run, removing all unreferenced points as well as those without normals.

3.2.2 Quality Metrics

We implemented two quality metrics to evaluate the quality of SPSR. We chose Chamfer Distance since it gives an idea about the general reconstruction quality, as well as Hausdorff Distance, which gives an idea about a local worst case. For all optimizations, we minimized the Chamfer Distance, but also give the Hausdorff Distances in the result tables. All used point clouds are scaled to the unit cube, so the Chamfer Distance can be used as a comparison between different clouds as well.

Chamfer Distance

For the first metric of the quality of a reconstruction, we chose the Chamfer Distance [BTBW77]. The Chamfer Distance of a reconstructed model is defined as the sum of the minimal distances of each reconstructed point to the nearest point on the original model. Alternatively, the average of minimal distances can be used ([DDR16]). The Chamfer Distance of a reconstruction R to the ground-truth G is given in Equation 3.4.

$$CD(R,G) = \sum_{r_i \in R} \min_{g_j \in G} |r_i - g_i| \tag{3.4}$$

Hausdorff Distance

A second value we calculate for each test is the Hausdorff Distance. It is a distance metric between two sets of points that can be visualized as a 'worst-case' travel distance from one set to the other. The Hausdorff Distance is defined as the maximal minimal distance from any point on one set to any point on the other set, as expressed in Equation 3.5 [HKR93].

$$H(A,B) = max(h(A,B), h(B,A)) \tag{3.5}$$

where

$$h(A,B) = \max_{a \in A} \min_{b \in B} ||a - b|| \tag{3.6}$$

3.3 Visualization

The goal of the visualization framework was to help with the understanding of the selected strategies. Originally, this meant being able to find possible errors during the implementation that might not be visible in the final result or any debugging output. If the intermediate states of the strategy did not conform with what we expected, we were able to check if there was something wrong in the code, or if we needed to adjust our understanding of the algorithm. In the later stages, it was important to understand how each algorithm worked in order to be able to select promising algorithm parameter candidates for the first part of the testing process. Since we could not run the strategies recursively, we needed to choose good candidates based on our understanding of the individual algorithms.
We implemented three different visualizations that can be used independently of each other and update during the optimization process.

3.3.1 Parameter Change Visualizer

The first visualization method is the **Parameter Change Visualizer**. It shows the development of one specific parameter during the optimization process. For each iteration, the average of the parameter across all tested parameter configurations, as well as the average over the last iteration's configurations, are plotted. Additionally, for each of these values, the variance of the parameter in the respective time frame is shown to indicate whether the algorithm is focusing on said parameter value, or if the parameter does not have a lot of influence on the result. Lastly, the graph also shows the value the tracked parameter has taken in the currently best found configuration. All this can be seen in Figure 3.2. It can be seen here that the parameter 'samplesPerNode' focuses on a value around 6 throughout the optimization.

3.3.2 Correlation Visualizer

The second implemented visualization is the **Correlation visualization**. It shows the whole parameter space projected onto a two-dimensional grid spanned by two

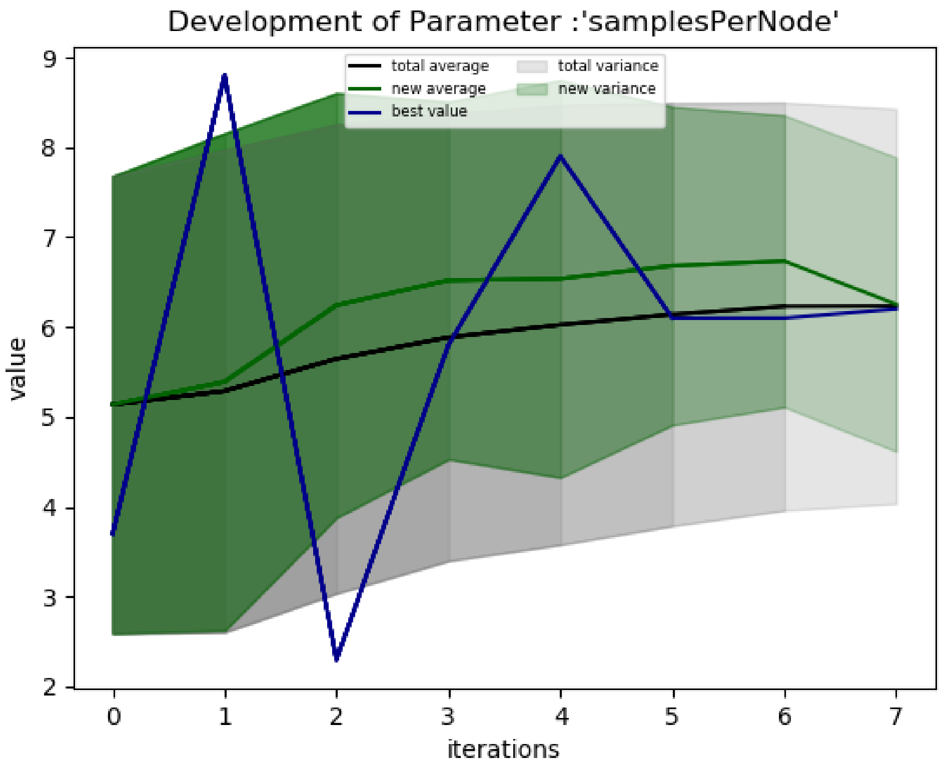


Figure 3.2: Parameter Change visualization of an IF-Race run on SPSR for the 'original - Armadillo' point cloud showing the optimization of the parameter 'samplesPerNode'.

selected parameters. At each point of the grid, a dot represents all tests done with the representative parameter combination. The size of the dot specifies the number of tests, and the color indicates the quality of the solution. One can select between showing the average of all values achieved with the corresponding combination of the two parameters, or the best result with said configuration. This means that a larger dot is a parameter combination chosen by the strategy to be tested more often, and a greener dot implies a better result. As an example, a large red dot in this view would show that the strategy does not work correctly, or there is an error in the strategy's logic since it does not make sense to test a bad parameter configuration multiple times. It is also possible in the Correlation Visualization to highlight some configurations with a circle, depending on the strategy. This is done, for instance, for IF-Race (Section 3.1.3) to show the parameter configurations that were selected to create the next probability distribution from. An example of a Correlation Visualization can be found in Figure 3.3. Here, we see the correlation between the parameters 'depth' and 'samplesPerNode' of a GEIST run on SPSR for the 'original - Armadillo' point cloud. As can be seen here, both parameters have a high impact on the quality of the result, with a high 'depth' and

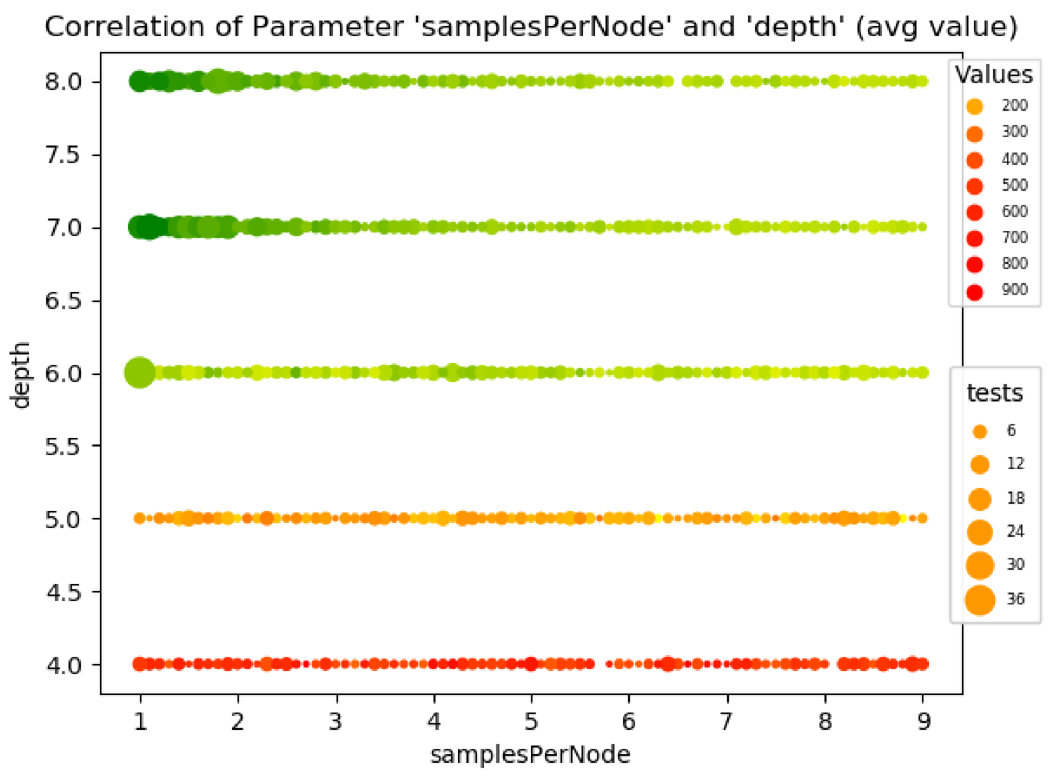


Figure 3.3: Correlation visualization of a GEIST run on SPSR for the ‘original - Armadillo’ point cloud showing the correlation between ‘depth’ and ‘samplesPerNode’.

low ‘samplesPerNode’ yielding the best results.

3.3.3 MDS Visualizer

The last visualization we implemented is the **MDS Visualizer**. MDS, or multidimensional scaling, projects the parameter space onto a 2D plane. It does so in a way to have similar points (parameter configurations with similar resulting values in our case) end up close to each other. As for the Correlation Visualizer, we color all dots according to their resulting value. For size, we use one selectable parameter. This way, it is very easy to see how much influence a parameter has on the outcome of SPSR. If only dots with similar sizes are within the green area of the plot, then it is important for the selected parameter to be of that size. If sizes are spread around the whole plot and do not correlate with color, then the parameter has no significant impact on the resulting value. As an example, in Figure 3.4 we visualized the ‘samplesPerNode’ parameter of a GEIST run on SPSR for the ‘original - Armadillo’ point cloud. As can be seen here, the color and the size of the dots correlate, and only very small values of ‘samplesPerNode’ led to good results.

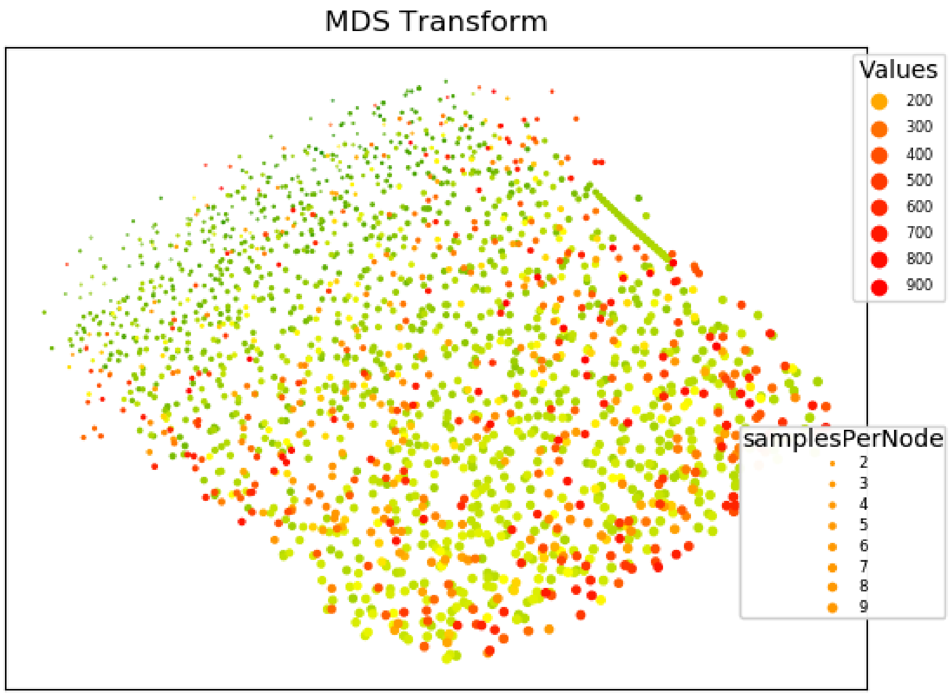


Figure 3.4: MDS visualization of a GEIST run on SPSR for the ‘original - Armadillo’ point cloud. The parameter highlighted by the dot color is ‘samplesPerNode’. ‘Values’ is the Chamfer Distance of the resulting construction.

On the other hand, in Figure 3.5 we have the same optimization run but visualized the ‘scale’ parameter. Here, there is no correlation between the size and color of the dots, which means that the ’scale’ parameter has no strong impact on the resulting Chamfer Distance. There is still a green cluster on the right side of the graph, which indicates another parameter being influential on the value.

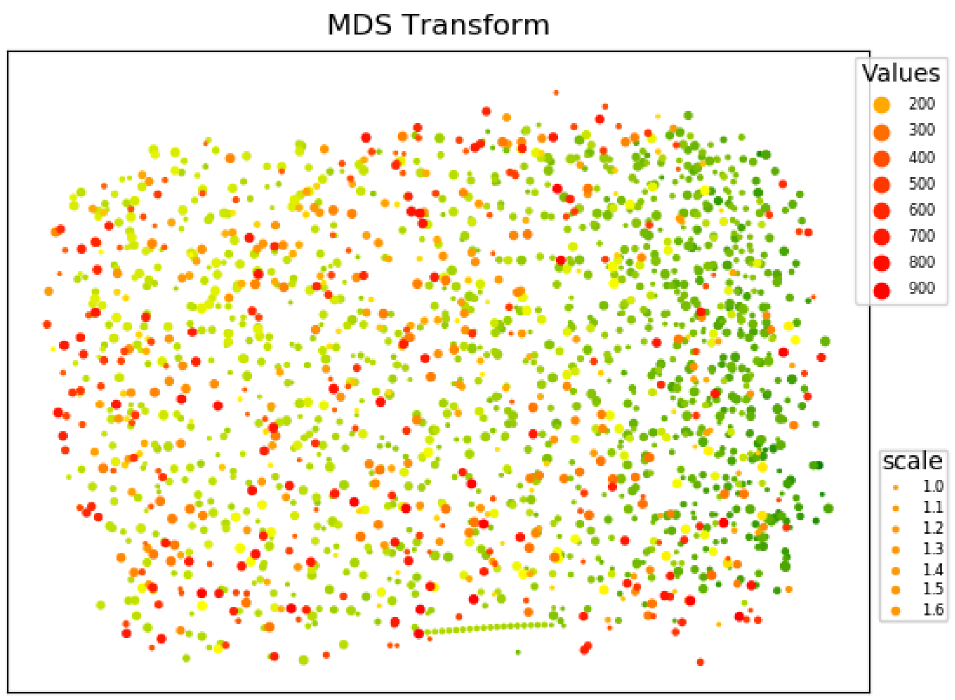


Figure 3.5: MDS visualization of a GEIST run on SPSR for the 'original - Armadillo' point cloud. The parameter highlighted by the dot color is 'scale'. 'Values' is the Chamfer Distance of the resulting construction.

CHAPTER 4

Experiments

To determine the best parameter-space optimization algorithm for use with SPSR, as well as the optimal parameters for our test set, it was necessary to conduct a large number of tests, some manual, some automated. All tests were run on the same computer, an Ubuntu 20.04 machine with an AMD Ryzen 7 3700X CPU, to be able to fairly compare run times. Only a few tests to generate output with specific parameters without the need of timing an optimization process were conducted on a second computer, another Ubuntu 20.04 machine with an Intel I7-9700K, 32GB memory, and a Nvidia GTX 970. We used two different datasets in our tests. The first is the 'famous' dataset assembled by Erler et al., the second one is an extract from the 'ABC' dataset [KMJ$^+$19]. For both, we used the preprocessed versions generated for Points2Surf [EGO$^+$20]. Both datasets contain point clouds of varying quality and ground-truth meshes to compare the reconstruction to. For the famous dataset, each point cloud is available in its original, noise-free, extra noisy, sparse, and dense version. The different qualities of the 'Armadillo' point cloud can be seen in Figure 4.1. In the ABC dataset, a default, noise-free, and extra noisy version are available, they are visible in Figure 4.2. We tested the effects of these different quality levels and the results can be found in Section 4.3. For the ABC dataset, we use synonyms for the used point cloud since they do not have speaking names:

- **cloud 1**: 00010218_4769314c71814669ba5d3512_trimesh_013
- **cloud 2**: 00017012_cd8dbafbc2a3422eb55090d7_trimesh_000
- **cloud 3**: 00991549_ecacd48a6851e8a6a3f8e137_trimesh_001
- **cloud 4**: 00018869_72063a0c38b94f71a6524566_trimesh_000

We conducted tests in two phases. The goal of the first phase was to find good algorithm configurations for each parameter-space optimization algorithm, as described in Section

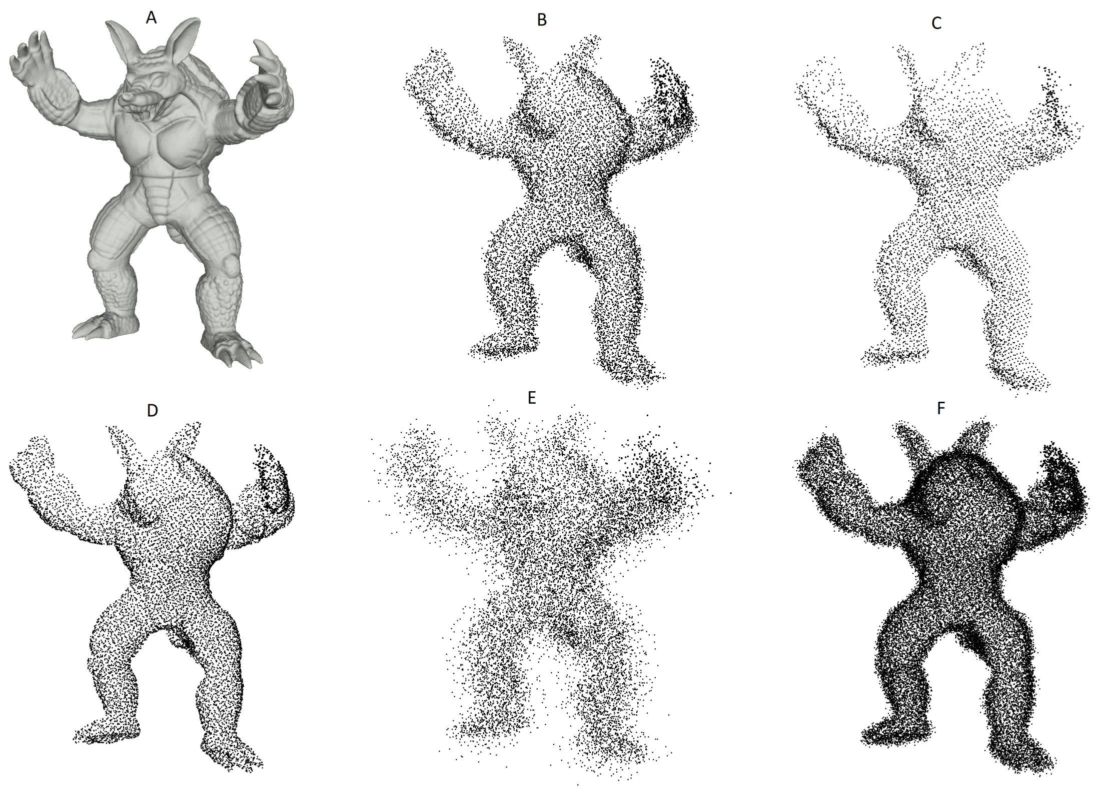


Figure 4.1: Different qualities of the 'famous' - 'Armadillo' point cloud. Image A shows the ground-truth mesh. B-F show the point cloud in the following datasets: B: original, C: sparse, D: noise-free, E: extra_noisy, F: dense

4.2. In the second phase, we tested the previously found algorithm configurations against different point clouds (Section 4.4), as well as different qualities of the same point cloud (Section 4.3).

4.1 Test Equation

To ensure our algorithms work, we first tested them on our test equation $a - b + c - d + e$ with all values being from the range $[-5, 5]$. The expected optimum for a minimization would of course be $a = -5, b = 5, c = -5, d = 5, e = -5$.

4.1.1 GEIST

Our GEIST implementation found the absolute best solution. A correlation visualization between a and b can be found in Figure 4.3. In this graph, it is very clear that the algorithm focuses in on the optimal region and does not need to test a lot of parameter configurations in the rest of the parameter space.

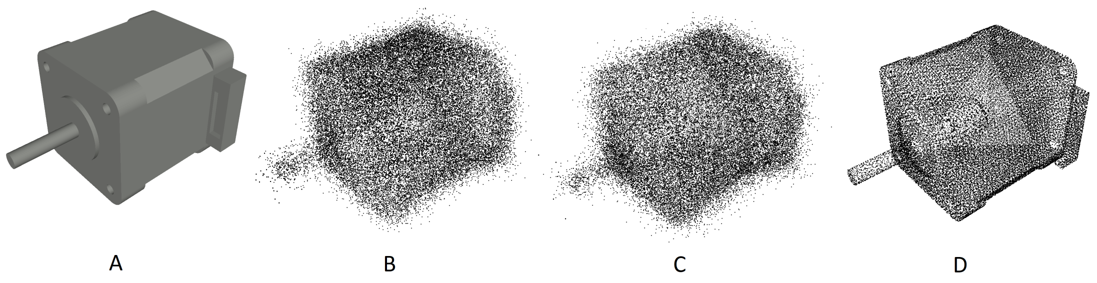


Figure 4.2: Different qualities of the ‘ABC’ - ‘cloud1’ point cloud. Image A shows the ground-truth mesh. B: original, C: extra_noisy, D: noise-free

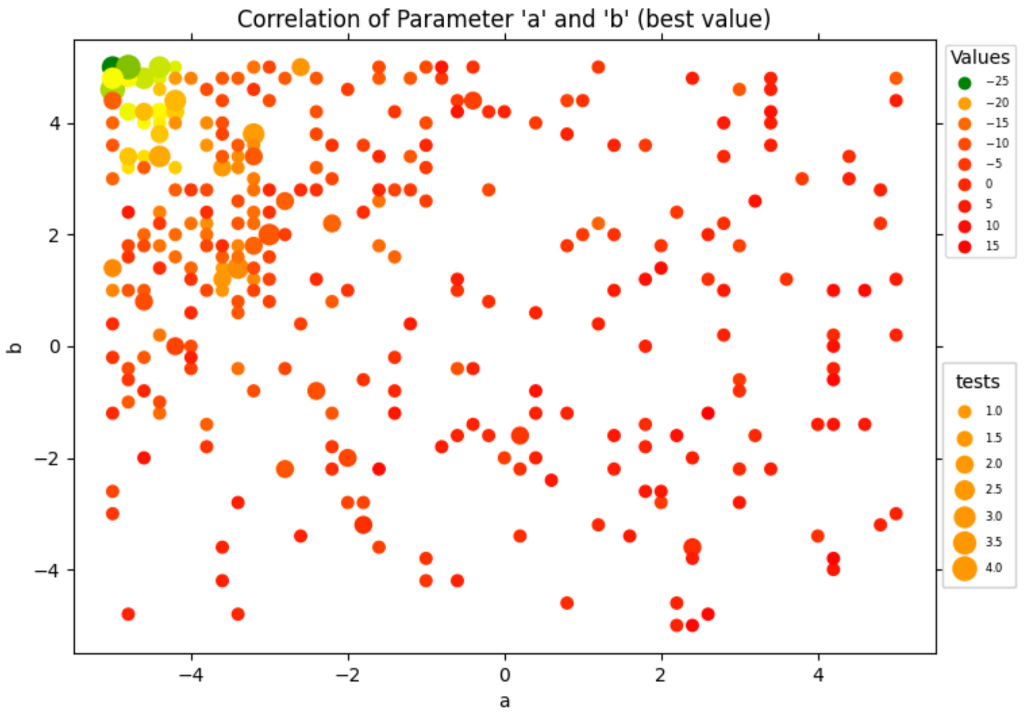


Figure 4.3: Correlation between parameters ‘a’ and ‘b’ of a GEIST run on our test equation

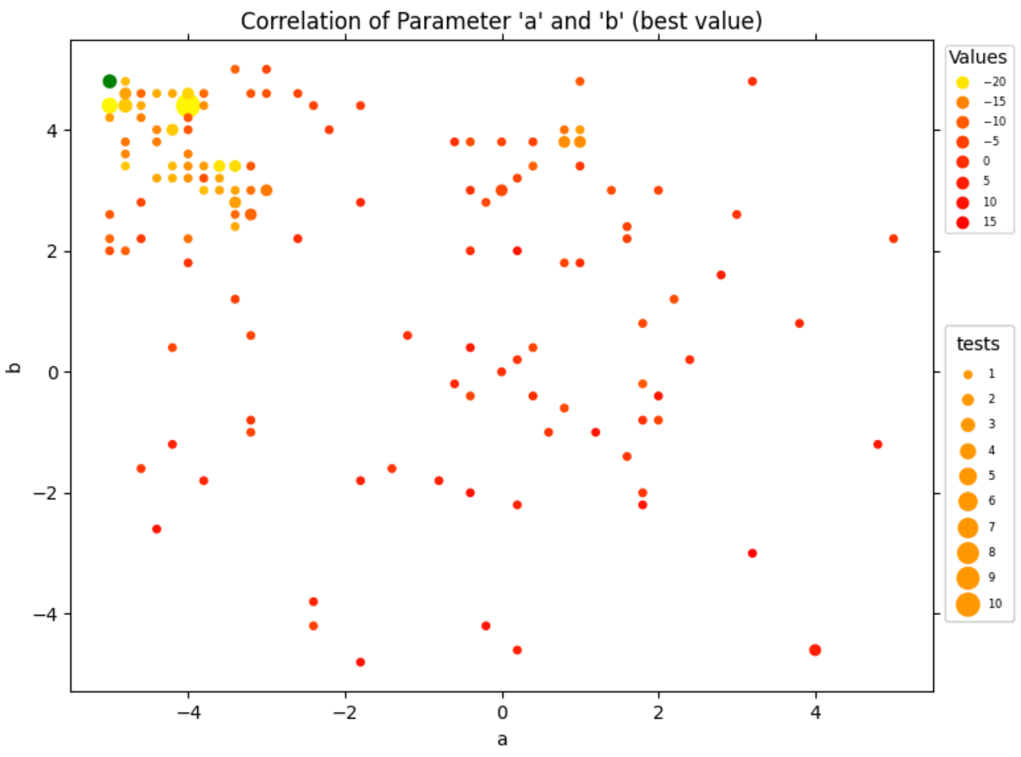


Figure 4.4: Correlation between parameters ‘a’ and ‘b’ of an IF-Race run on our test equation

4.1.2 IF-Race

IF-Race did not find the perfect solution, the best one it reached is $a = -5.0, b = 4.8, c = -2.0, d = 5.0, e = -5.0$. This it did after far fewer tests than GEIST, as can also be seen in Figure 4.4. In this graph, far fewer points can be found than in the GEIST graph (4.3), indicating that the algorithm took far fewer tests to terminate.

4.1.3 ParamILS

Like GEIST, ParamILS also found the optimal solution. The correlation visualization in Figure 4.5 for ParamILS shows very clearly how the algorithm behaves. It first tested some random parameter configurations and then started to move on from the best it could find – in this run this was around $a = -2.5, b = 4.8$. From there it started to find better neighbors and move to them. It is clearly visible how it had to check the area around its initial optimum and optimize other dimensions, until finally moving towards $a = -5$; even the single tests in the negative b direction when looking for better neighbors

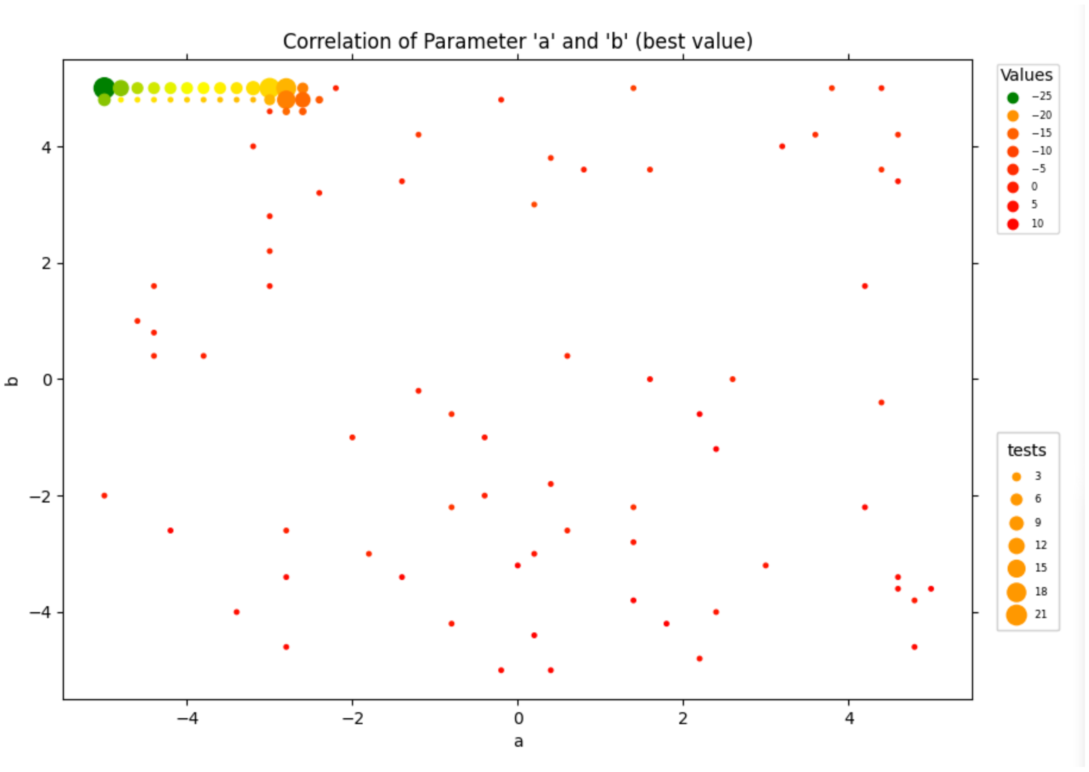


Figure 4.5: Correlation between parameters ‘a’ and ‘b’ of a ParamILS run on our test equation

can be seen.

4.1.4 PostSelection

The best parameter configuration PostSelection could find was $a = -3.4, b = 4.2, c = -5.0, d = 5.0, e = -5.0$. As can be seen in Figure 4.6, a lot of tests had to be conducted to reach this result, but again not many of them are in the uninteresting regions of the parameter space, indicating the algorithm works as expected. Still, for this rather small example, PostSelection is probably a too complex algorithm and does a lot of unnecessary work.

4.2 Algorithm Configuration

The first round of tests we conducted aimed to find desirable algorithm configurations to start the different parameter-space optimization algorithms with. These desirable configurations can either yield the best results or terminate the quickest. To find said configurations, we used manually selected tests instead of any automation due to the immense size of the parameter space. While it would be possible to apply any of the

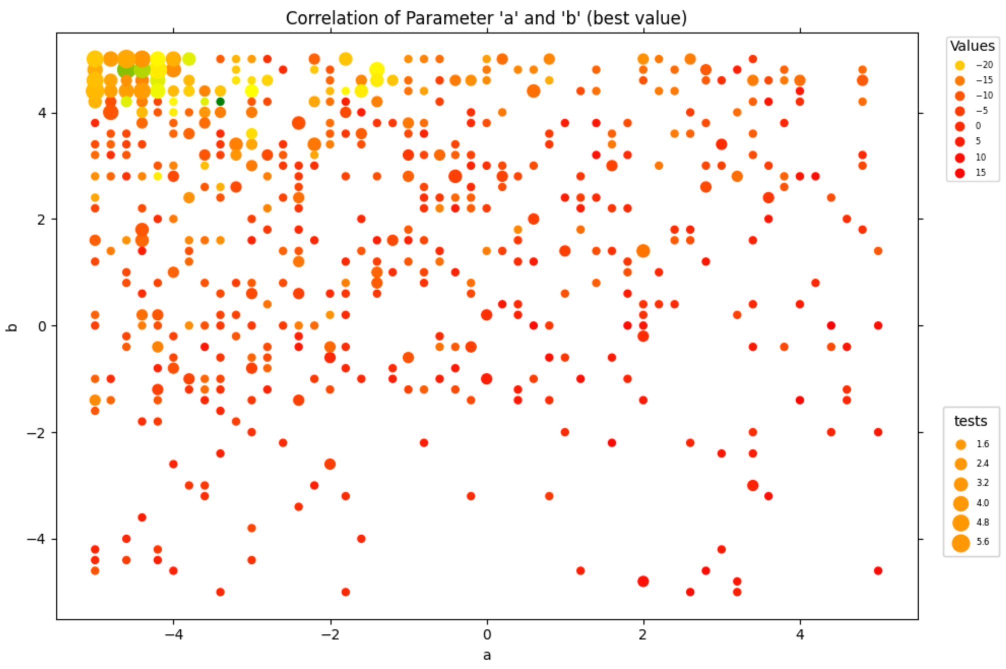


Figure 4.6: Correlation between parameters ‘a’ and ‘b’ of a PostSelection run on our test equation

algorithms recursively to itself to find a good algorithm configuration, this is not feasible due to the long run times. Each of the hundreds of necessary configurations for one optimization would have to be tested multiple times, each test taking up to four hours (in the case of GEIST and PostSelection).
The parameter space for these tests we selected as follows ($[min, step, max]$):

- $cgDepth \in [0, 1, 1]$
- $depth \in [4, 1, 8]$
- $fullDepth \in [5, 1, 8]$
- $iters \in [6, 1, 10]$
- $pointWeight \in [2.0, 0.2, 8.0]$
- $preClean \in \{True, False\}$
- $samplesPerNode \in [1.0, 0.1, 9.0]$
- $scale \in [0.9, 0.1, 1.7]$

- $visibleLayer \in \{True, False\}$

The limits of this parameter space were heuristically determined by tests, leaving some space around the expected optimum. *cgDepth* only ever took the value of 0 or 1, so no more options were allowed. An exception is *depth*, which often improves the resulting value the higher it is, but also increases the run time significantly. So while higher *depth* could lead to even better results, the parameter was limited to 8 to enable more tests in the same time. The $confidence$ parameter we fixed to $False$ for comparison sake, since not all datasets contain weighted normals that would be needed for that feature. This results in a 9-dimensional discrete parameter space with $18,079,200$ possible parameter configurations. Using our testing machine, each individual SPSR run takes three seconds using a *depth* of 8, including calculating Chamfer and Hausdorff Distance. This means that testing the whole parameter space would take roughly one and a half years.

4.2.1 GEIST

There are several parameters that can be tweaked when starting our version of the GEIST algorithm. To control the optimal threshold, its starting value can be set via *optimal_threshold*, the fixed change each iteration with $fixed\_threshold\_change$, and the additional change to speed up convergence using *threshold_change*. Additionally, the number of starting parameter configurations as well as the number of configurations to test per iteration can be set using $start\_size$ and run_size.
The algorithm configurations we chose to test based on previous experience while implementing GEIST as well as their corresponding results can be found in Table 6.1. A visualization of the results can be seen in Figure 4.7 on the example of the 'famous_original / Armadillo' dataset using SPSR. This graphic shows that GEIST leads to consistently good results, with the Chamfer Distance ranging from 117.3 to 119.8. Unfortunately, the run time is far higher than that of most other algorithms. Depending on the algorithm configuration, it can take from one up to five hours on our testing machine. As expected, the best results are achieved by a slow convergence with large iteration sizes. When trying to achieve good results in a reasonable time, the results suggest it would be best to choose large initial and iteration sizes but limit the number of iterations by selecting bigger threshold changes. Configurations 2 and 4 were run more often than the rest since they showed a very high standard deviation after the first 4 tests.

All results of our tests with different GEIST configurations can be found in Table 6.1. This table shows the eight algorithm configurations we selected with their individual results of at least three test runs. Additionally, an average and the standard deviation are given for easier comparison between the configurations. While there are configurations that might lead to better results, only these eight were chosen due to constraints on the run time. Not only is GEIST already one of the slower algorithms and an even slower version would not be recommendable, but we also had to consider the additional tests we were conducting with the selected algorithm configuration on different datasets and qualities.

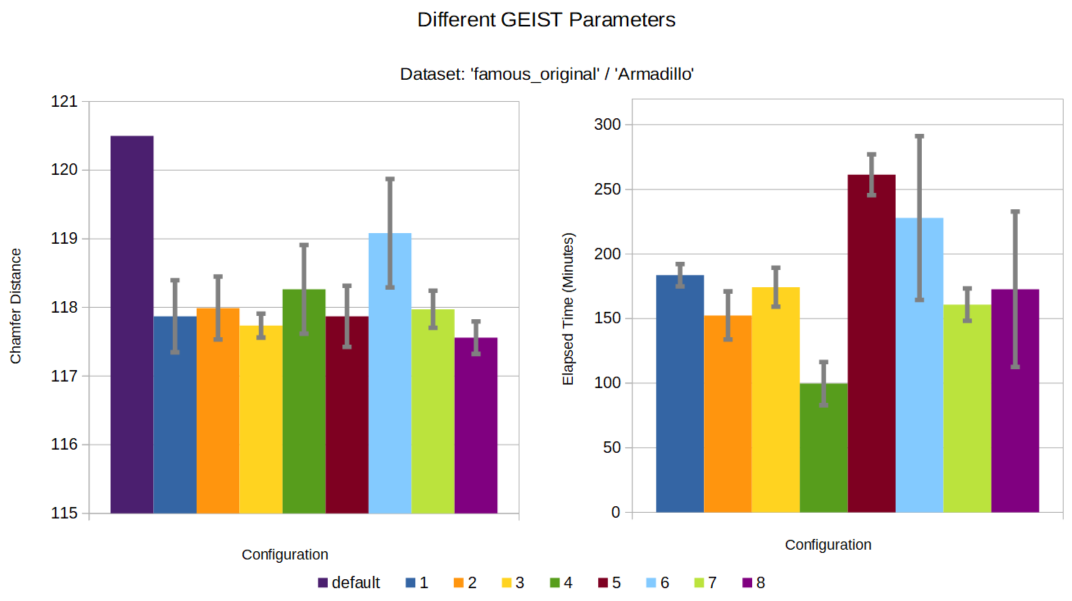


Figure 4.7: visualization of the results of the manual tests to find a good parameter configuration to run GEIST with. The left graph shows the resulting Chamfer Distance in comparison to the default parameters, the right graph shows the needed time in minutes. The grey lines on the bars show the standard deviation. Each bar corresponds to one configuration in Table 6.1 as specified by the configuration number.

The version we selected for testing on different point clouds and datasets in the next sections is config 8. This configuration takes an average of 173 minutes, being around average over all configuration, but achieves the best result of 117.6. Config 8 consists of the following parameters:

- $start\_size = 1500$
- $run\_size = 150$
- $optimal\_threshold = 0.3$
- $fixed\_threshold\_change = 0.02$
- $threshold\_change = 0.04$

4.2.2 IF-Race

Our version of the Iterated F-Race algorithm has multiple algorithm parameters needed to start it. *start_size* and *iteration_size* determine how many parameter configurations will be tested for the initial set of parameter configurations, or each iteration respectively. To decide how many of the best-known parameter configurations should be used to

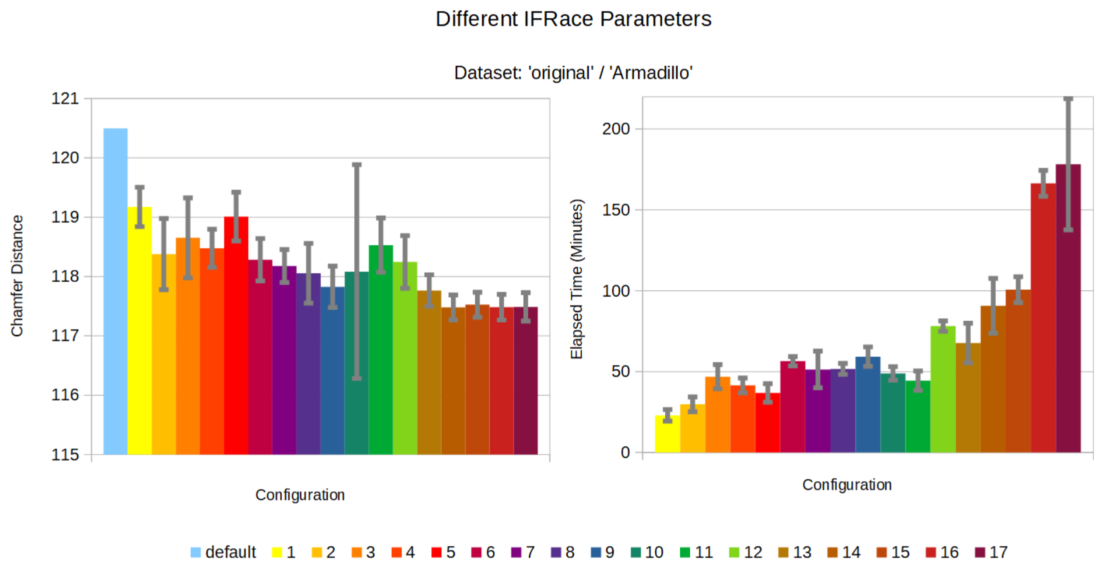


Figure 4.8: visualization of the results of the manual tests to find a good algorithm configuration to run IF-Race with. The left graph shows the resulting Chamfer Distance in comparison to the default parameters, the right graph shows the needed time in minutes. The grey lines on the bars show the standard deviation. Each bar corresponds to one configuration in Table 6.2 as specified by the configuration number.

calculate the next iteration's probabilities, $nr\_good\_results$ can be used. For controlling the algorithm's termination behavior, on the one hand, there is a hard set maximum of iterations, $max\_iterations$. On the other hand, $annealing\_factor$ controls how fast the standard deviation of the used normal distributions shrinks per iteration, according to Equation 4.1.

$$stddev_i = stddev_{i-1} * annealing\_factor \tag{4.1}$$

The standard deviation is multiplied with the $annealing\_factor$ each iteration, so to switch annealing off, one can simply set $annealing\_factor$ to 1.0.
Figure 4.8 shows the effects of different algorithm parameters on the run time and resulting Chamfer Distance of SPSR on the ‘famous_original / Armadillo‘ point cloud. As can be seen, a higher $annealing\_factor$ generally leads to a better result but causes a longer run time. While an $annealing\_factor$ of 0.9 results in a very consistent Chamfer Distance of under 118, it also takes around three hours to terminate. The fastest algorithm configuration, finishing in around 25 minutes, still reaches results of under 120. An $iteration\_size$ of 30 or 45 looks to be preferable over 15, causing a small increase in run time, but also a noticeable improvement in the result. For $nr\_good\_results$, again, a medium value of 10 seems to reach a good balance between speed and accuracy. All results of our manual tests can be found in Table 6.2. The algorithm configuration we chose for the next round of tests is number 14:

- $start\_size = 30$
- $iteration\_size = 45$
- $nr\_good\_results = 10$
- $max\_iterations = 20$
- $annealing\_factor = 0.6$

4.2.3 ParamILS

There are only three parameters relevant to starting our implementation of ParamILS. Firstly, there is the *starting_sample* to set the number of parameter configurations to randomly select for initializing the algorithm. This value is called r in the original algorithm, which can be seen in Algorithm 3.2. Secondly, to adjust the distance to move after finding a local optimum, *perturbation_distance* can be used. It denotes the maximal fraction of the parameter range to move for each parameter, so it is not equal to Hutter et al.'s s. The last parameter is *max_repetition*, which is the number of times the same local optimum can be reached before the algorithm terminates. While not having a direct equivalent in the original algorithm, this value controls the $TerminationCriterion()$ function of Algorithm 3.2.
As can be seen in Figure 4.9, the results for the ParamILS algorithm on the 'Armadillo' dataset are very inconsistent. For each algorithm configuration, 5 runs were performed and their value and run time average and standard deviation were visualized in the graphic. While the run time could be as low as under 2 minutes, it could also reach over 100 minutes at certain times, both with the same *perturbation_distance* of 0.6 and a *starting_sample* of 15, just allowing for 5 or 6 repetitions of the same optimum respectively.
The resulting values are just as inconsistent as the run times. They contain good results of 117.4, but also far worse values as high as 122.9. Due to the high standard deviation in both value and run time, no clear best configuration can be determined. All test results can be found in Table 6.3. In the end, we settled on the following parameter configuration to use in the second round of tests:

- $starting\_sample = 150$
- $perturbation\_distance = 0.6$
- $max\_repetitions = 5$

4.2.4 PostSelection

Our implementation of PostSelection needs four parameters to run. Firstly, there are *strategy_generate* and *strategy_evaluate*, which are of course the algorithms to be used

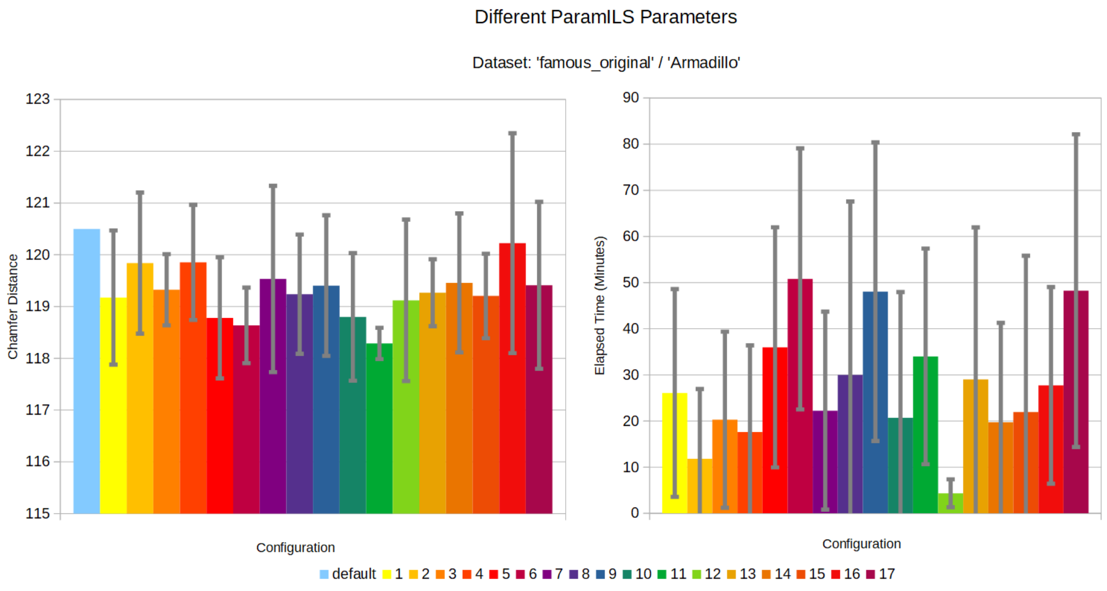


Figure 4.9: Visualization of the results of the manual tests to find a good algorithm configuration to run ParamILS with. The left graph shows the resulting Chamfer Distance in comparison to the default parameters, the right graph shows the needed time in minutes. The grey lines on the bars show the standard deviation. Each bar corresponds to one configuration in Table 6.3 as specified by the configuration number.

for the generate and the evaluate step respectively. These algorithms have themselves to be configured according to the algorithm parameters described in their sections. To control the execution of the generate step, it is possible to set the number of times the generate step is run via *generate_repetitions*. Using *max_generate_iterations*, we can limit how many iterations each generate step is allowed to take before it is terminated. Figure 4.10 shows the Chamfer Distance and run time of different PostSelection configurations. As can be seen here, the results vary greatly depending on the selection of generate- and evaluate-strategy. The results of these tests can be found in Table 6.4, and the algorithm configurations of the used generate and evaluate strategies are listed in Table 6.5. In general, PostSelection can achieve very good results, with the worst still being just over 118, but takes a very long time to run, the worst being around 6.5 hours. The algorithm configuration we chose consists of a slower IF-Race run as generate strategy and a short GEIST as evaluate step:

- $strategy\_generate = IF - Race$
  - $start\_size = 30$
  - $iteration\_size = 15$
  - $nr\_good\_results = 10$

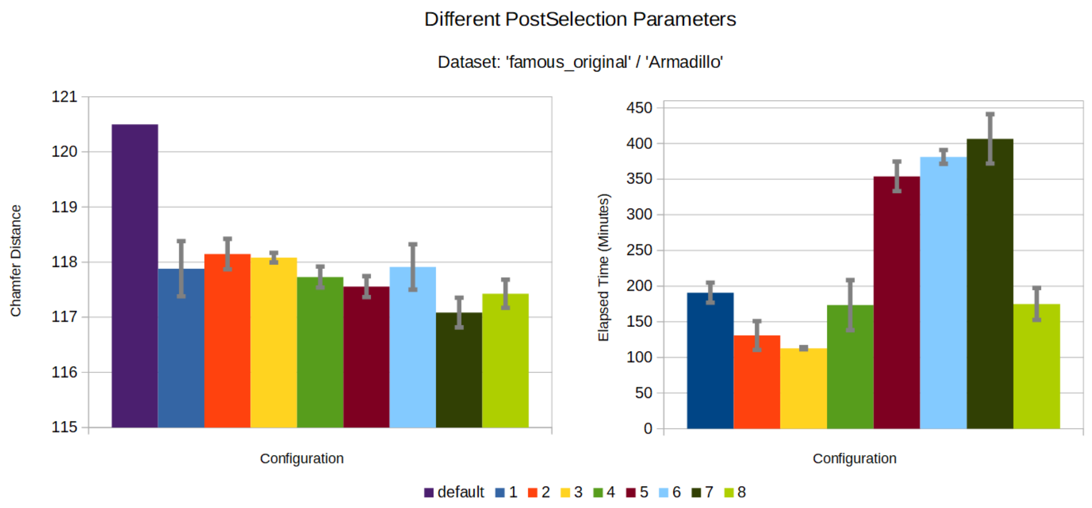


Figure 4.10: Visualization of the results of the manual tests to find a good algorithm configuration to run PostSelection with. The left graph shows the resulting Chamfer Distance in comparison to the default parameters, the right graph shows the needed time in minutes. The gray lines on the bars show the standard deviation. Each bar corresponds to one configuration in Table 6.4 as specified by the configuration number.

- $max\_iterations = 20$
- $annealing\_factor = 0.9$

- $strategy\_evaluate = GEIST$
  - $start\_size = 150$
  - $run\_size = 150$
  - $optimal\_threshold = 0.1$
  - $fixed\_threshold\_change = 0.02$
  - $threshold\_change = 0.04$
- $max\_generate\_iterations = 10$
- $generate\_repetitions = 5$

4.2.5 Brute-Force

Our Brute-Force algorithm simply selects random parameter configurations from the parameter space equal in number to the available processing cores. Selecting the same configuration multiple times is possible, but due to the size of the parameter space highly unlikely. There are two parameters relevant to our implementation of a Brute-Force

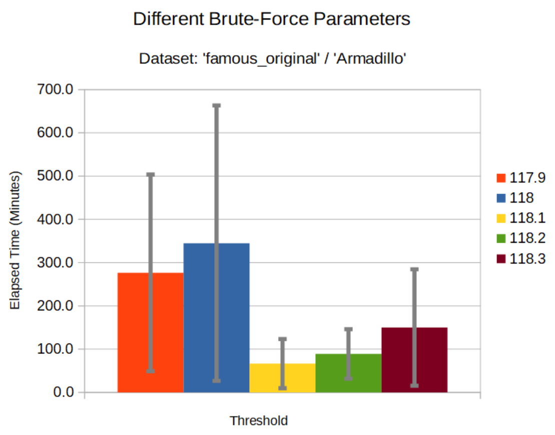


Figure 4.11: Visualization of the results of the manual tests to find a good threshold to run Brute-force with. The grey lines on the bars show the standard deviation.

algorithm. The first one is *threshold*, which is the value to reach as a termination criterion. The second one is *max_iterations*, which is a hard maximum after which the algorithm terminates. The effects of different *thresholds* can be seen in Figure 4.11. All results of these tests can be found in Table 6.6. As can be seen here, due to the random nature of the Brute-Force algorithm, it is not possible to predict how long the run time will be. One would expect to get a lower run time when choosing a higher *threshold*, but this assumption does not hold when looking at a statistically insignificant sample size, which is the intended use case of the developed framework.

4.3 Data Quality

After finding good algorithm configurations for each algorithm in Section 4.2, we tested these algorithms on different qualities of the same point cloud. We continued to use the ‘Armadillo’ cloud from the ‘famous’ dataset and selected all five available qualities for tests. For the ABC dataset, we used ‘cloud 1’ and tested it on the three available qualities, ‘original’, ‘noisefree’, and ‘extra_noisy’.
For this round of tests, we dropped the parameters *preClean* and *visibleLayer* since they are just used for Meshlab itself and don’t influence the algorithms. This means that for the following tests the parameter space encompasses only 4,519,800 nodes:

- $cgDepth \in [0, 1, 1]$
- $depth \in [4, 1, 8]$
- $fullDepth \in [5, 1, 8]$
- $iters \in [6, 1, 10]$

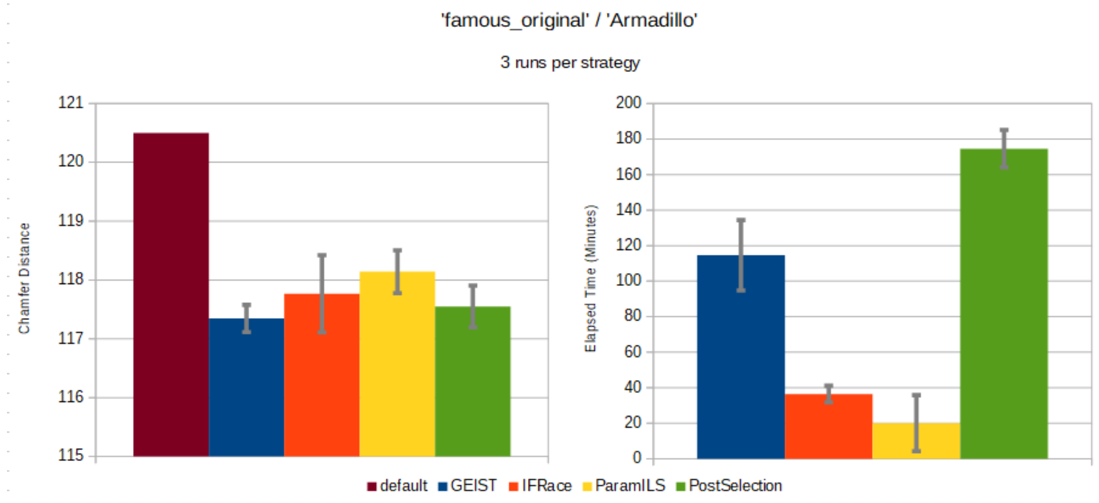


Figure 4.12: Comparing different algorithms on the ‘famous_original’ - ’Armadillo’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as grey lines.

- $pointWeight \in [2.0, 0.2, 8.0]$
- $samplesPerNode \in [1.0, 0.1, 9.0]$
- $scale \in [0.9, 0.1, 1.7]$

4.3.1 original

If we look at the results using the original_famous dataset in Table 6.7, we can see that all tested algorithms improve on the default parameter configuration significantly. The best results are reached using GEIST with a Chamfer Distance of only 117.3 on average. The worst result was achieved by ParamILS with 118.1, but this is still lower than the default result of 120.5. These values as well as the run time of each algorithm are also visualized in Figure 4.12. Here we can see that while GEIST and PostSelection achieve the best results, they also take the longest to run, taking nearly two or three hours respectively. In comparison, ParamILS was finished after just 20 minutes on average.

For the ABC dataset, the results can be found in Figure 4.28 and Table 6.18. As can be seen there, the run time looks very similar to the famous dataset, with PostSelection taking the longest and ParamILS the shortest. Again, GEIST leads to the lowest Chamfer Distance, but all results are very close together. Interestingly, here the improvement over the default solution is far higher than for the famous dataset. This suggests, that the default configuration is not very well suited for artificial geometric point clouds like ‘cloud 1’.

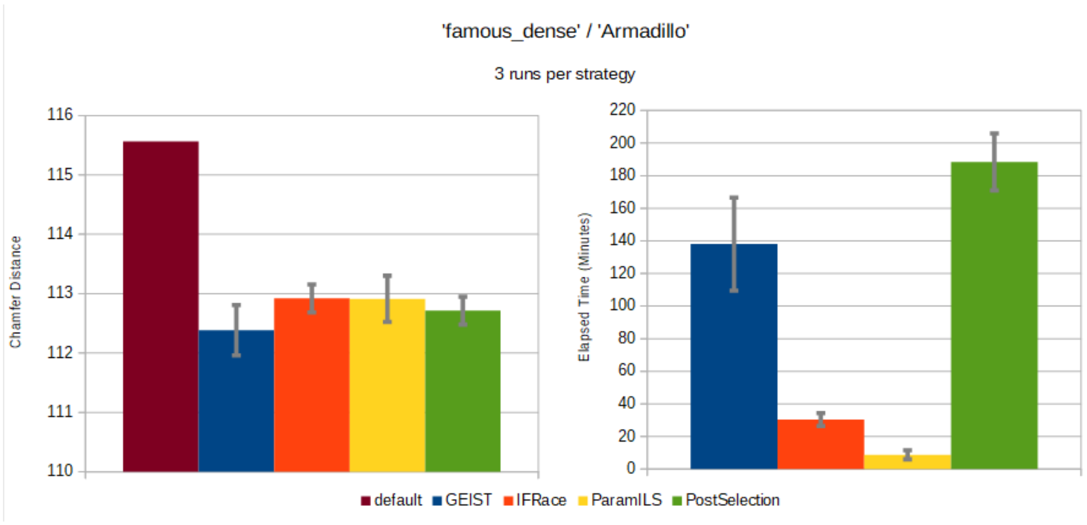


Figure 4.13: Comparing different algorithms on the ‘famous_dense’ - ‘Armadillo’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as grey lines.

4.3.2 dense

The results for the dense dataset look very similar to the original one. GEIST again reaches the best result with 112.4 against the default 115.6. The run time differences are even more pronounced than for the original dataset, with PostSelection taking over three hours and ParamILS not even nine minutes. All of these results can be found in Table 6.8 and Figure 4.13. The SPSR parameters calculated also look very similar to the original dataset. $iters$ (the number of Gauss-Seidel iterations to be performed at each level of the octree) tends to be lower for the dense dataset and $pointWeight$ (the importance of interpolating point values over gradients) seems to be lower, but more tests would be needed for a definitive answer here.

4.3.3 extra noisy

For the famous_extra_noisy dataset, the order of performance is the same again as for the two previous datasets, but the improvement over the default configuration is much higher. GEIST reaches an average Chamfer Distance of 166.6, and even ParamILS still results in only 169.5, compared to the default 239.7. The biggest difference in calculated SPSR configuration is that the $depth$ is only at five or six, while it usually reached eight in the famous_original dataset. When it comes to run time, the differences are substantial again, PostSelection takes over three hours on average, while ParamILS terminates after under nine minutes. All these results can be found in Table 6.9 and are visualized in Figure 4.14.

In the ABC_extra_noisy dataset, we get similar results to the famous dataset, a large

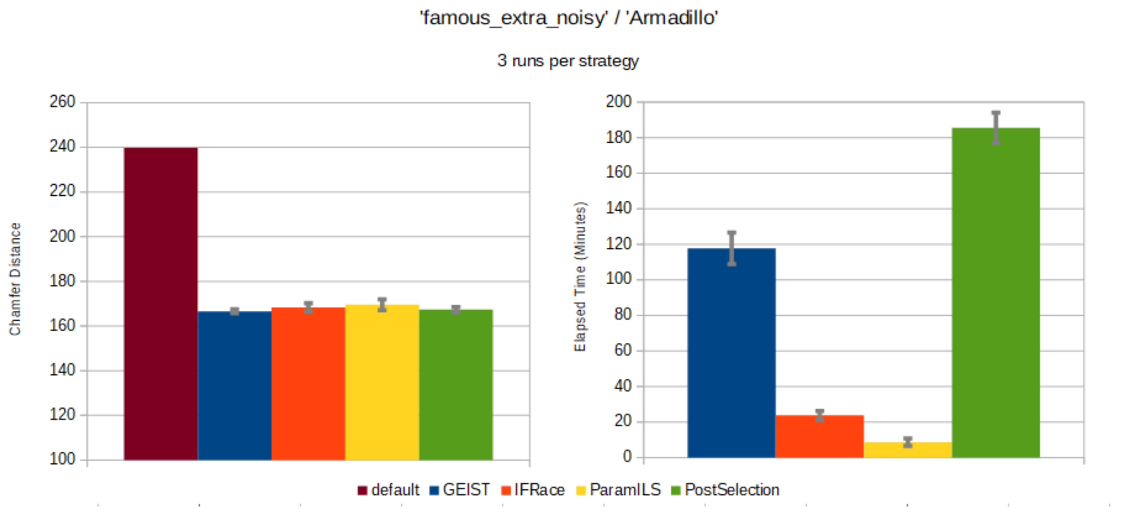


Figure 4.14: Comparing different algorithms on the 'famous_extra_noisy' - 'Armadillo' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as grey lines.

improvement over the default solution, but not a huge variety between the algorithms. A big difference can be found in the run time again. PostSelection takes on average nearly three hours, while ParamILS is finished after around ten minutes. All results can be found in Table 6.10 and are visualized in Figure 4.15.

4.3.4 noisefree

For the famous_noisefree dataset, the achieved improvement was the lowest. Compared to the default 112.2, GEIST only reached a Chamfer Distance of 110.5 and was even surpassed by IF-Race with 110.4. For this small of an improvement, GEIST still took 98.2 minutes on average. The slightly better IF-Race terminated after 38 minutes and the slowest PostSelection took 165.1 minutes. The full results can be found in Table 6.11 and in Figure 4.16. From these results, we can tell that for a noisefree point cloud, a high *depth* is beneficial for SPSR, and values over eight might have yielded even better results. In addition to that, the *pointWeight* (the importance of interpolating point values over gradients) calculated by all optimization algorithms is higher than for the famous_original dataset, circling around seven instead of three.

The ABC_noisefree results look similar to the famous_noisefree ones. Again, IF-Race reaches better results than GEIST. Only here, PostSelection results in an even lower Chamfer Distance of 194.2 compared to the default 198.5. As for the famous_noisefree dataset, *depth* is calculated at eight across the board, and *pointWeight* is higher than before. The run time looks like with most tests, with ParamILS being the fastest and PostSelection taking over three hours.

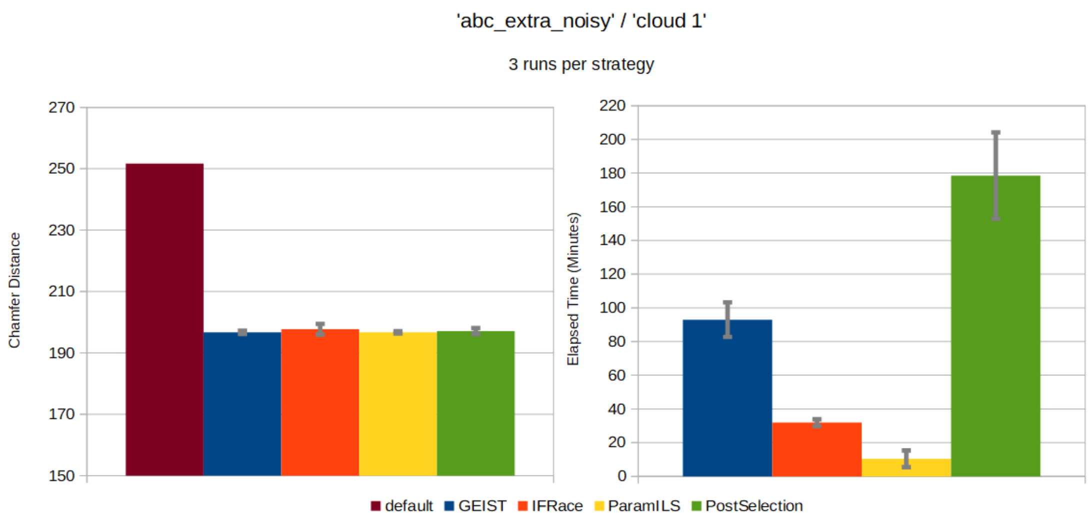


Figure 4.15: Comparing different algorithms on the 'ABC_extra_noisy' - 'cloud1' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as grey lines.

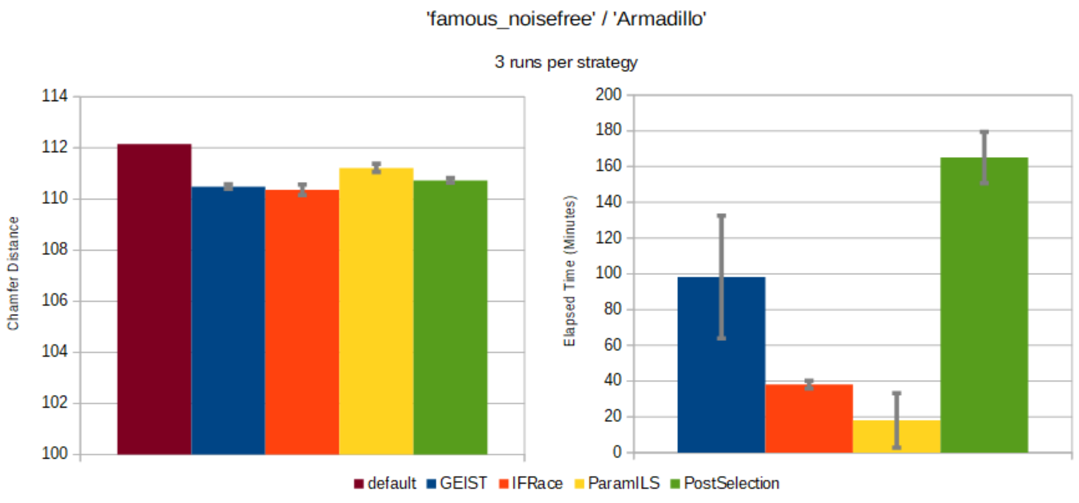


Figure 4.16: Comparing different algorithms on the 'famous_noisefree' - 'Armadillo' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

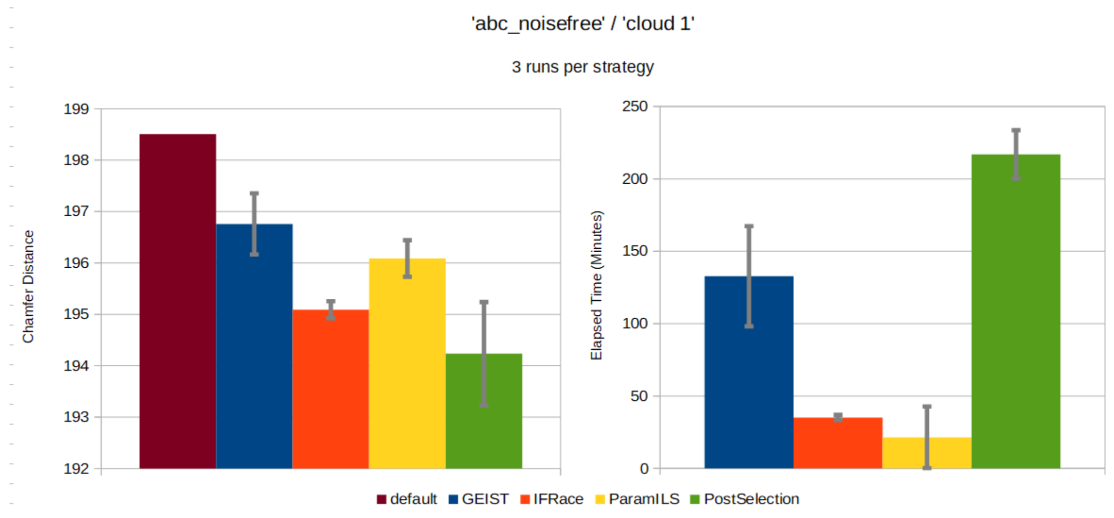


Figure 4.17: Comparing different algorithms on the ‘ABC_noisefree’ - ‘cloud1’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

4.3.5 sparse

For the famous_sparse dataset, all algorithms reach very similar results. Their averages lie between 141 and 142, with ParamILS achieving the lowest and IF-Race the highest. Compared to that, the default parameter configuration results in a Chamfer Distance of 149.1. For these results, ParamILS only takes 33 minutes, while the slowest, PostSelection, takes 169 minutes on average. The results of all these tests can be found in Table 6.13 and Figure 4.18. Similar to the famous_noisefree dataset, all tests resulted in a *depth* of eight, so an even higher *depth* might yield even better results. The *pointWeight* is also significantly higher than for the famous_original dataset, but not as high as for the famous_noisefree dataset.

4.4 Mesh Differences

The third round of tests looks at various point clouds from the same dataset and aims to find differences in which algorithm is better suited to which point cloud, as well as if other parameters are ideal for SPSR applied to these point clouds.

4.4.1 Dragon

The first point cloud we looked at is the Dragon point cloud (‘xyzrgb_dragon_clean’) from the ‘famous_original’ dataset. As can be seen in Table 6.14 as well as Figure 4.19, all algorithms achieve a reduction of the Chamfer distance compared to the default configuration by about three. While the resulting Chamfer Distances are very close, the run times of the algorithms differ greatly. While IF-Race only takes an average of

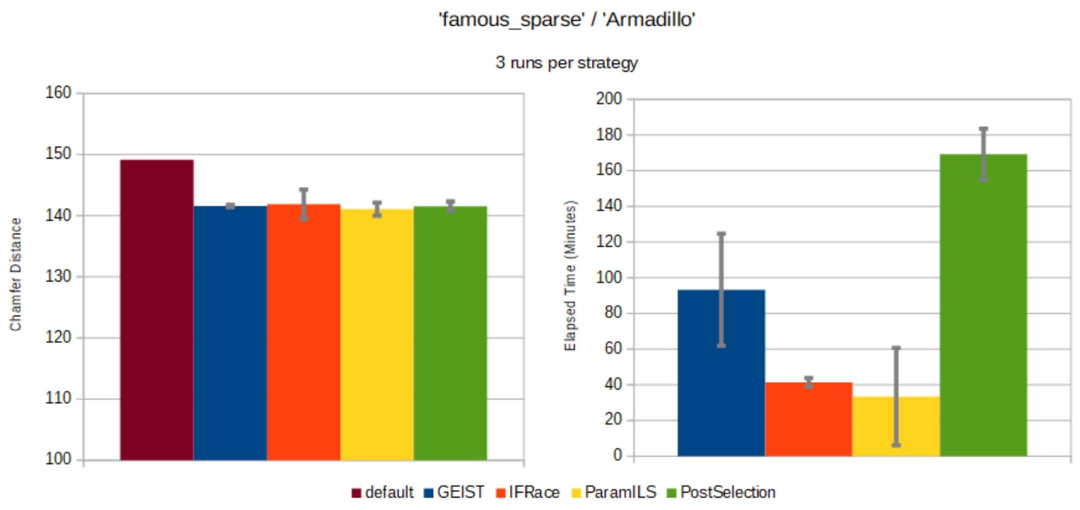


Figure 4.18: Comparing different algorithms on the ‘famous_sparse’ - ‘Armadillo’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

50 minutes, GEIST took nearly four hours to complete. It is noteworthy that for this test run, GEIST took longer to complete than PostSelection, while it was the other way around for most others. Looking at the low difference in achieved quality, it seems it would not make sense to run this optimization for each new point cloud. The resulting mesh does, however, show visual improvements over the default parameters. The model is far smoother, especially on the tail and on the side of the body. A comparison can be seen in Figure 4.20.

4.4.2 Yoda

The second point cloud we used for this set of tests is the ‘yoda’ point cloud, also from the ‘famous_original’ dataset. This is a far more difficult mesh to reconstruct since it is topographically very complex and has a lot of holes. The original mesh, as well as the solution using default SPSR parameters and our best calculated solution, can be seen in Figure 4.23. As can be seen here, many of the holes cannot be reconstructed successfully no matter which parameters are used, the result is however much better with our calculated configuration. On this mesh, the right one in the Figure, far more holes were detected correctly and the surface itself is smoother than on the default solution. This can also be seen when looking at the Chamfer Distances of both solutions. The key to getting better results for this mesh is setting the *pointWeight* higher than usual, forcing the reconstruction to stick closer to the actual points. This is supported by Figure 4.21, which shows the Correlation Visualization of the best IF-Race run focused on *pointWeight* and *depth*. As can be seen here, while a high depth is necessary for good results, only those parameter configurations with a high *pointWeight* reach the

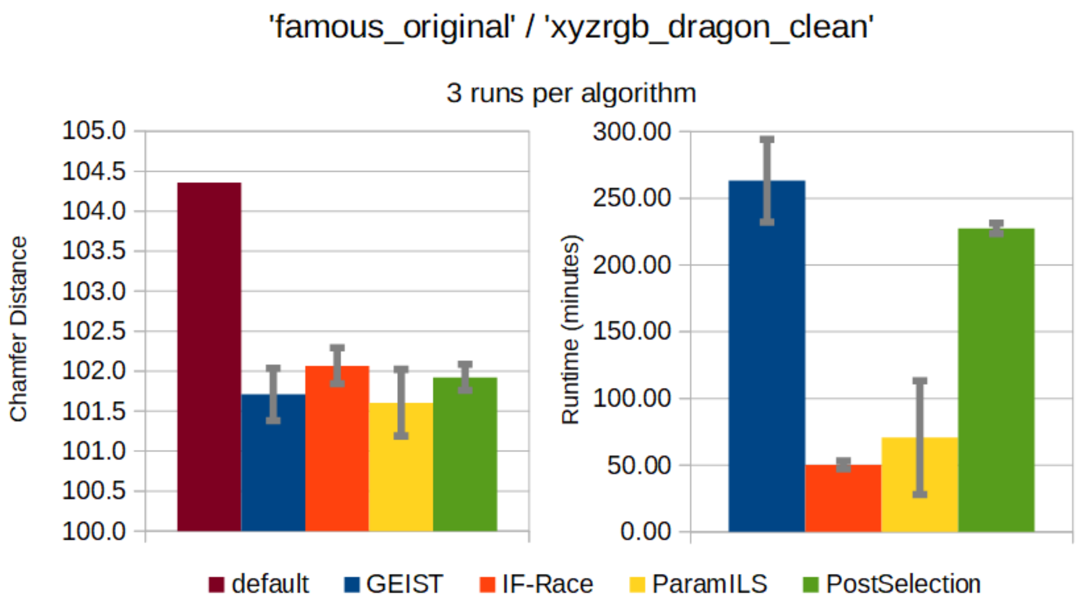


Figure 4.19: Comparing different algorithms on the 'famous_original' - 'xyzrgb_dragon_clean' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

lowest Chamfer Distances.

While the default parameters yield a value of 333, our best found solution only results in a Chamfer Distance of 298, which is an improvement of over 10%. These and all other results can be found in Table 6.15 and in Figure 4.22. Another thing visible from this graph is that all tested algorithms achieve a very similar result, with the worst one, ParamILS, being only at 303. The run time graph looks similar to most instances, with GEIST and PostSelection taking far longer (100 and 160 minutes) than IF-Race and ParamILS (35 and 27 minutes).

4.4.3 Bunny

The 'bunny' point cloud from the 'famous_original' dataset is a very well-known mesh, which does not offer anything out of the ordinary the SPSR parameters could be tuned to. As such, the achieved Chamfer Distances are not as much an improvement over the default as the previous point clouds. As can be seen in Figure 4.24, all algorithms are very close together, with IF-Race resulting in the lowest Chamfer Distance of 141 compared to the worst average value of 142 of ParamILS and 143 of the default. This small difference also results in only a very small difference in the resulting mesh. Figure 4.25 shows the original mesh, the default result, and our best calculated result. As can

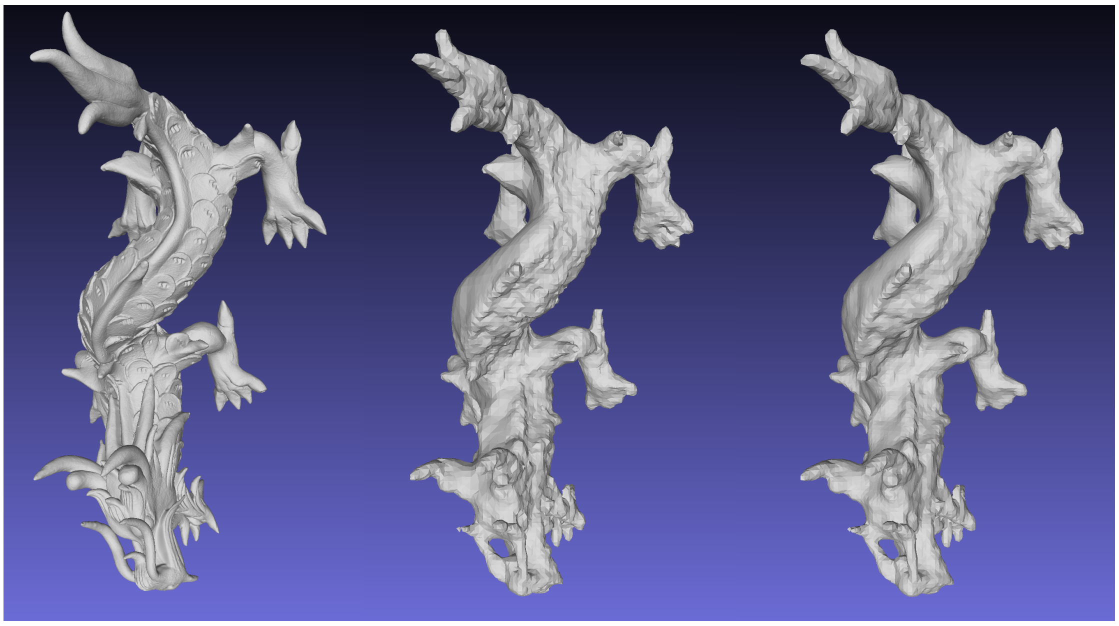


Figure 4.20: Comparison of reconstructions of the 'famous_original' / 'xyzrgb_dragon_clean' point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

be seen here our solution achieves slightly sharper features, for example on the ears, but results in a rougher surface on the body. The accurate numbers of all tests on the 'bunny' point cloud can be found in Table 6.16.

4.4.4 Flower

As a last point cloud from the 'famous_original' dataset, the 'flower' is a more geometric and artificial mesh. The original, as well as the result of the default SPSR parameters and our best result, can be seen in Figure 4.27. As is already visible here, our result looks much smoother than the default one. This is also supported by the numerical results in Table 6.17 and in the graph in Figure 4.24. Here one can see, that the best result was achieved using GEIST, resulting in a Chamfer Distance of 119. The worst strategy we got was with ParamILS at 120, compared to the default of 122. These improvements come at the cost of on average 124 minutes using GEIST or just 17 minutes using ParamILS.

4.4.5 Cloud 1

The first cloud from the ABC dataset, 'cloud 1', shows a big improvement over the default solution. The resulting meshes can be seen in Figure 4.29. It is visible that all surfaces are far smoother and the edges more pronounced, leading to a clearer shape. This is also supported by the values derived from the tests found in Table 6.18 and visualized in Figure 4.28. All algorithms reached a big improvement over the default configuration

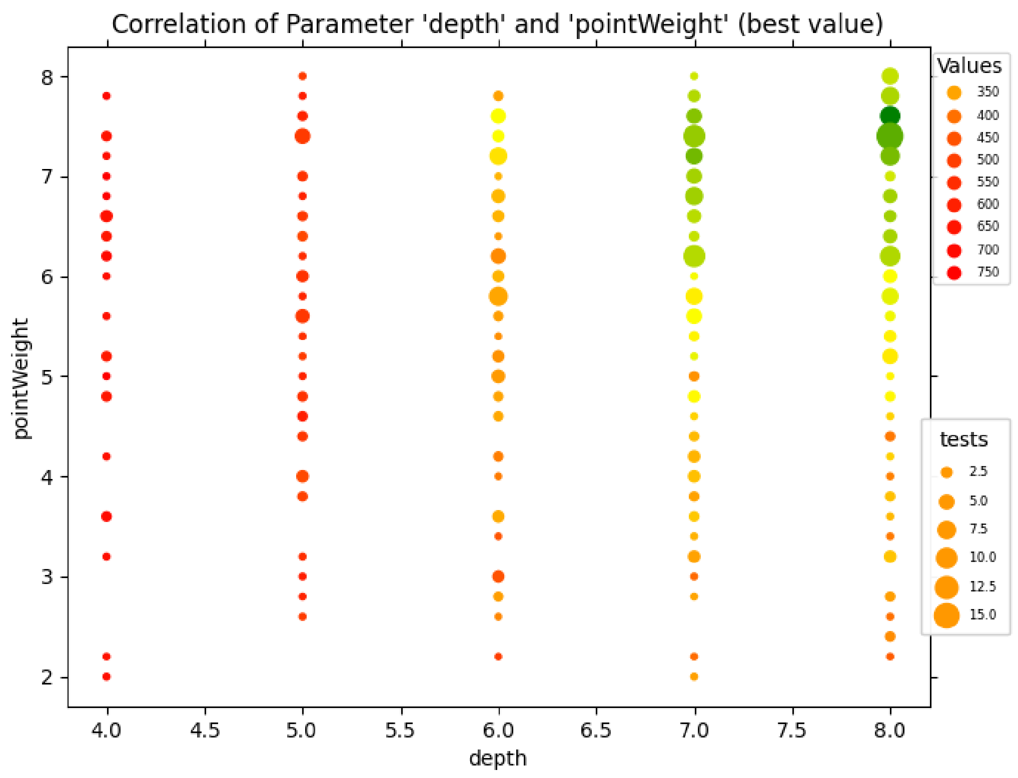


Figure 4.21: Correlation between *depth* and *pointWeight* of a IF-Race optimizatiion run on SPSR for the 'famous_original' - 'yoda' point cloud.

with a Chamfer Distance of 229.5. The lowest value was achieved by GEIST with an average of only 182.4, but even the worst, ParamILS, with 183.5 was still far below the average. Similar to most other tests, PostSelection was the slowest with a run time of nearly three hours, and ParamILS the fastest with an average of under ten minutes. Interestingly, the resulting configurations all have lower depths than what would be expected from the famous dataset, only being six or seven instead of the usual eight. This might be due to the ABC dataset consisting of more geometrical shapes with little to no detail in the surfaces themselves.

4.4.6 Cloud 2

'Cloud 2' from the ABC dataset is four girder-like objects next to each other and poses a challenge to SPSR due to its thin walls. The ground-truth as well as the default and our best reconstruction can be found in Figure 4.31. As can be seen here, SPSR does not manage to reconstruct each wall correctly, failing for the biggest girder in the middle with its small perpendicular walls. Our best solution is more accurate than the

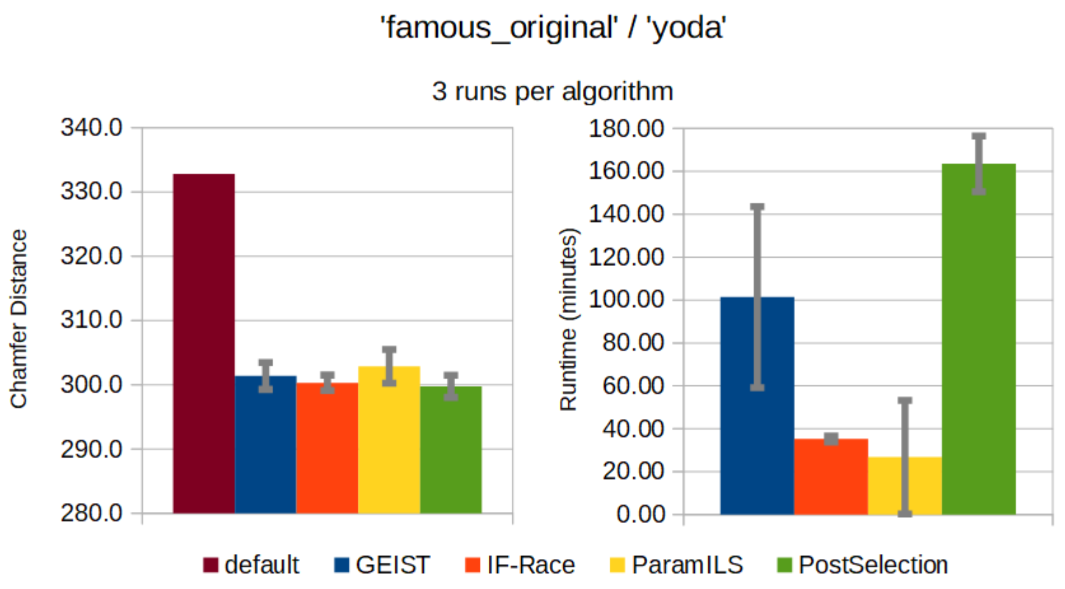


Figure 4.22: Comparing different algorithms on the 'famous_original' - 'yoda' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

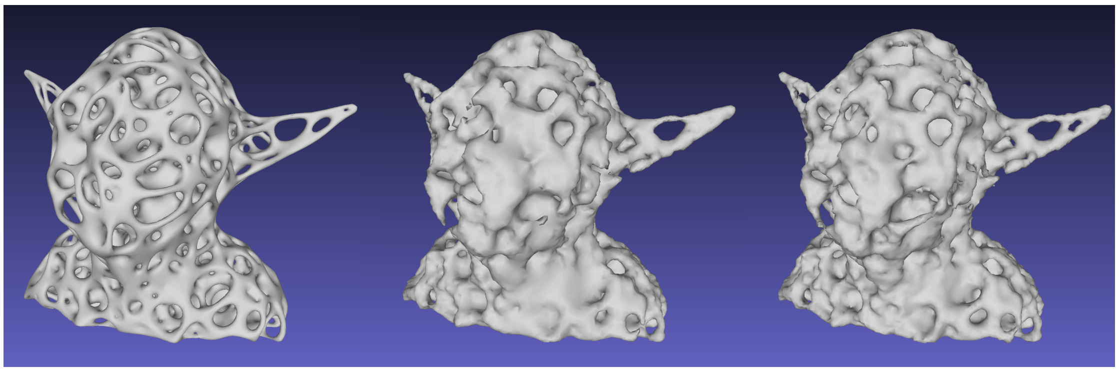


Figure 4.23: Comparison of reconstructions of the 'famous_original' / 'yoda' point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

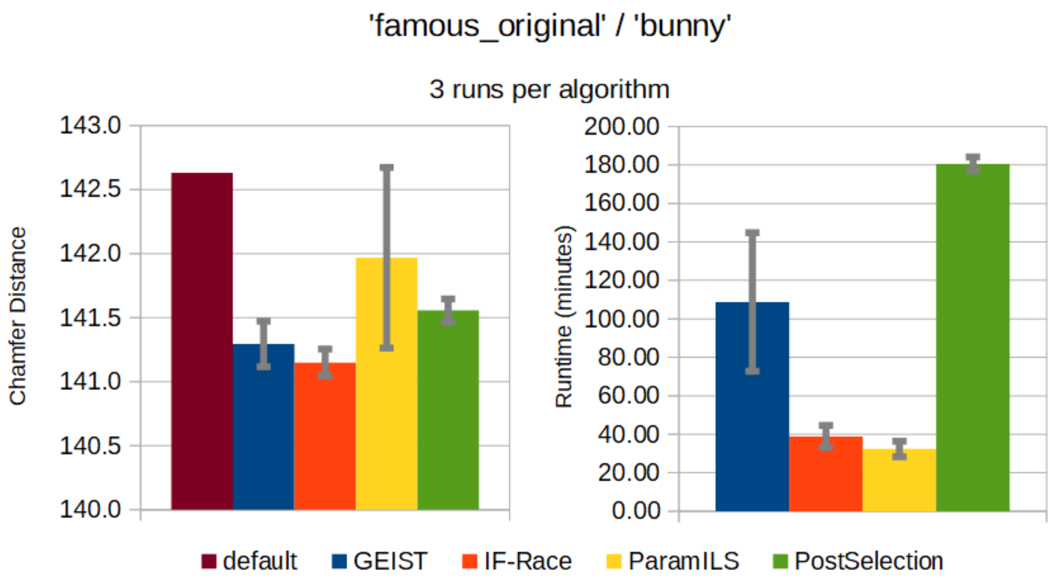


Figure 4.24: Comparing different algorithms on the 'famous_original' - 'bunny' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

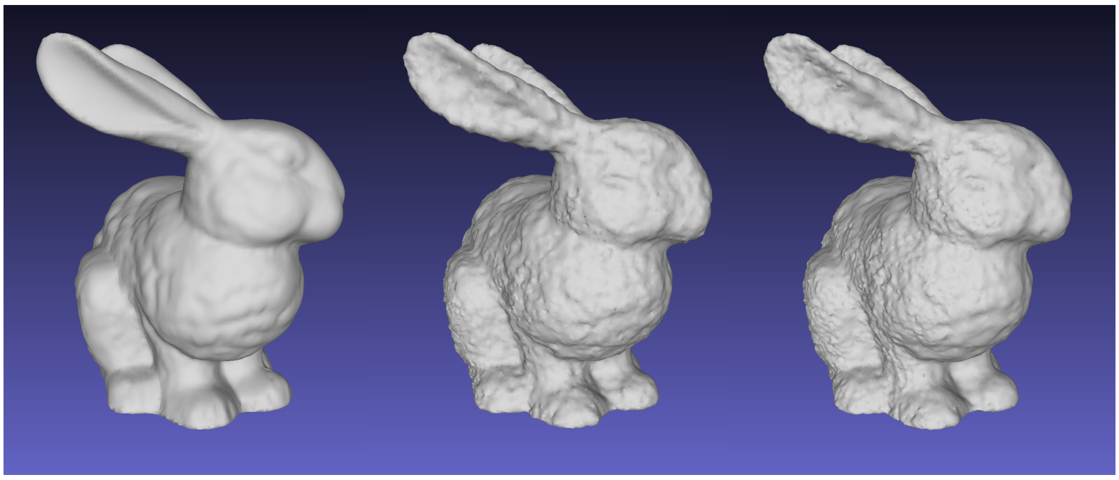


Figure 4.25: Comparison of reconstructions of the 'famous_original' / 'bunny' point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

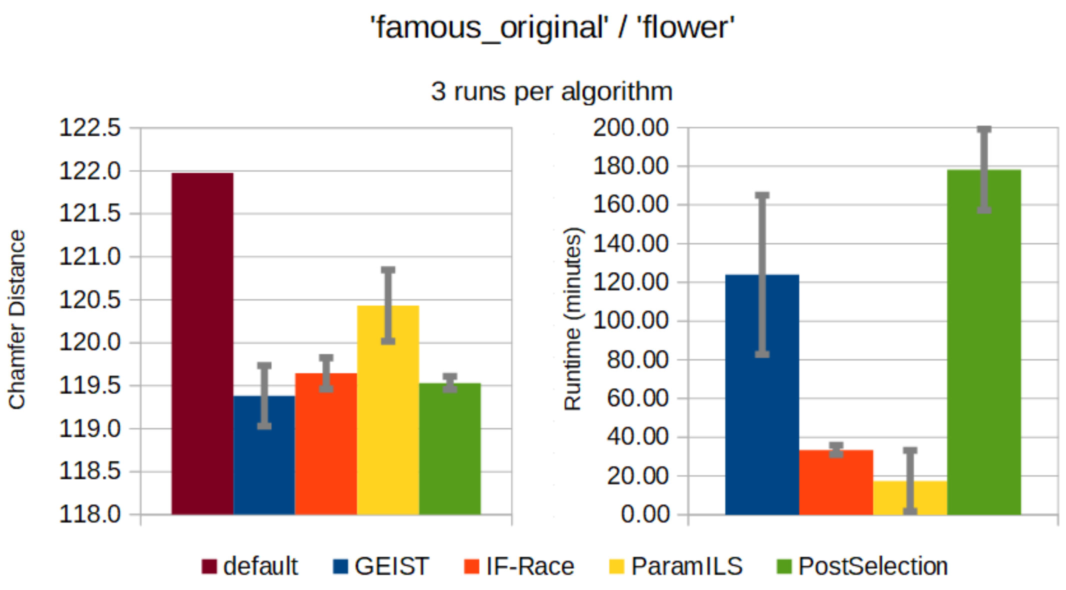


Figure 4.26: Comparing different algorithms on the 'famous_original' - 'flower' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

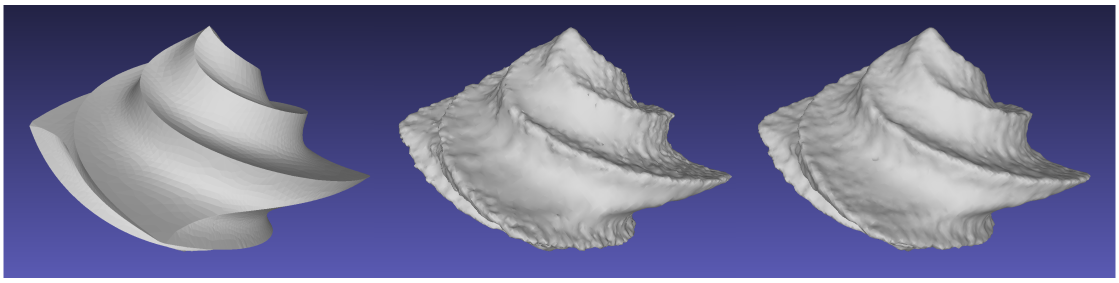


Figure 4.27: Comparison of reconstructions of the 'famous_original' / 'flower' point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

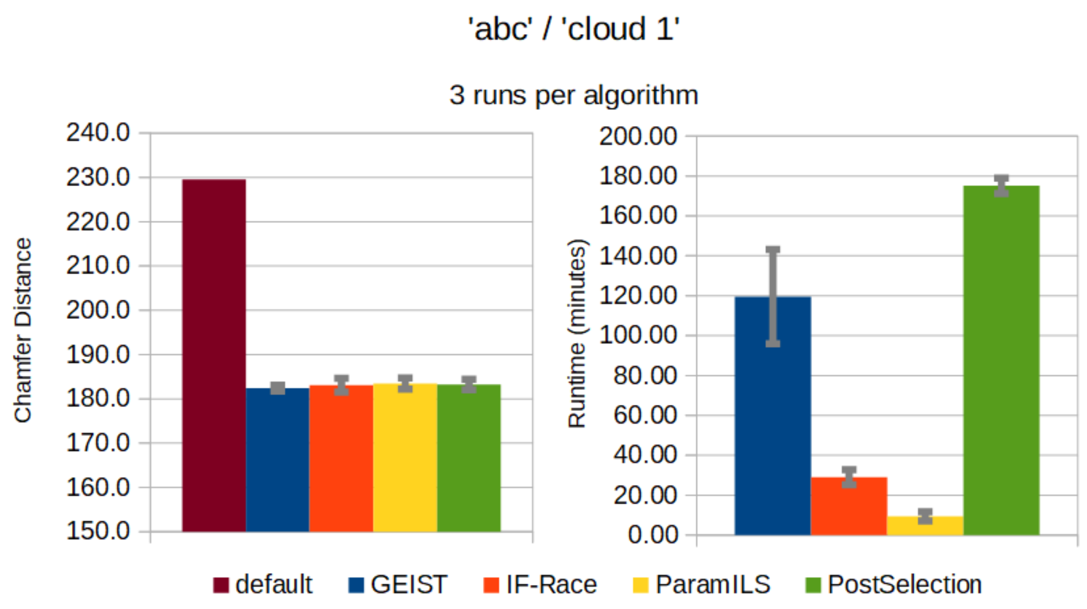


Figure 4.28: Comparing different algorithms on the 'famous_original' - 'cloud 1' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

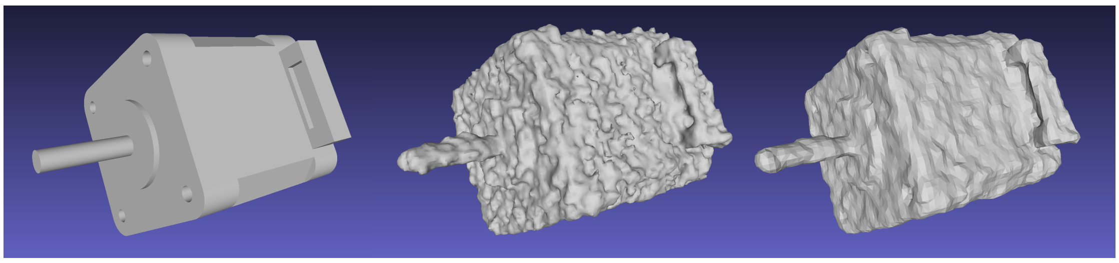


Figure 4.29: Comparison of reconstructions of the 'ABC_original' / 'cloud1' point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

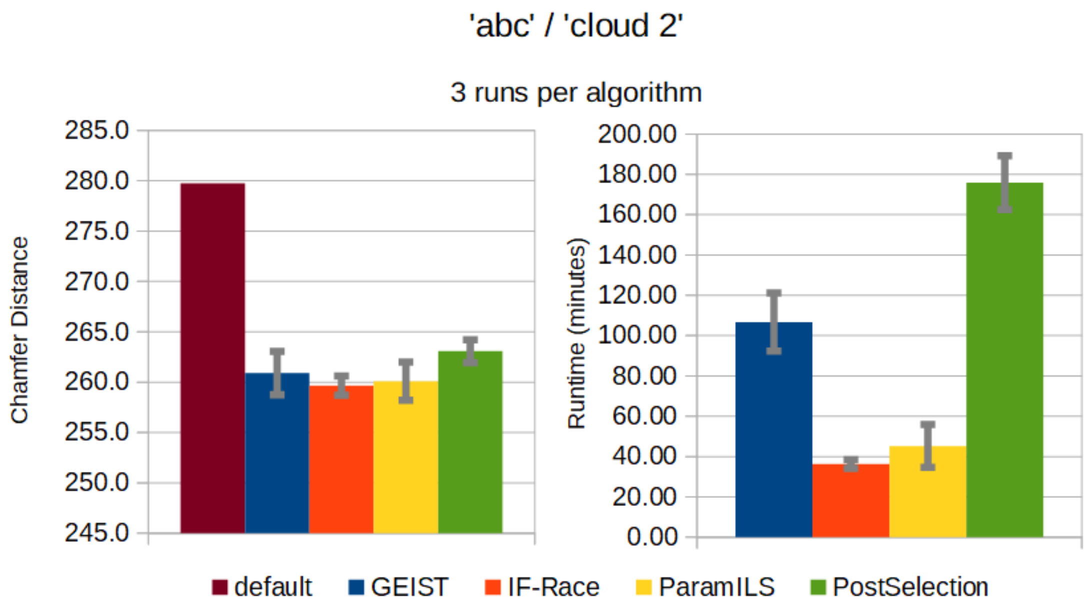


Figure 4.30: Comparing different algorithms on the 'famous_original' - 'cloud 2' dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

default, but it still does not manage to get all of these correct. The numbers support this observation, with all algorithms improving on the default. IF-Race reaches the lowest Chamfer Distance for this case with an average of 259.7 compared to the default of 279.7. IF-Race is also faster than ParamILS for cloud 2, taking only 36.3 minutes on average. For the fine details on this point cloud, the resulting *depth* is higher again, as well as the *pointWeight*, which is around seven in most configurations. All of these results can be found in Table 6.19 and in Figure 4.30.

4.4.7 Cloud 3

The third point cloud we selected from the ABC dataset is a large pipe with a small hole in it. SPSR had some trouble finding the correct interior wall of the pipe, both in the default and in our best solution. The resulting meshes can be found in Figure 4.33. Our solution has a rougher surface, but the erroneous region inside the pipe is smaller than the default. The resulting configurations look similar to the ones of cloud 2, with high *depth* and *pointWeight*, and there is an improvement over the default, but it is not as significant as for some of the other tests. As can be found in Table 6.20 and in Figure 4.32, the Chamfer Distance could only be improved to 561.6 from 600.7, again by IF-Race. The fastest algorithm was ParamILS with 28.7 minutes, and the slowest again PostSelection with over three hours.

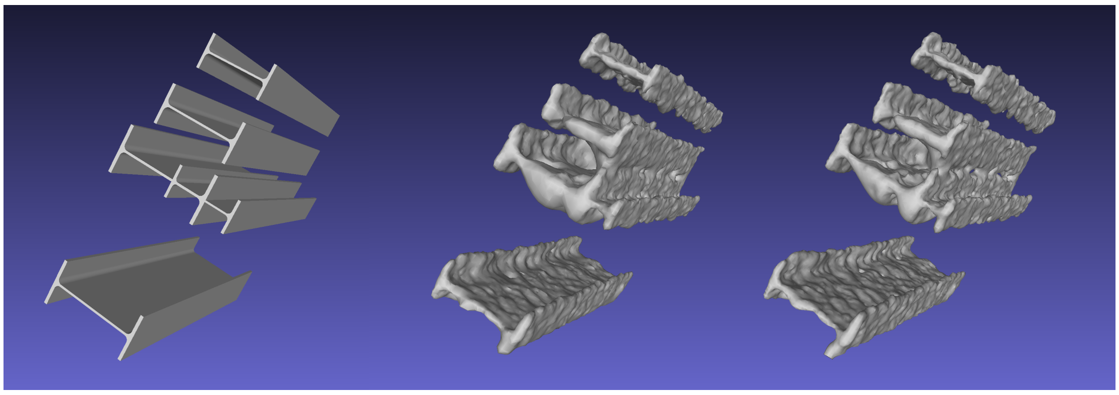


Figure 4.31: Comparison of reconstructions of the ‘ABC_original’ / ‘cloud2’ point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

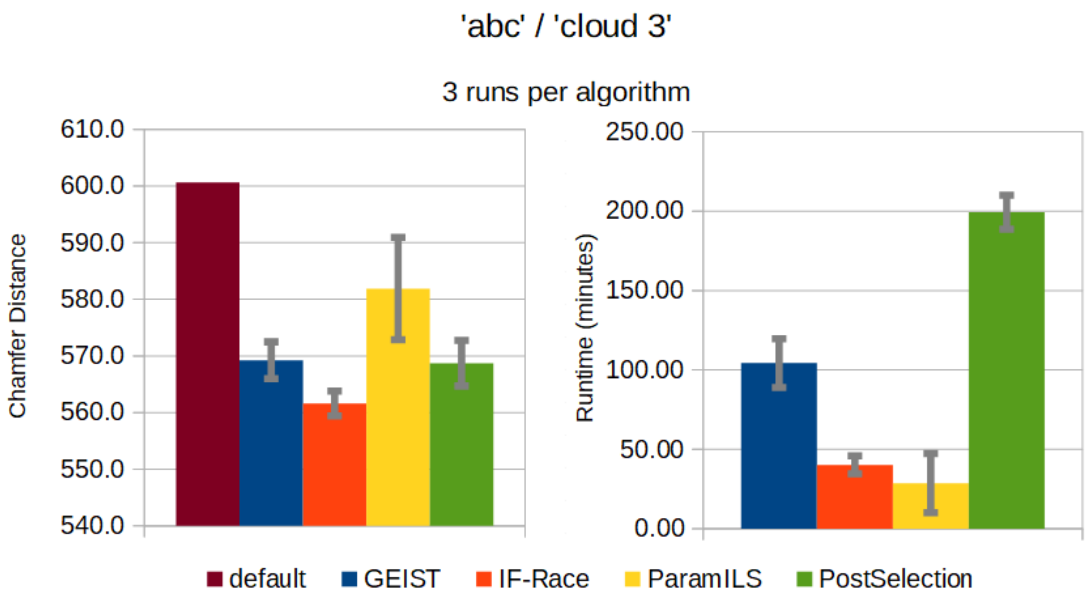


Figure 4.32: Comparing different algorithms on the ‘famous_original’ - ‘cloud 3’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

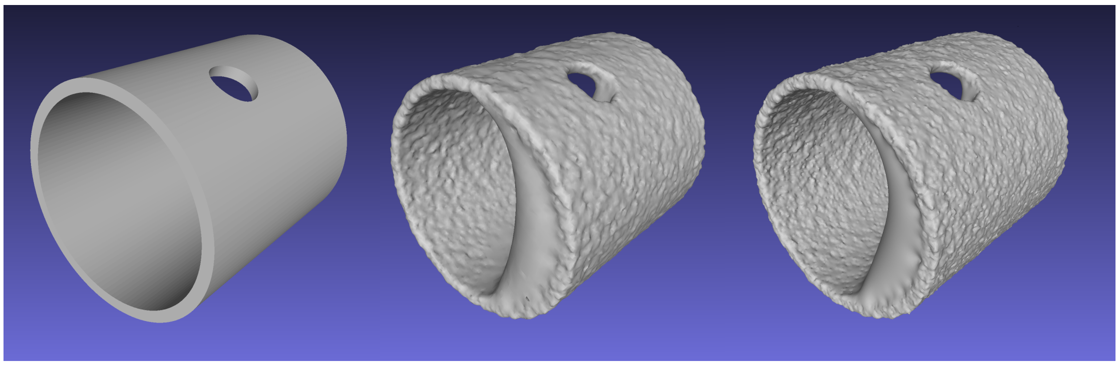


Figure 4.33: Comparison of reconstructions of the ‘ABC_original’ / ‘cloud3’ point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

4.4.8 Cloud 4

The last point cloud from the ABC dataset seems to be close to the ideal point cloud for the default parameter configuration. In the comparison, visible in Figure 4.35, differences are visible between the default reconstruction and our best version, but it is not clear which one is the better mesh. There are some differences around the holes in the corner, but the surface looks to be of the same roughness. A similar low improvement over the default can be found in the data presented in Table 6.21 and the visualization in Figure 4.34. While all algorithms achieved lower Chamfer Distances than the default, it is only by about seven. The lowest values were achieved by PostSelection with a value of 227.1 compared to the highs of 227.6 by GEIST and 235 by the default parameter configuration. The parameter values are similar to the previous two clouds, with high $depth$ and $pointWeight$. In terms of run time, ParamILS performs the best with an average of 32 minutes. PostSelection was the slowest with three and a half hours again.

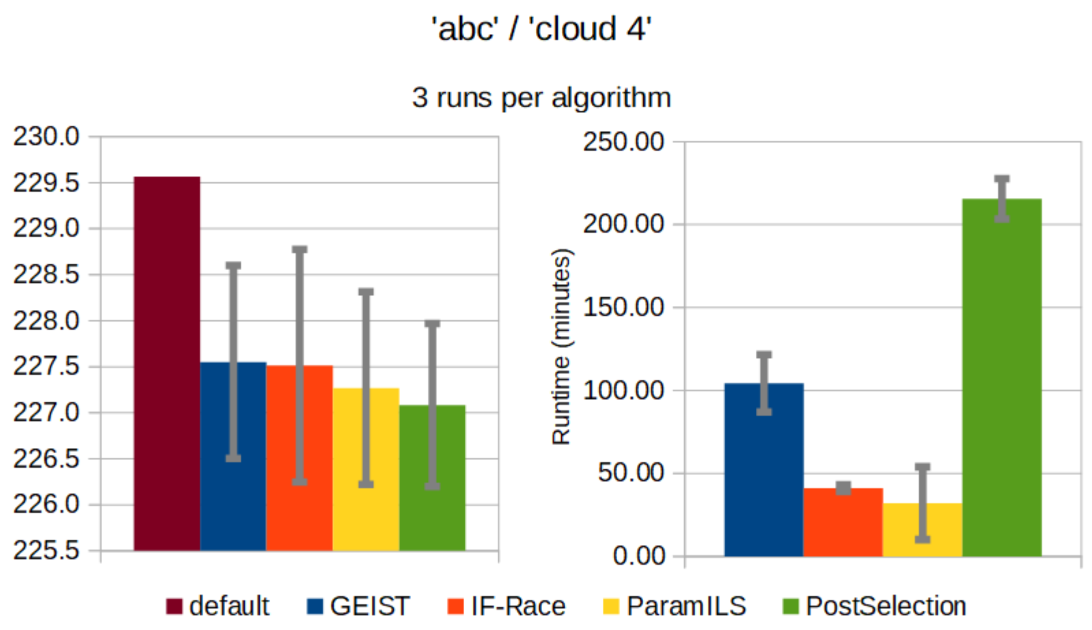


Figure 4.34: Comparing different algorithms on the ‘famous_original’ - ‘cloud 4’ dataset. The left graph shows the average achieved Chamfer Distance and the right graph the required average run time. Standard deviations for both values are shown as gray lines.

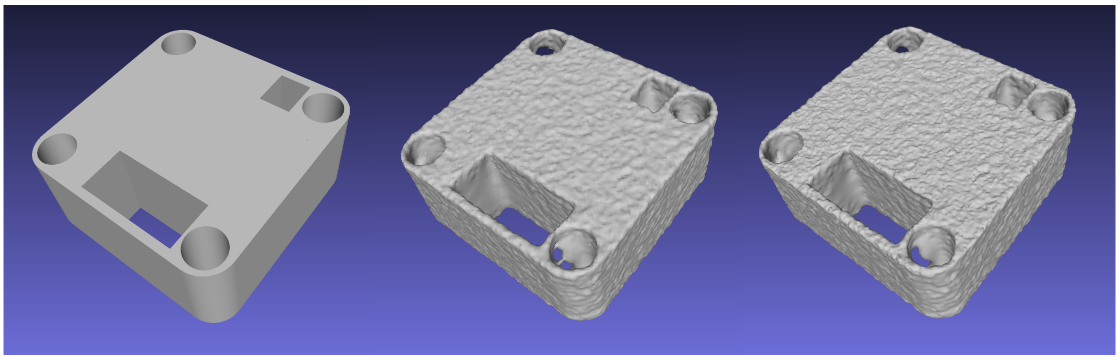


Figure 4.35: Comparison of reconstructions of the ‘ABC_original’ / ‘cloud4’ point cloud. Left: ground-truth, Middle: default parameters, Right: best optimization result

CHAPTER 5

Discussion and Future Work

In this chapter, we will discuss what we have achieved in this work, how well our approaches worked, as well as any future work that could be done to extend it.

5.1 Implementation

We have implemented a framework for optimizing different targets by the usage of various strategies. Both targets and strategies are easy to implement by inheriting from a super-class. An optimization or a single run of a target can be started easily by just defining a target and a strategy or a target and a single algorithm configuration. Our implementation also supports visualizations that can be updated after each iteration as well as after the whole run is finished. Currently, three are implemented, as described in Section 3.3, but more are implementable as easily as new targets or strategies.
The framework works very well, provides verbose output of its current state, and can be used as is, but there are a few improvements that could be made in the future. First of all, there are currently only two targets, SPSR and a test equation for easy debugging of strategies. This could be extended to include a general command line target, which calls a command line command with defined parameters. This would make our framework much more universally usable. A larger task would be to add a GUI to make usage of the system more user-friendly and get away from defining each test run in code.

5.2 Best SPSR configuration

We have consistently managed to improve on the standard SPSR parameter configuration, so there is definitely an improvement to be achieved. The difficult question is which configuration is the best overall. The answer is not straightforward since each point cloud has its own optimum depending on geometric features, noise, and density. For a relatively smooth and topologically simple mesh as the ‘famous_original’ / ‘bunny’, the default parameter configuration is already very good, but can still be improved. In this case, a near-optimal solution is:

- $cgDepth = 1$
- $depth = 8$
- $fullDepth = 7$
- $iters = 9$
- $pointWeight = 3.2$
- $samplesPerNode = 1.3$
- $scale = 1.2$

On the other extreme, the ‘yoda’ mesh or some of the geometric ABC clouds are very difficult to reconstruct in general for SPSR since they consist of many holes and fine details that cannot always be reconstructed correctly. In this case, the $pointWeight$ (α in Equation 3.3) needs to be much higher, putting more emphasis on the position of the points than on the surface gradient. A good parameter configuration for this case might look like this:

- $cgDepth = 0$
- $depth = 8$
- $fullDepth = 6$
- $iters = 8$
- $pointWeight = 8.0$
- $samplesPerNode = 1.0$
- $scale = 0.9$

Another problem that might occur is the quality of the point cloud. If there is a lot of noise, the position of the individual point is less important and a too deep octree can be detrimental to the reconstruction result. Another use case for this kind of configuration is simple geometric objects without holes that feature mainly flat surfaces. For the ‘famous_extra_noisy’ / ‘Armadillo’ point cloud, the following configuration has proven to be the best:

- $cgDepth = 1$

- $depth = 6$
- $fullDepth = 5$
- $iters = 9$
- $pointWeight = 2.2$
- $samplesPerNode = 1.0$
- $scale = 1.5$

To conclude, it is always important to know the properties of the point cloud that needs to be reconstructed. For most objects, the first mentioned parameter configuration will work well. If the data is very accurate, *depth* and $pointWeight$ can be increased. If there is a lot of noise or the points are less accurate, those values should be reduced. Should the point cloud describe a topologically difficult object, that cannot be reconstructed with this parameter configuration, it can be beneficial to set the $pointWeight$ very high to get a higher-quality reconstruction. This might, however, lead to overfitting if there is too much noise or outliers.

5.3 Best Algorithm to optimize SPSR

Choosing the best algorithm to optimize SPSR with is not easy, since there are different criteria to look at. We will therefore discuss each of them separately. An overview of all qualities and algorithms on the example of the 'Armadillo' point cloud can be found in Figure 5.1.
The first one is GEIST by Thiagarajan et al. [TJA$^+$18]. It is a very robust algorithm that achieved very low, if not the lowest, Chamfer distances throughout our tests. GEIST is highly configurable, making it easy to adjust to any specific needs. It is possible to have it terminate faster with a less accurate result or take longer and be more thorough. With different thresholds and iteration sizes, it can take more global data into consideration or focus more on the currently best area of the parameter space. There is, however, a big downside to GEIST, which is its run time. This algorithm took multiple hours to terminate across all tests on our system. While the selected algorithm configuration was aimed at optimizing the result and not the run time, this is still far above the two faster algorithms IF-Race and ParamILS. Only PostSelection, which runs multiple smaller algorithms, took even longer in most cases.
The second algorithm that is outstanding in a category is ParamILS by Hutter et al. [HHLBS09]. While this algorithm did not yield as much of an improvement over the default parameter configuration, it is still close to GEIST in most cases. The biggest advantage of ParamILS is its speed, taking only minutes instead of hours. This is achieved while still being tuned to yield good results and not to terminate quickly. Next to the slightly worse reconstruction quality, ParamILS also has the disadvantage of only taking local data into consideration. This means that even though some steps of the algorithm

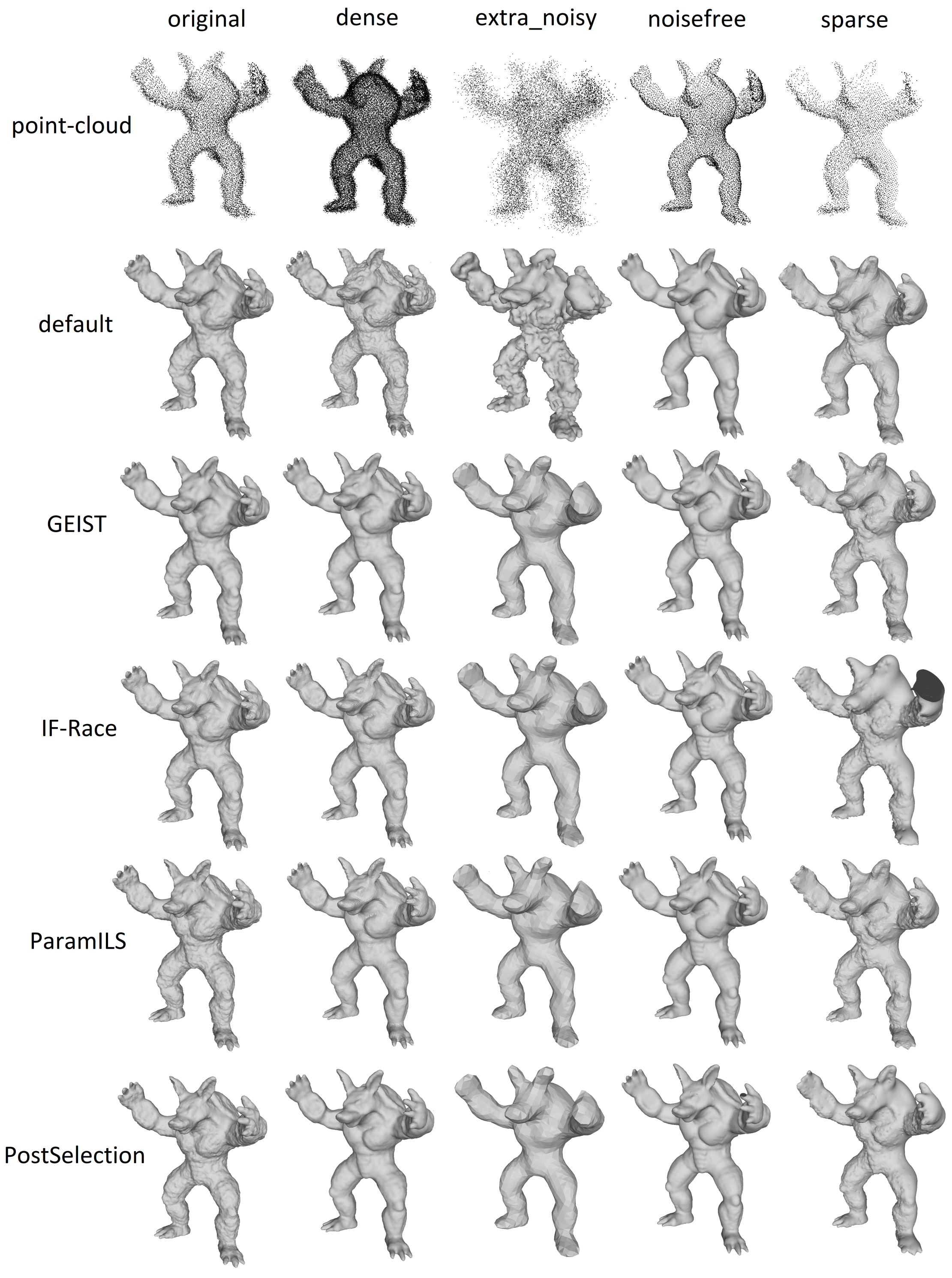


Figure 5.1: Reconstruction of the 'Armadillo' point cloud from different qualities using different algorithms.

are aimed to avoid this, it is far more likely to get stuck in a local optimum. This does not seem to be as much of a problem with SPSR due to how the target function behaves. Thirdly, IF-Race by Balaprakash et al. [BBS07] is a good compromise between the two previously mentioned algorithms. It reaches results comparable to GEIST, sometimes even beating it and still terminates in under one hour in all cases. Similar to GEIST, IF-Race takes global data into consideration. Our implementation is highly configurable, therefore giving the ability to tweak it depending on time constraints or quality requirements.
The last algorithm to mention is PostSelection by Yuan et al. [YSMdO$^+$13]. Its biggest advantage in our algorithm configuration is its consistency, having the lowest standard deviation in most cases. While not yielding exceptional results, the run time for PostSelection is, except for one test, significantly higher than for any other algorithm.
In general, if time is not a concern, GEIST will be the best solution to optimize SPSR. Otherwise, a good solution can be gotten far quicker by using ParamILS or IF-Race. Since IF-Race usually leads to better results, it should be used as long as time is not the primary concern. Another possibility in that case would be to use Brute-Force, which will be discussed in Section 5.4

5.4 SPSR Target Function

In this section, we gather some remarks on the properties of the target function for which we are trying to optimize the parameter input. This is the function that takes a parameter configuration as input and outputs the reconstruction's Chamfer Distance. The target function behaves very differently for each parameter. Not only are parameters of different importance, they also have a different kind of impact on the resulting values. Some parameters change the Chamfer Distance linearly like $pointWeight$, while others do so exponentially. The main example for this is *depth*, which can lead to extremely bad results when configured wrongly, but does not improve the resulting Chamfer Distance that much anymore closer to the optimum. An example of this behavior can be found in Figure 5.2, which shows the resulting Chamfer Distance average over all tests in a GEIST run on the 'famous_original' - 'Armadillo' dataset for each value *depth* took in that run. This leads to the fact that there are some parameter configurations that yield extremely bad results, but even a very short run of any algorithm can find some decent ones quickly. Closer to the optimum, any change in the configuration only affects the result minimally and it takes very long to find a better solution, so there is a very diminished return to any additional time spent optimizing. This also leads to the interesting possibility to use Brute-Force for optimizing SPSR in very time-constrained situations. While it will probably not find anything near the quality of ParamILS in the same time, it will yield a decent solution very quickly. This could be used for point clouds where the default configuration fails, like very noisy data or topologically difficult objects.

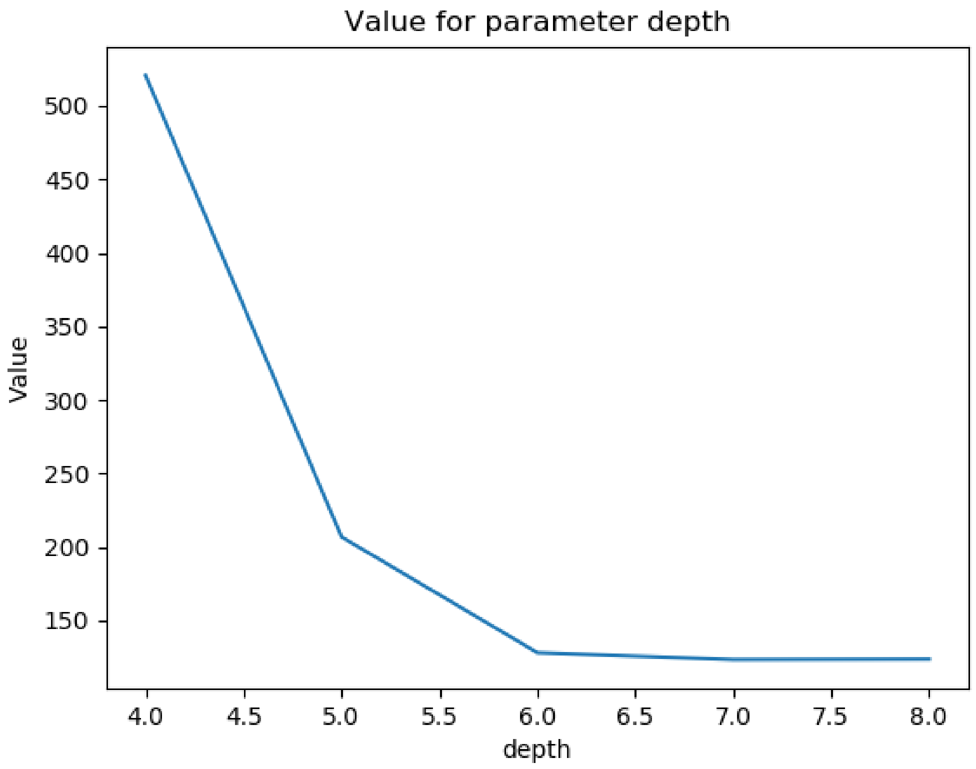


Figure 5.2: Chamfer Distance average for each depth value taken from the results of a ‘famous_original’ - ‘Armadillo’ run.

5.5 Recursive Optimization

One big problem during this work was to find optimal algorithm configurations to run each algorithm with. This task required manual testing and assumptions on what would work and what would not and probably did not find the actual best possible solutions. Another constraint was time, the algorithm configurations should not only perform well but also terminate in a reasonable time frame. Without any time constraints, it would have been possible to recursively optimize the optimization algorithm with another instance of itself, using a strategy algorithm as a target in the framework, seeing its algorithm configurations as parameter configurations. This leads to a hen-egg problem. So why stop there? While even one recursion layer is impossible with the hardware we have available, we might find a better algorithm configuration with an optimization algorithm that was found by an optimized optimization algorithm. This idea can of course be repeated indefinitely, so it would be an interesting thought experiment to see how far a recursion might make sense. In the near future, it might be possible to add one layer of optimization and see how that affects the resulting reconstructions.

5.6 Machine Learning

Two big problems of optimizing SPSR for point clouds is that it takes a long time for each object and that there is not one solution that fits all point clouds. A possibility to alleviate these problems would be to use a neural network to predict optimal SPSR parameter configurations. A training set of point clouds and their optimized SPSR configurations could be created using the presented framework. This training set would have to include as many different types of objects and qualities of point clouds as possible. Alternatively, Reinforcement Learning could be used to avoid that task, since we already have a framework for calculating the quality of a reconstruction. A common problem would be how to supply the neural network with some representation of the point cloud as a feature vector. A possibility might be to use PointNet or PointNet++ by Qi et al. [QSKG17][QYSG17] or Poco by Boulch and Renaud [BM22] for this task. If successful, it should then be possible to get an optimal SPSR parameter configuration for any point cloud nearly instantly.

5.7 Other Targets

It would be interesting to apply the used algorithms to different targets to see if the conclusions we reached also hold on those. This could include similar ones, for example, Points2Surf by Erler et al. [EGO+20], which could also use the Chamfer Distance as a quality metric, or completely different ones that might be more suited to other algorithms than GEIST and ParamILS. It would also be interesting to construct target functions specifically to better understand certain algorithms. For example, a function with many local optima could be used to test the avoidance of those present in ParamILS.

CHAPTER 6

Tables

Table 6.1: Test Results of different GEIST algorithm configurations. Optimizing SPSR for the 'Armadillo' point cloud from the 'famous_original' dataset.

	threshold			**size**		**evaluation**	
config	start	change	fixed	start	iter	chamfer	time
1	0.4	0.04	0.02	150	15	117.393	177.47
1	0.4	0.04	0.02	150	15	118.433	179.63
1	0.4	0.04	0.02	150	15	117.785	193.45
avg	0.4	0.04	0.02	150	15	117.870	183.52
stddev						0.525	8.67
2	0.3	0.02	0.01	150	15	117.817	145.58
2	0.3	0.02	0.01	150	15	118.096	154.14
2	0.3	0.02	0.01	150	15	117.552	146.61
2	0.3	0.02	0.01	150	15	118.860	130.17
2	0.3	0.02	0.01	150	15	117.836	151.14
2	0.3	0.02	0.01	150	15	117.785	186.37
avg	0.3	0.02	0.01	150	15	117.991	152.34
stddev						0.460	18.61
3	0.2	0.01	0.005	150	15	117.925	158.19
3	0.2	0.01	0.005	150	15	117.585	176.19
3	0.2	0.01	0.005	150	15	117.693	188.17
avg	0.2	0.01	0.005	150	15	117.735	174.18
stddev						0.174	15.09
4	0.3	0.04	0.02	150	15	119.438	66.16
4	0.3	0.04	0.02	150	15	117.959	82.73
4	0.3	0.04	0.02	150	15	118.406	108.97
4	0.3	0.04	0.02	150	15	117.432	112.64
4	0.3	0.04	0.02	150	15	117.605	101.22

Table 6.1 – *Continued from previous page*

	threshold			**size**		**evaluation**	
config	start	change	fixed	start	iter	chamfer	time
4	0.3	0.04	0.02	150	15	118.363	106.41
4	0.3	0.04	0.02	150	15	118.876	122.07
4	0.3	0.04	0.02	150	15	117.753	95.22
4	0.3	0.04	0.02	150	15	118.543	101.22
avg	0.3	0.04	0.02	150	15	118.264	99.63
stddev						0.646	16.72
5	0.3	0.01	0.005	150	15	117.809	272.80
5	0.3	0.01	0.005	150	15	117.459	267.81
5	0.3	0.01	0.005	150	15	118.342	243.30
avg	0.3	0.01	0.005	150	15	117.870	261.30
stddev						0.445	15.79
6	0.3	0.04	0.02	15	15	119.745	300.53
6	0.3	0.04	0.02	15	15	119.292	184.54
6	0.3	0.04	0.02	15	15	118.207	198.27
avg	0.3	0.04	0.02	15	15	119.081	227.78
stddev						0.790	63.38
7	0.3	0.04	0.02	1500	15	117.684	156.16
7	0.3	0.04	0.02	1500	15	118.012	174.93
7	0.3	0.04	0.02	1500	15	118.221	151.10
avg	0.3	0.04	0.02	1500	15	117.972	160.73
stddev						0.270	12.55
8	0.3	0.04	0.02	1500	150	117.626	220.38
8	0.3	0.04	0.02	1500	150	117.756	105.02
8	0.3	0.04	0.02	1500	150	117.296	192.55
avg	0.3	0.04	0.02	1500	150	117.559	172.65
stddev						0.237	60.20

Table 6.2: Test Results of different IF-Race algorithm configurations. Optimizing SPSR for the ´Armadillo' point cloud from the ´famous_original' dataset.

	parameters					**evaluation**	
config	start	iters	nr_good	maxIter	anneal	chamfer	time
1	30	15	10	20	0.3	119.221	22.35
1	30	15	10	20	0.3	119.520	18.80
1	30	15	10	20	0.3	119.272	24.48
1	30	15	10	20	0.3	119.227	25.57
1	30	15	10	20	0.3	118.621	23.61
avg	30	15	10	20	0.3	119.172	22.96
stddev						0.332	2.61
2	30	30	10	20	0.3	118.050	26.53

Table 6.2 – *Continued from previous page*

	parameters					**evaluation**	
config	start	iters	nr_good	maxIter	anneal	chamfer	time
2	30	30	10	20	0.3	117.437	25.51
2	30	30	10	20	0.3	118.259	27.72
2	30	30	10	20	0.3	118.485	30.62
2	30	30	10	20	0.3	118.786	23.68
2	30	30	10	20	0.3	119.501	33.10
2	30	30	10	20	0.3	118.524	33.53
2	30	30	10	20	0.3	117.784	32.15
2	30	30	10	20	0.3	118.018	32.80
2	30	30	10	20	0.3	118.924	32.13
avg	30	30	10	20	0.3	118.377	29.78
stddev						0.601	3.59
3	30	45	10	20	0.3	118.620	41.61
3	30	45	10	20	0.3	118.667	42.85
3	30	45	10	20	0.3	118.072	47.35
3	30	45	10	20	0.3	118.143	51.97
3	30	45	10	20	0.3	119.757	50.54
avg	30	45	10	20	0.3	118.652	46.86
stddev						0.674	4.57
4	30	30	10	20	0.4	117.826	38.63
4	30	30	10	20	0.4	118.769	43.59
4	30	30	10	20	0.4	118.362	42.44
4	30	30	10	20	0.4	118.635	47.17
4	30	30	10	20	0.4	118.933	47.77
4	30	30	10	20	0.4	117.838	32.13
4	30	30	10	20	0.4	118.406	37.60
4	30	30	10	20	0.4	118.864	41.56
4	30	30	10	20	0.4	118.249	41.21
4	30	30	10	20	0.4	118.865	42.51
avg	30	30	10	20	0.4	118.475	41.46
stddev						0.411	4.58
5	30	15	10	20	0.4	118.577	42.89
5	30	15	10	20	0.4	118.930	39.50
5	30	15	10	20	0.4	118.892	43.71
5	30	15	10	20	0.4	119.383	26.81
5	30	15	10	20	0.4	119.267	30.97
avg	30	15	10	20	0.4	119.010	36.78
stddev						0.321	7.51
6	30	45	10	20	0.4	118.620	52.02
6	30	45	10	20	0.4	118.313	52.59
6	30	45	10	20	0.4	117.770	55.70

Table 6.2 – *Continued from previous page*

	parameters					**evaluation**	
config	start	iters	nr_good	maxIter	anneal	chamfer	time
6	30	45	10	20	0.4	118.603	55.53
6	30	45	10	20	0.4	118.107	66.18
avg	30	45	10	20	0.4	118.283	56.40
stddev						0.357	5.71
7	30	15	10	20	0.5	118.323	46.71
7	30	15	10	20	0.5	118.282	52.35
7	30	15	10	20	0.5	117.766	54.16
7	30	15	10	20	0.5	118.034	52.79
7	30	15	10	20	0.5	118.472	50.53
avg	30	15	10	20	0.5	118.175	51.31
stddev						0.278	2.88
8	30	30	10	20	0.5	118.116	59.39
8	30	30	10	20	0.5	117.805	61.04
8	30	30	10	20	0.5	117.461	60.98
8	30	30	10	20	0.5	118.410	64.13
8	30	30	10	20	0.5	118.448	65.06
8	30	30	10	20	0.5	118.382	40.47
8	30	30	10	20	0.5	117.659	38.34
8	30	30	10	20	0.5	117.744	39.62
8	30	30	10	20	0.5	117.506	46.55
8	30	30	10	20	0.5	119.012	41.23
avg	30	30	10	20	0.5	118.054	51.68
stddev						0.503	11.31
9	30	45	10	20	0.5	117.703	56.20
9	30	45	10	20	0.5	117.537	55.11
9	30	45	10	20	0.5	118.183	61.29
9	30	45	10	20	0.5	118.215	62.83
9	30	45	10	20	0.5	117.495	60.97
avg	30	45	10	20	0.5	117.826	59.28
stddev						0.349	3.41
10	30	15	10	20	0.6	117.825	53.21
10	30	15	10	20	0.6	117.858	45.17
10	30	15	10	20	0.6	118.027	55.00
10	30	15	10	20	0.6	119.102	48.49
10	30	15	10	20	0.6	118.256	46.75
10	30	15	10	20	0.6	118.377	43.29
10	30	15	10	20	0.6	117.708	53.84
10	30	15	10	20	0.6	117.433	50.19
10	30	15	10	20	0.6	117.979	49.12
10	30	15	10	20	0.6	118.266	43.63

Table 6.2 – *Continued from previous page*

	parameters					**evaluation**	
config	start	iters	nr_good	maxIter	anneal	chamfer	time
avg	30	15	10	20	0.6	118.083	48.87
stddev						0.457	4.21
11	30	15	5	20	0.6	117.463	38.33
11	30	15	5	20	0.6	118.229	37.77
11	30	15	5	20	0.6	118.008	50.49
11	30	15	5	20	0.6	121.675	46.30
11	30	15	5	20	0.6	117.267	49.04
avg	30	15	5	20	0.6	118.528	44.39
stddev						1.802	5.98
12	30	15	15	20	0.6	117.511	73.86
12	30	15	15	20	0.6	118.321	78.64
12	30	15	15	20	0.6	118.691	82.79
12	30	15	15	20	0.6	118.465	77.89
12	30	15	15	20	0.6	118.234	77.37
avg	30	15	15	20	0.6	118.245	78.11
stddev						0.445	3.20
13	30	30	10	20	0.6	117.591	91.13
13	30	30	10	20	0.6	117.519	65.74
13	30	30	10	20	0.6	117.719	73.38
13	30	30	10	20	0.6	117.446	67.40
13	30	30	10	20	0.6	118.042	77.02
13	30	30	10	20	0.6	117.447	76.02
13	30	30	10	20	0.6	117.943	50.06
13	30	30	10	20	0.6	118.024	52.85
13	30	30	10	20	0.6	118.169	60.60
13	30	30	10	20	0.6	117.725	62.71
avg	30	30	10	20	0.6	117.763	67.69
stddev						0.266	12.24
14	30	45	10	20	0.6	117.507	93.65
14	30	45	10	20	0.6	117.346	97.64
14	30	45	10	20	0.6	117.214	97.23
14	30	45	10	20	0.6	117.561	103.77
14	30	45	10	20	0.6	117.764	61.09
avg	30	45	10	20	0.6	117.478	90.67
stddev						0.210	16.93
15	30	30	10	20	0.7	117.475	96.07
15	30	30	10	20	0.7	117.401	88.32
15	30	30	10	20	0.7	117.558	106.09
15	30	30	10	20	0.7	117.325	104.92
15	30	30	10	20	0.7	117.868	107.88

Table 6.2 – *Continued from previous page*

	parameters					**evaluation**	
config	start	iters	nr_good	maxIter	anneal	chamfer	time
avg	30	30	10	20	0.7	117.525	100.66
stddev						0.210	8.26
16	30	30	10	20	0.8	117.753	158.23
16	30	30	10	20	0.8	117.169	163.89
16	30	30	10	20	0.8	117.565	163.00
16	30	30	10	20	0.8	117.521	167.27
16	30	30	10	20	0.8	117.401	179.51
avg	30	30	10	20	0.8	117.482	166.38
stddev						0.216	8.02
17	30	30	10	20	0.9	117.481	220.50
17	30	30	10	20	0.9	117.817	222.80
17	30	30	10	20	0.9	117.540	163.17
17	30	30	10	20	0.9	117.142	140.18
17	30	30	10	20	0.9	117.460	144.65
avg	30	30	10	20	0.9	117.488	178.26
stddev						0.240	40.54

Table 6.3: Test Results of different ParamILS algorithm configurations. Optimizing SPSR for the ´Armadillo' point cloud from the ´famous_original' dataset.

	parameters			**evaluation**	
config	start	perturbation	repetitions	chamfer	time
1	15	0.1	5	121.701	1.89
1	15	0.1	5	119.236	4.00
1	15	0.1	5	118.367	39.35
1	15	0.1	5	118.543	15.86
1	15	0.1	5	118.967	36.56
1	15	0.1	5	118.228	58.82
avg	15	0.1	5	119.174	26.08
stddev				1.294	22.52
2	15	0.2	5	119.853	4.87
2	15	0.2	5	121.320	9.71
2	15	0.2	5	121.405	4.53
2	15	0.2	5	117.941	42.16
2	15	0.2	5	118.831	7.90
2	15	0.2	5	119.683	1.71
avg	15	0.2	5	119.839	11.81
stddev				1.363	15.12
3	15	0.3	5	118.258	4.83
3	15	0.3	5	118.680	11.61

Table 6.3 – *Continued from previous page*

	parameters			**evaluation**	
config	start	perturbation	repetitions	chamfer	time
3	15	0.3	5	119.548	22.37
3	15	0.3	5	119.891	9.50
3	15	0.3	5	119.673	16.16
3	15	0.3	5	119.895	57.26
avg	15	0.3	5	119.324	20.29
stddev				0.688	19.07
4	15	0.4	5	119.551	54.25
4	15	0.4	5	118.766	9.93
4	15	0.4	5	121.998	7.19
4	15	0.4	5	119.557	1.91
4	15	0.4	5	119.860	15.76
4	15	0.4	5	119.379	16.76
avg	15	0.4	5	119.852	17.63
stddev				1.112	18.76
5	15	0.5	5	120.189	5.90
5	15	0.5	5	117.610	49.41
5	15	0.5	5	119.512	73.62
5	15	0.5	5	117.418	36.95
5	15	0.5	5	118.267	42.84
5	15	0.5	5	119.693	7.09
avg	15	0.5	5	118.782	35.97
stddev				1.170	26.02
6	15	0.6	3	119.358	57.49
6	15	0.6	3	117.724	67.02
6	15	0.6	3	118.129	44.86
6	15	0.6	3	118.608	79.14
6	15	0.6	3	119.357	5.48
avg	15	0.6	3	118.635	50.80
stddev				0.730	28.29
7	15	0.6	4	118.131	5.60
7	15	0.6	4	121.891	7.47
7	15	0.6	4	118.347	39.80
7	15	0.6	4	118.233	50.89
7	15	0.6	4	121.060	7.49
avg	15	0.6	4	119.533	22.25
stddev				1.799	21.46
8	15	0.6	5	119.484	5.01
8	15	0.6	5	117.982	6.60
8	15	0.6	5	120.506	23.99
8	15	0.6	5	120.175	5.11

Table 6.3 – *Continued from previous page*

	parameters			**evaluation**	
config	start	perturbation	repetitions	chamfer	time
8	15	0.6	5	120.321	11.52
8	15	0.6	5	120.695	8.04
8	15	0.6	5	117.644	61.26
8	15	0.6	5	119.957	11.44
8	15	0.6	5	118.456	116.61
8	15	0.6	5	118.213	5.27
8	15	0.6	5	118.181	75.12
avg	15	0.6	5	119.238	30.00
stddev				1.151	37.58
9	30	0.6	5	118.435	96.46
9	30	0.6	5	118.095	8.24
9	30	0.6	5	119.053	53.08
9	30	0.6	5	121.475	32.86
9	30	0.6	5	119.961	49.51
avg	30	0.6	5	119.404	48.03
stddev				1.358	32.35
10	75	0.6	5	117.757	5.19
10	75	0.6	5	118.189	9.64
10	75	0.6	5	120.922	7.29
10	75	0.6	5	118.628	12.17
10	75	0.6	5	118.499	69.21
avg	75	0.6	5	118.799	20.70
stddev				1.233	27.24
11	150	0.6	5	118.744	11.80
11	150	0.6	5	118.232	36.81
11	150	0.6	5	118.399	51.18
11	150	0.6	5	117.953	61.44
11	150	0.6	5	118.110	8.80
avg	150	0.6	5	118.288	34.01
stddev				0.303	23.37
12	15	0.6	6	118.116	1.22
12	15	0.6	6	119.886	5.21
12	15	0.6	6	121.505	3.92
12	15	0.6	6	118.313	9.02
12	15	0.6	6	117.777	2.34
avg	15	0.6	6	119.119	4.34
stddev				1.560	3.03
13	15	0.6	7	119.712	7.83
13	15	0.6	7	119.022	69.99
13	15	0.6	7	118.309	59.70

Table 6.3 – *Continued from previous page*

	parameters			**evaluation**	
config	start	perturbation	repetitions	chamfer	time
13	15	0.6	7	119.318	4.32
13	15	0.6	7	119.969	3.16
avg	15	0.6	7	119.266	29.00
stddev				0.647	32.97
14	15	0.7	5	119.262	57.56
14	15	0.7	5	118.774	34.11
14	15	0.7	5	118.956	6.62
14	15	0.7	5	118.880	7.34
14	15	0.7	5	118.697	7.45
14	15	0.7	5	122.167	5.25
avg	15	0.7	5	119.456	19.72
stddev				1.342	21.56
15	15	0.8	5	119.141	4.56
15	15	0.8	5	119.174	8.90
15	15	0.8	5	117.758	89.35
15	15	0.8	5	119.351	2.47
15	15	0.8	5	120.247	23.03
15	15	0.8	5	119.554	3.25
avg	15	0.8	5	119.204	21.93
stddev				0.816	33.90
16	15	0.9	5	121.272	47.54
16	15	0.9	5	122.933	48.51
16	15	0.9	5	119.330	5.36
16	15	0.9	5	121.881	3.76
16	15	0.9	5	118.385	17.28
16	15	0.9	5	117.543	43.90
avg	15	0.9	5	120.224	27.72
stddev				2.124	21.31
17	15	1	5	119.043	90.00
17	15	1	5	119.083	68.66
17	15	1	5	122.449	30.80
17	15	1	5	119.486	72.88
17	15	1	5	117.661	25.18
17	15	1	5	118.739	2.00
avg	15	1	5	119.410	48.25
stddev				1.612	33.89

Table 6.4: Test Results of different PostSelection algorithm configurations. Optimizing SPSR for the ´Armadillo' point cloud from the ´famous_original' dataset.

	strategies		**parameters**		**evaluation**	
config	generate	evaluate	gen_it	gen_rep	chamfer	time
1	Fast-IFRace	Fast-GEIST	10	5	117.378	199.14
1	Fast-IFRace	Fast-GEIST	10	5	118.380	174.49
1	Fast-IFRace	Fast-GEIST	10	5	117.882	198.77
avg	Fast-IFRace	Fast-GEIST	10	5	117.880	190.80
stddev					0.501	14.13
2	Fast-paramILS	Fast-GEIST	10	5	117.998	120.43
2	Fast-paramILS	Fast-GEIST	10	5	118.464	117.78
2	Fast-paramILS	Fast-GEIST	10	5	117.974	154.03
avg	Fast-paramILS	Fast-GEIST	10	5	118.145	130.75
stddev					0.276	20.21
3	Fast-IFRace	Fast-paramILS	10	5	117.996	114.47
3	Fast-IFRace	Fast-paramILS	10	5	118.077	111.54
3	Fast-IFRace	Fast-paramILS	10	5	118.171	112.54
avg	Fast-IFRace	Fast-paramILS	10	5	118.081	112.85
stddev					0.088	1.49
4	Fast-GEIST	Fast-paramILS	10	5	117.693	238.04
4	Fast-GEIST	Fast-paramILS	10	5	117.711	161.16
4	Fast-GEIST	Fast-paramILS	10	5	117.425	143.31
4	Fast-GEIST	Fast-paramILS	10	5	117.680	153.55
4	Fast-GEIST	Fast-paramILS	10	5	117.965	155.70
4	Fast-GEIST	Fast-paramILS	10	5	117.902	188.09
avg	Fast-GEIST	Fast-paramILS	10	5	117.729	173.31
stddev					0.191	35.09
5	Good-IFRace	Fast-paramILS	10	5	117.601	333.29
5	Good-IFRace	Fast-paramILS	10	5	117.717	374.84
5	Good-IFRace	Fast-paramILS	10	5	117.345	353.79
avg	Good-IFRace	Fast-paramILS	10	5	117.554	353.97
stddev					0.190	20.77
6	Good-IFRace	Fast-GEIST	10	5	118.231	383.08
6	Good-IFRace	Fast-GEIST	10	5	117.446	389.86
6	Good-IFRace	Fast-GEIST	10	5	118.058	370.76
avg	Good-IFRace	Fast-GEIST	10	5	117.912	381.23
stddev					0.412	9.68
7	Good-IFRace	Short-GEIST	10	5	117.177	387.72
7	Good-IFRace	Short-GEIST	10	5	117.296	385.34
7	Good-IFRace	Short-GEIST	10	5	116.781	446.55
avg	Good-IFRace	Short-GEIST	10	5	117.085	406.54
stddev					0.270	34.67

Table 6.5: Different algorithm configurations used in our PostSelection tests.

config	**parameters**				
IFRace	start	iters	nr_good	maxIter	anneal
Fast-IFRace	30	15	10	20	0.3
Good-IFRace	30	15	10	20	0.9
paramILS	start	perturbation	repetitions		
Fast-paramILS	15	0.6	6		
GEIST	threshold			size	
	optimal	change	fixed	start	iter
Fast-GEIST	0.3	0.04	0.02	150	15
Short-GEIST	0.1	0.04	0.02	150	15
Thorough-GEIST	0.1	0.04	0.02	150	150

Table 6.4 – *Continued from previous page*

	strategies		**parameters**		**evaluation**	
config	generate	evaluate	gen_it	gen_rep	chamfer	time
8	Thorough-GEIST	Fast-paramILS	10	1	117.455	149.61
8	Thorough-GEIST	Fast-paramILS	10	1	117.663	192.58
8	Thorough-GEIST	Fast-paramILS	10	1	117.155	182.54
avg	Thorough-GEIST	Fast-paramILS	10	1	117.425	174.91
stddev					0.255	22.48

Table 6.6: Test Results of different Brute-Force algorithm configurations. Optimizing SPSR for the ´Armadillo' point cloud from the ´famous_original' dataset.

Theshold	**Chamfer Distance**	**run time**
117.9	117.78	176.3
117.9	117.87	35.1
117.9	117.83	610.2
117.9	117.66	396.3
117.9	117.89	163.7
avg	117.81	276.3
stddev	0.09	227.3
118	117.54	3.9
118	118.00	627.0
118	117.94	666.4
118	117.99	402.7
118	117.34	24.6
avg	117.76	344.9
stddev	0.30	318.3
118.1	117.96	31.4
118.1	118.00	6.4

Table 6.6 – *Continued from previous page*

Theshold	Chamfer Distance	run time
118.1	118.08	57.5
118.1	117.79	82.3
118.1	118.06	154.7
avg	117.98	66.5
stddev	0.12	56.9
118.2	118.05	103.8
118.2	118.13	60.0
118.2	118.01	158.1
118.2	118.16	7.8
118.2	118.00	114.3
avg	118.07	88.8
stddev	0.07	57.2
118.3	118.19	43.6
118.3	118.13	307.4
118.3	118.25	88.4
118.3	118.14	27.7
118.3	118.21	282.8
avg	118.18	150.0
stddev	0.05	134.6

Table 6.7: Test Results of optimizing SPSR for the ´Armadillo’ point cloud from the ´famous_original’ dataset using different algorithms.

run	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	run time
default	0	8	5	8	4	1.5	1.1	0.0183	120.50	
GEIST										
1	0	7	8	10	3.2	1	1.5	0.0205	117.53	91.9
2	0	7	7	10	2.2	1	1	0.0179	117.41	128.9
3	1	7	7	9	2	1	1.1	0.0185	117.09	122.9
average								0.0190	117.34	114.6
stddev								0.0014	0.23	19.8
IFRace										
1	1	7	6	9	4.6	1.3	1.5	0.0249	118.51	41.0
2	1	7	7	9	2.2	1.1	1.1	0.0181	117.50	36.7
3	0	7	5	10	2.4	1	1.2	0.0176	117.28	31.7
average								0.0202	117.76	36.5
stddev								0.0041	0.66	4.6
ParamILS										
1	1	8	8	6	3.6	1.2	1.2	0.0179	117.89	38.2
2	0	7	7	8	3.4	1.2	1.2	0.0182	117.97	10.7
3	0	7	7	6	3	1.5	1.1	0.0204	118.56	11.1

Table 6.7 – *Continued from previous page*

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
average								0.0188	118.14	20.0
stddev								0.0014	0.36	15.8
PostSelection										
1	0	7	8	9	2.8	1.1	1.4	0.0182	117.36	185.7
2	1	8	8	8	3.6	1.2	1	0.0186	117.96	164.6
3	0	8	8	8	2.2	1.1	1.1	0.0173	117.33	173.6
average								0.0181	117.55	174.6
stddev								0.0007	0.35	10.6

Table 6.8: Test Results of optimizing SPSR for the ´Armadillo’ point cloud from the ´famous_dense’ dataset using different algorithms.

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
default	0	8	5	8	4	1.5	1.1	0.0176	115.56	
GEIST										
1	1	7	5	6	2	1.2	1.1	0.0164	111.90	126.4
2	1	7	7	6	2	1.1	1	0.0171	112.62	117.3
3	1	7	6	7	2	1.8	1.1	0.0180	112.64	170.6
average								0.0171	112.39	138.1
stddev								0.0008	0.42	28.5
IFRace										
1	0	7	8	8	2.8	1.4	1.2	0.0187	112.76	35.0
2	0	7	5	8	2.8	1	1.2	0.0172	112.81	28.6
3	1	7	6	7	3.8	1.9	1.4	0.0173	113.19	27.7
average								0.0177	112.92	30.4
stddev								0.0009	0.24	4.0
ParamILS										
1	1	7	5	7	2.2	1	1.2	0.0160	112.48	6.8
2	0	7	7	7	3.8	1.1	1.2	0.0180	113.02	11.9
3	0	8	5	9	2	1.9	1.3	0.0175	113.24	7.5
average								0.0172	112.91	8.8
stddev								0.0011	0.39	2.7
PostSelection										
1	1	7	7	7	4	1	1.1	0.0171	112.80	208.1
2	0	7	5	10	2	1	1.1	0.0185	112.45	182.4
3	1	7	5	9	2.6	1.2	1	0.0171	112.90	174.8
average								0.0176	112.71	188.5
stddev								0.0008	0.23	17.5

Table 6.9: Test Results of optimizing SPSR for the ´Armadillo' point cloud from the ´famous_extra_noisy' dataset using different algorithms.

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
default	0	8	5	8	4	1.5	1.1	0.0584	239.7	
GEIST										
1	0	6	8	7	2.4	1.4	1.7	0.0407	167.58	108.3
2	0	6	6	6	2	1.2	1.6	0.0399	166.41	125.9
3	1	6	5	9	2.2	1	1.5	0.0405	165.82	118.8
average								0.0404	166.60	117.7
stddev								0.0004	0.90	8.8
IFRace										
1	0	5	6	9	3.4	1.1	0.9	0.0465	167.54	21.8
2	1	5	7	10	2.2	3	0.9	0.0467	170.47	22.6
3	0	6	7	6	2.2	1.2	1.5	0.0383	167.00	26.6
average								0.0438	168.34	23.7
stddev								0.0048	1.87	2.5
ParamILS										
1	1	5	7	10	3.8	1.9	0.9	0.0436	171.05	8.5
2	0	5	8	7	4.2	1.6	0.9	0.0437	170.78	6.7
3	1	6	8	9	2.4	1.3	1.6	0.0347	166.73	10.8
average								0.0407	169.52	8.6
stddev								0.0052	2.42	2.1
PostSelection										
1	1	6	6	8	2.8	1.1	1.7	0.0371	166.06	180.1
2	1	5	8	9	2.2	1.6	0.9	0.0486	168.48	195.4
3	0	6	8	6	3.4	1	1.7	0.0357	167.40	181.2
average								0.0405	167.32	185.6
stddev								0.0071	1.21	8.5

Table 6.10: Test Results of optimizing SPSR for the ´cloud 1' point cloud from the ´ABC_extra_noisy' dataset using different algorithms.

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
default	0	8	5	8	4	1.5	1.1	0.0519	251.70	
GEIST										
1	0	6	8	7	4	1.6	0.9	0.0707	197.39	92.9
2	0	6	6	10	5.6	1.2	0.9	0.0621	196.20	155.9
3	0	6	8	9	4	1.7	0.9	0.0621	196.69	149.3
average								0.0650	196.76	132.7
stddev								0.0049	0.60	34.6
IFRace										
1	0	6	8	9	2.4	4.1	0.9	0.0502	195.74	30.4

Table 6.10 – *Continued from previous page*

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
2	0	6	7	10	3.4	3.6	1	0.0523	198.36	31.0
3	0	6	7	10	5.8	2.1	1	0.0545	199.06	34.2
average								0.0524	197.72	31.9
stddev								0.0021	1.75	2.0
ParamILS										
1	1	7	7	10	2.6	1.2	1.5	0.0671	196.27	16.2
2	0	6	6	7	3	2.6	0.9	0.0578	197.03	7.6
3	1	6	6	8	2.6	2.8	0.9	0.0633	196.75	7.6
average								0.0627	196.68	10.5
stddev								0.0047	0.39	4.9
PostSelection										
1	1	6	7	10	2.4	2.7	0.9	0.0647	196.55	208.1
2	1	7	7	9	2.8	2	1.7	0.0621	198.24	162.3
3	1	6	8	9	2.2	3	0.9	0.0683	196.54	165.1
average								0.0650	197.11	178.5
stddev								0.0031	0.98	25.7

Table 6.11: Test Results of optimizing SPSR for the ´Armadillo’ point cloud from the ´famous_noisefree’ dataset using different algorithms.

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
default	0	8	5	8	4	1.5	1.1	0.0185	112.15	
GEIST										
1	0	8	8	10	7.8	1.1	1.3	0.0191	110.57	58.9
2	0	8	7	7	7.4	1	1.2	0.0185	110.50	122.4
3	0	8	5	7	6.8	1	1	0.0189	110.39	113.1
average								0.0188	110.49	98.2
stddev								0.0003	0.09	34.3
IFRace										
1	0	8	8	6	6.4	1.1	1.1	0.0181	110.58	35.8
2	1	8	8	9	7	1	1.4	0.0200	110.18	40.1
3	0	8	8	8	7.6	1	1.6	0.0160	110.32	38.2
average								0.0180	110.36	38.0
stddev								0.0020	0.20	2.2
ParamILS										
1	0	8	6	6	7.4	1.4	1.1	0.0194	111.09	11.0
2	0	8	8	8	2.4	1.2	1.2	0.0178	111.40	35.5
3	1	8	5	9	5.2	1.2	1.2	0.0172	111.18	7.5
average								0.0181	111.22	18.0
stddev								0.0011	0.16	15.3
PostSelection										

Table 6.11 – *Continued from previous page*

run	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	run time
1	1	8	6	6	4.4	1.1	1	0.0185	110.62	181.7
2	1	8	8	8	7.6	1.1	1.2	0.0163	110.77	156.7
3	0	8	5	8	5.6	1	1	0.0176	110.79	157.0
average								0.0174	110.73	165.1
stddev								0.0011	0.09	14.4

Table 6.12: Test Results of optimizing SPSR for the ´cloud 1' point cloud from the ´abc_noisefree' dataset using different algorithms.

run	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	run time
default	0	8	5	8	4	1.5	1.1	0.0843	198.51	
GEIST										
1	1	8	5	8	7.8	1	1.1	0.0855	197.20	87.9
2	0	8	8	6	4.2	1	1	0.0852	196.13	86.3
3	1	8	8	6	2.2	1	1	0.0860	196.82	104.8
average								0.0856	196.72	93.0
stddev								0.0004	0.54	10.3
IFRace										
1	0	8	5	8	4.4	1	1	0.0829	195.04	37.2
2	0	8	7	6	6.4	1.1	1.5	0.0835	194.95	34.2
3	1	8	7	9	7.8	1	1.1	0.0843	195.28	34.0
average								0.0836	195.09	35.2
stddev								0.0007	0.17	1.8
ParamILS										
1	1	8	8	8	8	1.2	1.4	0.0841	195.74	45.8
2	1	8	6	9	4.4	1	1.2	0.0847	196.06	11.9
3	0	8	5	10	6.2	1	1.4	0.0853	196.45	6.7
average								0.0847	196.09	21.5
stddev								0.0006	0.35	21.3
PostSelection										
1	0	8	8	7	6.6	1	1	0.0837	194.27	210.2
2	1	8	7	6	6.4	1	1	0.0842	193.21	204.5
3	0	8	8	6	3.4	1	1.4	0.0850	195.22	235.9
average								0.0843	194.23	216.8
stddev								0.0007	1.00	16.7

Table 6.13: Test Results of optimizing SPSR for the ´Armadillo' point cloud from the ´famous_sparse' dataset using different algorithms.

run	**cgD**	**D**	**fD**	**it**	**w**	**Sa**	**Sc**	**Hausdorff**	**Chamfer**	**run time**
default	0	8	5	8	4	1.5	1.1	0.0643	149.13	
GEIST										
1	0	8	8	8	5.8	1.2	1.1	0.0531	141.70	103.8
2	1	8	6	9	7.2	1.2	1.3	0.0509	141.72	57.9
3	0	8	6	10	5.8	1.1	1.3	0.0504	141.34	118.0
average								0.0515	141.58	93.3
stddev								0.0015	0.22	31.4
IFRace										
1	0	8	7	9	4.8	4.8	1.5	0.0571	144.41	41.4
2	1	8	7	10	5.2	5.2	1	0.0506	139.64	43.8
3	0	8	7	10	4	4	1	0.0505	141.57	39.0
average								0.0527	141.87	41.4
stddev								0.0038	2.40	2.4
ParamILS										
1	1	8	8	10	5.6	1.4	1.3	0.0507	142.17	64.0
2	0	8	6	10	5.4	1.3	1.1	0.0506	141.04	25.1
3	1	8	6	10	6	1.3	1.3	0.0506	140.04	11.3
average								0.0506	141.08	33.4
stddev								0.0001	1.06	27.3
PostSelection										
1	0	8	8	8	5.6	1.1	1.2	0.0520	140.68	181.7
2	1	8	7	10	4.6	1	1.3	0.0555	142.02	153.6
3	1	8	7	10	4.8	1	1.4	0.0531	141.97	172.5
average								0.0535	141.56	169.3
stddev								0.0018	0.76	14.3

Table 6.14: Test Results of different algorithms optimizing SPSR for the ´famous_original' - ´xyzrgb_dragon_clean' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
dragon	default	0	8	5	8	4	1.5	1.1	0.02808	104.357	
dragon	GEIST	1	8	7	8	2	1	1.1	0.02633	101.329	293.25
dragon	GEIST	0	8	8	6	2.6	1	1	0.02703	101.915	231.20
dragon	GEIST	0	8	8	10	2.2	1	1.2	0.02684	101.881	265.08
	avg								0.02673	101.709	263.18
	stddev								0.00036	0.329	31.07
dragon	IF-Race	0	8	8	9	3.6	1	1	0.02694	101.838	51.65
dragon	IF-Race	0	8	6	9	4.6	1	1	0.02637	102.286	46.89
dragon	IF-Race	1	8	8	7	6	1	1.2	0.02690	102.078	52.09

Table 6.14 – *Continued from previous page*

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
	avg								0.02674	102.067	50.21
	stddev								0.00032	0.224	2.88
dragon	ParamILS	0	8	8	8	6.4	1	1.1	0.02689	101.379	119.46
dragon	ParamILS	1	8	8	9	3.2	1.1	1.3	0.02613	102.086	50.56
dragon	ParamILS	1	8	8	6	6.2	1	1	0.02675	101.351	41.80
	avg								0.02659	101.605	70.61
	stddev								0.00041	0.416	42.54
dragon	PostSelection	0	8	8	8	4.2	1	1.1	0.02702	101.806	224.55
dragon	PostSelection	0	8	5	10	2.8	1	1.1	0.02674	102.108	231.98
dragon	PostSelection	1	8	8	9	2.8	1	1.2	0.02679	101.854	225.62
	avg								0.02685	101.923	227.39
	stddev								0.00015	0.162	4.02

Table 6.15: Test Results of different algorithms optimizing SPSR for the ´famous_original' - ´yoda' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
dragon	default	0	8	5	8	4	1.5	1.1	0.12694	332.823	
yoda	GEIST	1	8	6	7	7.8	1.1	1	0.11925	302.931	119.79
yoda	GEIST	0	8	7	6	8	1	1.2	0.11706	298.987	131.23
yoda	GEIST	0	7	7	10	7.8	1	1.1	0.11749	302.200	53.09
	avg								0.11793	301.373	101.37
	stddev								0.00116	2.098	42.20
yoda	IF-Race	1	8	7	9	7.6	1	1.2	0.12141	300.113	34.24
yoda	IF-Race	0	7	7	6	7.8	1	1.2	0.12028	301.591	34.71
yoda	IF-Race	1	7	8	10	7.6	1	1.3	0.12457	299.202	36.86
	avg								0.12209	300.302	35.27
	stddev								0.00223	1.206	1.40
yoda	ParamILS	0	8	7	6	7	1.1	1.5	0.12277	304.270	9.63
yoda	ParamILS	0	8	6	9	6	1	1	0.12674	304.461	13.47
yoda	ParamILS	1	8	7	6	8	1	1.6	0.11789	299.839	57.18
	avg								0.12247	302.857	26.76
	stddev								0.00444	2.615	26.42
yoda	PostSelection	1	8	7	7	8	1	1.3	0.11836	300.129	174.86
yoda	PostSelection	0	8	6	8	8	1	0.9	0.12069	297.888	166.29
yoda	PostSelection	0	8	6	6	7.8	1	1.1	0.12187	301.259	149.39
	avg								0.12031	299.759	163.51
	stddev								0.00179	1.716	12.96

Table 6.16: Test Results of different algorithms optimizing SPSR for the ´famous_original' - ´bunny' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
dragon	default	0	8	5	8	4	1.5	1.1	0.03078	142.631	
bunny	GEIST	1	7	7	6	5.4	1	1.2	0.03187	141.377	89.27
bunny	GEIST	1	8	7	9	2.4	1.1	1.2	0.03105	141.418	86.68
bunny	GEIST	0	7	8	8	6.6	1	1.3	0.03049	141.091	150.30
	avg								0.03114	141.295	108.75
	stddev								0.00069	0.178	36.01
bunny	IF-Race	0	8	8	8	6.4	1.5	1	0.03105	141.240	40.91
bunny	IF-Race	1	8	7	9	3.2	1.3	1.2	0.02944	141.035	43.29
bunny	IF-Race	0	7	7	8	7	1.7	1.6	0.03063	141.175	32.41
	avg								0.03037	141.150	38.87
	stddev								0.00083	0.105	5.72
bunny	ParamILS	1	7	8	10	4.2	1	1.4	0.03230	141.229	34.65
bunny	ParamILS	1	7	8	10	5.4	1	1.6	0.03138	142.039	27.68
bunny	ParamILS	0	8	8	10	3.4	2.2	1.4	0.03060	142.635	34.78
	avg								0.03143	141.968	32.37
	stddev								0.00085	0.706	4.06
bunny	PostSelection	0	7	7	9	5.2	1.1	1.3	0.03139	141.659	178.01
bunny	PostSelection	1	7	7	8	7.2	1.1	1.6	0.03095	141.525	179.18
bunny	PostSelection	1	7	8	10	5.2	1.6	1.5	0.03238	141.486	184.66
	avg								0.03157	141.557	180.62
	stddev								0.00073	0.090	3.55

Table 6.17: Test Results of different algorithms optimizing SPSR for the ´famous_original' - ´flower' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
dragon	default	0	8	5	8	4	1.5	1.1	0.02378	121.978	
flower	GEIST	1	7	5	10	2.4	1	1.2	0.01937	119.015	76.89
flower	GEIST	1	7	7	10	2.4	1	1.1	0.01848	119.715	142.06
flower	GEIST	0	8	8	6	2.4	1.1	1.3	0.01943	119.417	152.77
	avg								0.01909	119.383	123.91
	stddev								0.00053	0.351	41.07
flower	IF-Race	0	7	7	6	2.6	1	1.3	0.01830	119.473	32.30
flower	IF-Race	1	8	7	10	2.4	1	1	0.01757	119.622	36.10
flower	IF-Race	1	8	5	8	2	1.1	1.2	0.01985	119.836	31.72
	avg								0.01857	119.644	33.38
	stddev								0.00116	0.182	2.38
flower	ParamILS	0	8	8	6	3.6	1.3	1.6	0.01856	120.786	35.58
flower	ParamILS	1	7	5	9	4.2	1.2	1.6	0.01949	120.535	9.48

Table 6.17 – *Continued from previous page*

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
flower	ParamILS	1	7	5	8	3	1	1.5	0.01914	119.974	7.28
	avg								0.01907	120.432	17.45
	stddev								0.00047	0.416	15.74
flower	PostSelection	1	8	6	8	2	1	1.5	0.02012	119.608	158.84
flower	PostSelection	1	7	7	6	2.6	1.2	1	0.02254	119.531	175.40
flower	PostSelection	1	8	7	7	2	1.1	1.6	0.01941	119.458	200.38
	avg								0.02069	119.532	178.21
	stddev								0.00164	0.075	20.91

Table 6.18: Test Results of different algorithms optimizing SPSR for the ´abc' - ´Cloud 1' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
1	default	0	8	5	8	4	1.5	1.1	0.04527	229.563	
1	GEIST	0	6	7	8	2	4.7	0.9	0.04191	181.993	100.65
1	GEIST	0	6	8	10	2	4.6	0.9	0.03904	182.070	111.94
1	GEIST	0	6	6	6	7.2	1	0.9	0.04117	183.221	146.06
	avg								0.04071	182.428	119.55
	stddev								0.00149	0.688	23.64
1	IF-Race	1	6	6	9	3	6.3	1	0.03864	184.881	28.27
1	IF-Race	1	6	7	9	2.2	4.2	0.9	0.03622	182.033	25.60
1	IF-Race	1	6	8	8	2.4	3.4	0.9	0.03569	182.309	33.17
	avg								0.03685	183.074	29.01
	stddev								0.00157	1.571	3.84
1	ParamILS	0	6	8	9	2.2	3.2	0.9	0.04314	182.357	9.78
1	ParamILS	1	6	6	8	6	2.2	1	0.04034	184.859	6.96
1	ParamILS	1	7	7	9	2	2.5	1.6	0.03773	183.151	11.57
	avg								0.04041	183.456	9.44
	stddev								0.00271	1.278	2.32
1	PostSelection	0	7	7	6	3.6	1	1.5	0.04731	182.947	179.46
1	PostSelection	1	6	6	10	5.2	2.2	1	0.04301	184.520	173.81
1	PostSelection	1	6	6	9	2.2	3.7	0.9	0.04013	182.182	172.02
	avg								0.04348	183.216	175.10
	stddev								0.00361	1.192	3.88

Table 6.19: Test Results of different algorithms optimizing SPSR for the ´abc' - ´Cloud 2' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
2	default	0	8	5	8	4	1.5	1.1	0.09171	279.736	
2	GEIST	1	8	7	8	3.4	1	1	0.07925	262.779	102.69
2	GEIST	1	7	8	6	7.4	1	1	0.07458	261.385	94.75
2	GEIST	0	7	8	10	7.8	1	1	0.07360	258.549	122.79
	avg								0.07581	260.904	106.74
	stddev								0.00302	2.156	14.45
2	IF-Race	1	8	7	9	7	1	1	0.08123	258.566	38.46
2	IF-Race	0	8	7	7	7	1	1	0.08411	260.129	34.28
2	IF-Race	0	8	6	10	7.6	1	1	0.07072	260.280	36.12
	avg								0.07869	259.658	36.29
	stddev								0.00705	0.949	2.09
2	ParamILS	1	8	7	8	6.6	1	1	0.07054	258.155	33.15
2	ParamILS	0	8	7	7	6.6	1	0.9	0.08934	260.203	49.35
2	ParamILS	1	8	8	10	5.4	1	1.6	0.07535	261.950	53.24
	avg								0.07841	260.103	45.25
	stddev								0.00977	1.900	10.66
2	PostSelection	1	7	8	8	8	1	1.1	0.07214	261.996	191.20
2	PostSelection	1	8	5	7	7.6	1	1	0.07412	262.959	168.75
2	PostSelection	0	7	8	10	6.2	1.1	1	0.07930	264.277	167.44
	avg								0.07519	263.077	175.80
	stddev								0.00370	1.145	13.35

Table 6.20: Test Results of different algorithms optimizing SPSR for the ´abc' - ´Cloud 3' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
3	default	0	8	5	8	4	1.5	1.1	0.37586	600.651	
3	GEIST	0	8	8	10	7.2	1.6	1.4	0.35730	569.413	104.17
3	GEIST	1	8	7	9	3.8	1.3	1.3	0.35768	572.401	119.71
3	GEIST	0	8	7	9	7.4	1.2	1.3	0.35161	565.893	89.03
	avg								0.35553	569.236	104.30
	stddev								0.00340	3.258	15.34
3	IF-Race	1	8	7	10	7.4	1	1.1	0.34640	564.078	37.97
3	IF-Race	1	8	8	10	5.2	1	1.1	0.34719	560.830	46.63
3	IF-Race	1	8	8	10	6.6	1	1.3	0.34440	559.892	36.04
	avg								0.34600	561.600	40.21
	stddev								0.00144	2.197	5.64
3	ParamILS	1	8	7	9	3	1.8	1.3	0.36789	583.338	19.63
3	ParamILS	1	8	6	10	5.8	1.4	1.1	0.35788	572.220	16.33

Table 6.20 – *Continued from previous page*

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
3	ParamILS	0	8	8	9	2	1.4	1.6	0.37348	590.087	50.20
	avg								0.36642	581.882	28.72
	stddev								0.00790	9.022	18.68
3	PostSelection	0	8	6	10	6.6	1.2	1.3	0.35141	567.463	206.74
3	PostSelection	0	8	8	10	3.6	1.2	1.5	0.36005	573.227	186.98
3	PostSelection	1	8	8	10	8	1	1.6	0.35790	565.467	204.20
	avg								0.35645	568.719	199.31
	stddev								0.00450	4.029	10.75

Table 6.21: Test Results of different algorithms optimizing SPSR for the ´abc' - ´Cloud 4' point cloud.

Cloud	Algorithm	cgD	D	fD	it	w	Sa	Sc	Hausdorff	Chamfer	time
4	default	0	8	5	8	4	1.5	1.1	0.08055	234.982	
4	GEIST	1	8	7	7	6.8	1	1.3	0.08040	226.770	124.01
4	GEIST	0	8	7	6	7	1	1	0.08002	227.139	97.43
4	GEIST	1	8	7	10	7.8	1.4	1.4	0.07871	228.744	91.63
	avg								0.07971	227.551	104.35
	stddev								0.00089	1.049	17.27
4	IF-Race	1	8	7	7	6	1.2	1.4	0.08105	228.762	40.58
4	IF-Race	0	8	6	8	8	1.2	1	0.07519	226.234	39.39
4	IF-Race	0	8	7	9	7.8	1.2	1.4	0.07987	227.537	43.28
	avg								0.07870	227.511	41.08
	stddev								0.00310	1.264	1.99
4	ParamILS	0	8	7	10	8	1	1.1	0.08043	226.108	44.99
4	ParamILS	1	8	7	10	5.8	1.2	1.3	0.08164	227.559	44.40
4	ParamILS	0	8	5	6	7.8	1.4	1.1	0.08173	228.140	6.73
	avg								0.08127	227.269	32.04
	stddev								0.00073	1.047	21.92
4	PostSelection	0	8	7	8	7	1	1	0.08006	226.541	209.44
4	PostSelection	0	8	6	6	8	1	1	0.08119	226.606	207.63
4	PostSelection	1	8	7	7	8	1.1	1	0.07717	228.104	229.56
	avg								0.07947	227.084	215.54
	stddev								0.00208	0.884	12.18

List of Figures

List of Tables

List of Algorithms

Bibliography

[ADL06] Belarmino Adenso-Díaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations Research*, 54(1):99–114, 2006.

[BBS07] Prasanna Balaprakash, Mauro Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In Thomas Bartz-Beielstein, María José Blesa Aguilera, Christian Blum, Boris Naujoks, Andrea Roli, Günter Rudolph, and Michael Sampels, editors, *Hybrid Metaheuristics*, pages 108–122, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BGW13] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.

[BKvS$^+$23] Thomas Bäck, Anna V. Kononova, Bas van Stein, Hongya Wang, Kirill Antonov, Roman Kalkreuth, Jacob De Nobel, Diederick Vermetten, Roy de Winter, and Furong Ye. Evolutionary algorithms for parameter optimization—thirty years later. *Evolutionary Computation*, 31:81–122, 2023.

[BM22] Alexandre Boulch and Renaud Marlet. Poco: Point convolution for surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6302–6314, June 2022.

[Bot98] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[BSPV02] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, page 11–18, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[BTBW77] Harry G Barrow, Jay M Tenenbaum, Robert C Bolles, and Helen C Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proceedings: Image Understanding Workshop*, pages 21–27. Science Applications, Inc Arlington, VA, 1977.

[DDR16] Lakshitha Dantanarayana, Gamini Dissanayake, and Ravindra Ranasinge. C-log: A chamfer distance based algorithm for localisation in occupancy grid-maps. *CAAI Transactions on Intelligence Technology*, 1(3):272–284, 2016.

[EGO$^+$20] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Michael Wimmer, and Niloy Mitra. Points2surf: Learning implicit surfaces from point clouds. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, volume 12350 of *Lecture Notes in Computer Science*, pages 108–124, Cham, October 2020. Springer International Publishing.

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22, 2012.

[GLM04] Robert B. Gramacy, Herbert K. H. Lee, and William G. Macready. Parameter space exploration with gaussian process trees. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 45, New York, NY, USA, 2004. Association for Computing Machinery.

[Han06] Nikolaus Hansen. *The CMA Evolution Strategy: A Comparing Review*, pages 75–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.*, 36(1):267–306, September 2009.

[HKR93] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[HLY20] C. Huang, Y. Li, and X. Yao. A survey of automatic parameter tuning methods for metaheuristics. *IEEE Transactions on Evolutionary Computation*, 24(2):201–216, 2020.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP ’06, page 61–70, Goslar, DEU, 2006. Eurographics Association.

[KH13] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Transactions on Graphics (TOG)*, 32(3):29, 2013.

[KMJ$^+$19] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master’s thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See http://llvm.cs.uiuc.edu.*

[LMS03] Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle. *Iterated Local Search*, pages 320–353. Springer US, Boston, MA, 2003.

[MAJ$^+$17] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. Performance modeling under resource constraints using deep transfer learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’17, New York, NY, USA, 2017. Association for Computing Machinery.

[MBG20] Harshitha Menon, Abhinav Bhatele, and Todd Gamblin. Auto-tuning parameter choices in hpc applications using bayesian optimization. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 831–840, 2020.

[MS78] Robert E Mercer and JR Sampson. Adaptive search using a reproductive meta-plan. *Kybernetes*, 1978.

[QSKG17] Charles Ruizhongtai Qi, H. Su, M. Kaichun, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, 2017.

[QYSG17] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5099–5108. Curran Associates, Inc., 2017.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

[TJA+18] Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, Alfredo Gimenez, Rahul Sridhar, Aniruddha Marathe, Tao Wang, Murali Emani, Abhinav Bhatele, and Todd Gamblin. Bootstrapping parameter space exploration for fast tuning. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 385–395, New York, NY, USA, 2018. Association for Computing Machinery.

[WKB+20] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, H. Finkel, P. Hovland, V. Taylor, and Mary W. Hall. Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization. *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, null:61–70, 2020.

[WR06] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[YFK16] Yuto Yamaguchi, Christos Faloutsos, and Hiroyuki Kitagawa. CAMLP: confidence-aware modulated label propagation. In Sanjay Chawla Venkatasubramanian and Wagner Meira Jr., editors, *Proceedings of the 2016 SIAM International Conference on Data Mining, Miami, Florida, USA, May 5-7, 2016*, pages 513–521. SIAM, 2016.

[YSMdO+13] Zhi Yuan, Thomas Stützle, Marco A. Montes de Oca, Hoong Chuin Lau, and Mauro Birattari. An analysis of post-selection in automatic configuration. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, page 1557–1564, New York, NY, USA, 2013. Association for Computing Machinery.