Informatics

Global Optimisation and Learning for Lighting Design

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Viktoria Petersen
Matrikelnummer 11924496

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Michael Wimmer, Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn.
Mitwirkung: David Hahn, PhD

Wien, 30. September 2024

	Viktoria Petersen	Michael Wimmer

Informatics

Global Optimisation and Learning for Lighting Design

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Viktoria Petersen
Registration Number 11924496

to the Faculty of Informatics

at the TU Wien

Advisor: Michael Wimmer, Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn.
Assistance: David Hahn, PhD

Vienna, September 30, 2024

Viktoria Petersen	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Viktoria Petersen

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.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. September 2024

Viktoria Petersen

Kurzfassung

Global optimisation and learning for lighting design ist eine Erweiterung des Tamashii Rendering Framworks der Rendering and Modeling Gruppe der TU Wien. Tamashii bietet ein graphisches User Interface und die Implementierung für eine lokale Optimierung mit dem Ziel, die vorgegegbene Zielbeleuchtung einer Szene zu rekonstruieren, indem die dafür nötige Position einer oder mehrerer Lichtquelle gefunden wird. Unsere Arbeit ergänzt das Framework mit einer globalen Suche für die optimiale Lichtposition und mit Surrogaten als alternative Optimierungsmethode aus dem Machine Learning Bereich. Wir untersuchen die Leistungsfähigkeit der verschiedenen eingesetzten Algorithmen zur Optimierung in Hinblick auf Effizient, Genauigkeit und Vielseitigkeit. Außerdem vergleichen wir die Optimierungsalgorithm im Einsatz auf dem ursprünglichen Tamashii-Model mit dem Einsatz auf Surrogaten.

Abstract

Global optimisation and learning for lighting design is a work in extension of the Tamashii rendering framework by the Rendering and Modeling Group at TU Wien. Tamashii offers a user interface and implementation to perform a local optimisation task on a scene to find the position of the scene's light object in order to recreate a given target illumination. We extend the existing framework by implementing a global search for the optimum and additional surrogate models to provide machine learning alternatives to apply various optimisation algorithms. We examine the performance of several different optimisation methods with respect to efficiency, accuracy, and versatility. Furthermore, we compare the use of algorithms in combination with surrogate models versus optimising on the Tamashii model directly.

Contents

CHAPTER 1

Introduction

Lighting design is an important aspect to consider when creating an environment, be it in interior design, art, or a professional work context. Lighting configurations can have various effects on the surroundings, for example, creating different moods in an interior context or generating a good or bad display of objects in art. Tamashii is a rendering framework developed by the Rendering and Modeling Group at TU Wien and offers automated lighting design with extensive user interaction [LHEN$^+$24]. The user can load a scene or an object into the framework and set some lighting requirements in various forms. The lighting requirements are represented by one or more light sources, characterised by their type of light source, position, intensity or light colour. They define the optimal illumination of certain objects, like desks in an office space or illumination of an art object in a certain way. What Tamashii does is determine the best lighting position in order to match the desired (target) illumination. To achieve this task, Tamashii performs a local optimisation, which is the leverage point for this work. Our goal is to extend the existing framework by broadening the optimisation from a local one to a global one. So far, Tamashii relies on one initial starting point for performing the optimisation task and returns the first found solution. We want to be more flexible and do not want to give any input parameters as a specific starting point. We also want to determine if there is a global optimum to be found, which we will accomplish by applying different approaches in our experiment to figure out the best possible extension for the Tamashii framework.

Global optimisation is a challenging task as it is impossible or computationally infeasible to certify that a global optimum is found in an unknown problem. A plethora of heuristics have been developed over time to perform the search for a global optimum. We will first review a handful of approaches, which we will later employ in our experiments, and give an overview of pitfalls and problems that come with them. Every optimisation task is heavily problem-dependent, so we want to examine if we can find an algorithm that would work particularly well in combination with Tamashii. Since Tamashii solves the illumination requirements for several different applications, we do have to be careful not to find a solution too closely linked to one specific problem. To get a better general picture, we first want to examine the algorithms in an isolated environment with different test functions, which we pick on the basis of the inherent challenges they bear for optimisation. By doing this, we can already see if there are specific problems to focus on.

Machine learning is taking up more and more space in the world of informatics. Utilising the power of machine learning offers a wide range of applications that can also be found in the context of optimisation. We want to work out if machine learning provides an applicable extension to the Tamashii framework. Performing the optimisation task in the framework directly is a time-consuming step and limits user interaction at the time being. By finding a solution with less computational expense, it could be possible to allow the users to interact with the application in real-time, for instance, repeatedly performing optimisations or changing the optimisation parameters on the go. Surrogate models are machine learning-based applications known to be fast-running approximations of problems that are costly to evaluate. We want to build surrogate models to test if these could be the extensions we seek. As one of Tamashii's main accomplishments is the efficient computation of gradients by implementing an adjoint method, we want to utilise this given information as well. Hence, we are selecting two gradient-enhanced implementations for surrogate modelling on which to run our experiments.

We will now first go into details of the global optimisation background in chapter 2.1, containing different optimisation algorithms as well as an overview of surrogate models, followed by more information about Tamashii in chapter 2.2. In chapter 3, the problem is defined in a formal way, and we explain how exactly the chosen algorithms are applied in our experiments. We then describe our two experiments in chapter 4, the first being done in a test function environment, the second one on Tamashii. In chapter Results 5, we elaborate on our findings and the problems we encountered during the experiments and will conclude with a summary and outlook in the end.

CHAPTER 2

Related Work

2.1 Local and Global Optimisation

In this chapter, we want to cover optimisation in general and related work in this field. First, we want to start by giving a brief overview of what optimisation is, of different approaches, and recurring problems. We then want to go into more detail about our applied optimisation methods. Being one of the most popular methods, it seems to be natural to start with Gradient Descent. We then continue with Adaptive moment estimation (Adam) [KB14], another first-order optimisation algorithm. To show a different non-gradient-based idea in optimisation, we include Simulated Annealing and finally conclude with learning-based optimisation using surrogate models of the Surrogate Modelling Toolbox (SMT) [SLB$^+$24], in particular, Gradient Enhanced Neural Network (GENN) [Ber24] and Gradient Enhanced Kriging with Partial Least Squares (GEKPLS) [BM19].

The overall goal in optimisation is to find the best solution for a given problem, usually the problem's minimum or the maximum. The basis of all is optimisation in a mathematical context, where it simply denotes finding a function's minimum or maximum. We could also apply optimisation in a computational context in engineering, which would mean minimising runtime or necessary resources. Finding the optimal efficiency or quality would be the goal in a business context. All of these requirements need to be interpreted in a function in order to perform some way of minimising or maximising. This function, called cost or objective function, provides a way to quantitatively measure the output. By the nature of every problem being fundamentally different, it becomes obvious how much the objective function and the applied methods depend on the desired outcome.

Weise et al. [WZCN09] and Weise [Wei09] give a detailed overview of common pitfalls in optimisation and cover several well-known methods. Overall, optimisation methods can be divided into two categories, deterministic and probabilistic algorithms [Wei09].

Deterministic algorithms efficiently explore the so-called search space to find the right solution, while probabilistic algorithms are not that obvious, so they need heuristics in order to find suitable candidates. Heuristics are tools to find approximate solutions for a problem that don't guarantee to be the best-existing one but work well enough in that given context.

In optimisation, we also want to differentiate between local and global optimisation. Local optimisation stops once it converges into the first possible solution that fulfils all criteria for local extrema. This solution is accepted, although it could very well only be a local optimum with a better solution (the global optimum) not being found or, in the worst case, even a saddle point. Conversely, global optimisation aims to find the globally best solution in a restricted search space. To overcome local optima, global optimisation algorithms often incorporate the concept of iterative restarting at mostly randomly chosen positions to eliminate the risk of converging in the wrong region. This is presumably the biggest issue in optimisation and often happens if the problem is multimodal, meaning it consists of several optima. Although this seems to be a recurring issue, several others exist, as covered by Weise et al. [WZCN09] and visualised in Figure 2.1.

Premature termination can be interpreted in two different ways in terms of global optimisation. It could mean the just-mentioned effect of an algorithm accidentally converging towards a local optimum instead of a global one or not reaching the optimum due to a chosen termination criterion in the algorithm configuration. Another challenge is ruggedness or deceptiveness [WZCN09]. Ruggedness happens when a function is multimodal and, in addition to that, has very steep ascends and descends, making it difficult for an algorithm to overcome these bumps. Deceptiveness happens when the function's gradients lead away from a global optimum towards a local one. Plateaus or so-called neutrality are especially hard to overcome for gradient-based optimisation as, in this case, no helpful gradient can be found in any direction. Noise also complicates the computation of the next direction. In machine learning-based optimisation, the problematic part is to train the model accordingly to avoid the two extremes of an inadequate model: overfitting and oversimplification. Overfitting results in an overly complicated model in an attempt to be as accurate as possible. Oversimplification is the opposite, where the training of a model terminates at a state where the built model represents the input data but is not detailed enough to match the actual problem.

As individual as these problem-specific challenges are the eligible algorithms. Choosing the right one heavily depends on the goal, whether efficiency, accuracy, or computational effort are needed. The No Free Lunch Theorem [WM97] addresses exactly this problem. Wolpert and Macready state that the sum over performances of all objective functions is always the same over all optimisation methods. This means that for a certain problem, a well-working, high-performative optimisation method exists, whilst the same algorithm may eventually find a solution to another problem, but it will require much more time and computational effort. The takeaway for us is to focus not only on one single optimisation approach but also compare and study others in order to find the best match.

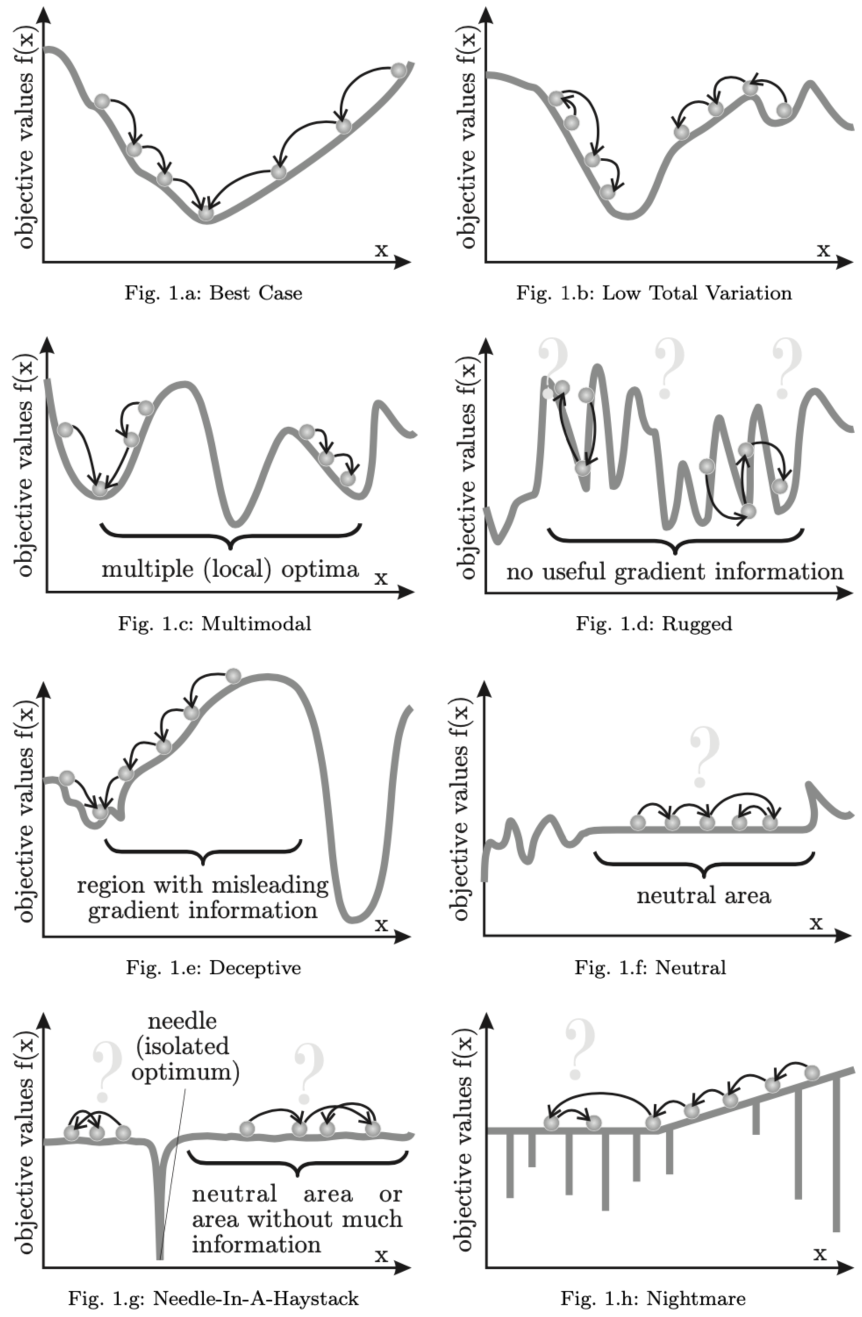
objective values f(x)
x
Fig. 1.a: Best Case
Fig. 1.b: Low Total Variation
multiple (local) optima
Fig. 1.c: Multimodal
no useful gradient information
Fig. 1.d: Rugged
region with misleading gradient information
Fig. 1.e: Deceptive
neutral area
Fig. 1.f: Neutral
needle (isolated optimum)
neutral area or area without much information
Fig. 1.g: Needle-In-A-Haystack
Fig. 1.h: Nightmare

Figure 2.1: A visual representation of various optimisation problem types encountered in machine learning and numerical optimisation [WZCN09].

2.1.1 Gradient Descent

Gradient Descent (GD) is an efficient and straightforward to realise optimisation algorithm. It is one of the most widely used methods and can be extended to Stochastic Gradient Descent as well. Stochastic Gradient Descent (SGD) is particularly popular for optimising deep neural networks [Lu22]. Gradient Descent's core is finding a minimum on a differentiable function, which can be convex and non-convex. As previously mentioned, in the context of an optimisation task, this function is called cost or objective function. A gradient is a vector pointing in the direction of the steepest increase, therefore following this vector ultimately leads to a (possibly local) optimum. If the gradient is computed using an adjoint method as in Tamashii [LHEN+24] or in backpropagation in Neural Network training [RHW86], then the computational complexity of getting the gradient is equivalent to the complexity of evaluating the function itself.

Gradient Descent is performed on a fixed number of iteration steps. In each iteration, a step of a specific length is added to the current position in a certain direction, which is determined via the direction of the gradient vector. The step size, often called the learning rate, can either be constant or can be adapted over time. A small and constant learning rate leads to slow progress and bears the risk of premature termination once the number of iterations is reached. A larger but still constant learning rate can lead to an optimum faster, but the algorithm often oscillates back and forth as the step size is too large to reach the actual region. The solution is using an adaptive learning rate, for which several approaches can be used. Exponential decay is the one most commonly used as it results in relatively large learning rates in the beginning, leading to quick progress and then quickly slowing down, therefore avoiding the risk of oscillation. Nevertheless, Gradient Descent's worst enemy is being trapped at a local minimum, which can only be overcome by restarting the entire process from a new location.

Stochastic Gradient Descent follows the same structure as Gradient Descent but employs a different computation of the gradient data. Instead of performing a function evaluation, the gradient is replaced by a random variable that only equals the gradient in expectation [SSBD14, Section 14.3]. By this, some random movement is introduced to avoid following the strict path of a standard Gradient Descent and potentially escape local minima. In machine learning, this often corresponds to mini-batch Gradient Descent [Lu22], where instead of calculating the gradient using the full dataset, it is computed using a small randomly selected subset or *mini-batch* of the data. Through this random subset, some variability is introduced, and the calculation is sped up.

2.1.2 Adam

Adam is another optimisation method utilising gradients for optimal performance. Adam stands for *adaptive moment estimation* [KB14] and is a combination of two previously existing methods. It combines AdaGrad [DHS11], an algorithm with good performance on sparse data, and RMSProp [TH12], efficiently working on non-stationary settings, like adaptive learning rates or reinforcement learning. What differentiates Adam from a regular Gradient Descent is the adaptation of the learning rate based on estimates of the gradients' first and second-order moments. Moments are statistical measurements; the first-order moment refers to the mean, the running average, of the gradients standing for the general direction at the time. The second-order moment is the uncentered variance representing the magnitude of the gradients. Including the first and second-order moment makes Adam especially suitable for noisy or sparse gradients, where the gradient value is mostly zero or close to zero except during certain steps. This algorithm requires little memory and is very efficient for optimisation over a large dataset or high-dimensional parameters. The speed of the learning rate adaptation can be controlled via hyperparameters. The convergence is generally faster than that of a regular Stochastic Gradient Descent, making Adam an efficient tool for training in machine learning. The reliable convergence is also due to an included bias-regulation factor. When starting the iterations, the first-and second-order moments are initialised to zero, as they are running averages, therefore not correctly representing the exact state at the starting position. With the help of the bias correction term, Adam ensures that the steps of the first iterations are computed correctly as well.

2.1.3 Simulated Annealing

Heuristics are a common way to solve complex optimisation methods with smaller computational requirements. They are problem-specific and need to be chosen according to the desired outcome. Two main categories separate the heuristic algorithms, as stated by Kirkpatrik et al. [KGJV83]. The first one is the idea of divide-and-conquer, in which a problem is split into smaller subproblems, which are consecutively solved, and the solution afterwards joined to an overall solution. Inherently, this category requires that a problem can be treated as smaller disjoint subproblems in the first place. The process of combining sub-solutions comes with small errors, which in total should not outweigh the positive effect of the benefit of the computational reduction. Alternatively, iterative approaches treat problems in their entirety. They start at a known configuration, iteratively rearranging the parameters until no further improvement can be accomplished. Accepting only steps towards either the descending or ascending direction, depending on whether a minimum or a maximum is searched for, results in a fast convergence but does not consistently deliver reliable solutions, as the found one does not have to be the globally best one. Simulated Annealing (SA) is one of the most successful optimisation algorithms that will also, from time to time, consider steps toward the other direction leading away from the seemingly right path. This makes it less prone to getting stuck at a local optimum. It does not take any gradient information into account [KGJV83], so this follows a completely different idea than the previously discussed methods.

Annealing stems from material science and refers to the process of heating up metal and then slowly cooling it down again with the aim of improving its properties. This process is described in detail by Weise [Wei09]. Small defects in metal originating from dislocated ions reduce the overall hardness of the material. By heating the metal, the energy and, thus, movement of the ions is increased, and their dislocation can be dissolved.

Slowly cooling the metal down allows the ions to rearrange their structure and find their equilibrium state.

The Metropolis Algorithm [MRR+53] is a direct translation of this process. It provides a probabilistic way to calculate whether a solution is accepted as the new state, laying the foundation for the Simulated Annealing algorithm. Simulated Annealing extends the Metropolis Algorithm by implementing a temperature annealing schedule controlled by input parameters. At each iteration, the algorithm determines which direction to go next. Even solutions with a higher energy will be accepted with a certain probability, giving the opportunity to possibly escape local regions.

Simulated Annealing is a reasonably reliable optimisation algorithm as it combines both categories of heuristic techniques. It performs an adaptive form of divide-and-conquer in addition to iteratively exploring the search space. In this case, divide-and-conquer is realised such that higher temperatures lead to a bigger probability of steps with higher energy than the current being accepted. As the temperature cools, so does the probability of accepting new positions with a higher temperature, slowly only leading towards lower energy and the convergence towards the optimum.

2.1.4 Surrogate Models

Surrogate Models, sometimes called meta models or response surfaces [BM19], are becoming increasingly interesting in the field of optimisation. They are fast-running approximations of problems that are too costly to be evaluated several times in order to perform an optimisation process. With the increasing complexity of many engineering problems, a solution is needed that requires less computational effort to solve them. The use cases for surrogate models are widespread and could be design-space exploration, uncertainty quantification, or optimisation [SLB+24]. The basic idea behind using a surrogate model is that it offers an increase in runtime performance and accuracy, which is most of the time contradicting. Forrester et al. [FSK08] give a detailed overview of what is necessary to set up a model and different ways to explore it (Figure 2.2). The construction of the surrogate models starts after deciding on a suitable sampling plan, for instance, picking random points, setting up a regular grid, or using Latinsquare Hypercube Sampling (LHS). The sampled data is fed into the model, and model-specific parameters are set to determine characteristics like the depth of the model, the number of training epochs, etc. Finally, building the model depends on the chosen form of approximation, which could be kriging, quadratic interpolation, or least squares regression, to name a few options [SLB+24].

Exploring and exploiting a surrogate model yields similar problems to performing an optimisation process on a real function. They often consist of many local optima and, therefore, can be deceiving to the optimisation algorithms. Sampling the training data for the models can be a quite costly step, therefore the setup often starts with as little data as possible. To later improve the model, we have the option to add infill points. These points are usually in areas of interest that appear during an optimisation iteration.

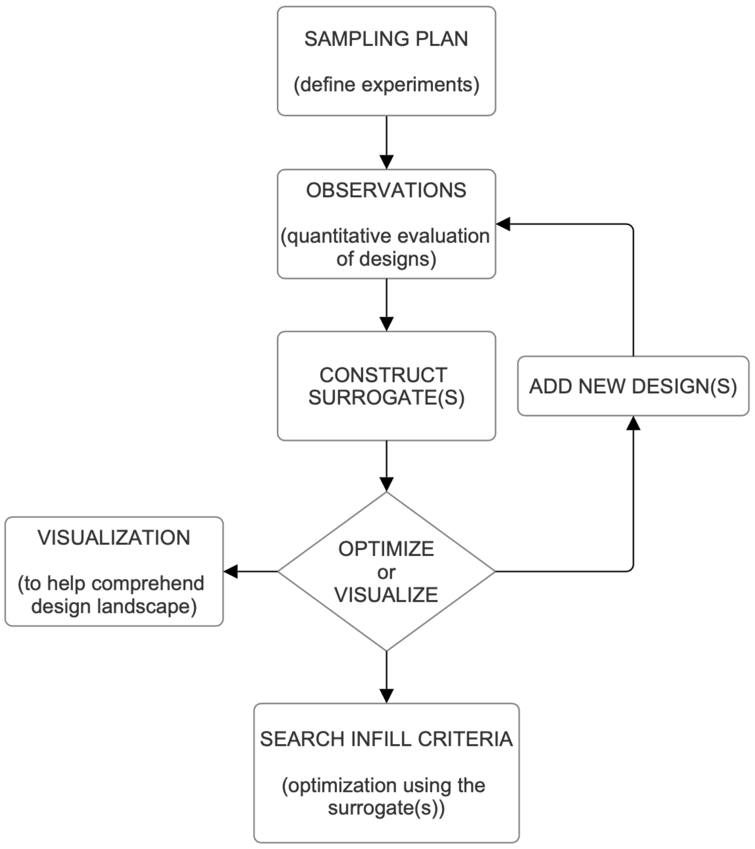

Figure 2.2: Illustration of the process of surrogate modelling, often employed in design optimisation workflows. The steps show an iterative approach of creating and recreating a model based on new designs [FSK08].

Forrester et al. [FSK08] give useful examples of where to look for further points to add to the model data and how they could influence the model. One method they call prediction-based exploration is where new points are added where the optimisation algorithms converge, and an optimum on the current model is found. The found point is evaluated in the real problem and fed back into the model. Another method could be error-based exploration, where the area to add more points is selected where the uncertainty of the current predictions is the highest. It is also important to determine a termination criterion when to stop adding additional points. This could be done either if the model is saturated, meaning no further improvement can be achieved, or by using other validation techniques. After all, one must always weigh the pros and cons of how many sample points are used to train a surrogate model to balance the computational cost of setting it up with the benefit of receiving a well-functioning, quick-to-evaluate model. This decision applies to both the initial training data as well as the later added infill points.

They also give useful advice on how a surrogate model could be visualised. When a model's size exceeds three dimensions, most humans struggle to imagine the model structure. Furthermore, visualisation helps understand the problem on a deeper level by seeing model characteristics and areas of interest. They show an example of visualisation in the form of slices through the model in combination with contour lines so that even the inner parts of the model are visible, a feature we want to incorporate into our implementation as well.

Surrogate Modeling Toolbox (SMT) [SLB+24] is an open-source Python package to build various kinds of surrogate models and is the main source for our work. Therefore, we want to go into detail about what the background of this work is and what makes it noteworthy. The current version of SMT is an improvement of an already existing one, implementing significant extensions to established features complemented by new ones. The main feature separating SMT from other machine learning tools is the support of gradients, both for training and predictions. There are several popular machine learning packages for different programming languages, like Python or Matlab. The most popular would be skiKit-learn [PVG+18], PyTorch [PGM+19] and TensorFlow [AAB+16] for Python, or SUMO [GCD+10] and GPML [RBLR04] for Matlab. What unites them is that none of them focus on taking gradients into account. SMT, however, developed several surrogate model methods with derivatives like GENN, short for Gradient-Enhanced Neural Network, or GEKPLS, short for Gradient-Enhanced Kriging with Partial Least Squares, covered in detail in the following subsections. The main benefit of extending a surrogate model with derivatives is the significant improvement in accuracy with fewer training points. Since retrieving training points is one of the major time-consuming parts, this is a relevant improvement in the area of surrogate model application. Furthermore, they provide sampling plans to import some of the previously mentioned methods as pre-implemented libraries, test functions for the models, or ways of updating and retraining the model.

GENN

A Gradient-Enhanced Neural Network, short GENN, is a fully connected multilayer perceptron [SMT22b]. It is a surrogate modelling technique that combines neural networks with gradient information for improved accuracy and the ability to predict first derivatives as well. This feature makes GENN particularly useful for optimisation algorithms utilising gradient information for improved performance.

Its implementation is based on the Jacobian Enhanced Neural Network (JENN) library [Ber24]. Jacobian is a similar concept as a gradient, with the difference that it accounts for a vector-valued function with more than one output instead of a scalar-valued function, which would be the case for GENN. Berguin developed JENN and performed interesting test cases to show the significantly improved performance compared to regular neural networks. JENN is a more complex version of GENN and offers more functionality to fine-tune the model. While GENN gives the option to retrain a model with updated training sets, JENN also makes weighing individual sample points possible in order to focus on specific regions, called *polishing*. The reasoning behind that is the following:

During training, the model's goal is to minimise its prediction error. Regions with larger changes inevitably result in higher penalties for wrong predictions, while flat regions like valleys with changes close to zero only produce smaller prediction errors. Unfortunately, these sort of flatter regions are often regions in close proximity to a local optimum, thus of high interest for optimisation, but don't receive much attention during the learning process, resulting in higher inaccuracies. The concept of polishing provides the option to pay more attention to these areas by giving them a higher weight and, thus, a higher importance for correct predictions.

Berguin uses the Rosenbrock function [Ros60] to test his implementation in comparison with the true response of the function, a regular neural network, and a polished version of his model. The Rosenbrock function is a challenging test case for surrogate model construction, as it consists of a shallow valley where the gradients are close to zero, being the perfect fit to test the polishing method. Berguin shows that in the first iterations, neither the neural networks nor JENN find the true minimum of the Rosenbrock function, but after applying polishing, JENN is able to find it. Generally, JENN outperforms regular neural networks in all of his test cases, proving that including derivatives results in a measurable performance improvement.

GEKPLS

GEKPLS stands for Gradient Enhanced Kriging with Partial Least Squares. Kriging is one of the most used interpolation models, also known as Gaussian Process Regression [BM19]. It is one of the new methods featured in the updated SMT library [SMT22a].

The original version of kriging was developed by Matheron [Mat63]. For a long time, kriging was not overly useful for machine learning, as a high number of function evaluations was necessary to build an accurate model. As for other surrogate models, using gradient information holds the opportunity to improve the model's accuracy. However, with Kriging, gradients are not easily included since adding more data immensely affects the size of the correlation matrix used for the prediction process. For each sample point, the corresponding information must be added to the correlation matrix in every direction of the design space. The matrix thereby increases proportionally to the number of input data and samples [BM19]. Many close sampling points produce a different issue and lead to a *quasi-linearly-dependent* matrix, hindering solving the problem. Many independent variables also used to be a performance problem due to the number of hyperparameters that must be estimated in the training process.

Following this work, Bouhlel et al. [BBOM16] developed Kriging with Partial Least Squares (KPLS), which results in a quicker construction process of the kriging model. Partial Least Squares is a statistical method used to find the relation between two matrices. Reducing the dimension of the data and using only the relevant information is the key factor helping to improve the size of the correlation matrix. Sample points in close proximity are checked for their relevance, and only the points with the highest influence will be added. Due to this selection, producing an ill-conditioned, quasi-linearly-dependent matrix can be avoided, and the memory requirement can be kept down as well. Bouhlel et al. show in their work that including Partial Least Squares improves the model's accuracy and runtime performance up to 450 times.

Finally, by combining GEK (Gradient-Enhanced Kriging) and KPLS (Kriging with Partial Least Squares), Bouhlel and Martins develop GEKPLS. This new method utilises the improvement in model accuracy by including gradient information while still benefiting from the performance improvement and controlling the size of the correlation matrix and number of hyperparameters by using Partial Least Squares.

2.2 Tamashii

2.2.1 What is Tamashii

Tamashii is a rendering framework developed by the Rendering and Modeling Group at TU Wien. Its purpose is to realise automated lighting design across various fields of application. In their work, Lipp et al. [LHEN$^+$24] implemented a novel method for global illumination in an interactive and view-independent format. This method's goal is to perform optimisation algorithms on light parameters to match a lighting target represented in different forms, for example, as a pre-computed, baked illumination map, lighting requirements in interior designs, or artistic sketches. They provide an interactive user interface where the user is not only able to preselect the desired lighting parameters to be optimised or draw the target directly into the scene but also to follow and adapt the optimisation process. The optimisation is supported by an innovative way to compute the global radiance data, the according objective function between the current state and the target state, as well as the corresponding gradients. These gradients are computed using an adjoint method in a backward pass, where rays are traced from the image back into the scene to determine how changes in scene parameters affect the rendered result. Leveraging GPU-based raytracing makes efficient interaction in the user interface possible. Although there are existing lighting design tools [DIA22, Rel22], they don't offer the same range of automatisation and user interaction as Tamashii.

There is other relevant research in this area that intersects with the goals of Tamashii. Radiative backpropagation [NDSRJ20], for instance, is a method by Nimier-David et al. exploring a similar direction by investigating differentiable simulation of light but in view-dependent path tracing. Additionally, other work on differential rendering [LHJ19] and differentiable path tracing, such as Mitsuba2 [NDVZJ19], contribute to this field. Still, none address the challenge of view-independent differentiable global illumination, making Tamashii a standalone application at the time.

Differentiable rendering is one of the key concepts in the work of Lipp et al. The term refers to the process of computing derivatives of the rendered image or an objective function that evaluates the image with respect to scene parameters. Most of the time, it is used to optimise physical properties like materials or geometry to match a target output. In contrast to inverse rendering, which aims to find parameters that reproduce a given target, differential rendering includes a more efficient gradient-based optimisation utilising the computed derivatives of the rendered image or objective function. In Tamashii, however, the focus is not on optimising physical properties but on matching the scene illumination to the desired target illumination. The user can select which lighting parameters in particular they want to optimise, for instance, the light's position, the colour, or the intensity. All of these parameters or just a subset can be selected in the graphical user interface and then exported into the code.

Traditional rendering methods in computer graphics can be classified into forward and backward rendering approaches. Path tracing [Kaj86] and ray tracing [Whi80] are examples of backward rendering methods and are view-dependent, which means they trace rays starting from the camera to determine the visibility of the scene's objects or lighting. Therefore, they rely on the camera position to calculate the rendered scene. However, these view-dependent rendering techniques are inefficient for differential rendering of complex scenes as the objective function would need to be calculated from many different camera angles, making the process very time-consuming.

Forward rendering methods, on the contrary, typically trace rays starting from the light sources to simulate how the light interacts with the scene and are view-independent. Being view-independent allows for the calculation of the global illumination of more complex scenarios independently from the current camera position by following the rays leaving the scene's lighting objects in a forward approach. Each ray is traced up to a predetermined number of ray-surface-intersections, where the contribution to the global radiance data is updated. The gradients are computed in a backward pass employing an adjoint method during this process. A backward pass in the context of rendering means tracing the ray backwards into the scene to determine how changes in settings, like the light position, affect the overall scene. It is often used in combination with an adjoint method, as also done in Tamashii and is responsible for significant performance acceleration. In a traditional forward gradient computation, the gradient is calculated separately for each optimisation parameter, which can be quite time-consuming for several parameters. In contrast to that, an adjoint gradient computation is able to compute the gradient for all parameters in a single reverse mode, resulting in a much better performance.

The provided gradient data can then be visualised via the Tamashii user interface as well to help the user gain deeper insight into the problem. The objective function or so-called cost function of the calculated radiance data and corresponding gradients can be used for an optimisation task, which is a crucial step that enables the use of first-order optimisation algorithms and is the key feature that our work is based on.

2.2.2 Python Extension

Tamashii's source code is written in C++, an object-oriented programming language often used in computer graphics. One of the main advantages of C++ is that programmers have direct access to resources and can freely allocate and deallocate needed memory.

While this can be very beneficial, it makes the code quite complicated and lengthy at the same time. On the other hand, higher-level scripting languages like Python provide other options. Their higher abstraction level makes the code more intuitive and easier to use but comes at the cost of performance, mainly due to the interpreter overhead. Nevertheless, a vast amount of applications are implemented using Python, especially in the field of machine learning. As machine learning and optimisation go hand in hand, it is highly relevant to bridge the gap between these two programming languages and make Tamashii extendable with machine learning libraries. Preymann [Pre23] solved this by adding a scripting automation layer for Tamashii. He implemented the interface to access and manipulate relevant data from the C++ code and incorporate it into the new Python functionality. Being able to access the light objects in a scene directly and retrieving the calculated gradients and cost function opens the door to performing optimisation outside of the Tamashii framework and in a Python environment. Using Nanobind, he created custom bindings for the Interactive Adjoint Light Tracer (IALT), the core module needed for our work. The integration process and generation of build files involves the build system CMake. Following that, building the project and creating all necessary Python files can be done via Visual Studio.

We have now given an overview of global optimisation in general as well as some selected optimisation algorithms, which we identified as useful to apply in our context. We also gave some insight into surrogate modelling and presented the two models in detail that we will employ ourselves. We will now continue with a formal problem definition and how exactly these algorithms and models are being used in our experiments.

CHAPTER 3

Background

In Tamashii, the goal is to match a scene's lighting to a design-space-dependent target lighting (Figure 3.1). Therefore, the scene is initialised with a default position for one or more lights, and the designated target illumination is set. The optimisation parameters, the loss values, and gradients are provided by the Tamashii implementation and can be accessed via the Python - C++ interface. In our case, the scene contains one light source, so we start with an initial position for the corresponding object and a given target position to be found by the optimisation algorithm.

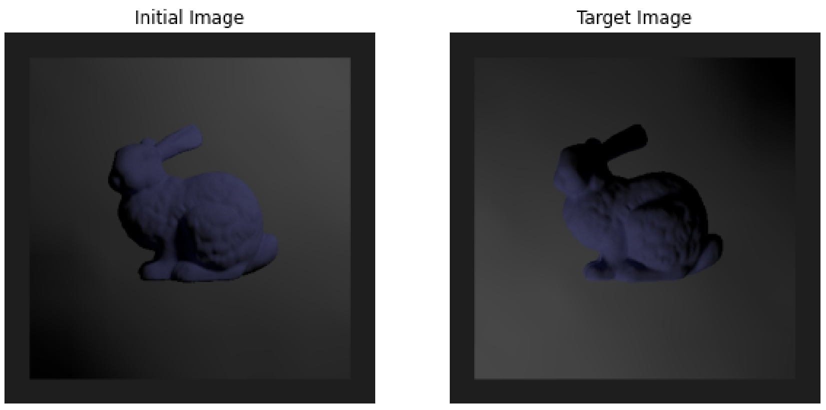

Figure 3.1: Rendered rabbit scene with the initial light position (left) and the target light position (right).

In the following, we will give a more formal overview of our optimisation problem based on the original problem definition and adjoint gradient computation in Tamashii [LHEN$^+$24].

3.1 Problem Definition

Lighting Position:

Let $\mathbf{p} = (x, y, z) \in \Omega \subset \mathbb{R}^3$ represent the 3D position of a light in the scene, where x, y, and z are the coordinates of the light in the scene and the parameters to be optimised. Here, Ω is the surface of the scene.

Lighting Function:

Let $L : \Omega \times \mathbb{R}^3 \to \mathbb{R}$, $(\mathbf{p}, \omega) \mapsto L(\mathbf{p}, \omega)$ denote the *radiance field*, representing the illumination on the scene's surface depending on the light position $\mathbf{p}$ and the outwards direction ω.

Target Illumination:

Let $L^* : \Omega \times \mathbb{R}^3 \to \mathbb{R}$ represent the *target illumination*, the desired illumination at various points in the scene, with the light object located at the target position.

Loss Function:

The loss or objective function is the squared difference between the radiance field and the target illumination across the scene, i.e.,

$$\mathcal{L}(\mathbf{p}) = \frac{1}{2} \int_{\Omega} \frac{1}{2\pi} \int_{H^2} \alpha(\mathbf{x}, \omega) \left(L(\mathbf{x}, \omega) - L^*(\mathbf{x}, \omega) \right)^2 \, d\omega \, d\mathbf{x},$$

where:

- L^* represents the target illumination,
- Ω is the scene's surface,
- H^2 denotes the hemisphere of outgoing light directions,
- $\mathbf{x}$ represents points on the scene's surface,
- ω is the direction of the radiance,
- α is a weighting function that can prioritise certain parts of the scene or lighting directions.

Optimisation Problem:

The optimisation problem is to find an optimal light position $\mathbf{p}^*$ that minimises the loss function $\mathcal{L}(\mathbf{p})$:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \mathcal{L}(\mathbf{p})$$

Adjoint Method for Gradient Computation:

Tamashii employs an adjoint method to compute the gradient of the objective function $\mathcal{L}(\mathbf{p})$ with respect to the lighting position $\mathbf{p}$. The exact calculation can be found in the work of Lipp et al. [LHEN+24]. This method is called *adjoint method*, as it computes the gradient for the objective function in a single reverse pass instead of needing to calculate the gradient for each individual parameter separately in forward mode. It takes the overall changes of the scene illumination, which is a contribution of all optimisation parameters simultaneously and propagates the changes backwards, starting from the loss function following the negative direction of the light traces. The name *adjoint* stems from the *adjoint operators* used for this computation, being the transpose of the operators applied in the forward computation. We, therefore, only need one forward pass to compute the overall scene illumination and one backward pass to compute the corresponding gradients.

General Optimisation Process:

To find $\mathbf{p}^*$, the light position $\mathbf{p}$ is updated iteratively using an update rule, which depends on the applied optimisation algorithm. Concretely, for $\mathbf{i} \in \mathbf{N}$

$$\mathbf{p}_{i+1} = \mathcal{U}(\mathbf{p}_i, \mathcal{L}, i)$$

where:

- $\mathbf{p}_i$ is the current light position at iteration i,
- $\mathcal{L}$ is the loss function,
- $\mathcal{U}$ is the algorithm's update function,
- i refers to the current iteration index.

As we have elaborated in the previous chapters, performing an optimisation is very problem-dependent and does not have a one-fits-all solution. To find the most suitable algorithm, we set up a range of experiments. These experiments aim to find out if a certain algorithm outperforms others in our problem setting from various viewpoints, which are classified as finding the target position, reliability in finding a good solution, or runtime performance. We want to compare the classic Gradient Descent with a version of a Stochastic Gradient Descent, Adam, and Simulated Annealing. As Tamashii enables the use of gradients for optimisation, the first three are the most obvious choices. Nonetheless, we also want to experiment with a non-gradient-based algorithm to see if the use of gradients is really beneficial in this application. We experiment with different settings for our algorithms, such as starting from one predetermined position or restarting from several random starting points to circumvent the problem of premature termination in a local minimum. We also experiment with different parameter settings for algorithm-specific variables like learning rates, iterations per optimisation run, or cooling schedules for Simulated Annealing.

Evaluating the loss value of Tamashii is quite a time-consuming step, so Tamashii would be one of the textbook applications for the use of surrogate modelling. We want to explore different options for surrogate modelling to find out if a beneficial application exists for our problem. We decide to explore the already introduced GENN and GEKPLS models since they are both gradient-enhanced surrogate models and can make use of the provided gradient data in Tamashii. As one option in the new SMT2.0 version is to update the original training data of the surrogate models, we test if iterative training would improve our model accuracy. This relearning is done in several iterations, and the outcome is compared to the initial surrogate model. Additionally, certain parameters are to be set when initialising a surrogate model, so we are testing the impact of adjusting these as well.

Since Tamashii offers an interface where several scenes can be uploaded and optimised on, we do not want to find a solution too specific to one designated design space. We, therefore, perform some basic experiments on test functions first to acquire a deeper insight into the optimisation algorithms and surrogate models before applying the knowledge we have gained to Tamashii. As test functions, we choose the Rosenbrock function [Ros60], a sine function combined with a parabola to create a unique global minimum and the same sine function with added noise.

T1 (Rosenbrock Function): The function `rosenbrock(x, y, b)` is defined as:

$$f(x, y) = b \cdot (y - x^2)^2 + (x - 1)^2, \quad \text{for } x, y \in \mathbb{R}$$

where $b \in \mathbb{R}$ is a constant with a default value of 10.

T2 (Sine Function): The function `modified_sine(x, y)` is defined as:

$$f(x, y) = \sin(x) + 0.1 \cdot y^2, \quad \text{for } x, y \in \mathbb{R}$$

T3 (Sine with Noise Function): The function `modified_sine_noise(x, y,` δ`)` adds noise in the form of another sine wave to the previous sine function. It is defined as:

$$f(x, y, \delta) = \sin(x) + 0.1 \cdot y^2 + \delta \cdot \sin(100 \cdot (x + y)), \quad \text{for } x, y \in \mathbb{R}$$

where $\delta > 0$ defines the amplitude of the added sine noise.

3.2 Methods

We will now go further into detail on how we applied the algorithms to the test functions, the surrogate models, and Tamashii and explain our implementation of restarting and relearning.

3.2.1 Gradient Descent

The Gradient Descent Algorithm (GD) has a straightforward implementation and does not require many parameter settings. We start from a single starting point containing the coordinates in the room. The number of iterations determines the number of times the algorithm will update its current position on the way towards an optimum before terminating. The learning rate denotes the step size in the update process. We also input the currently used test function or surrogate model as input and the scene's bounding box. At every iteration, the algorithm steps in the negative gradient direction from the current position. The gradient is either computed by the given test function or predicted by the surrogate model. It points in the direction of the greatest local increase, so consequently, following the opposite direction means moving locally along the greatest decrease. We test updating with different step sizes like a constant one or a step size scaled depending on the current iteration count as mentioned by Shalev-Shwartz and Ben-David [SSBD14, Section 14.4.2] and applied in the following update rule.

Gradient Descent Update Rule:

$$\mathbf{p}_{i+1} = \mathbf{p}_i - \left(\frac{\eta}{\sqrt{i+1}}\right) \cdot \nabla_{\mathbf{p}}\mathcal{L}(\mathbf{p}_i), \quad \text{for } i \in \mathbb{N}$$

where:

- $\mathbf{p}_i \in \mathbb{R}^d$ is the current point at iteration i, where $d = 2/3$ for the test function experiments or Tamashii respectively,
- $\eta > 0$ is the learning rate,
- $\nabla_{\mathbf{p}}\mathcal{L}(\mathbf{p}_i) \in \mathbb{R}^d$ is the gradient of d at $\mathbf{p}_i$.

We want to perform the optimisation within a given space, determined by each scene. We define the bounding box for the respective scene at the beginning of the experiments and use it to limit the movement of the algorithm. By calling the function `projectToBoundingBox` with the point's updated position $\mathbf{p}_{i+1}$, we can check if the new position is still within our search space or else readjust it to be just at the border of the space. Finally, after termination, the algorithm returns all traversed points with the last point being the found solution treated as the minimum.

Algorithm 1 Modified Gradient Descent

```
Function: GradientDescent(starting_point, iterations, learning_rate, function / model, boundingBox, noise_variance=0.01)

• Initialise $x$ as a 2D array of size $(\text{iterations}, \text{len}(starting\_point))$
• Set $x[0,:]$ to the $starting\_point$
• For $i = 0$ to $iterations - 2$:
    – Initialise $grad$ as a 1D array of size $\text{len}(starting\_point)$
    – If $function$ is provided:
        ∗ Compute $grad$ as gradients of the given function at $x[i,:]$
    – Else if $model$ is provided:
        ∗ For each $j$ in $gradient\ dimensions$:
            · Predict gradient $grad\_j$ by model.predict_derivatives
            · $grad[j] \leftarrow grad\_j$
    – Compute $next\_point \leftarrow x[i,:] - \left(\frac{\text{learning\_rate}}{\sqrt{i+1}}\right) \cdot [grad]$
    – $x[i+1,:] \leftarrow \text{ProjectToBoundingBox}(next\_point, boundingBox)$
• Return $x$
```

While we can compute the point's gradients for the test functions or have them predicted by the surrogate models, we have to follow a different approach for optimisation on a Tamashii scene. The overall algorithmic structure stays the same, but the computation of the gradients and the update process is slightly different, which is not covered in the given pseudocode for the purpose of a better overview. We want to give a quick summary here to then only refer to it briefly in the following subsections.

In Tamashii, the current position is represented by the exported optimisation parameters, which can be accessed by the function:

$$parameters = scene.lightsToParameterVector()$$

To receive the corresponding value, which in the case of Tamashii is the loss value and the gradients for the current parameter settings, two function calls are necessary. We perform a forward and a backward pass with the parameters:

$$ialt.forward(parameters)$$

$$loss, gradients = ialt.backward()$$

IALT is the *Interactive Adjoint Light Tracing* module implemented in Tamashii and is responsible for the gradient and loss calculation. Instead of updating the current point's position, we are updating the parameters to move along the negative gradient direction with the same update rule as for the test functions and surrogate models.

3.2.2 Stochastic Gradient Descent

In our interpretation of a Stochastic Gradient Descent (SGD) algorithm, we extend the classic Gradient Descent with some stochastic noise. After calculating the gradient, we add some Gaussian noise between 0 and 1 with a standard deviation given as an input variable called *noise_variance* following the classical definition of SGD as described by Shalev-Shwartz and Ben-David [SSBD14]. By this, we avoid following the strict path down the negative gradient at each point and include some unpredictable movement. By doing this, we hope to escape local minima and find a more reliable result [SSBD14].

Algorithm 2 Stochastic Gradient Descent

```
Function: StochasticGradientDescent(starting_point, iterations, learning_rate, function / model, boundingBox, noise_variance=0.01)

• Initialise $x$ as a 2D array of size (iterations, len(starting_point))
• Set $x[0,:]$ to $starting\_point$
• For $i = 0$ to $iterations - 2$:
    – If $function$ is provided:
        * Compute $grad$ as gradients of the given function at $x[i,:]$
    – Else if $model$ is provided:
        * For each $j$ in $gradient\ dimensions$:
            · Predict gradient $grad\_j$ by model.predict_derivatives
            · $grad[j] \leftarrow grad\_j$
    – Add stochastic noise to the gradients:
        * $grad = grad +$ random noise (Gaussian with standard deviation $noise\_variance$)
    – Compute $next\_point \leftarrow x[i,:] - \left(\frac{\text{learning\_rate}}{\sqrt{i+1}}\right) \cdot grad$
    – $x[i+1,:] \leftarrow$ ProjectToBoundingBox($next\_point, boundingBox$)
• Return $x$
```

3.2.3 Adam

In Adam, the update per step also includes the gradient but follows a more advanced structure. The gradient computation is done as in Gradient Descent, which is a direct test function evaluation in T1, T2 or T3, a prediction by the surrogate models, or a forward-backward pass in Tamashii. To update the position, we include first and second-moment estimates, being the running average of the gradients and the running average of the squared gradients, respectively. The update rule also includes a bias correction term, which aims to equalise the incorrect computation of the first and second moment estimations in the early stages of the algorithm.

Algorithm 3 Adam

```
Function: Adam(starting_point, iterations, learning_rate, function / model, boundingBox, beta1=0.9, beta2=0.999, epsilon=1e-8, noise_variance=0.01)

• Initialise $x$ as a 2D array of size $(\text{iterations}, \text{len}(starting\_point))$
• Set $x[0,:] \leftarrow$ starting_point
• Initialise $m$ as a zero array of size $\text{len}(starting\_point)$
• Initialise $v$ as a zero array of size $\text{len}(starting\_point)$
• For $i = 0$ to $iterations - 2$:
    – If $function$ is provided:
        ∗ Compute $grad$ as gradients of the given function at $x[i,:]$
    – Else if $model$ is provided:
        ∗ For each $j$ in $gradient\ dimensions$:
            · Predict gradient $grad\_j$ by model.predict_derivatives
            · $grad[j] \leftarrow grad\_j$
    – Update the biased first-moment estimate: $m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot grad$
    – Update the biased second raw moment estimate: $v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot (grad^2)$
    – Compute the bias-corrected first moment estimate: $m\_hat \leftarrow \frac{m}{1-\beta_1^i}$
    – Compute the bias-corrected second raw moment estimate: $v\_hat \leftarrow \frac{v}{1-\beta_2^i}$
    – Compute $next\_point \leftarrow x[i,:] - \frac{\text{learning\_rate} \cdot m\_hat}{\sqrt{v\_hat} + \epsilon}$
    – $x[i+1,:] \leftarrow \text{ProjectToBoundingBox}(next\_point, boundingBox)$
• Return $x$
```

Adam Update Rule:

$$\mathbf{p}_{i+1} \leftarrow \mathbf{p}_i - \frac{\eta \cdot \hat{m}}{\sqrt{\hat{v}} + \epsilon}, \quad \text{for } i \in \mathbb{N}$$

where:

- $\eta > 0$ is the learning rate,
- $\hat{m} \in \mathbb{R}$ is the bias-corrected first-moment estimate in the i-th iteration,
- $\hat{v} \geq 0$ is the bias-corrected second raw moment estimate in the i-th iteration,
- $\epsilon > 0$ is a small constant added to avoid division by zero,
- $\mathbf{p}_i \in \mathbb{R}$ is the updated position at iteration i, where $d = 2/3$ for test functions or Tamashii, respectively,

3.2.4 Simulated Annealing

Simulated Annealing (SA) is a type of optimisation algorithm that does not use gradients during the computation of the next step. Instead, it finds the next point by generating a random neighbour and accepting the new position following specific acceptance rules. The neighbour is produced by taking the current position and adding some random Gaussian noise. The value of the current position and the neighbour position are then evaluated depending on the method used. The difference between the two values is computed, which is then used for the acceptance probability together with the current temperature of the cooling stage. The temperature is similar to our previously mentioned learning rate, which decays over time, allowing the algorithm to take larger steps at early stages and smaller steps at more advanced stages. The new neighbour is then accepted probabilistically depending on the current temperature, making it possible to accept worse solutions with a larger probability in the beginning and getting stricter the further the algorithm progresses.

Simulated Annealing Update Rule:

For a random neighbour point p_n, we set $p_{i+1} = p_n$ if the value at p_n is smaller than the value at p_i or a random number uniformly distributed on $[0, 1]$ is less than δ,

where:

- $\delta = \exp\left(\frac{-(\mathcal{L}(p_n) - \mathcal{L}(p_i))}{\text{temp}}\right)$,
- temp is the current temperature at the i-th iteration.

Algorithm 4 Simulated Annealing Algorithm

```
Function: SimulatedAnnealing(starting__point, iterations, initial__temp, cooling__rate, function / model, boundingBox, noise__variance=0.01)

• Initialise current__point to $starting\_point$
• If $function$ is provided:
    – Compute $current\_value$ depending on given function at $current\_point$
• Else if $model$ is provided:
    – Predict $current\_value$ by model.predict__value at $current\_point$
• Initialise temperature, $temp = initial\_temp$
• Initialise $x$ as an array of size (iterations, len(starting__point))
• Set $x[0,:]$ to $starting\_point$
• For $i = 0$ to $iterations - 2$:
    – Generate a random neighbour and project to Bounding Box:
        ∗ neighbour = current__point + random noise (Gaussian with standard deviation $noise\_variance$)
        ∗ neighbour = ProjectToBoundingBox(neighbour, boundingBox)
    – If $function$ is provided:
        ∗ Compute $neighbour\_value$ depending on given function at $neighbour$
    – Else if $model$ is provided:
        ∗ Predict $neighbour\_value$ by model.predict__value at $neighbour$
    – Compute the difference:
        $$\Delta value = neighbour\_value - current\_value$$
    – Compute the acceptance probability:
        $$acceptance\_probability = \exp\left(\frac{-\Delta value}{temp}\right)$$
    – If $\Delta value < 0$ or random number < acceptance__probability:
        ∗ Accept the neighbour: $current\_point = neighbour$, $current\_value = neighbour\_value$
    – Decrease the temperature: $temp* = cooling\_rate$
    – Set $x[i+1,:] = current\_point$
• Return $x$
```

3.2.5 Restart

To overcome getting stuck in local optima or saddle points and to find potentially better solutions, we apply the restart method to our algorithms. We hand over a list of randomly generated starting points to our function, each of which is used to start an entirely new cycle in the optimisation routine. Each time, we get all points of the iteration returned so we can compare the outcome with our previous results. For that, we compute the values for traversed points, with the last one being our currently found solution in a manner equivalent to computing the values in the algorithm, which would either be a direct function evaluation in T1-T3, a prediction by the surrogate model, or a forward-backward-pass in Tamashii. We compare the new result with the already saved solution and overwrite it with the new version if it provides an improvement. Finally, we return the overall best solution found during this procedure.

Algorithm 5 Optimisation with Restarts

```
Function: Restart(restart_points, iterations, learning_rate, boundingBox, function=None, model=None)

• Initialise empty lists: $best\_descent$, $values\_descent$, $found\_minima$, and set $path$ to $None$
• For each point in restart_points:
    – Initialise empty list: $descent$
    – If $function$ is provided:
        ∗ Perform optimisation method and return all iterates:
              $descent = \text{optimisation\_method}(point, ...)$
        ∗ Compute corresponding values to iterates:
              $values\_temp = \text{function}(descent)$
    – Else if $model$ is provided:
        ∗ perform optimisation method on surrogate model and return all iterates:
              $descent = \text{optimisation\_method\_on\_surrogate\_model}(point, ...)$
        ∗ Predict values on surrogate model:
              $values\_temp = model.predict\_values(descent)$
    – $minimum \leftarrow$ [last entry of $descent$, last entry of $value\_temp$]
    – Append minimum to found_minima
    – If last entry of $values\_temp$ is less than last entry of $values\_descent$:
        ∗ $values\_descent = values\_temp$
        ∗ $path = [descent, values\_temp]$
        ∗ $best\_descent = descent$
• Sort found_minima by the last column in descending order
• Return $best\_descent$, $values\_descent$, $sorted\_minima$, $path$
```

3.2.6 Relearning

Relearning is applied to the surrogate models to test if an iterative addition of new points would help to increase the model's accuracy. For that, we implement the following approach. We pass the current sample points, the sampled loss values, and gradients for the respective sample points to the function. Similar to setting up the initial surrogate model, we first perform a validation-test-split. We choose to use two-thirds of our data for training purposes and one-third for the validation of the model. We then split the sample points, losses, and gradients accordingly. To avoid overwriting the initial model, we create copies, one for the retraining and a temporary one for the current iteration. The relearning counter denotes the number of iterations we add new points to our data sets and train the model anew. We also pass the optimisation method to the function to determine by which algorithm the new point should be searched for.

For each iteration, we set the sample points, loss values, and gradients for our models and train them. We then perform an optimisation from a new randomly created starting point, looking for the current model's minimum. The found solution is the new sample point, and the corresponding value and the gradient are computed. The new sample point, the sample point's value, and the gradient are then combined with the data sets for the next iteration. By inserting the new data at the first position in the existing sets, we ensure that the initial base data for validation stays the same and is always a subset of the current validation split.

We calculate and save the validation error to determine whether the current model is an improvement compared to the previous versions. If the validation error improves, we save the model for the next iteration. Otherwise, the same model is used in the following iteration, where the next infill point is searched for by starting another optimisation rerun from a new randomly determined starting point. After the last iteration, we return the current version of the retrained model for further use.

Algorithm 6 Relearning for Surrogate Models on Tamashii

```
Function: Relearning(samplePoints, sampleLosses, sampleGrads, model, method, iterations, learning_rate, boundingBox, relearning_counter=10)

• Initialise new training data from 2/3-split: $newSamplePoints$, $newSampleLosses$, $newSampleGrads$
• Initialise empty lists: $descent$, $values$, $found\_minima$ and $error\_retraining$
• Create a copy of the model:
      $model\_retrained = \text{copy.deepcopy}(model)$
• For $i = 0$ to $relearning\_counter - 1$:
    – Create a temporary copy of the retrained model:
          $model\_temp = \text{copy.deepcopy}(model\_retrained)$
    – Set updated values to train the model:
        ∗ $model\_temp.set\_training\_values(newSamplePoints, newSampleLosses)$
        ∗ For each $dimension$ in $gradient\_dimensions$:
            · $model\_temp.set\_training\_derivatives$
    – Train the model
    – Generate random starting point to perform search for minimum depending on the given method and predict values
          $descent = optimisation\_method\_on\_surrogate\_model(point, ...)$
          $values\_temp = model\_temp.predict\_values(descent)$
    – Compute loss value and gradients in Tamashii with forward and backward passes:
          $ialt.forward(descent[-1, :])$
          $phi, grads = ialt.backward()$
    – Update training data:
        ∗ Combine $descent[-1, :]$ with $newSamplePoints$
        ∗ Combine $phi$ with $newSampleLosses$
        ∗ Combine $grads$ with $newSampleGrads$
    – Recalculate 2/3-split from $newSamplePoints$ size
    – Calculate validation error and append to $error\_retraining$:
          $valError = \frac{\|values\_temp - descent\|^2}{\text{samplePoints.shape}[0] - split}$
    – Save the improved model if validation error improves:
        ∗ If $valError \leq \min(error\_retraining)$:
            · $model\_retrained = model\_temp$
• Return $model\_retrained$, $newSamplePoints$, $values$, $error\_retraining$
```

CHAPTER 4

Experiments

4.1 Test Function Experiments

As was already stated by Berguin [Ber24], the Rosenbrock function is a suitable test function for both surrogate models and optimisation in general. Consequently, we choose this function to train our surrogate models and compare the optimisation results on the models with direct application on the real functions. Our first experiments compare the classic Gradient Descent with Adam and Simulated Annealing. We test these algorithms on three functions: Rosenbrock (T1), sine (T2), and sine with added noise (T3). These test functions are also used to train a GENN and a GEKPLS model.

For each optimisation method, we perform two tests. We first start the method with a fixed starting point and perform one iteration only. We then restart the same method ten times, each time with a new starting point, and save the best solution. The restart starting points are randomly generated at the beginning of the experiment, and the same set of points is then used for all algorithms. This way, we can ensure that the conditions are similar for all algorithms throughout the entire experiment.

For the surrogate models, we examine two different approaches. The first one is setting up a model with sampled data once and then applying the optimisation; the second one is stepwise retraining it to improve its accuracy. The retrained model is tested by performing an optimisation again and comparing the output.

4.1.1 Logarithm Mapping

During the surrogate model setup, we encounter a similar problem that Berguin already described in his work on JENN. He determines one of the main problems in setting up the surrogate model is the internal weighing of different regions during the training process. Regions with high changes result in a high penalty for wrong approximations, while regions with small changes close to zero result in a low penalty for inaccuracy. Inevitably, areas with smaller changes suffer from underfitting, although these are often the areas of interest in an optimisation task. Rosenbrock, for example, has quite steep descents with extremely large function values at the sides and a stretched flat valley close to zero in the middle as displayed in the top left image of Figure 4.1.

In our test models, we also experience this effect's impact. The built model shows steep slopes at the edges and a flat area in the centre. This results in premature convergence and, thus, wrong solutions. While Berguin solves this in his work by applying polishing, a method where specific areas are marked as more important than others, we can not recreate this as GENN and GEKPLS do not offer this weighing. Moreover, we intend to find a more generally applicable solution than manually flagging areas of interest for each individual test problem. We develop an approach to alleviate the problems of very high function values while preserving the function's main characteristics, like differentiability or the exact location of optima. Concretely, we propose composing the test function with the function $x \mapsto \log(1 - \ell + x)$, where ℓ is a lower bound on the original test function. In applications, the test function is usually a loss function and is non-negative, so $\ell = 0$.

T1-L (Rosenbrock Function with Logarithm Mapping): The function `log_rosenbrock(x, y, b)` is defined as:

$$f(x, y) = \log(1 + (b \cdot (y - x^2)^2 + (x - 1)^2)), \quad \text{for } x, y \in \mathbb{R}$$

where $b \in \mathbb{R}$ is a constant with a default value of 10.

T2-L (Sine Function with Logarithm Mapping): The function `log_sine(x, y)` is defined as:

$$f(x, y) = \log(2 + (\sin(x) + 0.1 \cdot y^2)), \quad \text{for } x, y \in \mathbb{R}.$$

T3-L (Sine with Noise Function with Logarithm Mapping): The function `log_sine_noise(x, y, δ)` is defined as:

$$f(x, y, \delta) = \log(2 + \sin(x) + 0.1 \cdot y^2 + \delta \cdot \sin(100 \cdot (x + y))), \quad \text{for } x, y \in \mathbb{R}$$

where $\delta > 0$ defines the amplitude of the added sine noise.

We map our test functions T1-T3 by adding a constant value before taking the logarithm. By this, we ensure that we work with positive values before computing the logarithm, so we are not changing the function's characteristics. The adjusted function remains differentiable and, therefore, applicable for training our surrogate model.

To demonstrate that our mapping is keeping the function characteristics, we examine the general function $g(x) = \log(c + f(x))$, where c is a constant and $f(x)$ is a given function. For this, we analyse the monotonicity and differentiability of $g(x)$.

Monotonicity

Let $f : \mathbb{R}^d \to \mathbb{R}$ be such that $f(x) + c \geq 1$ for all $x \in \mathbb{R}^d$ and a $c \in \mathbb{R}$. Therefore, for x, y, with $f(x) < f(y)$ it follows by the strict monotonicitiy of log that $\log(c + f(x)) < \log(c + f(y))$.

In particular, for a global minimum x^* of f, x^* is also a global minimum of $\log(c + f)$ and vice versa.

Differentiability

If f is differentiable and $c + f(x) \geq 1$ for all x, then $g = \log(c + f)$ is (everywhere) differentiable by the chain rule.

The derivative of g at x is computed using the chain rule:

$$g'(x) = \frac{d}{dx} \log(c + f(x)) = \frac{1}{c + f(x)} \cdot f'(x).$$

Conditions

In order for $g(x) = \log(c + f(x))$ to be well-defined and differentiable, we require that:

$$c + f(x) \geq 1.$$

After converting the test function to the logarithm of the test function, our models are noticeably more accurate and reliable in finding the real minimum. Figure 4.1 shows the impact of using T1 as the basis for both optimisation methods and surrogate models compared to using the logarithm mapped function values of T1-L. The plots show that all function characteristics are preserved, but the features are enhanced overall and extremely large function values are toned down.

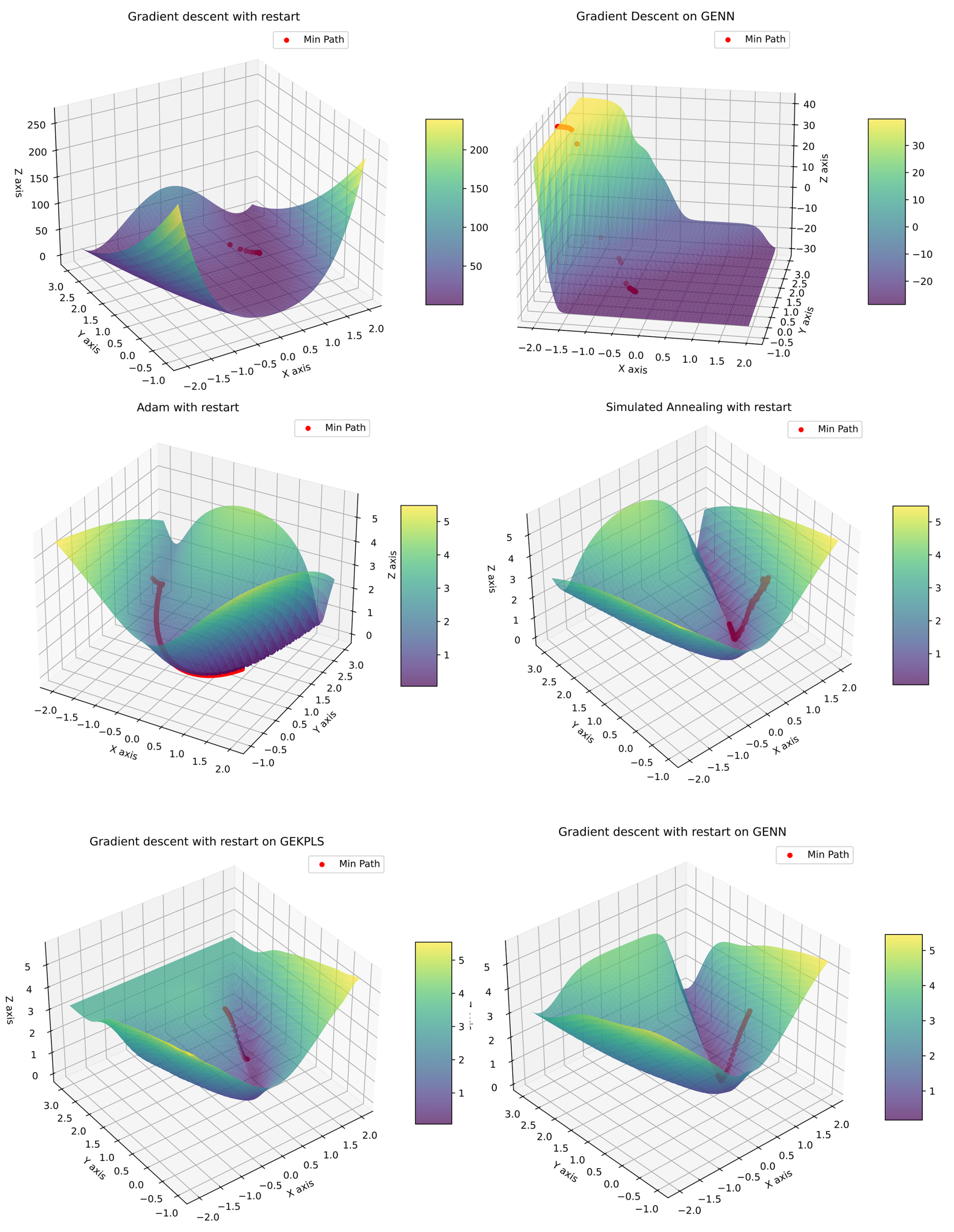

Figure 4.1: Best path finding the minimum of different algorithms on Rosenbrock (T1) and the logarithm mapped Rosenbrock (T1-L) to compare the improved features of applying the logarithm. **Top left**: Gradient Descent with Restart on T1. **Top right**: Gradient Descent on GENN based on T1. **Middle left**: Adam on T1-L. **Middle right**: Simulated Annealing on T1-L. **Bottom left**: Gradient Descent on GEKPLS based on T1-L. **Bottom right**: Gradient Descent on GENN T1-L.

4.2 Optimisation on Tamashii

Equivalent to the experiments on the test functions, we apply Gradient Descent, Adam, and Simulated Annealing with GENN and GEKPLS on the Tamashii models.

We consider the following two scenes for our Tamashii experiments:

S1 **Rabbit**: The rendered Stanford bunny as shown in Figure 3.1.

S2 **Simple Office**: A rendering of an office with two tables to be illuminated as visualised in Figure 4.2.

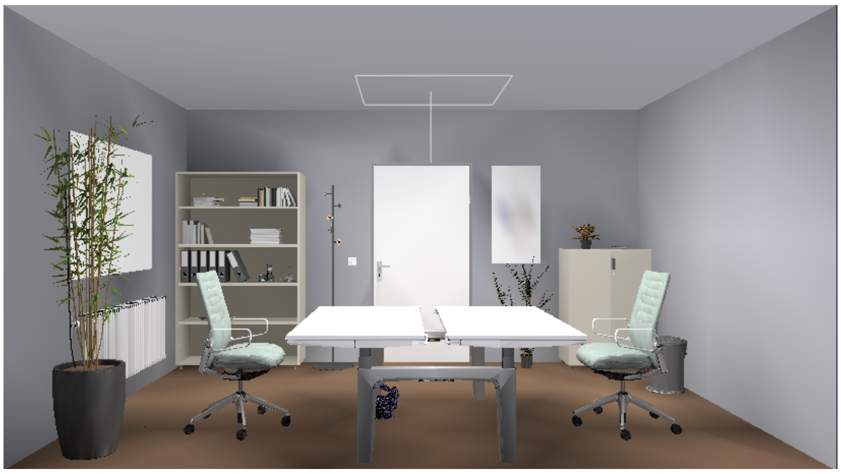

Figure 4.2: Rendered Simple Office scene showing a room with one light source. The goal is to create lighting for the two desks according to a given target illumination.

For the surrogate models, we also add Stochastic Gradient Descent (SGD) to the list of optimisation algorithms. These algorithms are applied to the real model, as well as to the surrogate models and the retrained versions. Finding an accurate representation of the actual model is significantly more challenging than finding a representation of the test functions. We test various sampling methods, parameter settings for the surrogate model initialisation, different learning rates for the algorithms, as well as improvements for retraining the models.

4.2.1 Algorithm Parameters

As a first step, we apply Gradient Descent (GD), Adam and Simulated Annealing (SA) directly to the Tamashii model to create a ground truth for our surrogate model experiments. We quickly notice that performing the optimisation on the test scenes is very time-consuming as it requires many direct model evaluations. The main parameters to fine-tune for Gradient Descent and Adam are the learning rate and the number of steps per optimisation run. The main difference between direct optimisation on the model and optimisation on the surrogate model is the difference in the step count. In Tamashii, we opt for a step count of only 100 instead of 1000, as used for the test function experiments and surrogate models, to keep the runtime within a reasonable timespan. For Simulated Annealing, we had the initial temperature and the cooling rate to experiment with. At first sight, Simulated Annealing does not appear as promising in finding the global minimum as Adam or Gradient Descent, so we want to see whether Simulated Annealing is just not a suitable algorithm or if we can improve the convergence by adjusting the parameter settings. We will elaborate more on the results of the Simulated Annealing adjustment in the chapter 5.

4.2.2 Sampling Plan

A sampling plan needs to be made for setting up a surrogate model (Figure 2.2). In order to find the best method, we compare different numbers of sampling points. The underlying idea is to have a model with as few sampling points as possible since calculating the loss value and gradient is the most time-consuming step in Tamashii. Furthermore, we test sampling with an equidistant grid, with random sampling points and with LHS. As we previously found promising results in working with the logarithm-mapped function values in the test function experiments, we also apply this on Tamashii. We visualise the sampled data and the built models by slicing through the model and plotting tiles of contour curves with colour-mapped loss values. This helps to understand the three-dimensional room that is otherwise hard to imagine visually. It also gives a better comparison of how accurately a model represents the actual space.

For a better understanding, we visualise the arrangement of the contour tiles in the room. In Figure 4.3, we can see, with slightly see-through versions of the tiles, how the x, y, and z slices are structured in the 3D space. The colourmapping displays the loss values, with dark purple being the smallest and bright yellow the highest. This means that the bright yellow area is where the rabbit is located in the 3D space, shown in the bottom right image of Figure 4.3, and a light position would lead to a high loss value in the overall scene illumination.

Figure 4.4 shows contour tiles of the Rabbit (S1, 3.1), comparing logarithm-mapping and normal sample values. For this, we used 1000 sample points, a high number which is only used for visualisation purposes. Comparing the normal values with the logarithm mapping, we can see that, equivalent to the Rosenbrock function, all features are preserved but enhanced simultaneously, which we would evaluate as a positive effect for the time being.

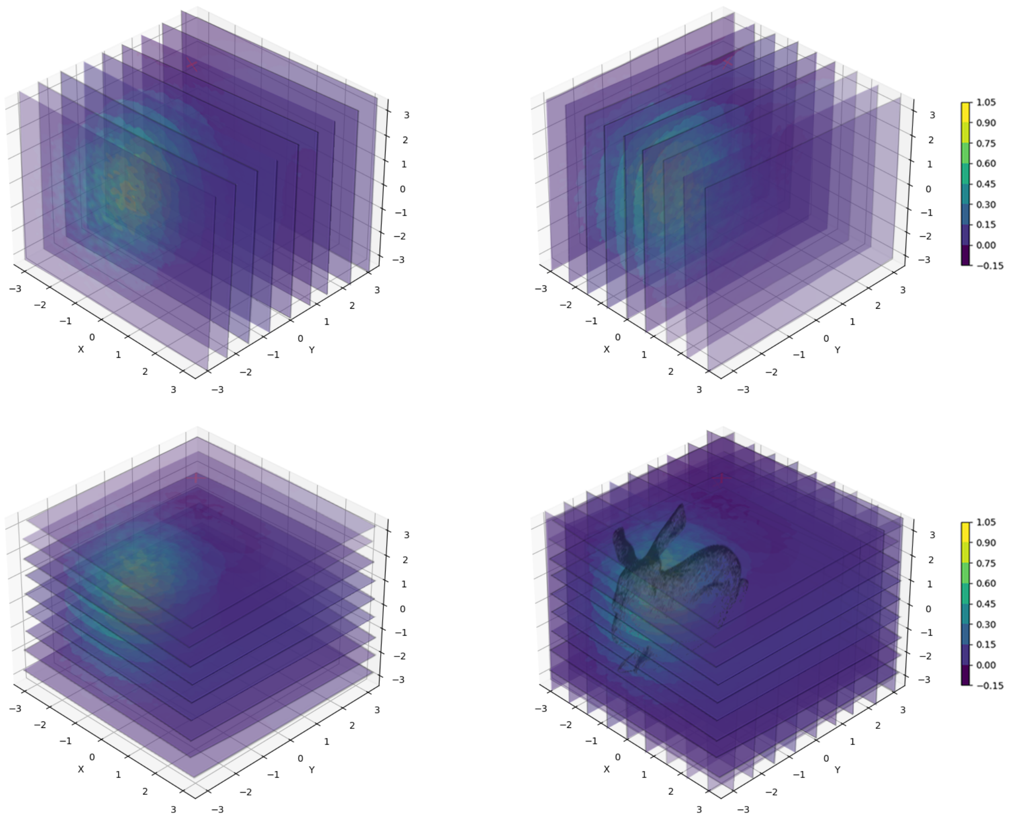

Figure 4.3: Overview of how the contour tiles are arranged in the design space. **Top left**: y-contour tiles with fixed y values, **top right**: x-contour tiles with fixed x values, **bottom left**: z-contour tiles with fixed z values, **bottom right**: all contour tiles combined with rabbit object.

Figure 4.5 shows a juxtaposition of the built GENN and GEKPLS models based on 100 sampling points in the Rabbit space. Overall, both models show very similar representations of the real space, with GEKPLS having slightly more extreme features than GENN.

With this gained knowledge, we want to continue our experiments and find out if the enhanced features of logarithm mapping or the GEKPLS model yield an advantage in the overall optimisation process.

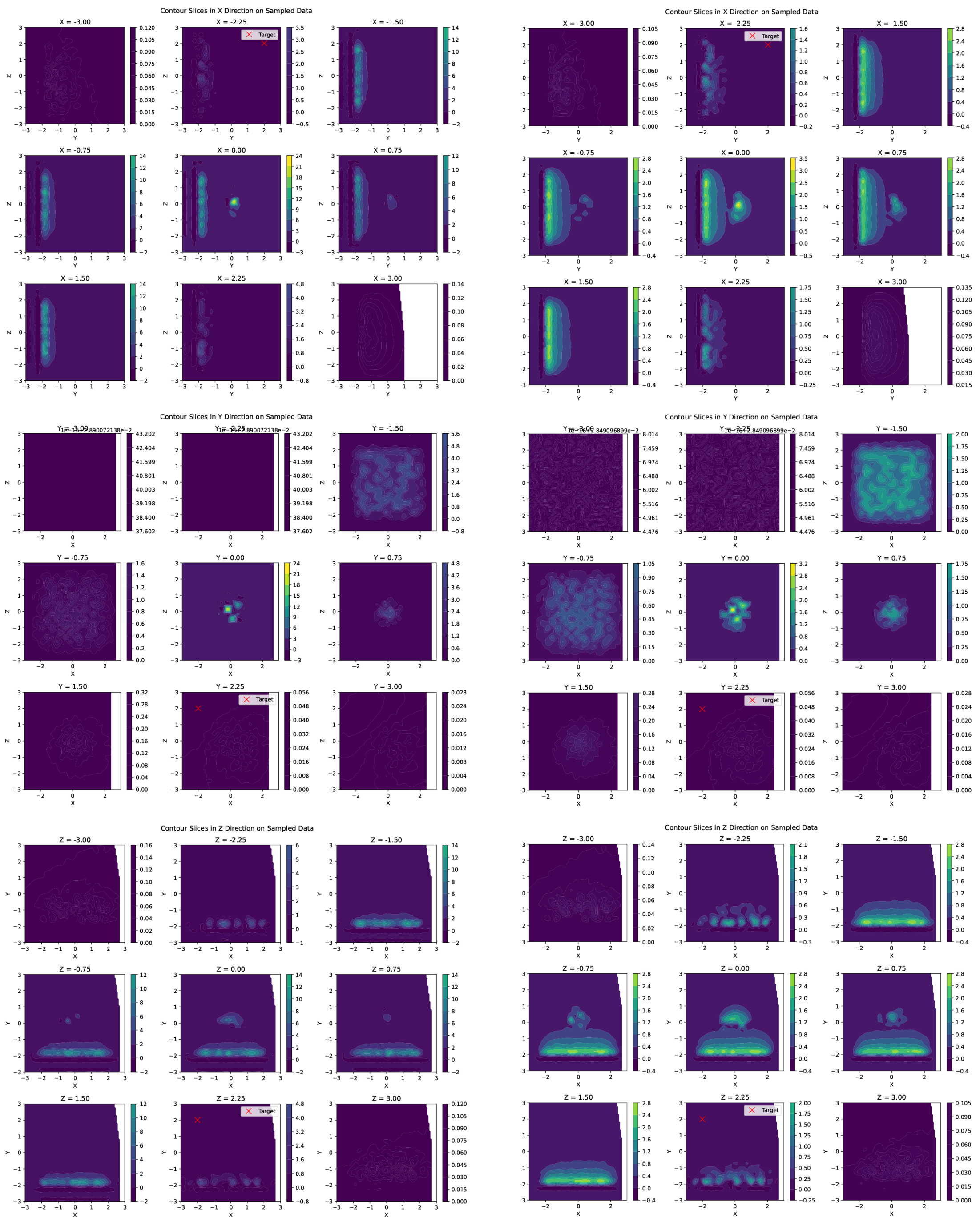

Figure 4.4: Contour tiles of the real model of the Rabbit (S1) in X, Y, and Z directions comparing real model values with logarithm mapping. **Left**: 1000 points sampled in a regular grid without logarithm. **Right**: 1000 points sampled in a regular grid with logarithm mapping.

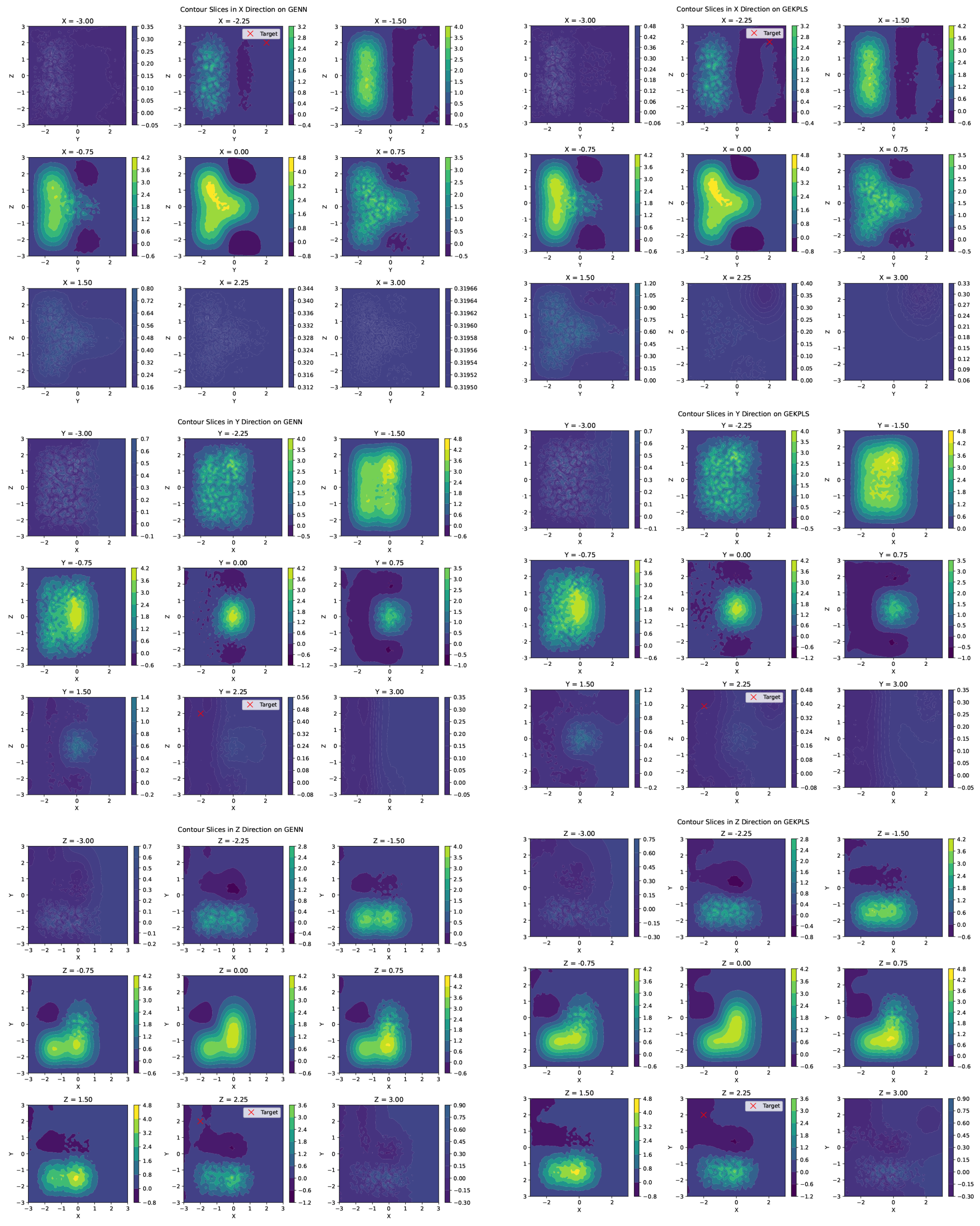

Figure 4.5: Contour tiles of the surrogate models of the Rabbit (S1) in X, Y, and Z directions. Sampling method: 100 sample points in a regular grid without logarithm. **Left**: Contour tiles of GENN, **right**: Contour tiles of GEKPLS.

4.2.3 Surrogate Model Parameters

We also want to find the best possible initialisation parameters for the surrogate models. Our initial setup is based on the SMT implementations ([SMT22b, SMT22a]). We perform tests for the surrogate model parameters using the same sampling method for all parameter settings. As the surrogate models differ every time they are set up, we choose to work with grid sampling since this method would always return the same sampling points for our training data, thus creating an environment as coherent as possible.

The GENN implementation provides several parameters to adjust. First, we test different settings for lambda, the regularisation coefficient. The regularisation coefficient is responsible for the overall smoothness of the built model. Having a small regularisation coefficient leads to a model representing the data very closely but comes with potential overfitting. On the other hand, a large regularisation coefficient might result in a model that is too simple for the training data used. We try a variety from very small to very large values for the regularisation coefficient, ranging from 0.001, 0.01, 0.1, 1 to 10. We continue testing the learning rate alpha for the values 0.1 and 0.05. Continuing, we test hidden_layer_sizes for [6, 6], [12, 12], [20, 6], [100] and [1000], defining the number of neurons per layer of the neural network. Typically, in machine learning, this setup is found via experimenting with different sizes to figure out a problem-dependent solution. For the two-layer setting, we keep the number of neurons reasonably low, as a bigger network would require a larger amount of training data to find a good general representation without overfitting. However, our general approach is to find a solution to work with as little data as possible, so we focus on smaller networks instead.

The GEKPLS model does not have as many parameters to fine-tune as GENN. We test the parameter extra_points, which is the number of extra points per training point. This parameter is used to modulate the model's accuracy. The higher the number of extra points per training point, the higher the chance for better accuracy but also higher computational cost. A lower input for this parameter would come at the cost of accuracy but could improve the speed during the training process. The SMT default is 0, but the example implementation is set to 1, therefore we test both versions. We also try different values for the nugget parameter, which is jitter for numerical stability. This stability is achieved by adding this nugget-input value to the diagonal of the covariance matrix used for Kriging. Especially when gradients are included in the computation, the covariance matrix contains both the function values and the gradient information. When function values or gradient information are close to each other, this matrix can easily become ill-conditioned, leading to errors during the computation in the experiment. By adding a small nugget, we can reduce the risk of these errors. The nugget also works as a regularisation parameter, making the overall model a bit more robust. Here, we have the same constraints to consider as for the lambda parameter in GENN. A small nugget might not regularise the data enough, leading to potential underfitting while at the same time not providing the needed numerical stability. A larger nugget, on the other hand, might provide reliable numerical stability but could result in overfitting. We experiment with the values 1e-10, 1e-8, and the SMT default of 2.220446049250313e-14.

4.2.4 Algorithm Reliability

For each algorithm run, we return the best path towards the found minimum in the search space. We plot the target marked as a red cross and the individual paths in the 3D space to visualise the algorithm's behaviour and convergence (Figure 4.6, 4.7, 4.8). The individual algorithm paths show a colour gradient from yellow (high loss value) to the assigned colour in the legend (low loss value). The starting points are also marked as crosses corresponding to the algorithm's colour. Ideally, we want to see a path starting somewhere in the room, making its way towards the red cross and converging close to it. Nevertheless, we can see that the algorithms, most of the time, terminate somewhere else in the room and don't find the target. This leads us to test the reliability of the individual algorithms on the surrogate models in isolation. In order to find out if the termination in the wrong area is due to an unreliable algorithm or an inaccurate model, we compare the approximated minimum of the surrogate model to the location of termination of each individual algorithm.

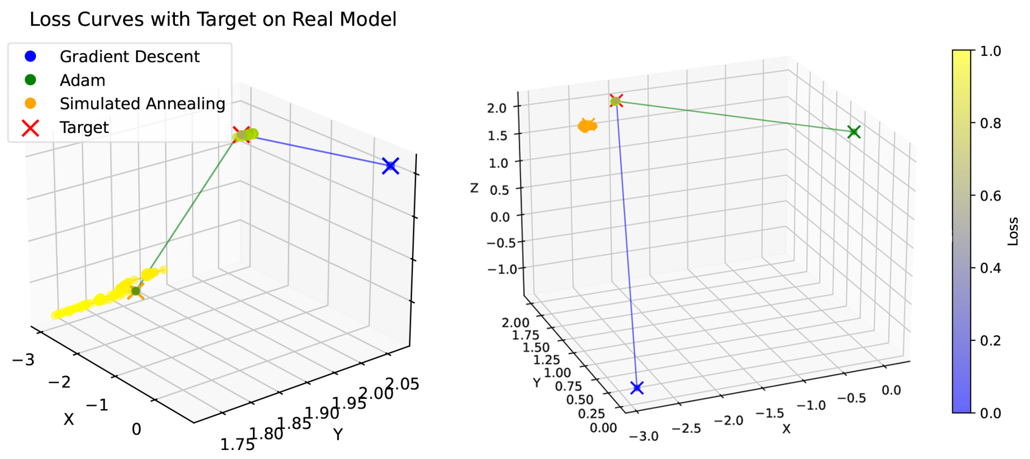

Figure 4.6: Found minimum per algorithm with corresponding path on the Tamashii model for two different experiment runs. The red cross marks the target, the differently coloured crosses mark the starting point for the corresponding algorithms.

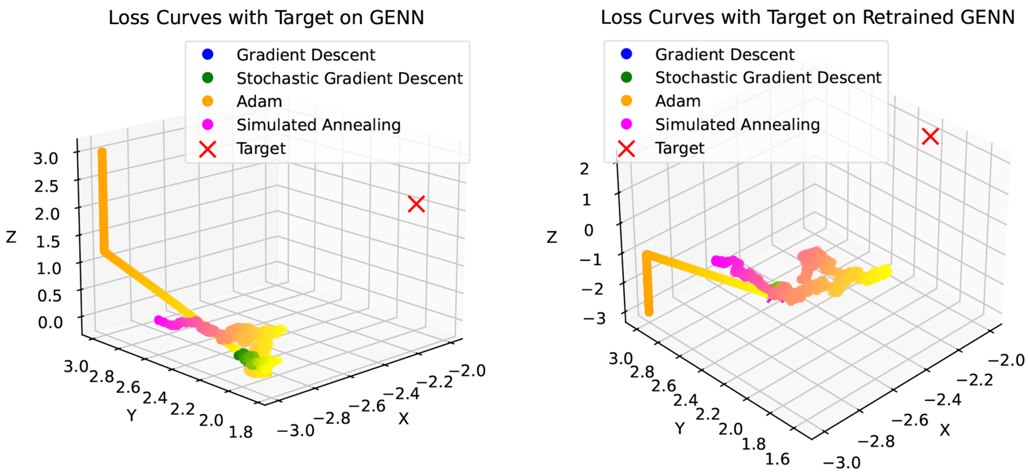

Figure 4.7: Found minimum per algorithm with corresponding path on GENN model and GENN model after relearning. Sampling method: 50 sample points in a grid with logarithm mapping.

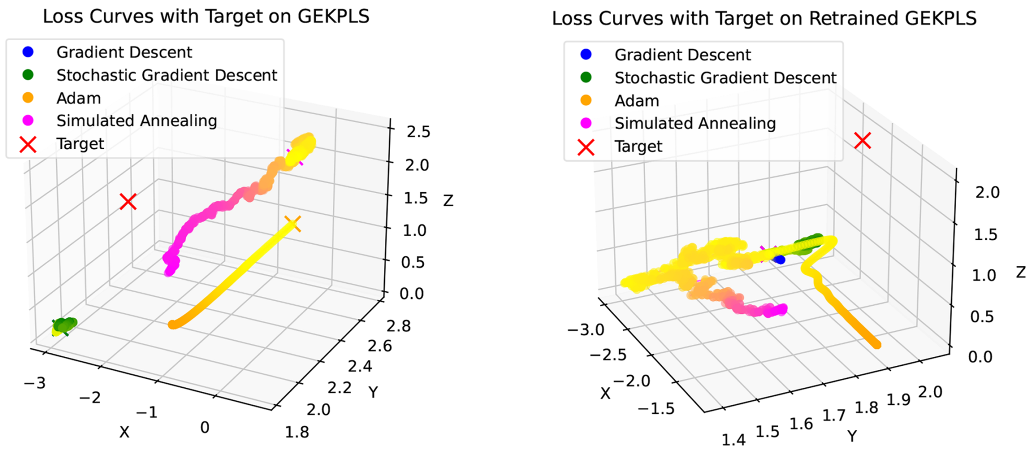

Figure 4.8: Found minimum per algorithm with corresponding path on GEKPLS model and GEKPLS model after relearning. Sampling method: 50 sample points in a grid with logarithm mapping.

CHAPTER 5

Results

Having described our experiment setup, we now want to go into detail about our results. We will first go over our test function experiments and then continue with the insight we gained during the Tamashii optimisation. An overall realisation is that it appears to be challenging to find an appropriate approximation of the individual problems for our surrogate models. Although we figured out improvements in both the algorithm parameter settings and the surrogate model initialisation parameters, we experience performing the optimisation directly on the Tamashii model to be the most reliable approach compared to optimising on the surrogate models.

5.1 Test Function Experiments

In our experiments on the test functions Rosenbrock, sine, and sine with noise (T1-L-T3-L), we compare the impact of performing one optimisation with several restarts per optimisation. For one iteration, we predefine fixed starting points separately for the individual functions. This is because we already know the global minimum and want to avoid starting our optimisation too close to that point. Certainly, in an unknown environment, this could very well be the case, but for now, we want to prioritise testing the functionality of our algorithms, so we seek to avoid this scenario. In Figure 5.1, 5.2, and 5.3, we plot the convergence of each algorithm with one iteration versus the best restart iteration from 10 random starting points towards the real function's minimum. Especially in the plot for the Rosenbrock convergence (Figure 5.1), we can see that performing restarts outperforms a single algorithm iteration. We can also see that Gradient Descent, in particular, suffers from premature termination. Working with different learning rates and step counts per optimisation cycle shows that we can not completely solve that problem by simply adjusting these parameters. We test constant learning rates as well as decaying learning rates, but as a constant learning rate often comes with the risk of oscillation once an optimum is about to be reached, we opt for the second approach. Similarly, for the step count, we experience that changing it results in improved termination in some functions while worsening it for others. This effect is very obvious comparing Figure 5.1 to 5.2 and 5.3. With our parameter setting, we achieve overall good convergence on T1-L, while several algorithms find a close approximation of the minimum in T2-L and T3-L but bounce out of this region and terminate somewhere else. We find a count of 1000 steps to be a good middle ground for our test functions and the surrogate models.

In all three plots, we find Gradient Descent without restart to be the worst-performing algorithm. Another noticeable effect is the convergence curve of Simulated Annealing compared to the gradient-based algorithms Gradient Descent and Adam. While they show smooth lines, Simulated Annealing fluctuates a lot. This is due to the randomness in creating the next position during the algorithm execution and is especially prominent in Figure 5.2.

Overall, we can observe that the optimisations performed on the surrogate models find similar minima as optimising on the test functions itself, leading us to conclude that our model setup and the logarithm mapping have the desired effect of creating reliable approximations. Furthermore, restarting seems essential to improve the algorithms concerning premature termination, which counts particularly for Gradient Descent.

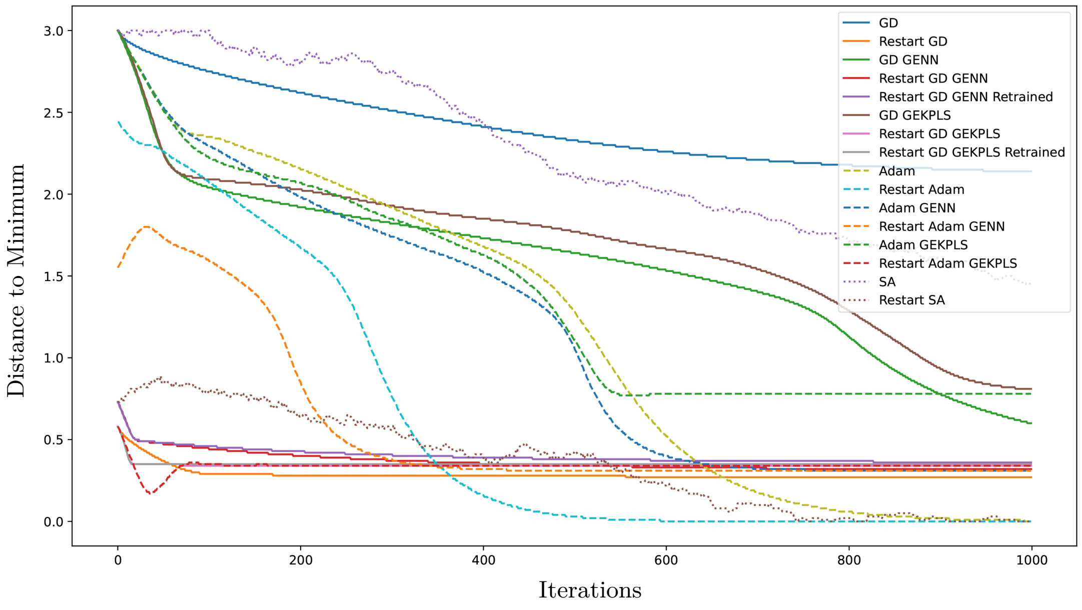

Figure 5.1: Convergence of the different algorithms towards the logarithm mapped Rosenbrock (T1-L) minimum with one optimisation start and the best result of 10 restarts.

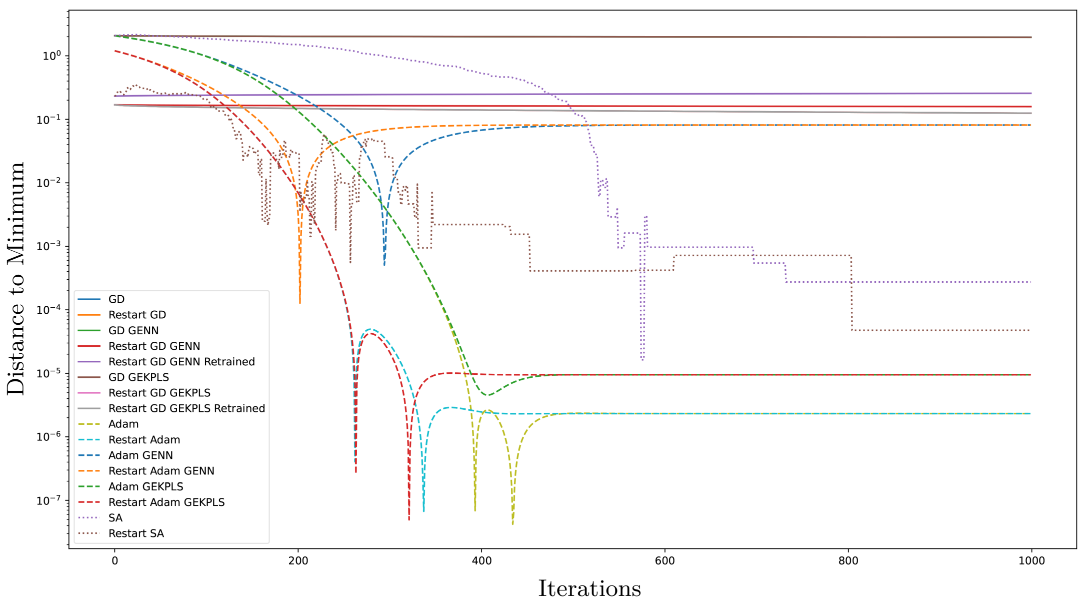

Figure 5.2: Convergence of the different algorithms towards the logarithm mapped sine (T2-L) minimum with one optimisation start and the best result of 10 restarts.

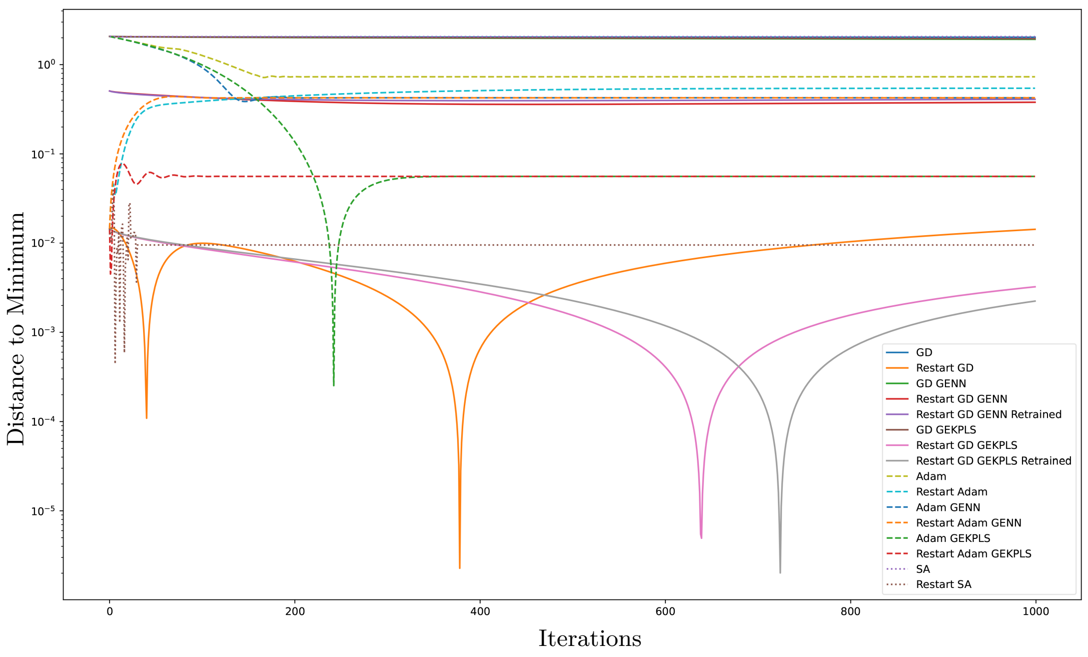

Figure 5.3: Convergence of the different algorithms towards the logarithm mapped sine with noise (T3-L) minimum with one optimisation start and the best result of 10 restarts.

5.2 Tamashii

5.2.1 Optimisation on Tamashii

The first and most obvious result we immediately notice is that applying the optimisation algorithms directly to the real Tamashii model results in the best outcome. However, it comes with massively increased time consumption needed to perform the task. Therefore, we compare the algorithms by applying them once, as we did in the test function experiments, with starting from 10 randomly chosen starting positions. We aim to create a setup that is as versatile as possible, hence, we define a universal starting point for performing only one algorithm iteration to be in the center of the room.

Algorithm Minima on Real Model									
RABBIT, 1 Iteration, 100 steps									Average
GD	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**
Adam	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	**0.3**
SA	3.37	3.59	3.47	3.39	3.46	3.38	3.44	3.55	**3.44**
SIMPLE OFFICE, 1 Iteration, 100 steps									Average
GD	0.0	0.0	0.0	0.0	3.91	3.91	0.0	0.0	**0.98**
Adam	0.0	0.0	0.0	0.0	3.91	3.91	0.0	0.0	**0.98**
SA	0.7	0.79	0.75	0.8	0.83	0.63	0.84	0.81	**0.77**
RABBIT, 10 Iterations, 100 steps									Average
GD	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**
Adam	0.03	0.03	0.03	0.04	0.03	0.03	0.03	0.03	**0.03**
SA	2.36	2.83	2.24	0.93	1.73	0.93	4.01	1.13	**2.14**
SIMPLE OFFICE, 10 Iterations, 100 steps									Average
GD	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**
Adam	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**
SA	2.34	2.52	2.00	1.57	3.69	2.82	1.67	3.16	**2.47**

Table 5.1: Distances of found optimum to real optimum for different iteration counts in independently repeated experiments for GD, Adam and SA.

We learn that both Gradient Descent and Adam reliably find the target minimum, although there is a slight uncertainty when starting only one time from the centre. We trace that back to the built-in noise in the Tamashii gradients, which could, at worst, lead the algorithm away from the target minimum towards a local one. One could argue that one single algorithm start is working well enough on Tamashii in the case of Adam and Gradient Descent for the majority of test runs, and the lesser time consumption outweighs the small risk of not finding the optimum. But again, we want to create a universally applicable solution. Seeing that in the Simple Office scene S2 (Figure 4.2), we occasionally experience iterations finding a solution far away from the target, we want to avoid this happening in other design scenes either, so consequently, we can not limit our search to only one function call. Figure 4.6 shows that Gradient Descent and Adam mostly jump straight to the desired location and find the target. Simulated Annealing, on the other hand, turns out to be fairly unreliable, usually not finding the target minimum (Table 5.1).

5.2.2 Timing

Measuring the times during our experiments, we receive an overview of the computational cost of the individual approaches. As already pointed out, performing a direct evaluation on Tamashii requires quite some time and is a factor to be considered when deciding on the applied method. For the surrogate models, we need to sample the loss values and gradients for a certain number of sample points instead. As we can read in Table 5.3, sampling the loss values for 50 points just takes 9 seconds, while sampling for 100 points takes 18 seconds already, and 500 points will take around 80 seconds. This also explains the long duration for direct optimisation on Tamashii, as every algorithm iteration needs to perform 100 steps. As previously tested, restarting the algorithms seems to be crucial in order to ensure good results, so we perform ten restarts on Tamashii as well. This eventuates in long durations, such as circa 120 seconds for Gradient Descent and Adam or 215 seconds for Simulated Annealing on the Rabbit scene S1 (Table 5.2). The much longer times needed for Simulated Annealing stem from the need for two calculations of loss values per step instead of one needed in Gradient Descent and Adam. Interestingly, the Simple Office scene does require significantly less time to calculate the loss values and gradients.

Duration for Optimisation on Tamashii

Algorithm	**Rabbit**		**Simple Office**	
	1 Iteration	10 Iterations	1 Iteration	10 Iterations
GD	9	120	1.7	15
Adam	9	129	1.4	16
SA	23	215	3.2	90

Table 5.2: Overview of spent time in seconds to perform the different optimisation methods on the Tamashii models S1 and S2.

Although optimising on the test scenes seems to be fairly time-consuming, one has to bear in mind that creating a surrogate model also comes with a lot of computational costs. Depending on the sampling plan, one has to first sample the loss values and gradients for the defined sampling points, then train the model, potentially retrain it and finally perform the optimisation itself. Since we plan to use as few sampling points as possible, we want to review the computational expense for only 50 sampling points to understand

Duration for Surrogate Models				
		50 points	100 points	500 points
	Sampling	9	18	80
Training	GENN	2.5	2.7	4.2
	GEKPLS	0.3	0.8	5.4
Relearning	GENN	26	30	45
	GEKPLS	24	37	196
Optimisation on Surrogate Models, 10 Restarts, 1000 steps				
GENN, Retrained GENN	GD, SGD, Adam	4.5	4.5	4.5
	SA	0.9	1	0.9
GEKPLS, Retrained GEKPLS	GD, SGD, Adam	17	28	115
	SA	5	7	21

Table 5.3: Overview of spent time in seconds for sampling, training and retraining the surrogate models as well as performing the different optimisation methods with 10 restarts on the surrogate models based on the Rabbit S1.

the time requirements. As stated before, sampling the loss values and gradients for 50 points in the Rabbit scene takes 9 seconds. Training a GENN model takes about 2.5 seconds, while training the GEKPLS model only requires 0.3 seconds. Retraining those models 10 times can be done in about 25 seconds. Gradient Descent, Stochastic Gradient Descent, and Adam usually all take the same time of 4.5 seconds on GENN and about 17 seconds on GEKPLS. So, would we decide to train a surrogate model with only 50 sample points and perform ten optimisation restarts, it overall consumes 16 seconds on GENN, 26.3 seconds on GEKPLS or, in case we decided to include relearning, 41 seconds on GENN versus 51.3 seconds on GEKPLS. Generally, evaluating a GENN model is quicker than a GEKPLS model, which is even more noticeable the more sampling points are used. While Gradient Descent, Stochastic Gradient Descent, and Adam more or less need the same time to run on GENN or retrained GENN regardless of the used sampling count, their time to run on GEKPLS or the retrained version increases drastically. Also, the relearning process takes up much more time, increasing from 24 seconds for 50 sampling points to 196 seconds for 500 sampling points. From a performance point of view, this has to be considered when setting up the optimisation environment. The main use of surrogate models is to benefit from faster model evaluations by creating an approximation of the real problem. It seems wise to opt for as few sample points as possible to keep the computation times as low as possible to be able to exploit the benefit from the surrogate models.

5.2.3 Algorithm Improvement

We quickly figured out that Simulated Annealing did not perform well on the real Tamashii models nor on the surrogate models, so we decide to test different parameter settings for the initial temperature and the cooling rate. We find improvements for a combination of a high initial temperature with a slow cooling rate as well as for a low initial temperature combined with a faster cooling rate. We take the two best combinations and test if a higher step count within each algorithm iteration would improve this optimisation method (Table 5.4). Indeed, a doubled step count shows better results but comes with the cost of a doubled runtime of circa 430 seconds compared to the previous 215 seconds. Balancing accuracy against consumed time, we conclude that the gained accuracy is out of proportion in comparison to the amount of time needed to perform double the steps. As Simulated Annealing is already the slowest algorithm regarding runtime, we decide to opt for the slightly worse setting with fewer steps to find a balance between efficiency and accuracy.

Simulated Annealing Parameters 10 Iterations, 100 steps			
Initial Temperature	**Cooling Rate**		
	0.8	**0.9**	**0.99**
1	1.66	1.9	3.46
10	3.27	2.55	2.73
100	1.7	2.86	1.67
1000	2.59	2.18	2.76
10 Iterations, 200 steps			
1	1.03		
100			2.25

Table 5.4: Overview of the influence of different parameter settings for Simulated Annealing on Tamashii (S1) with an average distance to the target optimum over several experiment repetitions.

5.2.4 Surrogate Model Improvement

The surrogate model generally results in a quite inaccurate representation of the search space, leading to wrong minima. As described in the previous chapter, our first approach is to test if we can improve the model accuracy by adjusting the hyperparameters in the initialisation process. We, therefore, fix all necessary parameters and just change one at a time within a certain range.

GENN Parameters RABBIT, 50 Samples, Grid, Log						
		lambda				
		0.001	**0.01**	**0.1**	**1**	**10**
GENN	GD	6.68	5.39	4.49	3.66	3.85
	SGD	6.64	5.77	4.53	3.63	3.84
	Adam	4.81	6.08	4.72	3.66	3.85
	SA	4.87	5.21	4.41	3.66	3.82
Retrained GENN	GD	5.84	6.08	3.21	3.66	3.85
	SGD	5.84	5.91	3.22	3.7	3.87
	Adam	6.33	6.35	4.44	3.66	3.85
	SA	6.08	5.94	3.43	3.9	3.9
		alpha				
		0.1	**0.05**			
GENN	GD	4	4.77			
	SGD	3.97	4.81			
	Adam	4	4.77			
	SA	3.77	5.08			
Retrained GENN	GD	3.81	4.77			
	SGD	3.72	4.76			
	Adam	2.41	3.47			
	SA	2.26	3.81			
		hidden layer sizes				
		[6, 6]	**[12, 12]**	**[20, 6]**	**[100]**	**[1000]**
GENN	GD	4.77	2.9	3.07	2.83	2.85
	SGD	4.69	2.95	3.1	2.76	2.87
	Adam	4.77	2.9	3.07	1.73	1.73
	SA	4.87	3.02	2.87	3.16	3.76
Retrained GENN	GD	4.77	2.9	3.07	2.82	3.46
	SGD	4.73	2.9	3.07	2.91	3.46
	Adam	4.77	2.9	3.07	1.73	5.2
	SA	4.87	3.02	2.87	3.16	3.76
RABBIT, 50 Samples, Grid, Normal						
		hidden layer sizes				
		[6, 6]	**[12, 12]**	**[20, 6]**	**[100]**	**[1000]**
GENN	GD	3.17	2.14	2.94	3.59	3.41
	SGD	3.19	2.13	1.73	6.62	6.5
	Adam	1.73	1.73	1.73	6.62	6.5
	SA	3.13	3.13.	2.86	3.65	3.78
Retrained GENN	GD	3.17	3.17	2.95	3.71	2.31
	SGD	3.17	3.17	2.96	3.73	3.68
	Adam	1.73	1.73	2.98	5.96	5.27
	SA	2.97	2.97	2.85	4.72	2.37

Table 5.5: Impact of different initialisation parameters for GENN, comparing the average distance of minima found by the different algorithms to the target minimum. The sampling method was grid sampling with 50 sample points on S1.

GEKPLS Parameters RABBIT, 50 Samples, Grid, Log			
		extra points	
		1	**0**
GEKPLS	GD	4.33	3.04
	SGD	4.33	3.04
	Adam	3.14	1.52
	SA	3.64	2.54
Retrained GEKPLS	GD	3.59	3.53
	SGD	3.56	3.44
	Adam	2.6	1.85
	SA	3.63	3.34

		nugget		
		1e-10	**1e-8**	**2.220446049250313e-14**
GEKPLS	GD	3.04	3.55	1.88
	SGD	3.04	4.14	1.84
	Adam	1.52	1.78	1.78
	SA	2.54	1.63	1.86
Retrained GEKPLS	GD	3.53	3.81	1.87
	SGD	3.44	3.7	1.81
	Adam	1.85	1.91	1.63
	SA	3.34	1.61	1.54

Table 5.6: Impact of different initialisation parameters for GEKPLS, comparing the average distance of minima found by the different algorithms to the target minimum. The sampling method was grid sampling with 50 sample points on S1.

For each tested parameter, we perform individual runs of our experiments to see if the algorithms result in better or worse minima. The quality of the found minima is measured by the Euclidean distance to the target minimum. Since our algorithms perform several restarts per optimisation method with randomly selected starting points, the outcome can vary significantly. To have a more reliable result, we use the average of three entire experiment repetitions to determine the impact of the different parameter settings. Per parameter, the best setting is identified and fixed as the new basis for the following parameter tests. As we can see in the tables for 5.5, some parameters seem to impact the model accuracy more than others. The regularisation coefficient lambda, for instance, improves the accuracy a lot, while the learning rate alpha does not influence the outcome too much. For the last tested parameter, the hidden layer sizes, we want to see if logarithm mapping influences the model and the choices for our parameters. We test the different settings for both logarithm-mapped values and normal sample values and can see that the best settings indeed seem to vary depending on the sampling method.

We find the best result being a size of [100] in combination with logarithm mapping or [12, 12] for normal values. We decide to work with a value of [100] as we want to continue exploring the effect of logarithm mapping in Tamashii.

For the parameter adjustment in GEKPLS, we only have two parameters that we can test with. We extract from table 5.6, that by changing the nugget value we can significantly improve the model accuracy. At the same time, the extra_point parameter only has a small influence to the positive.

5.2.5 Impact of Sampling Plan

To find out if there is a sampling plan that is more problem-independent, we test on two different sampling spaces, one being the Rabbit S1 (Figure 3.1) and one being the Simple Office S2 (Figure 4.2). We choose a sampling count of 50, 100 and 500 points and three sampling methods, grid sampling, random sampling and LHS, to work on the surrogate models with. We also apply logarithm mapping as an option, just as we did on the test function experiments. To be able to compare the model's accuracy, we predict 100,000 loss values per model in a regular grid to find the area with the smallest loss values. The position of the loss minimum on the surrogate model is used to calculate the Euclidean distance to the target minimum to create a qualitative measurement. The Rabbit scene has a scale of [-3, 3] in every dimension, the Simple Office is only slightly smaller so we define a threshold of 1.73 in distance to mark the better solutions in Table 5.7. For each sampling method, we perform three reruns of our experiments to be able to compare a broader picture as the created models tend to differ a lot.

In the Rabbit scene, we find 50 sampling points with a grid distribution and logarithm-mapping to be a good possible sampling plan. It still results in a model that does not quite find the target minimum but gets the closest to the target compared to the other approaches. Unfortunately, we can not recreate this effect on the Simple Office space. We also find that increasing the number of sampling points does not necessarily result in a better model but instead only significantly increases the required time for both sampling and training the models. This strengthens our tendency to use smaller sampling counts to create the surrogate models.

Figure 5.4 and Figure 5.5 visualise the Rabbit scene with target illumination and illumination based on the different methods. The top left image displays the Rabbit with the light positioned at the target for this scene. We can clearly see that the light positioned at the minimum found on GENN and GEKPLS, as well as on the retrained models, differs noticeably from the target image, while the minimum found by applying the optimisation directly on Tamashii yields results very close to the optimum. Figure 5.4 is the outcome of the sampling method we found to work best on the Rabbit space for GEKPLS. Figure 5.5 is a different sampling method, clearly not producing well-working models.

Surrogate Model Minima												
	RABBIT						**SIMPLE OFFICE**					
	Grid 50		**Random 50**		**LHS 50**		**Grid 50**		**Random 50**		**LHS 50**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
GENN	7.09	5.00	2.84	6.06	3.43	5.2	3.01	2.11	3.82	2.4	**1.43**	0.69
	6.99	**1.56**	6.06	4.56	7.14	1.73	2.37	2.88	2.3	3.31	3.32	3.09
	6.96	**1.40**	2.4	**1.66**	5.74	6.06	3.01	3.11	2.15	2.48	3.24	2.24
Retrained GENN	7.09	5.00	2.09	6.1	3.43	2.48	3.02	2.52	3.81	3.29	2.19	**0.55**
	4.99	2.23	6.06	4.94	7.14	5.29	3.01	2.11	2.24	3.25	3.28	1.76
	5.26	4.48	5.1	5.36	5.76	1.86	3.01	3.16	**0.97**	2.29	3.02	4.2
GEKPLS	1.92	**1.66**	**1.42**	6.66	2.37	5.57	3.0	3.0	1.83	**0.12**	2.51	3.26
	2.03	1.81	7.21	4.9	4.54	5.46	3.0	3.07	2.6	2.0	2.88	1.82
	1.99	1.76	3.94	5.29	4.46	6.25	3.0	3.0	**1.57**	2.28	**1.4**	**0.21**
Retrained GEKPLS	1.92	1.98	**1.42**	6.66	2.27	**1.49**	3.49	3.35	4.48	3.23	3.53	4.38
	2.03	1.88	7.21	4.75	5.01	5.46	3.47	3.35	4.36	3.2	3.91	2.67
	1.99	1.87	7.21	5.29	4.5	6.25	3.41	3.54	4.86	4.15	4.01	4.34
Best Method	Grid 50 Log						LHS 50 Log					
Best Model	GENN						GEKPLS					
	RABBIT						**SIMPLE OFFICE**					
	Grid 100		**Random 100**		**LHS 100**		**100**		**Random 100**		**LHS 100**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
GENN	3.66	4.09	5.58	3.48	8.03	4.45	**1.42**	3.66	**1.57**	1.78	2.44	3.31
	3.68	4.16	5.97	7.07	4.28	5.36	**0.61**	2.07	**1.51**	3.42	3.91	**1.46**
	3.64	3.96	7.05	2.5	3.99	3.48	**1.41**	**1.41**	1.78	3.09	2.15	**1.4**
Retrained GENN	3.66	4.21	2.57	4.21	2.19	5.15	**1.53**	2.77	2.3	1.78	2.38	3.49
	3.89	3.34	5.97	7.08	3.5	6.06	3.1	3.19	**1.54**	2.79	**1.41**	2.67
	3.63	4.25	6.28	5.79	4.16	3.48	1.89	2.52	1.78	3.28	2.12	2.53
GEKPLS	2.54	2.35	4.13	6.26	4.67	6.67	3.29	3.14	**0.57**	1.81	2.42	3.5
	2.61	2.49	5.31	5.56	6.07	4.07	3.29	3.14	**1.14**	3.21	2.06	**1.7**
	2.6	2.56	5.73	4.6	4.98	7.13	3.07	3.14	1.81	2.47	**1.3**	**0.12**
Retrained GEKPLS	3.17	4.88	4.17	6.35	4.81	6.67	3.66	4.15	3.76	3.96	4.18	3.74
	4.8	2.6	5.24	2.27	**1.51**	4.33	4.39	2.93	4.34	4.18	3.91	3.82
	2.6	2.56	3.43	4.96	6.7	7.21	4.78	4.07	3.96	2.87	4.01	4.15
Best Method	LHS 100 Normal						Random 100 Normal					
Best Model	Retrained GEKPKLS						GENN					
	RABBIT						**SIMPLE OFFICE**					
	Grid 500		**Random 500**		**LHS 500**		**Grid 500**		**Random 500**		**LHS 500**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
GENN	5.41	7.73	3.25	3.68	4.66	3.97	2.79	2.16	2.23	3.31	4.18	3.11
	7.28	7.58	3.43	6.22	4.01	3.87	3.79	3.52	2.24	2.88	4.24	1.8
	7.28	7.97	3.36	4.42	3.54	3.55	3.79	2.31	4.53	2.47	**0.79**	3.41
Retrained GENN	5.97	3.44	3.3	3.84	4.74	3.93	1.42	3.45	3.56	3.05	2.92	3.23
	5.31	5.1	3.43	4.27	4.01	4.13	2.79	3.31	3.29	2.95	4.24	**1.27**
	5.95	5.1	3.36	4.39	3.54	3.55	2.82	**1.6**	4.32	**0.28**	2.01	3.53
GEKPLS	5.57	5.34	3.03	6.11	5.13	6.17	2.17	2.22	2.46	2.64	**0.89**	**1.14**
	5.57	5.34	4.25	5.11	3.78	4.87	**0.71**	2.22	3.88	4.63	2.78	3.85
	5.57	5.34	**1.34**	5.86	5.2	3.7	2.2	2.22	3.61	**1.02**	4.51	2.77
Retrained GEKPLS	5.43	5.17	3.12	5.16	5.13	4.89	4.09	3.38	4.28	3.87	3.61	4.69
	5.62	5.39	7.44	5.11	4.29	4.87	2.88	4.45	3.96	4.05	2.92	3.63
	5.57	5.39	**1.24**	5.86	3.97	5.56	4.15	3.51	2.16	4.65	3.97	3.97
Best Method	Random 500 Normal						Random 500 Log					
Best Model	Retrained GEKPLS						GEKPLS					

Table 5.7: Distances of surrogate model minima based on different sampling methods to the target minimum. The surrogate minima were evaluated by predicting 100,000 sample points to find the smallest loss value. The bold numbers mark the best solutions found with a threshold of a distance smaller than 1.73.

Applying the logarithm to the training values of the models does not have such a significantly positive effect as it had on the Rosenbrock function (T1). For each sampling count, we determine the best sampling method to see how often we decide for normal values or logarithm-mapping. We can see both methods equally often, so this does not give us a definite preference. Still, in the overall picture, we find the logarithm mapping resulting in slightly better or only minimally worse results in building the surrogate model. Therefore, we support the theory that this can be a helpful addition to finding a general setup. We also can not point out a sampling method that would work well on both the Rabbit and the Simple Office space, but we can say that LHS and random sampling seem to be a better choice than grid sampling. GEKPLS appears to produce minimally better optima than GENN, but in general, there is no clear trend visible on whether GENN or GEKPLS is more suitable for Tamashii(Table 5.7).

For the model accuracy, relearning does not have a noticeably positive effect. It does sometimes improve the found minimum, while sometimes it has no effect at all or even worsens the solution at times. To find out if we can improve the relearning process with a higher iteration counter, we test 20 and 30 relearning iterations as well. Unfortunately, we must realise that this, too, does not significantly help to build a more reliable representation. We can see a minimally positive effect, but a higher relearning counter also comes with increased computational resources. Considering the time needed to retrain the models, one must contemplate the benefit of having only a small chance of improvement over drastically increasing the runtime. Furthermore, we follow one more approach to improve the retrained models. Since we discovered that Gradient Descent is not the most reliable choice for our surrogate models to find the surrogate minima, covered in the following subsection, we question the fundamental idea of employing Gradient Descent in the relearning iterations. Nevertheless, in a separate test using Adam to find the current surrogate minima, we only see a slightly better outcome than with the initial Gradient Descent implementation 5.8. One aspect to point out, though, is that the GENN models seem to show marginally better responses in relearning than the GEKPLS models.

Surrogate Model Retraining												
Gradient Descent - 10 Relearning Iterations												
	RABBIT						**SIMPLE OFFICE**					
	Grid 50		**Random 50**		**LHS 50**		**Grid 50**		**Random 50**		**LHS 50**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
	0.00	0.00	0.75	0.00	0.04	**-2.72**	0.01	0.41	**-0.01**	0.84	0.76	**-0.14**
GENN/	**-2.00**	0.67	0.00	0.38	0.00	3.56	0.64	**-0.77**	**-0.06**	**-0.06**	**-0.04**	**-1.33**
Retrained GENN	**-1.70**	3.08	2.70	3.70	0.02	**-4.2**	0.00	0.05	**-1.18**	**-0.24**	**-0.22**	1.96
	0.00	0.32	0.00	0.00	**-0.10**	**-4.08**	0.49	0.35	2.65	3.2	1.02	1.12
GEKPLS/	0.00	0.07	0.00	**-0.15**	0.47	0.00	0.07	0.28	1.76	1.20	1.03	0.85
Retrained GEKPLS	0.00	0.09	3.27	0.00	0.04	0.00	0.41	0.54	3.29	0.92	2.61	4.13
Gradient Descent - 20 Relearning Iterations												
	Grid 50		**Random 50**		**LHS 50**		**Grid 50**		**Random 50**		**LHS 50**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
	0.23	0.00	1.16	2.73	**-3.22**	**-0.28**	0.13	**-0.87**	0.00	3.58	**-0.85**	**-0.72**
GENN/	**-1.78**	0.00	0.13	**-0.03**	**-1.58**	4.54	0.39	**-0.19**	0.00	**-0.21**	0.03	**-0.74**
Retrained GENN	2.37	0.00	1.88	0.07	**-2.25**	0.14	0.10	0.57	0.00	0.34	0.12	0.00
	3.31	4.38	0.11	**-0.26**	**-0.15**	**-0.05**	1.16	0.29	1.04	2.09	1.71	2.79
GEKPLS/	1.28	0.07	0.02	**-0.04**	**-2.20**	**-0.03**	1.27	0.23	2.29	1.90	3.78	0.83
Retrained GEKPLS	4.66	0.08	2.84	1.18	0.83	0.05	3.02	0.23	0.71	2.24	1.95	3.83
Gradient Descent - 30 Relearning Iterations												
	Grid 50		**Random 50**		**LHS 50**		**Grid 50**		**Random 50**		**LHS 50**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
	-1.94	3.42	**-0.86**	0.15	0.00	2.04	0.47	0.10	2.28	0.70	**-0.06**	0.05
GENN/	1.73	0.00	1.86	**-0.44**	0.55	0.44	0.33	0.37	**-1.35**	0.10	0.22	**-0.18**
Retrained GENN	0.00	3.72	0.38	0.69	**-0.05**	**-0.80**	0.02	1.07	0.18	**-1.52**	**-0.67**	**-0.49**
	0.00	3.55	0.96	0.07	2.47	**-0.08**	0.90	0.35	2.96	0.98	3.18	2.51
GEKPLS/	4.25	0.00	**-2.02**	3.11	0.01	**-2.11**	1.37	**-1.08**	3.16	1.50	0.50	2.46
Retrained GEKPLS	3.82	**-0.18**	1.30	3.39	**-1.87**	**-0.15**	1.36	0.14	0.92	2.19	1.59	1.59
Adam - 10 Relearning Iterations												
	Grid 50		**Random 50**		**LHS 50**		**Grid 50**		**Random 50**		**LHS 50**	
	normal	log	normal	log	normal	log	normal	log	normal	log	normal	log
	1.59	0.00	**-1.62**	0.00	**-3.02**	1.85	**-0.12**	0.00	**-1.09**	**-0.53**	**-0.56**	**-0.49**
GENN/	0.00	3.47	**-0.04**	0.07	0.51	0.15	0.00	**-0.57**	0.91	2.24	0.00	**-1.09**
Retrained GENN	0.00	3.47	**-2.09**	**-0.32**	1.11	**-3.96**	0.40	**-0.11**	**-0.22**	0.01	0.08	0.05
	0.11	0.00	0.13	**-0.98**	0.06	0.17	0.48	0.22	1.88	0.80	1.20	3.12
GEKPLS/	0.12	0.36	1.50	0.07	1.47	1.55	0.71	0.30	1.49	0.88	2.41	2.20
Retrained GEKPLS	0.12	0.00	**-0.22**	**-0.01**	0.11	0.01	1.10	0.28	3.76	3.88	2.34	2.46

Table 5.8: Differences in the surrogate minima between initial model and retrained model. The surrogate minima were evaluated by predicting 100,000 sample points to find the smallest loss value. The bold numbers mark improvements.

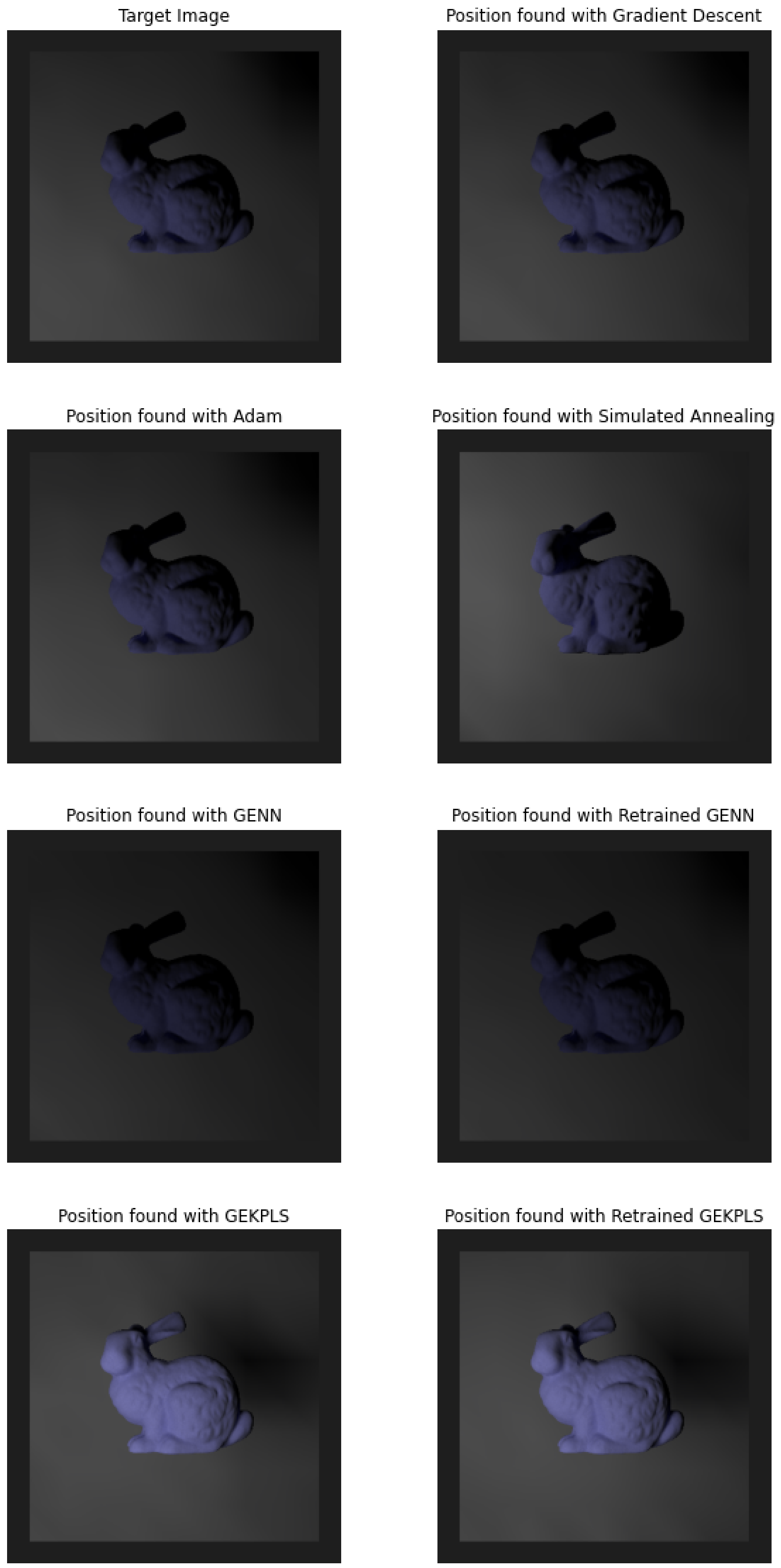
Target Image
Position found with Gradient Descent
Position found with Adam
Position found with Simulated Annealing
Position found with GENN
Position found with Retrained GENN
Position found with GEKPLS
Position found with Retrained GEKPLS

Figure 5.4: Rendered Rabbit scene (S1) with target illumination (**top left**) as ground truth and results of optimisation on Tamashii model (**top right**, **second row left** and **second row right**) as well as the surrogate minima before and after relearning. Sampling method: Grid, 50 sample points, logarithm-mapping

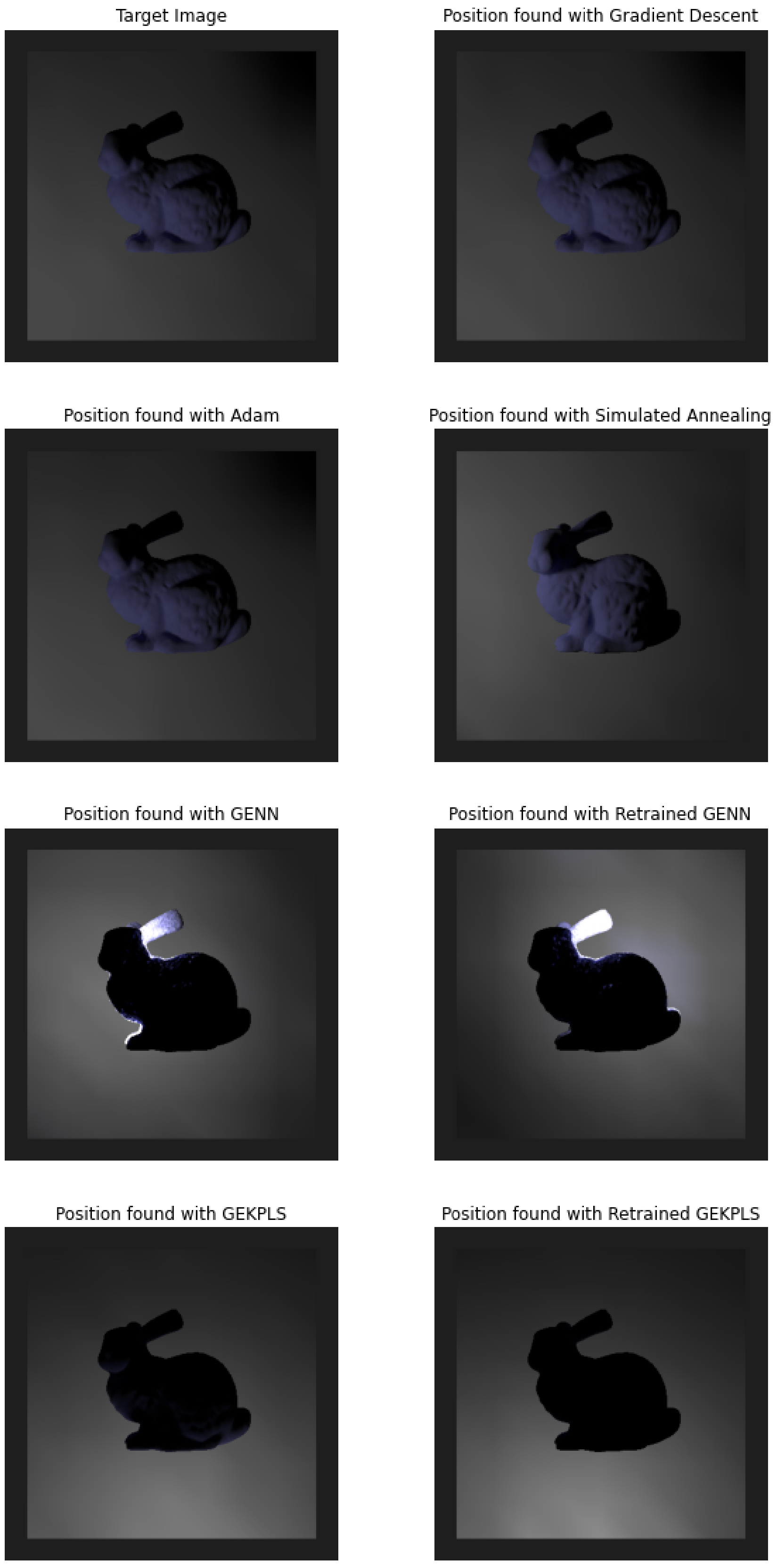
Target Image
Position found with Gradient Descent
Position found with Adam
Position found with Simulated Annealing
Position found with GENN
Position found with Retrained GENN
Position found with GEKPLS
Position found with Retrained GEKPLS

Figure 5.5: Rendered Rabbit scene (S1) with target illumination (**top left**) as ground truth and results of optimisation on Tamashii model (**top right**, **second row left** and **second row right**) as well as the surrogate minima before and after relearning. Sampling method: Grid, 100 sample points, no logarithm-mapping

5.2.6 Algorithm Reliability

Regarding the reliability of the individual algorithms, we find Adam to be the overall best fit for both models and retrained models in getting closest to the surrogate model's minimum. As done in the previous experiments, we perform three reruns of individual experiments to receive an average result. We compare the found minimum of each algorithm and calculate the Euclidean distance to the surrogate's minimum. This way, we can ensure that we examine only how well an algorithm performs in the task of finding the optimum and take the potential surrogate model inaccuracy out of the equation. For this experiment, we test all sampling methods (50, 100, and 500 sample points in combination with grid sampling, random sampling, and LHS) and logarithm mapping versus normal loss values. We can see how often an algorithm finds a good solution for the minimum and how close the solution is to the surrogate minimum. The proximity shows us how well an algorithm converges to the right area. At the same time, the percentage of good results gives insight into how reliable the method is working in different settings.

Furthermore, we evaluate the outcome of performing one optimisation starting with ten restarts and compare the Rabbit (S1, 3.1) with the Simple Office (S2, 4.2). Interestingly, some sampling methods seem to produce models where the algorithms work better than others. Unfortunately, although we are working with an average over three independent experiment runs, we can not point out one overall good solution for both the Rabbit and the Simple Office space. A particularly interesting takeaway is that a large number of sampling points does not improve the algorithm convergence on the models, which goes hand in hand with our previous findings. The average distance over all sampling methods for 50, 100, and 500 sampling points, respectively, is quite similar, leading to the conclusion that investing more time in sampling and building up a model with a larger sampling count does not benefit the algorithms. Besides Adam, we surprisingly see Simulated Annealing perform very well on the surrogate models after we adjusted the parameters. Gradient Descent and Stochastic Gradient Descent both tend to find more or less the same solution but are generally outperformed by Adam and Simulated Annealing (Table 5.9).

One outcome of this experiment is that performing ten restarts results in higher overall reliability, especially for Adam. Based on the averages of each algorithm per model (GENN, Retrained GENN, GEKPLS, Retrained GEKPLS) and sampling method, we also calculate the average of the algorithms in combination with each model over all sampling methods per sampling count. This gives us a good insight into how well an algorithm performs in combination with the respective model regardless of the sampling method.

Comparing the results of the overall distances between the surrogate model and the retrained surrogate model, it seems that retraining does not improve the surrogate model in terms of algorithm reliability. The convergence towards the surrogate minimum is overall slightly worse than it is on the initial model, which leads to the conclusion that here as well, spending the additional resources in retraining the model does not help to have a better optimisation setup. We occasionally see good fits between the retrained model and Adam or Simulated Annealing, but the result is only minimally better than the original model, so it seems reasonable to relinquish this improvement in regards to much better runtime.

We notice that there is no outstanding advantage in applying logarithm-mapping to the Tamashii model in terms of algorithm convergence. Normal sampling seems to yield better convergence on the Rabbit space, while logarithm-mapping has better results on the Simple Office space. The results are quite close, so one could argue that choosing one way over the other would give minimal advantages in one space and minimal disadvantages in the other and vice versa. So, on average, logarithm-mapping appears irrelevant when deciding for or against a sampling method.

There is also no trend visible on whether GENN or GEKPLS performs better in terms of algorithm reliability. We would argue that the algorithm convergence is slightly better on GENN in the Rabbit space and on GEKPLS in the Simple Office space. This also does not seem to contribute to a definite decision for one model over the other. Adam works equally well on both GENN and GEKPLS, so the final conclusion would be to always opt for Adam as the algorithm of choice for surrogate model optimisation regardless of the sampling plan or surrogate model.

Algorithm Reliability													
Model	**Algorithm**	**Sampling Method**						% Algorithm Dist < 0.5	% Algorithm Dist < 0.5 / Model	Average Model Dist	Best Algorithm / Model	Best Match with Dist	Best Sampling Method
RABBIT, 10 Restarts													
		Grid 50		**Random 50**		**LHS 50**							
		normal	log	normal	log	normal	log						
GENN	GD	3.21	2.11	1.61	3.34	5.42	4.59	0.00%	**0.39%**	**2.25**	**Adam**	**GENN + Adam (0.0)**	**Random 50 Normal**
	SGD	3.21	2.11	1.61	3.34	5.43	4.60	0.11%					
	Adam	0.00	0.00	1.25	0.05	2.34	0.00	0.89%					
	SA	1.65	2.00	1.45	0.05	4.69	0.00	0.56%					
Retrained GENN	GD	3.73	5.16	1.81	3.48	4.70	1.32	0.00%	**0.38%**	**2.31**	**Adam**	**Ret GENN + Adam (0.0)**	
	SGD	3.73	5.16	1.81	3.48	4.70	1.33	0.00%					
	Adam	0.03	0.00	0.05	0.03	3.91	0.02	0.89%					
	SA	2.61	4.81	0.05	0.06	3.33	0.02	0.61%					
GEKPLS	GD	2.92	2.05	3.69	4.32	5.55	5.16	0.06%	**0.36%**	**2.50**	**Adam**	**GEKPLS + Adam (0.05)**	
	SGD	2.93	2.05	3.69	4.32	6.04	5.15	0.06%					
	Adam	0.07	0.08	0.08	0.05	1.25	3.59	0.72%					
	SA	0.15	0.30	0.07	1.72	1.25	3.59	0.61%					
Retrained GEKPLS	GD	2.92	3.36	3.75	1.85	2.65	5.20	0.06%	**0.44%**	**2.00**	**Adam**	**Ret GEKPLS + Adam / SA (0.06)**	
	SGD	2.93	3.35	3.75	1.85	2.66	5.20	0.06%					
	Adam	0.08	0.08	0.06	0.07	0.06	3.71	0.89%					
	SA	0.07	0.54	0.06	0.07	0.06	3.78	0.78%					
	Average	**1.89**	**2.07**	**1.55**	**1.75**	**3.38**	**2.96**						
		Grid 100		**Random 100**		**LHS 100**							
		normal	log	normal	log	normal	log						
GENN	GD	0.07	0.06	2.87	3.77	3.91	2.51	0.39%	**0.56%**	**1.82**	**Adam, SA**	**GENN + SGD (0.05)**	**Grid 100 Normal**
	SGD	0.07	0.05	2.87	3.77	3.91	2.16	0.39%					
	Adam	0.07	0.07	2.51	2.39	4.23	0.06	0.72%					
	SA	10.07	0.07	2.51	2.40	4.23	0.18	0.72%					
Retrained GENN	GD	1.45	2.23	3.57	4.10	3.57	1.33	0.17%	**0.42%**	**1.98**	**Adam**	**Ret GENN + Adam (0.05)**	
	SGD	1.45	2.23	3.57	4.10	3.58	1.33	0.17%					
	Adam	1.45	0.06	3.35	0.05	1.87	0.06	0.72%					
	SA	1.45	1.14	3.34	0.23	1.87	0.06	0.61%					
GEKPLS	GD	2.23	2.98	2.49	4.40	2.96	5.69	0.00%	**0.31%**	**2.51**	**Adam, SA**	**GEKPLS + Adam / SA (0.05)**	
	SGD	2.36	2.098	2.49	4.40	2.97	5.69	0.00%					
	Adam	0.05	0.07	0.62	1.03	1.76	5.70	0.61%					
	SA	0.05	0.08	0.62	1.04	1.76	5.70	0.61%					
Retrained GEKPLS	GD	2.96	3.13	3.33	4.34	2.94	5.71	0.00%	**0.25%**	**2.91**	**Adam, SA**	**Ret GEKPLS + Adam / SA (0.05)**	
	SGD	2.96	3.13	3.34	4.34	2.94	5.71	0.00%					
	Adam	0.05	1.14	1.17	1.74	2.37	5.87	0.50%					
	SA	0.05	1.48	1.17	1.78	2.35	5.87	0.50%					
	Average	**1.06**	**1.31**	**2.59**	**2.74**	**2.95**	**3.33**						
SIMPLE OFFICE, 10 Restarts													
		Grid 50		**Random 50**		**LHS 50**							
		normal	log	normal	log	normal	log						
GENN	GD	2.23	4.04	3.90	1.22	4.46	0.07	0.11%	**0.26%**	**2.11**	**SA**	**GENN + SA (0.19)**	**LHS 50 Log**
	SGD	1.97	4.03	3.85	0.67	4.48	1.34	0.22%					
	Adam	1.67	2.53	2.10	0.39	2.19	0.06	0.33%					
	SA	1.95	3.19	3.03	0.19	0.72	0.28	0.39%					
Retrained GENN	GD	3.92	2.39	2.93	2.45	4.17	0.75	0.17%	**0.28%**	**2.23**	**Adam**	**Ret GENN + Adam (0.08)**	
	SGD	3.91	2.39	2.92	2.45	4.16	2.79	0.11%					
	Adam	2.84	2.71	1.27	0.08	1.73	0.64	0.44%					
	SA	1.19	2.80	1.96	1.67	0.54	0.86	0.39%					
GEKPLS	GD	0.93	1.73	0.06	3.54	2.19	2.59	0.50%	**0.64%**	**1.26**	**Adam, SA**	**GEKPLS + Adam / SA (0.05)**	
	SGD	0.93	1.73	0.06	3.55	2.19	2.59	0.06%					
	Adam	0.95	0.06	0.06	0.85	2.09	0.05	0.78%					
	SA	0.95	0.96	0.06	0.85	2.09	0.05	0.78%					
Retrained GEKPLS	GD	1.45	0.78	0.09	3.78	1.75	0.78	0.39%	**0.50%**	**1.31**	**Adam**	**Ret GEKPLS + Adam (0.06)**	
	SGD	1.78	0.78	0.09	3.86	2.00	1.16	0.39%					
	Adam	1.34	0.06	0.45	0.12	0.44	0.04	0.78%					
	SA	3.34	0.06	2.58	0.90	2.37	0.93	0.44%					
	Average	**1.96**	**1.83**	**1.59**	**1.67**	**2.37**	**0.94**						
		Grid 100		**Random 100**		**LHS 100**							
		normal	log	normal	log	normal	log						
GENN	GD	2.10	2.167	0.66	3.25	4.17	2.69	0.11%	**0.18%**	**2.40**	**Adam**	**GENN + Adam (0.07)**	**Random 100 Normal**
	SGD	2.11	2.67	0.66	2.67	4.17	3.56	0.11%					
	Adam	1.95	2.39	0.07	1.35	3.91	1.89	0.28%					
	SA	2.93	2.53	1.84	1.37	4.20	1.89	0.22%					
Retrained GENN	GD	1.59	1.39	2.04	3.72	4.48	3.86	0.00%	**0.15%**	**2.52**	**SA**	**Ret GENN + Adam / SA (0.6)**	
	SGD	1.60	1.98	2.04	3.39	4.48	3.86	0.00%					
	Adam	1.96	1.66	2.41	0.60	4.47	2.66	0.89%					
	SA	1.90	1.12	2.41	0.60	3.72	2.65	0.61%					
GEKPLS	GD	3.66	4.05	1.92	0.83	2.77	1.64	0.22%	**0.36%**	**2.50**	**Adam**	**GEKPLS + Adam (0.05)**	
	SGD	3.61	2.45	1.92	0.83	2.78	1.64	0.22%					
	Adam	3.43	1.86	2.02	0.06	2.77	0.04	0.50%					
	SA	3.42	2.86	1.84	0.05	2.78	0.04	0.50%					
Retrained GEKPLS	GD	3.62	2.96	1.70	4.22	2.40	4.57	0.16%	**0.25%**	**2.71**	**Adam**	**Ret GEKPLS + Adam (0.14)**	
	SGD	2.67	2.49	1.70	4.22	0.18	4.56	0.22%					
	Adam	0.58	0.28	2.10	0.11	0.14	2.09	0.61%					
	SA	5.95	3.00	2.10	4.18	5.18	4.50	0.00%					
	Average	**2.67**	**2.21**	**1.71**	**1.96**	**3.29**	**2.64**						

Table 5.9: Average distances of found minima by different algorithms to the surrogate minimum over three independent experiment repetitions based on S1 and S2.

5.3 Problems

The most prominent problem during our experiments is the inaccuracy of the built surrogate models. Overall, it was not possible to find a setup to create a reliable representation to perform our optimisation task on the surrogate models with a useful output. Although the surrogate model meets the requirements and speeds up the Tamashii optimisation process, we pay the price of not finding the target minimum anymore. Since we have quite varying optimisation spaces, it does not make sense to fit a model too closely to one specific scene, as the underlying idea was to utilise a surrogate model for many different scenes in Tamashii. We could improve our algorithms and models significantly by adjusting the hyperparameters, however, we can not pinpoint a universal sampling method.

To compare the efficiency of the applied methods, we track the time spent on the individual tasks. We hereby notice the rather long time span to perform the optimisation directly on the Tamashii model and the extensive time needed to sample a large amount of data. Building a surrogate model on a large data set requires a lot of time, and optimising on a GEKPLS built with such as well. One, therefore, has to weigh the pros and cons of efficiency against accuracy, a classic contradiction in optimisation. Efficiency can be achieved by working with a small dataset and performing the task on a surrogate model, accuracy can be obtained by directly optimising on the Tamashii model.

CHAPTER 6

Conclusion

During our experiments, we examined the effectiveness and efficiency of several different optimisation algorithms and optimisation methods. We performed experiments on the test functions Rosenbrock, sine, and sine with noise. In the early stages of our experiments, we tested the benefit of implementing several optimisation restarts over a single iteration and can say that several restarts evidently result in better outcomes regardless of the combination with a real function or surrogate model. We also implemented logarithm-mapping to improve our surrogate model in order to find representative approximations of the real functions. This drastically improved our surrogate model setup and the corresponding optimisation on these models.

Following the test functions, we continued our experiments on the Tamashii model. We quickly realised that creating a useful surrogate model was considerably more challenging than in the test function experiments. Moreover, it was not easy to determine when a good model approximation was found, as the visualisation of a 4-dimensional problem was not as straightforward as a 3-dimensional problem. We picked up the introduced visualisation idea by Forrester et al. [FSK08] and plotted the different models and scenes with slices through the space in every dimension with colour-mapped contour curves for the loss values. This gave us the option to gain more insight into the differences between the GENN versus the GEKPLS model or between real model values versus the logarithm-mapped ones. We also visualised the path of the best-found solution for each optimisation algorithm on the corresponding model to receive valuable insight into the behaviour of each algorithm.

Through this visualisation, we could observe that the target minimum was usually not found unless we applied the optimisation directly on Tamashii. This led us to test two different things. The first was the surrogate model accuracy, depending on different sampling strategies, to see how close the surrogate model's minimum would come to the target minimum; the second one was the algorithm reliability to figure out how likely an algorithm was to find the minimum altogether. As a result of these experiments, we found out that the surrogate minimum was, most of the time, not even close to the target minimum. Determining whether one model or sampling strategy was superior to the others, we, unfortunately, found pretty uniformly distributed results for both GENN or GEKPLS models as well as for logarithm-mapping and normal sample values. In regards to the surrogate minimum, we found GEKPLS having a marginal lead and for random sampling to produce a few better results. As for algorithm reliability, we see contradictory results for the different design scenes. Here, GENN seems to provide a better ground for convergence on the Rabbit scene, GEKPLS at the same time produces a better environment for the Simple Office space. Analogous, working with unmapped loss values yields better convergence in the Rabbit space, while logarithm mapping performs better in the Simple Office space. It becomes clear that none of the two models universally provide a good solution for the Tamashii problem. One main takeaway though is the reliable performance of Adam. Adam appears to find the target minimum on the Tamashii model itself but also the surrogate model's minimum. Although Gradient Descent also works very well on the Tamashii model, we find it not being reliable on the surrogate models and vice versa with Simulated Annealing.

Relearning did not improve our models significantly in correcting the initial inaccuracy from the first model training. Although we tested several improvements, we could not find a result that was good enough to vindicate the computational resources required to perform the relearning. In our experiments, we followed the approach of iteratively searching for the current minimum to add this found point to our training data. As Forrester et al. [FSK08] examine, there are other methods for infill points as well that could be tested. One example is error-based exploration, where points are added in the areas of highest uncertainty in the current prediction. Alternatively, playing with different termination criteria, like training the model until it is saturated or until a certain validation error threshold is reached, could be another approach. However, in this case, the increasing runtime for higher relearning counts has to be considered as well.

Surrogate models don't appear to be a useful extension to the Tamashii interface. Unfortunately, they could not match the results of direct optimisation and are, therefore, not applicable for real-time user interactions with optimisations in the Tamashii interface. If very short evaluation times are needed and accuracy is less important, one could consider defining individual surrogate model setups for each individual test scene. By this, it might be possible to determine the best sampling plan in combination with the best model choice and optimisation algorithm for every test scene to get an improved representation.

Alternatively, SMT provides more versions of surrogate models. Possibly, another surrogate model implementation might exist that fits the needs of the Tamashii scenes better than GENN and GEKPLS, although the reasoning behind using these was the benefit of incorporating the provided gradients. It does make sense to include the given gradient information for surrogate modeling to be able to build better overall models. If that was the path to be followed in the future, exchanging GENN for JENN could also be a possible solution. The polishing method provides a more delicate way to adjust a model closely to a specific problem.

We conclude that for now, performing the optimisation with Gradient Descent or Adam on the real Tamashii model gives the most accurate output. By including restarts, we can find the target minimum in a very high percentage of test cases. This justifies the longer duration required for direct optimisation in comparison to optimising on the surrogate model.

Overview of Generative AI Tools Used

Throughout this bachelor thesis, ChatGPT4o as AI tool was used to support the building of small pieces of code, but if applied, the generation was just used as a starting ground for further development or help to implement our own work.

ChatGPT was used to generate some plotting functions to visualise the created data. Here, the initial functions were created mainly to access the input data in the right form. After the initial setups, the functions were adjusted to our needs, i.e. changing the presentation to match the desired outcome or changing the input with further developed experiments.

ChatGPT generated a prototype for the pseudocode algorithms in this work based on our implementation. We also adjusted the output to our needs as well, for example, combining two algorithms into one or shortening it for a better overview.

We used ChatGPT to help write the mathematical formulas in Latex code or to fine-tune individual text passages. If used for text, we did not incorporate entire text passages generated by AI but instead used it to provide support for improvement and based on that improved selected wording.

List of Figures

List of Tables

List of Algorithms

Acronyms

Bibliography

[AAB$^+$16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016. `https://arxiv.org/abs/1603.04467`.

[BBOM16] Mohamed Amine Bouhlel, Nathalie Bartoli, Abdelkader Otsmane, and Joseph Morlier. Improving kriging surrogates of high-dimensional design models by partial least squares dimension reduction. *Structural and Multidisciplinary Optimization*, 53, 2016.

[Ber24] Steven H Berguin. Jacobian-enhanced neural networks. 2024. `https://arxiv.org/abs/2406.09132`.

[BM19] Mohamed A Bouhlel and Joaquim RRA Martins. Gradient-enhanced kriging for high-dimensional problems. *Engineering with Computers*, 35(1):157–173, 2019.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[DIA22] DIAL GmbH. Dialux evo 10.1, 2022. `https://www.dialux.com/en-GB/dialux`, Accessed: August 2022.

[FSK08] Alexander Forrester, Andras Sobester, and Andy Keane. *Engineering Design Via Surrogate Modelling: A Practical Guide.* John Wiley & Sons Ltd, 2008.

[GCD$^+$10] Dirk Gorissen, Ivo Couckuyt, Piet Demeester, Tom Dhaene, and Karel Crombecq. A surrogate modeling and adaptive sampling toolbox for computer based design. *Journal of Machine Learning Research*, 11:2051–2055, 2010.

[Kaj86] James T. Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20(4):143–150, 1986.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[KGJV83] Scott Kirkpatrick, Charles D. Gelatt Jr, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[LHEN$^+$24] Lukas Lipp, David Hahn, Pierre Ecormier-Nocca, Florian Rist, and Michael Wimmer. View-independent adjoint light tracing for lighting design optimization. *ACM Trans. Graph.*, 43(3), 2024.

[LHJ19] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Trans. Graph.*, 38(6), 2019.

[Lu22] Jun Lu. Gradient descent, stochastic optimization, and other tales. *ArXiv*, abs/2205.00832, 2022. `https://arxiv.org/abs/2205.00832`.

[Mat63] Georges Matheron. Principles of geostatistics. *Economic Geology*, 58(8):1246–1266, 1963.

[MRR$^+$53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[NDSRJ20] Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. Radiative backpropagation: an adjoint method for lightning-fast differentiable rendering. *ACM Trans. Graph.*, 39(4), 2020.

[NDVZJ19] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: a retargetable forward and inverse renderer. *ACM Trans. Graph.*, 38(6), 2019.

[PGM$^+$19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[Pre23] Matthias Preymann. Scripting automation for tamashii, 2023. Bachelor Thesis.

[PVG$^+$18] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel

Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python, 2018. `https://arxiv.org/abs/1201.0490`.

[RBLR04] Carl Rasmussen, Olivier Bousquet, Ulrike Luxburg, and Gunnar Rätsch. Gaussian processes in machine learning. *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures, 63-71 (2004)*, 3176, 2004.

[Rel22] Relux Informatik AG. Reluxdesktop, 2022. `https://reluxnet.relux.com/en`, Accessed: August 2022.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Ros60] Howard. H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, 1960.

[SLB$^+$24] Paul Saves, Rémi Lafage, Nathalie Bartoli, Youssef Diouane, Jasper Bussemaker, Thierry Lefebvre, John T. Hwang, Joseph Morlier, and Joaquim R.R.A. Martins. Smt 2.0: A surrogate modeling toolbox with a focus on hierarchical and mixed variables gaussian processes. *Advances in Engineering Software*, 188:103571, 2024.

[SMT22a] SMT Toolbox. *Surrogate Modeling Toolbox: GEKPLS*, 2022. `https://smt.readthedocs.io/en/latest/_src_docs/surrogate_models/gpr/gekpls.html`,Accessed: 2022-08-20.

[SMT22b] SMT Toolbox. *Surrogate Modeling Toolbox: GENN*, 2022. `https://smt.readthedocs.io/en/latest/_src_docs/surrogate_models/genn.html`, Accessed: 2022-08-20.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press, 2014.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. *University of Toronto, Technical Report*, 6, 2012.

[Wei09] Thomas Weise. *Global Optimization Algorithms - Theory and Application.* Self-Published, second edition, 2009. `https://api.semanticscholar.org/CorpusID:3079252`.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[WM97] David Wolpert and William Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.

[WZCN09] Thomas Weise, Michael Zapf, Raymond Chiong, and Antonio Nebro. *Why Is Optimization Difficult?*, volume 193, pages 1–50. 2009.