Informatics

Bidirectional MATLAB/C++ Interface for Lighting Design Optimization

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Matthias Zezulka
Matrikelnummer 11914298

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 9. November 2023

Matthias Zezulka	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Bidirectional MATLAB/C++ Interface for Lighting Design Optimization

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Matthias Zezulka
Registration Number 11914298

to the Faculty of Informatics

at the TU Wien

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

Vienna, 9th November, 2023

Matthias Zezulka   Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Zezulka

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

Wien, 9. November 2023

Matthias Zezulka

Danksagung

Als Erstes möchte ich mich bei meinen Eltern, aber auch meiner restlichen Familie, bedanken, da sie mir dieses Studium nicht nur ermöglicht haben, sondern mich auch während dessen tatkräftig dabei unterstützt und bestärkt haben.
Weiters möchte ich einen besonderen Dank an meinen Betreuer David Hahn aussprechen, der mir dieses interessante Thema ermöglicht hat und von Beginn der Phase der Ideenfindung, über die Entwicklung bis hin zum Schreiben der Arbeit immer mit ausführlichem Feedback und konstruktiven Ansätzen auf freundschaftliche Weise unterstützt hat.
Nicht zuletzt möchte ich mich noch bei allen Freunden bedanken, die mich während des Studiums immer wieder motiviert und bestätigt haben.

Acknowledgements

First of all, I would like to thank my parents, but also the rest of my family, as they not only made this study possible for me, but also actively supported and encouraged me during it.
Furthermore, I would like to express my special thanks to my supervisor David Hahn, who made this interesting topic possible for me and always supported me in a friendly manner with detailed feedback and constructive approaches from the beginning of the brainstorming phase, through the development to the writing of the thesis.
Last but not least, I would like to thank all my friends who have always motivated and encouraged me during my studies.

Kurzfassung

Die Beleuchtung von (virtuellen) Raum ist ein wichtiger Aspekt unseres täglichen Umfelds. Sie ermöglicht nicht nur kreativen Ausdruck, sondern ist oft auch ein notwendiger Faktor in professionellen Arbeitsumgebungen und künstlerischen Produktionen. Aufgrund der hohen Komplexität dieses Problems werden aktuelle Lösungen jedoch in der Regel in leistungsorientierten Programmiersprachen erstellt. Diese bieten einerseits eine detaillierte Low-Level-Ansicht der Anwendung, erschweren andererseits aber die Entwicklung von neuen Funktionalitäten. Dadurch wird es schwerer, Algorithmen bzw. bestimmte Programmteile gegen andere auszutauschen. Diese Arbeit baut auf dem bereits existierenden C++-Rendering-Framework *Tamashii* auf (Lipp et al. 2023 [20]), welches eine Blickwinkelunabhängige und gradientenbasierte globale Beleuchtungsdesign-Optimierung realisiert. Wir zeigen in dieser Arbeit eine Methode, MATLAB-Funktionen in den Optimierungsprozess zu integrieren, um somit nicht nur die Entwicklung von Optimierungsalgorithmen zu vereinfachen, sondern auch den Zugriff auf die bestehende MATLAB-Codebasis und numerische Analysewerkzeuge zu ermöglichen. Wir implementieren daher eine bidirektionale MATLAB/C++-Schnittstelle für den Austausch von Optimierungsdaten zwischen dem Rendering-Prozess und dem MATLAB-Prozess. Um diese Funktionalität zu ermöglichen, nutzen wir die MATLAB Engine API für C++ und die MATLAB MEX API, die beide nativ in MATLAB enthalten sind. Darüber hinaus implementieren wir einen Mechanismus für die prozessübergreifende Kommunikation unter Verwendung von Windows Named Pipes und einem selbst implementierten Kommunikationsprotokoll.
Darüber hinaus werden in dieser Arbeit auch verschiedene Optimierungsmethoden und die Verwendung der Surrogate-Based Optimization (SBO) für das globale Beleuchtungsdesignproblem kurz diskutiert. Wir zeigen, dass unsere Methode sehr leistungsfähig ist und evaluieren sie gegen native C++-Implementierungen an zwei Testszenen. Dies machen wir, indem wir nicht nur Optimierungsmethoden über die Schnittstelle testen, sondern auch das einfache Rendering neuer Beleuchtungskonfigurationen. Die Testergebnisse zeigen auch, dass die aktuelle MATLAB-native SBO-Implementierung in bestimmten Szenen hohe Effizienz für das von *Tamashii* realisierte Optimierungsproblem erbringen kann. Schließlich präsentieren wir einige andere Vorteile, wie die verbesserte Benutzerfreundlichkeit und den besseren Einblick in Optimierungsmethoden, die wir durch die Integration von MATLAB in *Tamashii* erzielen konnten.

Abstract

The lighting design of (virtual) space is an important aspect of our daily environment. It not only allows for creative expression but is often a necessary asset in professional work environments and artistic productions. However, due to the computational complexity of this problem, current solutions are usually built in performance-oriented programming languages that offer a detailed low-level view of the application on the one hand but do not allow for fast development and easy exchange of algorithms on the other. This work builds on the already existing C++ rendering framework *Tamashii*, proposed by Lipp et al. in 2023 [20], which offers view-independent and gradient-based global lighting design optimization. We propose a way to integrate MATLAB functions into the optimization process in order to not only allow for easier development of optimization algorithms but also enable access to MATLAB's existing code base and numerical analysis tools. We therefore implement a bidirectional MATLAB/C++ interface for exchanging optimization data between the rendering process and the MATLAB process. In order to achieve this functionality, we use the MATLAB Engine API for C++ and the MATLAB MEX API, which are both natively contained within MATLAB. Further, we implement a mechanism for inter-process communication using Windows Named Pipes and a custom communication protocol.
In addition, this work also briefly discusses various optimization methods and the use of Surrogate-Based Optimization (SBO) for the global lighting design problem. We show that our method achieves great performance and evaluate it against plain C++ implementations on two test scenes by not only testing optimization methods via the interface but also testing simple rendering of new lighting configurations. The test results also show that MALTAB's current SBO implementation can bring good performance to the optimization problem we encounter in *Tamashii* in certain scenes. Lastly, we discuss the increased usability and insight into optimization methods achieved by integrating MATLAB into *Tamashii*.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Background **3**
2.1 Optimization Problem 3
2.2 Tamashii 4
2.3 MATLAB 5

3 Optimization Algorithms **11**
3.1 Gradient Descent 12
3.2 ADAM 14
3.3 (L-)BFGS 16
3.4 CMA-ES 18
3.5 Surrogate Based Optimization 20

4 MATLAB/C++ Callback Interface **27**
4.1 Implementation 27
4.2 Using the Interface 37

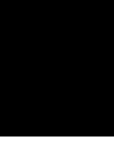
5 Evaluation and Comparison **41**
5.1 Evaluation of the bidirectional C++/MATLAB Interface 42
5.2 Comparison of Optimization Algorithms 47

6 Conclusions **51**

Acronyms **53**

Bibliography **55**

CHAPTER 1



Introduction

Lighting plays an important role in our daily lives, as it not only influences the way we perceive (virtual) space but also has a significant impact on our health and overall wellbeing. Consequently, the demand for computational lighting design arises, especially in professional work environments and artistic productions, which is, however, not easily solvable and oftentimes requires high-performance implementations with support of the graphics processing unit (GPU) for accelerated computation. Performance is therefore an enduring concern when developing software and can significantly influence the choice of programming language used. Currently, most programs that demand high performance or want to access GPU-accelerated computation via GPU-APIs such as OpenGL or Vulkan are written in C++ since it offers a great low-level view of the program. This approach allows for very efficient software; however, this way of programming can also significantly slow down the process of creating new functionality. Most programs that are not in need of high performance are therefore usually written in script-oriented languages, such as MATLAB or Python. These languages offer a high-level view of the program and allow for very fast development of functionality. This work builds on the already existing C++ rendering framework, *Tamashii*, developed at Vienna University of Technology (TU Wien). *Tamashii* offers global lighting design optimization of virtual scenes, i.e., it tries to find an optimal configuration of light sources in a scene. It does this by comparing the current light configuration with a user-defined target and then trying to minimize the difference between both. This optimization problem, i.e., finding optimal lighting parameters, is what we will consider the primary optimization problem in this thesis. In particular, we focus on the view-independent interactive adjoint light tracing (IALT) method proposed by Lipp et al. in 2023 [20]. The authors presented a method of computing gradient information for the current light configuration by using a GPU ray tracing-based method in conjunction with an adjoint method for gradient computation. The objective function that is used for this adjoint method is the lighting evaluation function that is implemented in C++ within *Tamashii*. Therefore, providing

this gradient information allows us to use many existing first-order optimization methods.

Currently, only a few optimization algorithms have been implemented in *Tamashii*. Though there are some libraries in C++ that can be used for building optimization methods, this process is usually still very time-consuming and does not allow for effortless switching of the algorithm used. Therefore, MATLAB promises easier development and a larger collection of already existing methods.

Conversely, running everything in MATLAB would also not be possible due to the needed GPU support and other performance-critical parts such as light tracing. This work proposes a way to integrate MATLAB scripts into the existing C++ process. We will also discuss the use of MATLAB black box optimizers in the global lighting design optimization problem, since MATLAB currently offers not only various toolboxes and out-of-the-box solutions for such problems but also more advanced learning and non-gradient-based methods. The most prominent toolboxes include the *Global Optimization Toolbox*, the *Statistics and Machine Learning Toolbox*, and the *Deep Learning Toolbox*. In particular, we will discuss MATLAB's surrogate model optimization (*surrogateopt*) found in the *Global Optimization Toolbox*.

In order to achieve the integration of MATLAB into the optimization process of *Tamashii*, we will create a bidirectional MATLAB/C++ interface, i.e., (1) a MATLAB interface for C++ and (2) a C++ callback interface for MATLAB, via which optimization data can be sent. On a technical level, the C++ application will then be able to run a MATLAB file that makes a call to one of the available optimization algorithms (e.g., *surrogateopt*) and returns the (local) optimum once it terminates. However, for this exchange to work, the MATLAB optimization function needs an objective function, for which the minimum has to be found. Since this objective function is implemented in the C++ rendering framework, the interface has to function both ways in order to call the objective function as a call-back function directly from MATLAB.

After the background chapter, this work will first start with a short overview of prominent optimization methods and then discuss the implemented MATLAB interface in more detail. The last part of this work explores the effective use of the interface by testing a number of modern approaches to the optimization problem. As mentioned above, the focus is thereby put on Surrogate-Based Optimization (SBO) which yields promising characteristics for the particular optimization problem in the *Tamashii* renderer. SBO is a more complex approach than traditional optimizers as it involves first constructing an approximate model of the objective function and then minimizing it with traditional methods; however, in this way, it is expected to need fewer objective function evaluations, especially when compared to other conventional methods. Further, we will also discuss the performance and usability of the interface as well as the performance of SBO against other optimization methods.

CHAPTER 2

Background

In this chapter, we will briefly give a background to the problems we will encounter throughout this work when implementing and using the bidirectional interface for the *Tamashii* renderer.

2.1 Optimization Problem

This thesis will mainly focus on the lighting design optimization problem that is given by the IALT implementation in the *Tamashii* renderer. For completeness, we will define the problem and the terminology that we will use in more detail in this section. The problem evolves around optimizing an objective function; let's call it J. This objective function is the lighting evaluation function that is provided by *Tamashii*. It works by rendering the illumination for the whole scene and comparing the result to a user-specified target. It then returns an objective value depending on how close the solution is to the given target, along with the gradient, i.e., the derivative of the objective function with respect to all parameters of the lighting configuration. As briefly mentioned in the introduction, this gradient is computed via an adjoint method (see [20]). We refer to the lighting configuration as the parameter vector; let's call it θ, which is a vector that consists of different light parameters that ought to be optimized (e.g., position, color, intensity, etc.). We define the size of this parameter vector, i.e., the dimensionality of the optimization problem, as d. Formally, we then want to find a θ^* that minimizes J, i.e.,

$$\theta^* \;=\; \underset{\theta \,\in\, \mathbb{R}^d}{\arg\min}\; J(\theta)$$

where $J : \mathbb{R}^d \to \mathbb{R}$. Note that for certain optimization algorithms (e.g., BFGS), we have to assume some notion of smoothness for J, which we, however, cannot generally guarantee. This is due to the fact that light sources can always move behind objects

in the scene, thus causing a discontinuous jump in the illumination. Further, for SBO, we will have to convert the unconstrained optimization problem to a constrained one, i.e., every parameter θ_i of the parameter vector is constrained to a restricted interval of values. Formally, this changes the optimization problem to

$$\begin{aligned} \theta^* &= \underset{\theta \in \mathbb{R}^d}{\arg\min} \; J(\theta) \\ \text{s.t.} & \quad l_i \leq \theta_i \leq u_i \\ & \quad \text{for } i = 1, ..., d \end{aligned}$$

where l_i and u_i are the lower and upper bounds for the parameter θ_i, respectively. Although SBO is usually considered a global optimization method, it needs these parameter bounds in order to achieve a reasonably detailed surrogate within the optimization process. Furthermore, searching for a minimum outside the sampled parameter range can be considered meaningless since there is no information about the underlying objective function. However, for the particular optimization problem that we encounter in *Tamashii*, these parameter bounds can easily be chosen by considering a reasonable domain for the specific parameters. Position parameters, for example, can be limited to a maximum distance from the scene; rotation parameters can be bounded between 0 and 2π; and intensity parameters can be restricted between 0 and a maximum intensity value that can realistically be reached by a physical light source.

2.2 Tamashii

The *Tamashii* renderer is a physically based renderer that is currently written completely in C++ on the central processing unit (CPU) side and GLSL/HLSL on the GPU side. The application is being developed for research purposes at the Vienna University of Technology and offers a variety of features. In its most basic functionality, it allows for virtual scenes to be loaded and displayed. Further, it is also possible to add light sources to the scene and edit the parameters of their configuration (e.g., intensity, color, etc.). As already mentioned, the focus of this thesis is now on the interactive adjoint lighting optimization feature, which we will briefly discuss in this section. For this feature to work, it is necessary to define a target for the scenes that are loaded into the application, which will serve as a reference for what an ideal lighting configuration of the scene should look like. This target is defined by vertex colors, which means it is either possible to draw the target before loading the scene into *Tamashii* in any 3D software (e.g., Blender) or to directly draw it in the application using *Tamashii's* drawing tools. An example of these targets for specific testing scenes can be seen in Figure 5.1b and Figure 5.2b, respectively. Once the scene is loaded and a target is defined, it is possible to choose an optimization algorithm for finding the best-matching lighting configuration for the scene. The algorithms available at the time of writing are

- Gradient Descent,

- ADAM,
- L-BFGS (currently built on LBFGSpp [1]),
- CMA-ES (based on Hansen's implementation [2]),

and are currently entirely implemented in C++. The IALT implementation of the renderer then provides a method of evaluating the current lighting configuration with respect to the target and not only returns a single objective function value but also a gradient, which is computed via an adjoint rendering method [20]. This makes it possible for the optimization algorithms to find a (local) minimum that, ideally, matches the lighting of the virtual scene as close as possible to the target. The result of the optimization can then be viewed in the renderer and exported as a glTF file. The default user interface of the application can be seen in Figure 2.1.

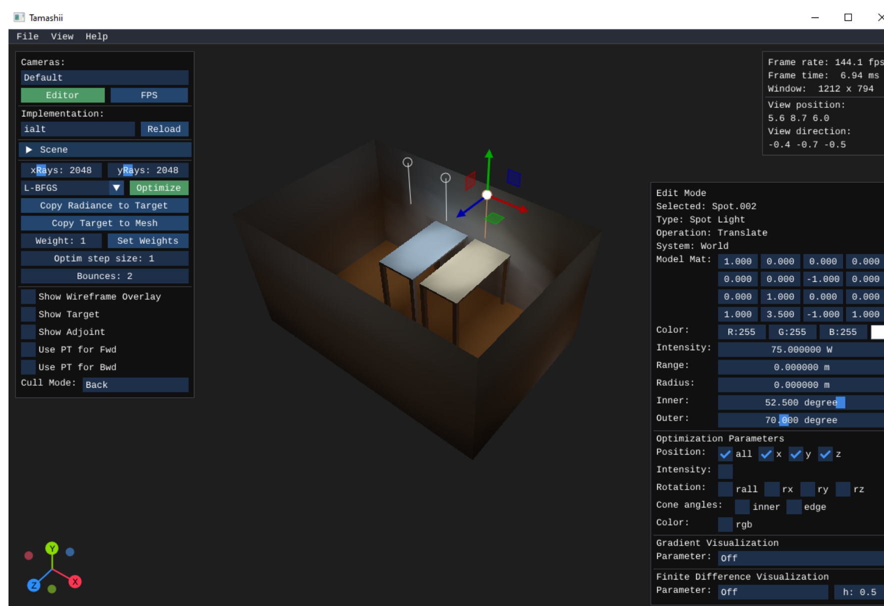


Figure 2.1: *Tamashii's* User Interface

2.3 MATLAB

MATLAB is currently one of the most used script-based programming languages and offers a wide variety of features, especially for numerical analysis. Further, MATLAB's File Exchange functionality allows for a very fast exchange of custom code and other resources. This richness of functionality in general offers a variety of ways to achieve a desired behavior; therefore, we will briefly discuss the basic MATLAB concepts that we will use to implement the bidirectional MATLAB/C++ interface (see Chapter 4) in this section. The approach we choose in this work is to use the MATLAB Engine API for

C++ in order to call MATLAB functions from C++ [3] and the C++ MEX API in order to call C++ functions from MATLAB [4]. Lastly, the MATLAB Data API for C++ will be used in both "directions" to facilitate data conversion between C++ and MATLAB [5]. The following sections will cover the use of these functionalities in more detail.

2.3.1 Scripts and Function Files

Before discussing the implementation aspects of the used MATLAB features in more detail, however, we will briefly introduce the concept of MATLAB script files (scripts) and MATLAB function files (functions), since these terms are often used in an interchangeable manner. Both functions and scripts are used to store sequences of commands in a code file. Scripts, however, are the simpler type since they store commands exactly as they are written on the command line. They also have no separate workspace and operate only in the base workspace. Functions, on the other hand, have their own workspace and are generally more flexible and easily extensible. Further, they contain input and output arguments as well as a function name. Note that the topmost function in a MATLAB function file is considered the only function with global access and that its name has to match the file name. The function file then consists of this entry function and an arbitrary number of local functions. Scripts can easily be converted into functions by surrounding them with a function declaration. This declaration includes the function keyword, the names of input and output arguments, and the name of the function [6]. Note that we are not able to call scripts but only functions with the approach we have taken, which therefore requires scripts to be surrounded by function declarations. Consequently, the rest of this thesis will only refer to MATLAB function files and assume that every script that ought to be run was converted to a function file beforehand. Note that we are also making assumptions about the input arguments and return values for these MATLAB functions in order to use them for optimization (see Subsection 4.1.5 for details).

2.3.2 Calling MATLAB from C++

MATLAB natively offers multiple external language interfaces that can be used to call MATLAB functions from other programming languages. Since *Tamashii* is implemented in C++, we choose the MATLAB engine API for C++, which offers a variety of methods to start and interact with MATLAB processes from a C++ process; however, in this section, we will focus primarily on the basic aspects that we will use to implement the interface. The first step in using the API is to start a new MATLAB instance from the C++ process. Since this step can take some time, we only do it once at startup and save the pointer to the engine object in a singleton class. This pointer can then be used to interact with the MATLAB instance in many ways. The most important one for most cases is to call a MATLAB function, which can be done synchronously or asynchronously. These functions can either be native MATLAB functions (e.g., *sqrt*) or custom global MATLAB functions as described in the previous subsection. In order to find custom functions, MATLAB uses its own search path configuration to look for these functions in the specified directories on the system.

Synchronous and asynchronous in this case means that the C++ process will either wait for the MATLAB function to terminate (synchronous) or continue its execution (asynchronous) and retrieve the result at a later time. This function call then returns either a MATLAB Data object if started synchronously or a future Matlab Data result if started asynchronously. An example of both cases can be seen in Listing 2.1. As mentioned above, the MATLAB Data API for C++ can then be used in both directions to convert data between C++ and MATLAB.

Listing 2.1: Starting a MATLAB instance and call function

```
#include "MatlabEngine.hpp"
#include "MatlabDataArray.hpp"

void callMatlabFunction()
{
    // Start MATLAB engine synchronously
    std::unique_ptr<MATLABEngine> matlabPtr = matlab::engine::startMATLAB();

    // Create input arguments for function
    matlab::data::ArrayFactory factory;
    matlab::data::Array args = factory.createScalar<int16_t>(1);

    // Call function synchronously
    matlab::data::Array result = matlabPtr->feval(u"functionName", args);

    // Call function asynchronously
    matlab::execution::FutureResult<matlab::data::Array> futureResult = matlabPtr->
        fevalAsync(u"functionName", args);

    // Wait for asnychronous results
    matlab::data::Array result = futureResult.get();
}
```

2.3.3 MEX Files

MEX (or MATLAB executable) files are written in C++ and then compiled with the MEX compiler, which compiles and links one or more C++ source files written with the C++ MEX API and MATLAB Data API for C++ into a binary MEX file. Both the C++ MEX API and MATLAB Data API for C++, as well as all other supported external language interfaces, are native MATLAB features that are contained within the base installation package of MATLAB. Note that we used CMake as a build and package management tool in this project, which offers a specific MEX target functionality that automatically compiles source files with the MEX compiler during the build process [7] (see Subsection 4.1.1). This MEX file can then simply be called like any other function in MATLAB. Further, it is then possible to create a function handle for this file, which can then be passed to any optimization algorithm as the function handle for an objective function. An example of this can be seen in Listing 2.2.

Listing 2.2: Calling a MEX file from C++

```
% calling the MEX function
result = mexFileName(parameter);

% creating a function handle for the MEX file
mexFunctionHandle = @(x) mexFileName(x);
```

The C++ source file used for the MEX compilation has to follow a certain structure in order to implement a MEX function. The function has to be implemented as a class called *MexFunction*, which is a subclass of `matlab::mex::Function` and overrides the function call operator, `operator()`. This implementation then creates a function object that is callable like any other function in MATLAB. Note that calling the MEX function from MATLAB instantiates this object; however, it will maintain its state across subsequent calls to this function and is therefore only instantiated once. The most basic form of a MEX function can be seen in Listing 2.3.

Listing 2.3: Structure of a MEX file

```
#include "mex.hpp"
#include "mexAdapter.hpp"

class MexFunction : public matlab::mex::Function {

private:
matlab::data::ArrayFactory factory;

public:
void operator()(matlab::mex::ArgumentList outputs, matlab::mex::ArgumentList inputs)
      {
        // Access input
        matlab::data::Array input1 = inputs[0];
        matlab::data::Array input2 = inputs[1];

         ...

        // Create output
        outputs[0] = factory.createScalar<int16_t>(1);
        outputs[1] = factory.createArray({ 1,2 }, { 1,2 });
      }

};
```

We have now given a brief overview of the MATLAB and C++ functionalities that we will use in this work to implement the bidirectional MATLAB/C++ interface. In particular we focused on the MATLAB Engine API for C++ in order to call MATLAB functions from C++ and the C++ MEX API to call C++ functions from MATLAB. Further, we introduced the basic functionality of *Tamashii* and defined basic concepts such as its

optimization problem and MATLAB function files. With this we will now conclude the background chapter and continue by discussing various optimization methods in more detail. Afterwards, we will present the implementation of our interface and further its performance evaluation.

CHAPTER 3

Optimization Algorithms

In this chapter, different approaches to finding the minimum of an objective function, i.e., solving a (non-)convex optimization problem, are discussed. Since we have no information about the convexity of our objective function, we have to assume it is non-convex. Consequently, we have to assume that solvers in general only return a local minimum, which differs from the global minimum in a way that its function value is only smaller than nearby points and that there may be another distant point with a smaller function value. In contrast, the global minimum is a point whose function value is smaller than all other feasible points. When discussing local and global optimization, one has to also think about starting points for local solvers since the local minimum that is returned by the optimization method in general differs for different starting points. However, choosing a "good" starting point, i.e., a starting point from which the optimizer finds a local minimum that has a smaller function value than other local minima, strongly depends on the structure of the underlying objective function and is therefore usually decided empirically for specific problems. Next to manually setting the starting point for the local optimizer, it is also common to randomly sample a point in the search space. This solves the problem of empirically finding a well-suited starting point on the one hand, but it also raises the question of what sampling method to use on the other. However, due to the limited scope of this work, we will not discuss this topic any further (see Chapter 7 in [25] for details). Note that we will use the same starting point in each scene for every (local) optimization method that we will use in Chapter 5 since the starting point in *Tamashii* is always given by the initial parameter layout of the scene that is specified by the user.

This work focuses on giving a deeper understanding of Surrogate Based Optimization, however, a short overview of prominent optimization algorithms is also given. Specifically, algorithms that are being compared in solving the problem given by the differentiable rendering framework (see Chapter 5) will be discussed in this chapter.

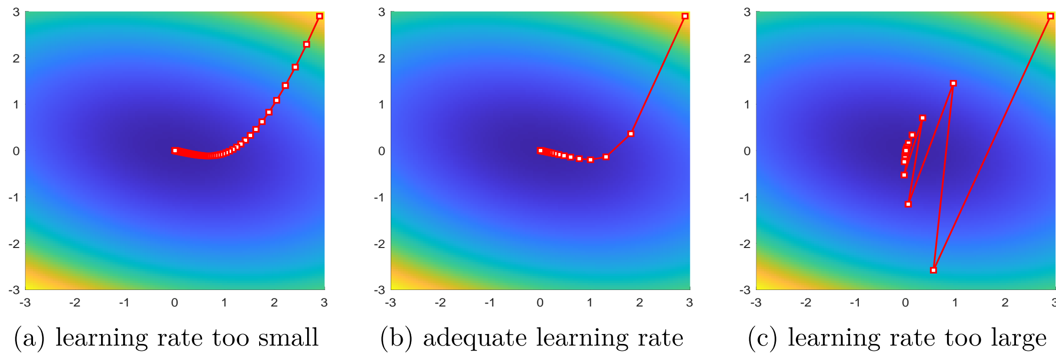
3.1 Gradient Descent

Gradient Descent (GD) is one of the most common and intuitive algorithms used for finding local minima of continuously differentiable functions. Further, it is a very popular method for optimizing hyper-parameters in neural networks and other machine learning tasks. The basic idea of this method is to always take the direction of the steepest descent in order to reach the function minimum as fast as possible. More formally, in its most basic version, we minimize an objective function given as $J(\theta)$ parameterized by a parameter vector $\theta \in \mathbb{R}^d$. As mentioned above, we then want to update this parameter vector in the direction of the steepest *descent* in order to reach the minimum as fast as possible. Since the direction of the steepest *ascent* corresponds to the objective function gradient $\nabla_\theta J(\theta)$ at this point, we update the parameters in the opposite direction of this gradient. This process then continues until any of the termination criteria are reached. Termination criteria largely depend on the kind of problem and can take many forms; nevertheless, we will briefly list the most prominent ones used with GD and similar optimization approaches. The algorithm usually terminates if a maximum number of iterations is reached, since this most likely indicates that the algorithm has failed to converge. Further, a lower threshold for the objective improvement and the gradient are often used as termination criteria since a small improvement (with non-zero gradient) can indicate that the algorithm is stuck near a non-smooth point of the objective function or the learning rate is too large to achieve any improvement. A gradient sufficiently close to zero, on the other hand, can indicate that the algorithm has converged to a local minimum (or saddle point). Additionally, the distance of the steps taken is also a termination criterion for many algorithms, i.e., if the distance of the steps taken is too small the algorithm terminates. Note that with GD this is equivalent to a gradient close to zero but it can, however, have a different interpretation for other algorithms. In order to determine the step size of the updates for GD, an additional learning rate hyper-parameter η is introduced. For completeness, we will define hyper-parameters as parameters that are set before the optimization process begins and that directly affect how well an optimization process will perform.

All things combined, we then have the parameter update rule for the vanilla Gradient Descent:

$$\theta = \theta - \eta \, \nabla_\theta J(\theta).$$

It can be shown that, when choosing an adequate learning rate hyper-parameter η, this version of GD converges to the global minimum for convex objective functions and to the local minimum for non-convex ones. However, we also encounter challenges when following this approach. The first problem is the question of choosing the learning rate η with respect to the objective function $J(\theta)$, as illustrated in Figure 3.1. If the learning rate is too small, convergence can become very slow, and the algorithm may terminate due to stop criteria before finding the actual (local) minimum. On the other hand, if the learning rate is too large, it may even hinder convergence by causing the parameters to fluctuate around the (local) minimum. This behavior may even lead to divergence



(a) learning rate too small (b) adequate learning rate (c) learning rate too large

Figure 3.1: Vanilla GD Example: Here we compare the trajectories of GD for different learning rate configurations on the "Polynomial" test function (as in Eq. (3.1)). We observe slow convergence for too small learning rates and oscillations around the minimum for too large learning rates.

in extreme cases. There are, however, multiple approaches to tackling this problem (e.g., line-search or learning rate schedules), which makes finding the correct step size an optimization problem itself. One approach of characterizing how well a step size is suited is formulated by the Wolfe conditions [31, 32], which are a set of inequalities for performing inexact line-search in order to find an acceptable step size in each iteration. Since these equations are mostly used in quasi-Newton methods, we will discuss them in more detail in Section 3.3. As already mentioned, another promising approach is to use (adaptive) learning rate schedules. Currently, the most popular results of this line of research is adding momentum. The idea of momentum and adaptive learning rate methods in general is to accelerate the descent of GD by adding a fraction γ of the update vector of the previous timestep to the current update vector:

$$v_t = \gamma v_{t-1} - \eta \, \nabla_\theta J(\theta),$$
$$\theta_t = \theta_{t-1} - v_t,$$

where γ is usually set to 0.9 or a similar value. Note that this momentum term increases if the gradient of the parameter vector has no large derivations of its direction over multiple timesteps and decreases if the gradient has rapid changes in direction. This not only results in an accelerated descent if the gradient does not change over some iterations, but also brings stability against small local minima by simply ignoring small bumps due to the momentum term. Although this method generally allows for more robust and faster convergence, it can also introduce oscillations due to the momentum term. This problem is again tackled by various adaptations, the most prominent one being ADAM, which will be discussed in Section 3.2.

There exists, however, another problem with the vanilla GD approach that becomes especially problematic in hyper-parameter tuning in machine learning applications (i.e.,

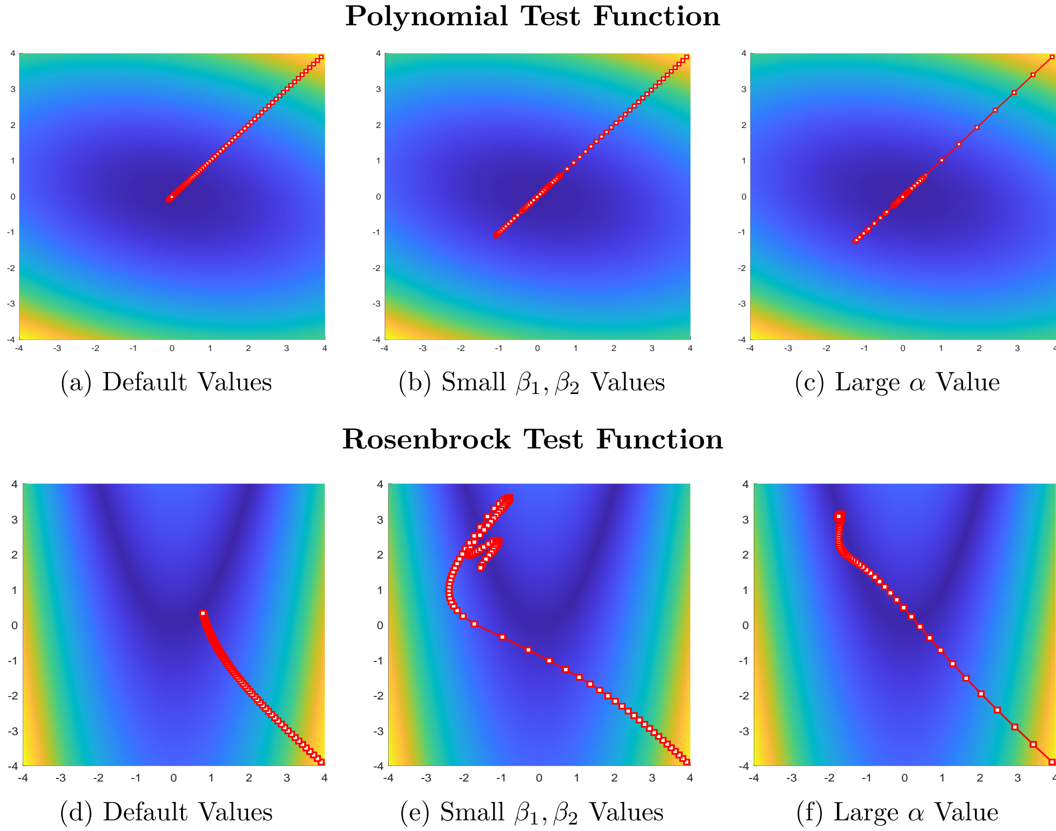
training of machine learning models). In such applications, computing the gradient for each sample in the entire training data set for each iteration would simply take too much time. To tackle this problem, several adaptations of vanilla GD were introduced. Note that these adaptations are not directly applicable to the problem we encounter in the *Tamashii* optimization problem and that we will therefore only briefly mention these approaches. The two most prominent ones of them beeing Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent, which both differ from vanilla GD in that they only compute the gradient of a subset of the entire data set. As mentioned above, this is on the one hand particularly useful in machine learning applications where computing the gradient of the entire data set is simply not possible; however, it also introduces a strong oscillation on the way to the minimum since the computed partial gradient has a high variance from the true gradient. In order to tackle this problem, there again exists a multiplicity of different adaptations and implementations to optimize SGD and Mini-Batch GD in a way that reduces the oscillations and accelerates the descent. As already mentioned, most modern approaches therefore focus on an adaptive learning rate method, such as Momentum [22], RMSProp [12] or AdaGrad [18]. However, this thesis will only focus on the most prominent one of them, ADAM, which we discuss in the next section.

3.2 ADAM

Adaptive Moment Estimation (ADAM) was first introduced by Diederik P. Kingma and Jimmy Ba in 2014 [19] and is an adaptation of GD that computes adaptive learning rates and combines the advantages of two different approaches: RMSProp [12] and AdaGrad [18]. Next to the parameters θ the algorithm also updates the exponential moving averages of the gradient (m_t) and the exponential moving averages of the squared gradient (v_t). Additionally, hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ are introduced to control the decay rates of these averages. We again let $J(\theta)$ be the objective function w.r.t. parameters $\theta \in \mathbb{R}^d$ and get:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\ \nabla_{\theta_t} J(\theta),$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\ ||\nabla_{\theta_t} J(\theta)||^2,$$

where the terms m_t and v_t are thought of as the running estimates of the mean (the first moment) and the variance (the second moment) of the gradients. Note that $||\nabla_{\theta_t} J(\theta)||^2$ in this case represents the scalar product of the gradient with itself. Further, the authors suggest a bias correction term for these estimates, since they observe that because of the initialization as a 0 vector, the estimates are biased towards zero. This behavior especially occurs during the initial timesteps and when the decay rates are slow, i.e. when β_1, β_2 are close to 1.



(a) Default Values (b) Small β_1, β_2 Values (c) Large α Value

(d) Default Values (e) Small β_1, β_2 Values (f) Large α Value

Figure 3.2: ADAM Example: Here we compare the optimization trajectories for ADAM using two test functions ("Polynomial" as in Eq. (3.1), and the well-known Rosenbrock function, Eq. (3.2)). We observe that the default configuration achieves the best results in both cases and that low damping introduces oscillations whereas large a large step-size tends to overshoot the minimum.

The bias-free moment estimates are then calculated as follows:

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t},$$

$$\hat{v}_t = \frac{v_t}{1-(\beta_2)^t}.$$

These estimates are then incorporated into the update, which yields the ADAM update rule:

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

The authors suggest $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as default values for the hyper-parameters in practice. Kingma and Ba show empirically in their article that this adaptive learning method performs well compared to similar approaches and has very limited memory requirements, which makes it very popular in machine learning tasks [19]. Example runs of ADAM can be seen in Figure 3.2, where we used a regular polynomial function and the Rosenbrock function for testing, which is a popular function for the testing of optimization algorithms. This is due to the narrow, parabolic-shaped valley, which makes finding the global optimum very hard for gradient-based optimization methods. More precisely, we defined the polynomial function as

$$f(x, y) = x^2 + xy + 3y^2, \tag{3.1}$$

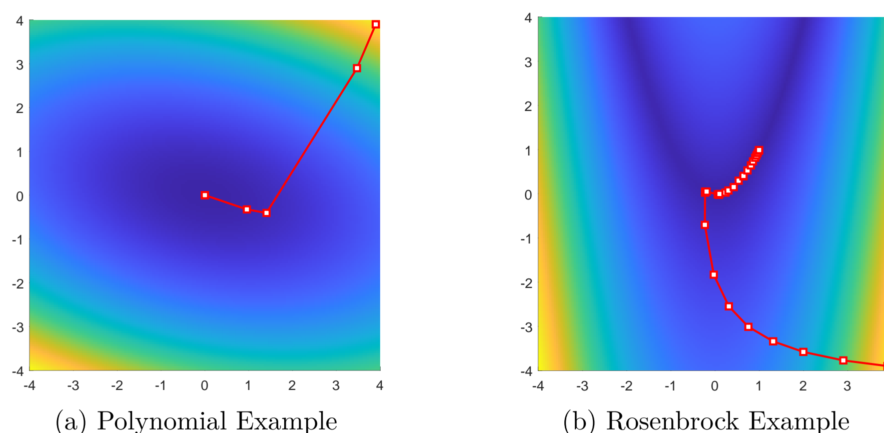
and the Rosenbrock function as

$$f(x, y) = 10^{-3}((1 - x)^2 + 100(y - x^2)^2), \tag{3.2}$$

with their respective derivatives. Note that as default values, we used the values that were suggested by Kingma and Ba for β_1, β_2 and ϵ.

3.3 (L-)BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is the most popular quasi-Newton optimization method and was independently introduced by Broyden, Fletcher, Goldfarb, and Shanno around 1970 [11, 13, 14, 29]. Newton's method in general tries to reduce the problem of minimizing any objective function $J(\theta)$ to the problem of solving a quadratic approximation of the objective function. The process begins by doing a second-order Taylor approximation of the objective function $J(\theta)$ at a certain point, i.e., fitting a quadratic function. Similar to other optimization methods, this starting point is either randomly sampled or specifically chosen as a hyper-parameter. The method then solves a linear system of equations in order to find the critical point of the quadratic approximation, which corresponds to a minimum if the function is convex (i.e., a positive-definite Hessian matrix). The entire method then starts over again by using this solution point as a starting point for another second-order Taylor approximation. This procedure repeats until the termination criteria are met. However, since this method needs the Hessian matrix, i.e., the matrix of second derivatives, for the Taylor approximation, it is often not feasible to use this method in practice. This is due to the fact that even if we assume that the objective function $J(\theta)$ is twice continuously differentiable, it is fairly likely that the Hessian matrix is dense, and therefore the cost of computing and storing all second-order derivatives is in the order of $O(d^2)$ where $d \in \mathbb{N}_{>0}$ denotes the dimensionality of the parameter vector $\theta \in \mathbb{R}^d$. Consequently, it can be very time-consuming to actually compute these derivatives, which led to the introduction of quasi-Newton methods.



(a) Polynomial Example (b) Rosenbrock Example

Figure 3.3: L-BFGS Example: Here we compare the optimization trajectories for L-BFGS again using two test functions ("Polynomial" as in Eq. (3.1), and the well-known Rosenbrock function, Eq. (3.2)). We use a default hyper-parameter configuration in both cases.

In contrast to the Newton method, the quasi-Newton method only approximates the Hessian matrix and thus only needs the first-order derivative, which is only in the order of $O(d)$ to store, and with adjoint methods, nearly $O(1)$ to compute, which significantly decreases the computation time. One algorithm that was introduced for this approximation is the BFGS method. However, the vanilla BFGS method still has one downside, namely the already-mentioned memory drain. Although it only approximates the Hessian matrix, its memory requirements are still in the order of $O(d^2)$ where $d \in \mathbb{N}_{>0}$ again denotes the dimensionality of the input vector $\theta \in \mathbb{R}^d$. Especially for large-scale problems, this can cause some difficulty, which led to the introduction of the Limited Memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method by Jorge Nocedal [23] in 1980. The basic idea of the L-BFGS method is not to store the approximated Hessian matrix but a (short) history of previous steps and gradients instead. The method then computes the product $H^{-1}v$ for any given vector v by looping twice through the history list, also known as two-loop recursion. This version of the algorithm is also the one that we will use for comparison in Chapter 5.

Formally, the algorithm now starts by setting an initial solution guess x_0, a history length m, coefficients β, β' where $0 < \beta' < \frac{1}{2}$, $\beta' < \beta < 1$ and a symmetric and positive definite starting matrix H_0. The following steps are then done iteratively until termination criteria are met. Note that the most prominent termination criteria again include a maximum number of iterations, a change in objective value that is below a certain threshold, a gradient that is sufficiently close to zero, or a step length that is too small. First compute:

$$d_k = -H_k g_k,$$

$$x_{k+1} = x_k + \alpha_k d_k,$$

where H_k is the *special BFGS matrix* approximating the inverse Hessian matrix and α_k is the step size. As already mentioned, the product $d_k = -H_k g_k$ is hereby computed in a two-loop recursion based on the history data (see Eq. (5) in [23] for details). Further, the step size α_k is chosen such that it satisfies the Wolfe conditions:

$$f(x_k + \alpha_k d_k) \leq f(x_k) + \beta' \alpha_k g_k^T d_k,$$
$$g(x_k + \alpha_k d_k)^T d_k \geq \beta g_k^T d_k.$$

The Wolfe conditions, first introduced by Philip Wolfe in 1969 [31, 32], are a set of inequalities for finding a suitable step size α_k. In order to avoid small objective function decreases when using large step sizes, the Wolfe conditions check whether the change in objective function value and the decrease of the gradient are “sufficient” for a certain step size. Note that the unit step length $\alpha_k = 1$ is always tried first, and the step length is then usually reduced by half if the objective function is not good enough or increased (doubled and a bit more) if the projected gradient has not decreased. We then let $\hat{m} = \min\{k, m-1\}$ and update the *special BFGS matrix* H_k based on Eq. (5) in [23] for a total of $\hat{m}+1$ times. Afterwards, we set $k := k+1$, and the next iteration begins by computing d_{k+1} and x_{k+2}.

3.4 CMA-ES

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a stochastic, or randomized, algorithm for finding (local) minima of (non-)convex functions. It was first introduced by Hansen and Ostermeier in 1996 [16]. The algorithm differs from the previously proposed algorithms in that it does not use the gradient of the objective function to find the (local) minimum. On the one hand, this can be a desirable characteristic for certain optimization problems since objective functions in general may be non-smooth (i.e. derivatives do not exist), and thus traditional gradient or quasi-newton methods cannot be applied. Further, due to its stochastic approach, CMA-ES is also able to cope with noisy or discontinuous functions, which is particularly useful in real-world applications. The downside, on the other hand, is that without gradient information (which in some cases can be computed relatively cheaply, e.g. adjoint method [10]), gradient-free methods suffer way more when the dimension of the problem grows, and generally converge more slowly than gradient-based approaches. More generally, CMA-ES can be grouped into a set of algorithms called **Evolutionary Algorithms (EAs)** or, as the name suggests, **Evolution Strategies (ES)**, which is a subset of EAs [30]. As with all EAs, ES are inspired by the natural evolution of species, and thus they maintain a set of sample instances (or solution candidates), the population, at each iteration of the algorithm. The best solutions from the population are then selected and used as parents for the next generation. The generation of a new population is a stochastic process, i.e.

the result is only known with a certain probability. This iterative process then continues until the termination criteria are met. Termination criteria for CMA-ES usually include a maximum number of iterations, a lower threshold for the change in mean between the previous and the next generation, or a lower threshold for the variance of the population. In the CMA-ES, a new population is now created by sampling from a multivariate normal distribution:

$$x_k^{(g+1)} \sim m^{(g)} + \sigma^{(g)}\mathcal{N}(0, C^{(g)}) \qquad \text{for } k = 1, ..., \lambda$$

where $\lambda \geq 2$ is the population count, $x_k^{(g+1)}$ is the k-th offspring from generation $g+1$, $m^{(g)} \in \mathbb{R}^n$ the mean sample value at generation g, $\sigma^{(g)} \in \mathbb{R}_{>0}$ the overall standard deviation, step size, at generation g and $C^{(g)}$ is the covariance matrix at generation g. ÍÍFurther, it is noted that

$$m^{(g)} + \sigma^{(g)}\mathcal{N}(0, C^{(g)}) \sim \mathcal{N}(m^{(g)}, (\sigma^{(g)})^2 C^{(g)}).$$

After the new population has been sampled, μ best solutions, i.e. the parents for the next generation, are used to compute the new mean as a weighted sum

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i x_{i:\lambda}^{(g+1)}$$

where

$$\sum_{i=1}^{\mu} w_i = 1, \qquad w_1 \geq w_2 \geq ... \geq w_\mu > 0 \qquad \mu \leq \lambda$$

and $x_{i:\lambda}^{(g+1)}$ is the i-th best ranked sample from generation $g+1$, in the sense that it has the i-th lowest objective value among the current generation, thus being most likely closer to the minimum of the objective function $J(\theta)$ than the higher-ranked samples.

In the next step, the covariance matrix, C, is updated. However, in order to achieve a fast search (which is contrary to a robust or more global search), the population size λ and thus also the parent size μ have to be small, which again makes the empirical estimation of C less reliable. In order to tackle this problem, there are many different ways to estimate C. In practice, the covariance matrix is often only updated with regards to the information of previous generations due to the lack of information in the current generation. This work will not go any further in the update of the covariance matrix C, additional information can be viewed in the tutorial by Hansen [17].

After the update of the Covariance matrix, the basic steps of CMA-ES are completed. Additionally, CMA-ES also computes the step-size $\sigma^{(g)}$, which corresponds to the scale

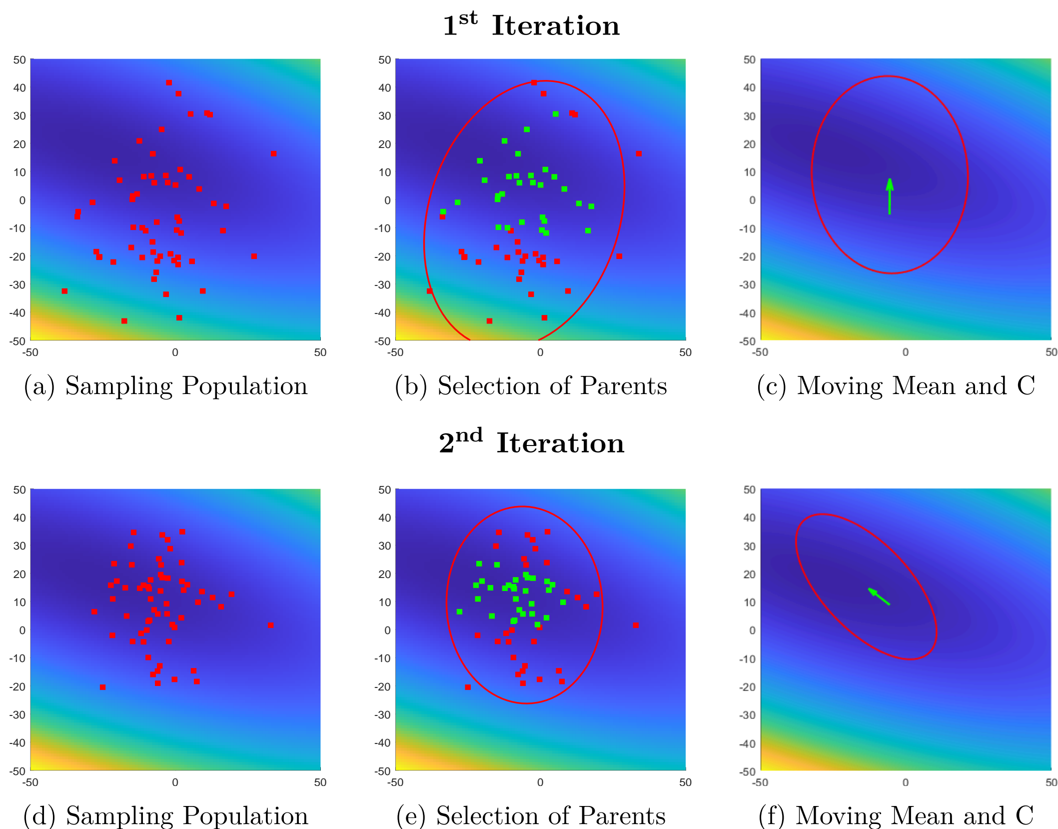


Figure 3.4: CMA-ES Example: Here we show the first two iterations of CMA-ES on the "Polynomial" test function (as in Eq. (3.1)). The red points mark all sampled points of the current iteration, and the green points indicate the selected parents for the next iteration. Further, we display the current covariance matrix as a red ellipse and a green arrow to show the movement of the mean.

of C, however, since this step is not important to understand the basic concept of this method, we will not cover it in any more detail here. As mentioned above, these steps are then applied iteratively until the termination criteria (e.g., a small change in objective value) are met. The first two iterations on a polynomial function of this algorithm are shown in Figure 3.4, where the green arrow denotes the movement of the mean and the red ellipse represents the covariance matrix of the currently sampled points.

3.5 Surrogate Based Optimization

Surrogate-Based Optimization (SBO) [26] represents a class of optimization methodologies that takes a different approach to solving the optimization problem. Conventional methods such as the ones presented above are used as sub-optimizations in this optimization process. Consequently, there exist many different possibilities for implementations, which can

differ quite drastically. For example, while some implementations work without gradient information, others include the gradients in the construction phase of the surrogate model. For this reason, this section will focus on the approach that the MATLAB implementation (*surrogateopt* [8]) takes, since it is also the version that we will use in Chapter 5. In its most basic idea, SBO alternates between the following stages:

a. **Construct Surrogate**,

b. **Search for Minimum within the Surrogate**,

which will be explained shortly. One benefit that the SBO approach inherently has is that usually fewer objective function evaluations compared to different approaches are needed due to the fact that the actual search for the minimum is only performed on the constructed surrogate. This is particularly useful if evaluations of the objective function are computationally expensive, e.g. running a simulation or rendering a scene. The downside, on the other hand, is that the construction of the surrogate becomes significantly more difficult with the increasing dimensionality of the optimization problem. Therefore, most SBO implementations (such as MATLAB's *surrogateopt*) demand parameter bounds to be set in order to restrict the space for which the surrogate has to be built.

3.5.1 Construction of the Surrogate

The construction stage starts by sampling the objective function that has to be approximated at certain points. Generally, these points are taken from a quasirandom sequence (see Chapter 7.8 in [25] for details) and transformed, scaled, and shifted to stay within bounds. However, if the number of variables exceeds 500, MATLAB takes these points from a Latin hypercube sequence [21]. Note that quasirandom is similar to pseudorandom, but with the difference of being more evenly spaced since pseudorandom has some tendency to cluster. Once these points have been sampled, the solver can construct a surrogate, which, ideally, should look as much like the real objective function as possible.

In the case of MATLAB's implementation, the construction is done by interpolation with a Radial Basis Function (RBF). RBFs are usually used for the reconstruction of an unknown function from known data. Some of the most prominent RBFs include the **inverse multiquadric**:

$$\Phi(x) := \frac{1}{\sqrt{1 + ||x||_2^2}},\ x \in \mathbb{R}^d,$$

or the **Gaussian**:

$$\Phi(x) := e^{-||x||_2^2},\ x \in \mathbb{R}^d,$$

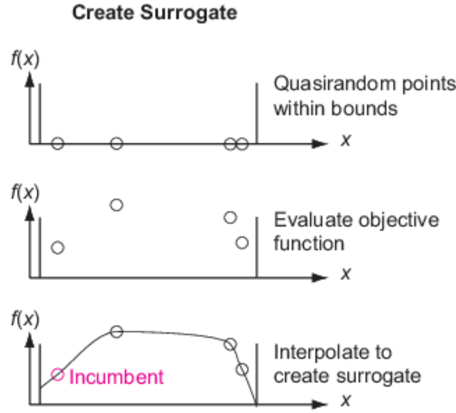


Figure 3.5: Creation of the Surrogate [8]

where $||\cdot||_2$ is the Euclidean norm in $\mathbb{R}^d$. This means RBFs are multivariate but reduce the vector argument x down to a scalar, which makes them radial in the sense that

$$\Phi(x) = \Phi(||x||_2) = \Phi(r),\ x \in \mathbb{R}^d,$$

where r represents the radius $r = ||x||_2$ with the scalar function $\Phi(x) : \mathbb{R} \to \mathbb{R}$. As R. Schaback mentions in his book [28], this makes RBFs very efficient in high-dimensional reconstruction and furthermore introduces invariance under orthogonal transformations. The resulting surrogate model $s(x)$ is then of the form

$$s(x) = \sum_{i=1}^{N} w_i \Phi(||x - y_i||),\ x, y_i \in \mathbb{R}^d,$$

i.e. a weighted sum of a single RBF with different centers y_i. The centers, or translations, are given by a set of vectors $\{y_1, ..., y_n\}$ where any $y_i \in \mathbb{R}^d$ and mark the evaluated points where data about the actual function is provided. Also note that N in this case is the number of samples (surrogate points). MATLAB's surrogate optimization implementation uses a cubic RBF with a linear tail in this phase, which changes the form of the resulting surrogate $s(x)$ to

$$s(x) = \sum_{i=1}^{N} w_i \Phi(||x - y_i||) + p(x),\ x, y_i \in \mathbb{R}^d,$$

where $p(x)$ is linear and $\Phi(r) = r^3$. As shown by H.-M. Gutmann [15], this form was chosen to minimize a measure of bumpiness in the resulting surrogate. MATLAB then solves a N-by-N linear system of equations where N is again the number of sampled surrogate points in order to find the weights w_i for the surrogate such as to best

approximate the given data $f(y_i)$. For many RBFs, it can be shown that this system of equations has a unique solution [24]. Note that in practice, many RBFs, and especially the RBF used by MATLAB's surrogate optimization implementation, are monotonically increasing. Consequently, this results in more weights being negative in order to solve the interpolation problem.

3.5.2 Search for Minimum within the Surrogate

After the surrogate has been constructed, the solver searches for the minimum objective value in a procedure that is related to local search. This search starts at the *incumbent*, which is the point with the lowest objective value out of all surrogate points that were used to construct the surrogate in the previous phase. Around this incumbent, it will then search within a search region radius that is relative to the overall parameter bounds. For this, the algorithm samples pseudorandom points within the search region, among which it will then search for the point with the minimal function value. However, it does not directly search for a minimal function value of the surrogate but for the minimum function value of a *merit function* instead. The merit function relates to the surrogate on the one hand and to the distance from existing search points on the other hand in order to have a balance between minimizing the surrogate and searching the space. More formally, the merit function $f_{merit}(x)$ consists of a weighted combination of two terms

- The scaled surrogate $S(x)$,
- The scaled distance $D(x)$,

which are defined as follows:

$$S(x) = \frac{s(x) - s_{min}}{s_{max} - s_{min}},$$

as the scaled surrogate, where s_{min} is the minimum surrogate value among the sample points, s_{max} is the maximum, and $s(x)$ is the surrogate value at point x. Note that $S(x)$ is zero at points x that have the minimal surrogate value. Secondly, the scaled distance is defined as:

$$D(x) = \frac{d_{max} - d(x)}{d_{max} - d_{min}}.$$

For this, define $y_j,\ j = 1, ..., k$ as the k evaluated points and d_{ij} as the distance from sample point i to evaluated point j. Now we can set $d_{min} = min(d_{ij})$ and $d_{max} = max(d_{ij})$ where the maximum and minimum is taken over all i and j. In contrast to the scaled surrogate, $D(x)$ is zero at sample points that have the maximum distance from the evaluated points.

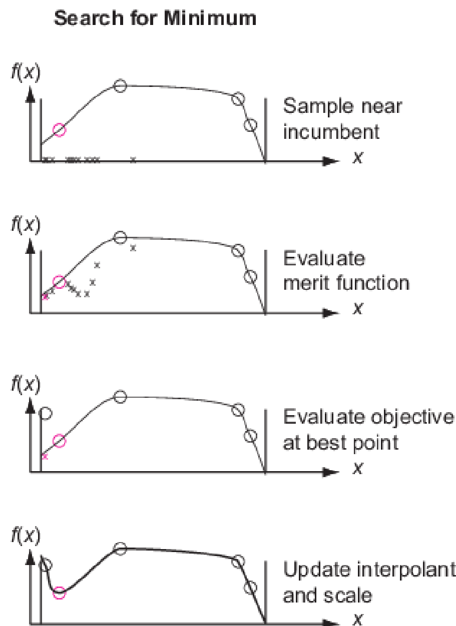


Figure 3.6: Minimum Search around Incumbent [8]

Lastly, we can define $f_{merit}(x)$ as:

$$f_{merit}(x) = wS(x) + (1 - w)D(x),\ 0 < w < 1.$$

We see that the choice of w significantly changes the search for minimum. On the one hand, if w is close to one, it leads the search to minimize the surrogate, and on the other hand, if w is close to zero, more importance is given to points that are far away from the evaluated surrogate points, which leads the algorithm to search in new regions. In the MATLAB implementation, the algorithm cycles through four different values for w in this searching phase: 0.3, 0.5, 0.8, and 0.95. These values were suggested by Regis and Shoemaker [27] and have the effect that early phases are more exploratory, whereas later phases focus more on local optimization.

After the minimal function value of f_{merit} among the samples has been found, the objective function value for this point is checked, and the surrogate is updated by this point. If it is sufficiently lower than the objective function value of the current incumbent, this point becomes the new incumbent. If it is not smaller, the search continues for the same incumbent. The solver will change the scale of the search region when certain criteria are met. Due to this, the algorithm will slowly reduce the scale of the search region to a minimum and converge at the incumbent with a minimal objective function

value. This is the solution of the optimization algorithm. Additionally, the algorithm can jump to the reconstruction phase of the surrogate in order to achieve a better surrogate with more evaluated points when certain criteria are met. This is then called a surrogate reset.

CHAPTER 4

MATLAB/C++ Callback Interface

After covering the basic functionalities of MATLAB and *Tamashii* in Chapter 2, we now give a more detailed overview and documentation about our implemented MATLAB/C++ callback interface, which we use for the evaluation of different optimizers regarding the lighting optimization rendering framework. Further, we provide a detailed description that can be used to reproduce the implementations and run the program.

4.1 Implementation

The aim of this work is to expand the already existing C++ rendering framework with the possibility of running custom MATLAB scripts that are again able to call the objective function, i.e. the lighting evaluation function of *Tamashii*, in order to find optima. In order to implement this functionality, we develop a bidirectional interface with the following control flow:

1. The C++ process starts a new MATLAB instance and runs a certain MATLAB function.

2. The running MATLAB function makes calls to the objective function running in the C++ process.

3. The C++ process renders the scene with the new parameters it obtains from MATLAB and evaluates the objective function. It then and returns the function value and the gradient for that point.

4. On termination, the MATLAB function returns the results to the C++ process.

5. The C++ process receives the results and continues its regular execution.

A visualization of this control flow can be seen in Figure 4.1. Note that steps two and three, i.e. the evaluation of the objective function and the return of the objective value (and gradient), can occur an arbitrary number of times. Further, any additional custom values may be returned in step three to the MATLAB process.

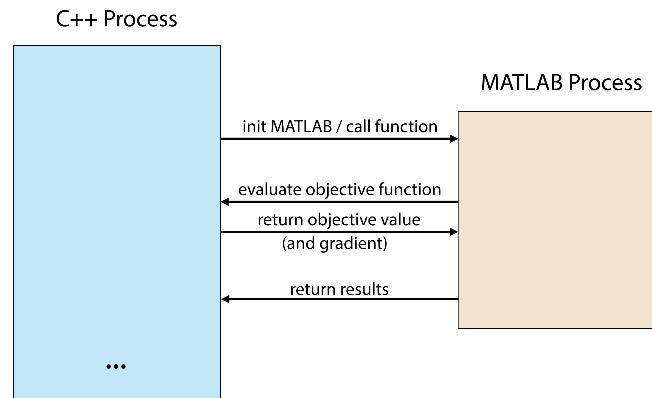


Figure 4.1: Control flow between C++ and MATLAB processes

After giving the high-level intuition of the interface, the upcoming sections will now explain our implementation in more detail. We will start by discussing our changes in the CMake configuration and then present our approach to inter-process communication (IPC). Afterwards, we will show the C++ side implementation of the interface and, lastly, show an example MATLAB function that runs SBO.

4.1.1 CMake Configuration

As mentioned above, this project uses CMake as a build and package management tool. In order to implement the MATLAB/C++ interface, we adapt certain parts of this Cmake configuration, and cover these changes in this section. First, we add an additional CMake option called `BUILD_MATLAB_INTERFACE`, which can either be toggled manually in the CMake GUI or added as a command-line argument in the console. This option specifies if the MATLAB/C++ interface should be included in the build configuration or not. Note that due to concurrent development, we made this option mutually exclusive with the option `BUILD_PYTHON_BINDINGS` in the current implementation. This means the MATLAB/C++ interface can only be built if the Python bindings are not being built. Otherwise, the Python bindings are always preferred if both options are checked. Since it is a global configuration setting, the `BUILD_MATLAB_INTERFACE` option resides in the root *CMakeLists.txt* file of *Tamashii*. All further CMake changes were then made in the *CMakeLists.txt* file of the subdirectory *./src/implementations/interactive_adjoint_light_tracing*, which is partially displayed in Listing 4.1. The check for the MATLAB and Python options can be seen in Listing 4.1 in line 2, where we also check

if the user is operating on a Windows machine since our implementation is currently Windows-specific.

Listing 4.1: CMake Configuration

```
1  # if build MATLAB interface
2  if (WIN32 AND NOT BUILD_PYTHON_BINDINGS AND BUILD_MATLAB_INTERFACE)
3
4   # using Matlab for mex compiler
5   find_package(Matlab REQUIRED)
6
7   matlab_add_mex(NAME mex_objective_interface SRC "./matlab/mex_objective_interface.cpp" "./
        matlab/ipc_handler.hpp" "./matlab/ipc_handler.cpp")
8
9   target_include_directories(mex_objective_interface PRIVATE "${INCLUDE_DIR}" "${
        EXTERNAL_DIR}/eigen")
10
11   install (TARGETS mex_objective_interface RUNTIME DESTINATION bin LIBRARY
        DESTINATION lib ARCHIVE DESTINATION lib)
12
13  # checking if all environment variables are set
14  SET(MATLAB_DLL_PATH "${Matlab_ROOT_DIR}/extern/bin/win64")
15      get_filename_component(INST_PATH_BIN "${CMAKE_INSTALL_PREFIX}/bin"
          REALPATH BASE_DIR ${CMAKE_BINARY_DIR})
16
17      string(REPLACE "/" "\\" MATLAB_DLL_PATH "${MATLAB_DLL_PATH}")
18      string(REPLACE "/" "\\" INST_PATH_BIN "${INST_PATH_BIN}")
19
20  message("Please make sure the following path is in your system variables: ${
        MATLAB_DLL_PATH}")
21
22  string(REPLACE "\\" "\\\\" INST_PATH_BIN "${INST_PATH_BIN}")
23
24  foreach(TARGET IN LISTS APP APP_SH)
25   target_compile_definitions(${TARGET} PUBLIC IALT_MATLAB_BINDIR="${
         INST_PATH_BIN}")
26   target_compile_definitions(${TARGET} PUBLIC IALT_BUILD_MATLAB="${
         BUILD_MATLAB_INTERFACE}")
27  endforeach()
28
29  endif()
```

We then first create a new MEX target and link the MEX source files to it (line 7). Note that source files that are linked to a MEX target are automatically compiled with MATLAB's MEX compiler. In lines 9 and 10, we then add the include directories to the MEX target and then specify the install directory. Afterwards, we create variables for the environment variables that have to be set in order for the program to find all MATLAB-related Dynamic Link Library (DLL) files needed to run the MATLAB libraries. However, since CMake is only able to set environment variables for its process and all child processes, the user needs to manually add the MATLAB_DLL_PATH variable to the system path. For simplicity, we print the absolute path directly to the console so the

user can directly copy and paste it into the path variable. Lastly, we add preprocessor directives in order for the C++ preprocessor to include the MATLAB code in all targets (which are excluded if the MATLAB/C++ interface is not to be built). Further, we also set the install path of the program as a preprocessor macro in order to then set this path as a MATLAB path variable from C++. We do this in order for MATLAB to find the MEX function, since the MEX function gets moved to the install folder after building. This concludes the CMake configuration needed to build the MATLAB/C++ interface.

4.1.2 Inter-Process Communication

In Chapter 2, we have seen how to initialize a MATLAB script from C++, and vice versa, how to call a C++ function (compiled as MEX) from MATLAB. However, since this MEX function is executed in the MATLAB process, a way of communication between the C++ process (the rendering framework, i.e., the objective function) and the MATLAB process (the optimization process) has to be established. In our approach, we use *Windows Named Pipes*, which is a Windows functionality that extends the traditional pipe concept found on Unix and Unix-like systems. Therefore, we lose support for Unix and MacOS systems in our implementation. It is, however, a high-performing method for IPC [9]. Due to the limited scope of this work, we will leave the implementation of Unix pipes and, consequently, the extension of the interface concept to Unix and MacOS systems for future work. In order to now realize this communication, we implement the server side of the pipe communication in the C++ rendering framework, and the client side of this communication in the MEX function. The main process now calls the MATLAB function asynchronously and then enters a listening loop to listen for client messages (see Listing 4.2). We then implement a simple communication protocol in order to control the message flow of the two processes. This protocol starts by the server waiting for the first message from the client. For the message encoding, we choose a comma-separated values (CSV) encoding, i.e., we use commas as delimiters between the values. Once it receives a message, the server first decodes it and then puts it into a C++ vector object. The client-side encoding can be seen in Listing 4.3. The server then checks weather the message from the client is equal to *ERROR* or *FINISHED*. These cases encode either errors occurring in the MEX interface or that the optimization process is finished. In both cases the server exits the listening loop and returns from the optimization process. However, if the message is not equal to these two cases, the server assumes a single vector (i.e., the parameter vector), which it then evaluates with the objective function. This evaluation returns an objective value and a gradient. The server then maps both the objective value and the gradient into one string, for which we again choose a CSV-style encoding. The objective value is put in the first position, followed by the gradient. Once this message is built, the server sends it to the client and again waits for a message from the client. As mentioned above, this process continues until either the *ERROR* or *FINISHED* flag occurs and the server exits the listening loop. The only thing then left for the main process is to wait for the asynchronous result to return in order to ensure the termination of the MATLAB function call and, optionally, query the solution of the optimization process (Listing 4.2, lines 95-99).

4.1.3 MATLAB Optimization Wrapper

In this section, we will discuss the implementation of the MATLAB optimization wrapper in C++. *Tamashii* has a base class called *OptimWrapperBase* that functions as a parent class for all optimization methods. Consequently, we first implement a derived class called *MatlabOptimWrapper*, which functions as a wrapper for our MATLAB/C++ optimization method. Note that the base class contains the operator that evaluates the objective function. When the optimization process starts, the *runOptimization* function is called with the current parameter vector. We first query our implemented MATLAB handler, which serves as a wrapper class for all needed MATLAB Engine functionality. As mentioned above, it is a singleton class in order to have just one MATLAB session running, which is started once at the startup of the application. We then create the arguments that we want to pass to the MATLAB function via the MATLAB Data API. In this case, we create a scalar for the maximum number of iterations, a vector for the light parameters, and a vector for types. This type vector contains enumeration types for each parameter in the parameter vector (e.g., X_POS, Y_POS, Z_POS, INTENSITY, etc.) and can be used to identify the meaning of each parameter entry in MATLAB. This information allows for interesting optimization options, for example, only optimizing the x-position variable of each light source. Additional parameters can also be passed using the MATLAB Data API [5]. The optimization function then calls the MATLAB function asynchronously in line 33. Note that the path to the specific MATLAB function file is saved in the MATLAB Engine wrapper class when choosing it in the GUI. Afterwards, the optimizer enters the listening loop, which we already explained in Subsection 4.1.2. Note that we put the code needed for our IPC approach in a wrapper class called *ipcHandler*. After the listening loop terminates, the optimizer waits for the MATLAB process to terminate and then sets and returns statistical optimization variables that are concurrently saved in the base class.

Listing 4.2: MATLAB Optimization Wrapper

```
1
2  MatlabOptimWrapper(LightTraceOptimizer* aSim, rvk::Buffer* aRadianceBufferCopy = nullptr) :
3    OptimWrapperBase<VectorType>(aSim, aRadianceBufferCopy), mStepSize(Real(1e-4)){}
4   OptimWrapperBase<VectorType>::OptimizationResult runOptimization(VectorType& aParams)
       override {
5
6    // Get Matlab Engine Handler
7    MatlabEngineHandler& matlabHandler = MatlabEngineHandler::getInstance();
8
9    // Create MATLAB data array factory
10   matlab::data::ArrayFactory factory;
11
12   std::stringstream pstr; pstr << aParams.transpose();
13   std::string tmp;
14   std::vector<double> light;
15   int counter = 0;
16   while(pstr >> tmp) {
17     light.push_back(std::stod(tmp));
```

```
18  }
19  std::stringstream tstr; tstr << mParamTypes.transpose();
20  std::vector<int16_t> types;
21  while (tstr >> tmp) {
22   types.push_back(std::stoi(tmp));
23  }
24
25  // Pass vector containing 2 scalar args in vector
26  std::vector<matlab::data::Array> args({
27   factory.createScalar<int16_t>(this->mMaxIters),
28   factory.createArray({1,light.size()}, light.begin(), light.end(), matlab::data::InputLayout::
        ROW_MAJOR),
29   factory.createArray({1,types.size()}, types.begin(), types.end(), matlab::data::InputLayout::
        ROW_MAJOR)
30   });
31
32  // Call MATLAB function
33  matlab::execution::FutureResult<matlab::data::Array> futureResult = matlabHandler.
        callScriptAsync(args);
34
35  // optimizer variables
36  this->mIters = 0; this->mEvals = 0;
37  VectorType dp;
38  Real phi;
39
40  // ipc handler
41  IPCServerHandler ipcHandler;
42
43  // listening loop
44  while (true) {
45   std::string message = ipcHandler.read();
46   std::string returnMessage = "ERROR";
47
48   if (strcmp(message.c_str(), "ERROR") == 0) {
49
50   } else {
51
52    if (strcmp(message.c_str(), "FINISHED") == 0) {
53     ipcHandler.cleanup();
54     break;
55    }
56
57    // map client message to params vector
58    std::replace(message.begin(), message.end(), ';', ' ');
59    std::stringstream ss(message);
60    std::vector<double> stdParams;
61    std::string temp;
62
63    while (ss >> temp) {
64     stdParams.push_back(std::stod(temp));
65    }
66
67    int length = stdParams.size();
```

```
68      VectorType cParams(length);
69
70      for (int i = 0; i < length; i++) {
71        cParams(i) = stdParams.at(i);
72      }
73
74      ++this->mIters;
75      phi = (*this)(cParams, dp);
76
77      std::stringstream rss; rss << dp.transpose();
78      returnMessage = std::to_string(phi);
79      returnMessage.append(";");
80      std::string tmp;
81      while (rss >> tmp) {
82        returnMessage.append(tmp);
83        returnMessage.append(";");
84      }
85    }
86
87    ipcHandler.write(returnMessage);
88    ipcHandler.reset();
89  }
90
91  // Wait for results
92  matlab::data::TypedArray<double> result = futureResult.get();
93
94  /*Display results(if needed)
95  std::cout << "Minimum at (Matlab): " << std::endl;
96  for (auto r : result) {
97    std::cout << r << " " << std::endl;
98  }*/
99
100 aParams = this->mBestRunParameters;
101 return { .bestObjectiveValue = this->mBestObjectiveValue, .lastPhi = this->
        mBestObjectiveValue };
102 }
```

4.1.4 MEX Interface Function

In this section, we discuss the implementation of the other side of the interface, the MEX file, in more detail. More precisely, the MEX layer lies in between the *Tamashii* process and the MATLAB process and handles the communication. On a high level, it receives an argument list as input from MATLAB and then sends it the *Tamashii* process. Afterwards, it waits for the return message from *Tamashii* and then returns it to the MATLAB process.

In our implementation (see Listing 4.3), we first follow the MEX base structure, which we already covered in Subsection 2.3.3. This means we create the class *MexFunction* and overwrite the function call operator. This operator marks the entry point to the MEX layer once it is called from MATLAB. It receives an *ArgumentList input* parameter

from MATLAB that contains the list of arguments that the MEX function was called with from MATLAB and an *ArgumentList output* parameter that is used to return values to MATLAB. In this operator, we first initialize our client-side wrapper object for IPC, which we use to communicate with the *Tamashii* process. Note that in our implementation, we expect to obtain one argument from MATLAB, namely the parameter vector. However, since we need to know when the optimization process in MATLAB is finished, in order to send the *FINISHED* flag to the *Tamashii* process, we define this special case by calling the MEX function with two parameters (see Listing 4.4 line 17 for the MATLAB side of the exit call). More precisely, only if the value of the second parameter equals `INT32_MAX` (note that the first parameter does not matter in this case). Note that this MEX call is necessary in all custom MATLAB functions that ought to be executed via the interface, since the C++ process will otherwise not exit the listening loop. Due to this convention, we first check if the *ArgumentList input* parameter contains more than one argument (lines 25–33). If this case enters, we send a *FINISHED* string as an exit flag to the *Tamashii* process in order for it to exit the listening loop, continue its regular execution (see Listing 4.2), and return 0 to the MATLAB process.

If this case does not enter, i.e., the *ArgumentList input* parameter contains only one element, we continue in the regular objective function evaluation case. This means we first encode the parameters given by the MATLAB process in the above-mentioned CSV-style format and then afterwards write them to the *Tamashii* process. We then wait for a response message containing the objective value and gradient, which we then map into the MATLAB *ArgumentList output* object using the MATLAB Data API for C++. Note that since we only work with a simple CSV-style encoding in our implementation, we fixed the first position of the CSV string to be the objective function value and the remaining entries to be the gradient. Lastly, we return the *ArgumentList output* object, containing the objective value in the first position and the gradient vector in the second position, to the MATLAB process.

Listing 4.3: MEX File

```
1
2  #include "mex.hpp"
3  #include "mexAdapter.hpp"
4  #include "ipc_handler.hpp"
5
6  using namespace matlab::data;
7  using matlab::mex::ArgumentList;
8
9  class MexFunction : public matlab::mex::Function {
10 private:
11   ArrayFactory mFactory;
12
13 public:
14   // entry point from MATLAB
15   void operator()(ArgumentList outputs, ArgumentList inputs) {
16
```

```
17 // ipc handler
18 IPCClientHandler ipcHandler;
19
20 std::string message;
21
22 bool finished = false;
23 TypedArray<double> inArray = inputs[0];
24
25 // check if MATLAB optimization is finished
26 int length = inputs.size();
27 if (length > 1) {
28  TypedArray<int> inFlag = inputs[1];
29  if (inFlag[0] == INT32_MAX) {
30   // set finished flag for Tamashii process
31   message = "FINISHED";
32   // return 0 to the MATLAB process
33   outputs[0] = mFactory.createScalar(0);
34   finished = true;
35  }
36 }
37 else {
38  message = "";
39  TypedIterator<double> paramsIter = inArray.begin();
40  for (paramsIter; paramsIter != inArray.end(); paramsIter++)
41  {
42   std::string val = std::to_string(*paramsIter);
43   message.append(val);
44   message.append(";");
45  }
46  outputs[0] = mFactory.createScalar(1);
47 }
48
49 // write message to the Tamashii process
50 ipcHandler.write(message);
51
52 // if the optimization process is finished we do not expect and answer anymore
53 if (finished) {
54  return;
55 }
56
57 // read the message from the Tamashii process
58 std::string retMessage = ipcHandler.read();
59
60 // map return message data into C++ containers
61 std::replace(retMessage.begin(), retMessage.end(), ';', ' ');
62 std::stringstream ss(retMessage);
63 std::vector<double> grad;
64 std::string temp;
65 bool firstSeen = false;
66 double phi = -1;
67
68 while (ss >> temp) {
69  if (!firstSeen) {
```

```
70        phi = std::stod(temp);
71        firstSeen = true;
72      }
73      else {
74        grad.push_back(std::stod(temp));
75      }
76    }
77
78    // return phi and gradient to MATLAB
79    outputs[0] = mFactory.createScalar<double>(phi);
80    outputs[1] = mFactory.createArray<double>({ 1, grad.size()}, grad.data(), grad.data() + grad.size
          ());
81  }
82 };
```

4.1.5 Surrogate Based Optimization Example

After explaining the server- and client-side implementation of the interface, we will briefly discuss an example of a MATLAB function that implements SBO as a black-box optimizer via the interface (Listing 4.4). In order to be able to use this function in the optimization process, the input first has to be matched with the input variables that come from C++. Currently, we pass three input arguments to MATLAB functions: the maximum number of iterations, the parameter vector, and the parameter type vector.

Listing 4.4: SBO Example

```
1   function [result] = surr_optim(~,params,~)
2
3       disp('MatLab File surr_optim in IALT')
4
5       % Creating MEX function handle
6       fg = @(param)(mex_objective_interface(param));
7
8       % Setting surrogate bounds
9       ub = ones(length(params),1) .* 4;
10      lb = -ub;
11
12      % Setting optimization parameters and plotting function
13      options = optimoptions('surrogateopt','PlotFcn','surrogateoptplot', 'InitialPoints', params
            );
14      [x, fval, exitflag ,output] = surrogateopt(fg,lb,ub,options);
15
16      % Callig the optimizer with abort parameters to close connection between processes
17      mex_objective_interface(x,intmax());
18
19      % Setting return value
20      result = x;
21  end
```

Note that the first argument, the maximum number of iterations, can be specified via *Tamashii's* graphical user interface (GUI). The second argument is the initial parameter vector that represents the lighting configuration at the beginning of the optimization process. Lastly, the third argument contains an integer encoding of each parameter in the parameter vector that characterizes the type of parameter (e.g., X_POS, Y_POS, Z_POS, etc.). This encoding is natively contained within *Tamashii* and can be used to identify the type of each parameter in the MATLAB file, which again allows for numerous use cases (e.g., only optimizing a specific parameter type). If input variables are not needed, such as the first and third ones in this example, MATLAB offers special syntax to ignore these parameters (using tilde). In this way, it is not necessary to always match the input parameters in C++ for each MATLAB function (which is far more tedious) or to have unused variables in the MATLAB workspace.
Afterwards, we create a function handle for the MEX function in line 6, which is then passed to the optimization function. Steps 9 and 10 are necessary since the surrogate optimization needs bounds in order to restrict the area where the surrogate has to be built. In this case, we just hard-coded the values to a reasonable range, which of course depends on the scene and lighting configuration. However, as mentioned in Chapter 2 we can use the type of each individual parameter in the parameter vector to conduct a reasonable range for the respective parameter (e.g., bound rotation parameter to interval $[0, 2\pi]$). The type information can again simply be queried from the parameter type vector that is passed to MATLAB. The next steps simply include setting options for the optimization algorithm, such as the plotting function, and running the algorithm. Note that we use the *params* variable as an initial guess, since this variable represents the current light configuration in the renderer at the time of starting the optimization process. However, it is also possible to pass other points as an initial guess or not to pass initial points at all to the algorithm (and let the algorithm choose its starting point depending on its implementation). This might result in an initial jumping behavior of the light sources for certain algorithms, but might also give a better initial guess than the manually placed light sources. Once the optimization is finished, we need to call the MEX interface function once more in order for it to send the *FINISHED* flag to the main C++ process, which will otherwise not exit the listening loop. We do this by calling it with the result of the optimization (that way it will again reduce unnecessary jumping behavior) and the needed *intmax* value. Lastly, we simply set the return variable.

4.2 Using the Interface

In this section, we will explain all necessary steps in order to run the *Tamashii* renderer with custom MATLAB optimization functions.

4.2.1 Configuration

In order to be able to use the bidirectional MATLAB/C++ interface, some configuration steps have to be taken. However, some of these steps are automated in the current

implementation, either by CMake or directly in C++, but for completeness, we will list them here as well. As already described in Subsection 4.1.1 we use CMake as a build and package management tool in this project, which makes it necessary to start the build process via it. Further, important messages for the usage of the IALT renderer are also displayed by CMake and only need to be followed in order to complete the configuration. First, MATLAB and Microsoft Visual Studio are required to be installed. We would recommend at least MATLAB R2022b and Visual Studio 17 since we tested the implementation with these versions. Further, *Tamashii* itself requires at least CMake version 3.14 and the Vulkan SDK to be installed. In the second part of the configuration, a path needs to be manually added to the path environment variable in order for the program to find all the DLL files needed to run the MATLAB libraries. This path will have the following structure:

- <MATLAB_ROOT_DIR>/extern/bin/win64

where <MATLAB_ROOT_DIR> represents the root directory of the MATLAB installation. Note that CMake will automatically find the MATLAB root directory and print the exact path that has to be added to the environment variable in the console when configuring the project. The last step is to select the option `BUILD_MATLAB_INTERFACE` and unselect the option `BUILD_PYTHON_BINDINGS` in CMake since currently only Python is built if both options are active. Once this step is done, the program is runnable, and all further configuration steps and additional environment variables will be set automatically. **Important:** It may be the case that the configuring step fails on the first execution because it cannot find the paths to the MATLAB Engine lib and the MATLAB Data Array lib. In this case, simply rerun the configuration step. Here is a list of all the configuration steps that are then done automatically:

- adding the MATLAB Engine lib path to includes, <MATLAB_ROOT_DIR>/extern/lib/win64/microsoft/libMatlabEngine.lib
- adding the MATLAB Data lib path to includes, <MATLAB_ROOT_DIR>/extern/lib/win64/microsoft/libMatlabDataArray.lib
- adding the install path (= compiled MEX function path) to the MATLAB path
- adding the path of the MATLAB script that has to be executed to the MATLAB path

where the first two points of the list are done by the CMake configuration and the last two points are implemented directly in C++.

4.2.2 Execution

Running optimization via the C++/MATLAB interface can be done easily once the configuration is done. First, a MATLAB function file has to be created, which matches

the structure of the example shown in Subsection 4.1.5. Further, the function needs to match the input variables passed from C++ that are needed for the optimization and can have an arbitrary number of output variables; however, output values are usually not needed since the C++ rendering framework already updates the position and the optimization statistics.

(a) Selecting Optimizer (b) Selecting MATLAB Script (c) Selected MATLAB Script

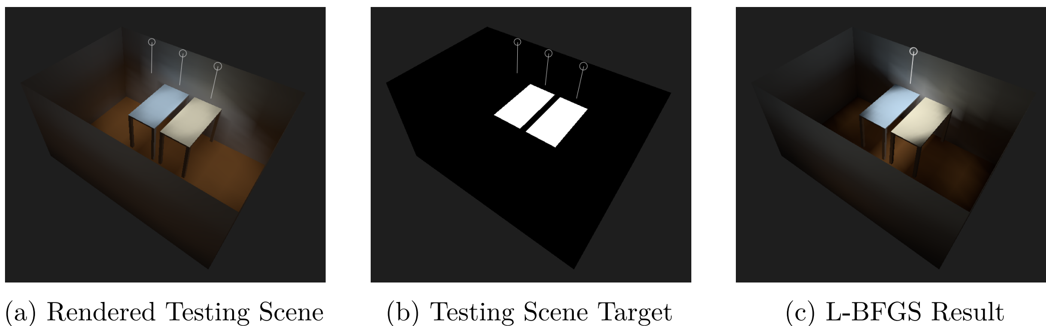
Figure 4.2: Tamashii User Interface

The function calls from MATLAB to the MEX interface can again be done as shown in Subsection 4.1.5. Once the MATLAB function is created, the *Tamashii* renderer is started regularly as explained in Section 2.2. Note that for the optimization process, an optimization target has to be defined. The target can manually be drawn by entering the drawing mode which can be accessed by pressing *D*. We extended *Tamashii's* user interface (see Figure 4.2) by adding a *MATLAB Optimizer* option which can then be selected in order to run optimization via the C++/MATLAB interface. Once the optimization method is selected, a file dialog can be opened by pressing *Select Script*. This file dialog can then be used to select the MATLAB function file that contains the optimization code. The name of the currently selected file is always displayed below the *Select Script* button. After choosing the correct MATLAB file, simply press *optimize* and the optimization process will run. Note that the MATLAB file can be located anywhere on your machine since its path is automatically added to the MATLAB path variable when choosing it. Once the optimization process terminates, the optimization statistics will be displayed and other *Tamashii* functionality is available as before.

CHAPTER 5

Evaluation and Comparison

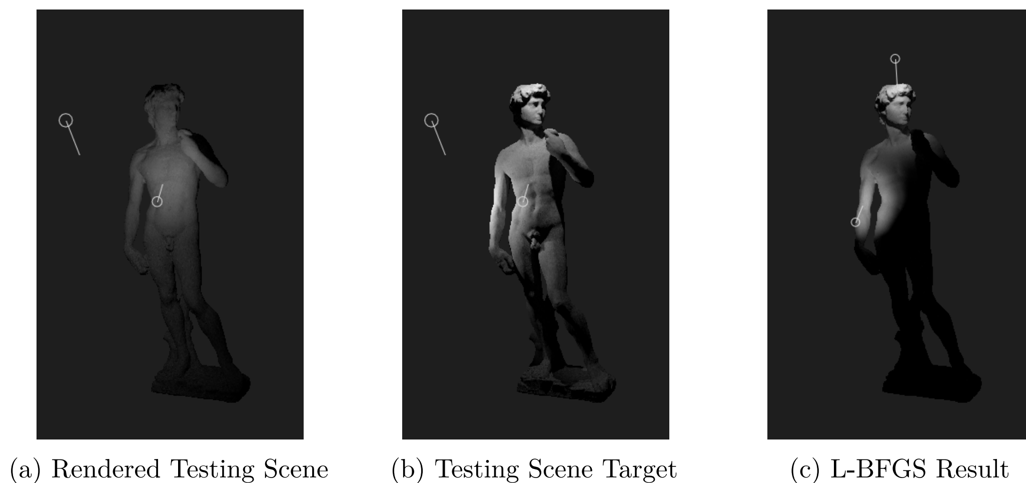
In this chapter we will present and compare the results of the tested algorithms presented in Chapter 3. Further, we will not only evaluate the performance of the different optimization algorithms but also compare the differences between the current C++ implementation and running the same optimization method via the MATLAB interface.



(a) Rendered Testing Scene (b) Testing Scene Target (c) L-BFGS Result

Figure 5.1: Small Office Testing Scene

For testing we consider two testing scenes: the *Small Office Scene* which contains 3 light sources (Figure 5.1) and the *David Statue Scene* which contains 2 light sources (Figure 5.2). Both scenes have the same fixed amount of light sources and the same predefined target in each optimization run. For simplicity, we only optimize the position parameters (i.e. coordinates) of each light source. Further, the application was run in release mode using a *NVIDIA GeForce RTX 2080 Super* GPU as testing hardware in all runs.



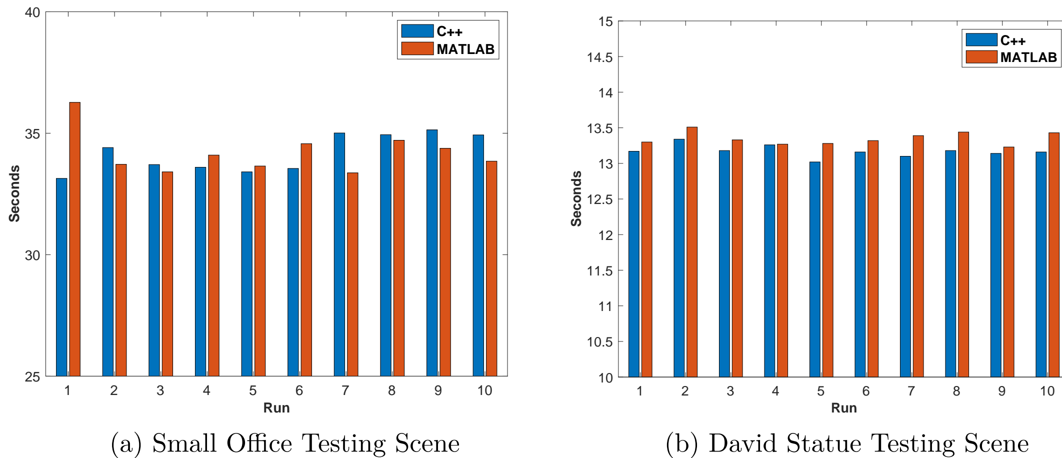
(a) Rendered Testing Scene (b) Testing Scene Target (c) L-BFGS Result

Figure 5.2: David Statue Testing Scene

5.1 Evaluation of the bidirectional C++/MATLAB Interface

In this section, we evaluate the performance of our implemented interface by comparing identical implementations running via MATLAB and running directly in C++. Further, we discuss the increased usability enabled by the MATLAB/C++ interface and explore the usage of MATLAB optimizers as black-box functions. In particular, for the first test, we implement a simple rerendering function that calls the redraw and evaluation functions of the IALT renderer 100 times and only slightly changes the positions by a fixed amount in each iteration. However, the position change is exactly the same in both implementations; this means that both implementations are identical, with the only difference being that the MATLAB implementation has the “overhead” of first calling the MATLAB function and then communicating over the implemented IPC method. Further, we also measure the times of the rerendering process at the same position for both implementations. We tested these rerendering functions with both test scenes 10 times each, and the results can be seen in Figure 5.3 and Table 5.1.

The results clearly show that the run-times of rerendering the scene 100 times using our MATLAB/C++ interface are almost identical to the plain C++ implementation. When looking at the statistical parameters for these runs in Table 5.1, we see that the C++ implementation is slightly faster, but we consider this difference negligible. Further, we also see that the rerendering times when using the MATLAB/C++ interface tend to have a higher deviation from the mean, i.e., high variance. This could be due to the increased complexity of the implementation, which causes the times to fluctuate more around the mean. However, from these results, we can still conclude that the “overhead” introduced by the inter-process communication is very small and that Windows Named Pipes are a very efficient solution for this particular problem. Note that the charts in Figure 5.3 are



(a) Small Office Testing Scene (b) David Statue Testing Scene

Figure 5.3: MATLAB and C++ Rerendering Comparison

Table 5.1: Overview of the results on the rerendering function. Columns: testing scene, rerendering implementation, run time sample mean in seconds, run time sample standard deviation in seconds.

Scene	Implementation	Mean	Standard Deviation
Small Office	C++	34.18	0.776
Small Office	MATLAB	34.20	0.862
David Statue	C++	13.17	0.085
David Statue	MATLAB	13.35	0.088

cut on the y-axis in order to give a better view of the differences in run-times.

Still, since simple rerendering was not the initial goal of this implementation, we also will test the performance difference of optimization via MATLAB compared to plain C++. For this, we compare an optimization algorithm implemented in C++ to an identical implementation in MATLAB. We choose ADAM as an example since identical implementations are already available in MATLAB and C++ and the algorithm does not depend on probabilistic sampling. In order to be able to compare both implementations, we also started all test runs with the same hyper-parameter configuration, which we list in Table 5.2 for reproducibility.

If we have a look at the results of the optimization processes (Table 5.3), we see that both implementations perform very similar, which does not come as a surprise since we compare identical implementations and already showed that the MATLAB "overhead" is (at least for simple rerendering processes) negligible. However, we still see some differences. Note that the C++ implementation is faster in the Small Office testing scene and that the MATLAB implementation is slightly faster in the David Statue testing scene. This behavior may again come down to the fluctuation of the optimization duration that we

Table 5.2: Overview of hyper-parameter configuration of ADAM. Columns: parameter, value.

Parameter	Value
max. iterations	200
α	0.1
β_1	0.9
β_2	0.999

Table 5.3: Overview of the results using ADAM optimization. Columns: testing scene, ADAM implementation, best achieved objective value of the optimization, number of iterations of the algorithm, total wall clock time for the entire optimization $t[s]$ in seconds. average time per iteration $t_\Delta[ms]$ in milliseconds.

Scene	Implementation	Best Objective	Iterations	t	t_Δ
Small Office	C++	1.5793	200	56.8	284
Small Office	MATLAB	1.5793	200	59.4	297
David Statue	C++	0.4032	200	28.5	142
David Statue	MATLAB	0.4031	200	29.1	145

were already able to observe in the rerendering process. Further, we see that in the Small Office testing scene, both algorithms achieve a more similar objective value compared to the David Statue testing scene, which may come down to rounding differences between MATLAB and C++. Note that the MATLAB implementation in this case is even able to find a better solution than the C++ implementation. All summed up, we see that the "overhead" introduced by the MATLAB/C++ interface is also in optimization processes hardly detectable.

Another interesting test case is the comparison between an integrated black-box optimization function of MATLAB and the in C++ implemented equivalent. In this case we choose MATLAB's *fminunc* function since it contains a L-BFGS implementation which we can compare to the L-BFGS implementation currently implemented in *Tamashii*. Note that we do not conduct any hyper-parameter tuning beforehand in order to test MATLAB's *fminunc* optimization function as an out-of-the-box solution. We only specify options for the function in MATLAB that are needed to have the function use the L-BFGS method as an Hessian approximation in conjunction with custom objective function gradients. The code for the MATLAB wrapper function for L-BFGS (*fminunc*) is shown in Listing 5.1.

If we have a look at the results of these runs (Table 5.4), we see that in both test scenes, the MATLAB implementation converges faster than the C++ implementation,

Listing 5.1: L-BFGS Example

```
1     function [result] = quasi_newton(maxIters,params,~)
2
3  disp('MatLab File quasi_newton in IALT')
4
5     % Creating MEX function handle
6     fg = @(param)(mex_objective_interface(param));
7
8     % Setting optimizier options and callback−function
9     options = optimoptions(@fminunc,'HessianApproximation','lbfgs','SpecifyObjectiveGradient',true);
10
11    % Calling the optimizer
12    [xo,fo,~,output] = fminunc(fg,params,options);
13
14    % Callig the optimizer with abort parameters to close connection between processes
15    mex_objective_interface(xo,intmax());
16
17    % Setting return value
18    result = xo;
19    end
```

which does however come with a sacrifice in solution quality. This behavior can be explained by termination criteria that are met earlier in the MATLAB implementation than in the C++ implementation, since not only the time that each method takes is reduced by half but also the number of objective function evaluations. This also shows that, by default, the MATLAB implementation tends to prioritize performance over solution quality. Still, we want to stress here that we are comparing different implementations and therefore can only conclude performance differences by looking at the optimization metrics, since we do not know how the MATLAB implementation internally works. Note that again, the time measurement is the same for both implementations, i.e., we include the time for starting and connecting to the MATLAB process in the overall optimization time. Nevertheless, we still think that MATLAB black-box optimizers, such as *fminunc*, are very usable in the context of *Tamashii* and have at least similar performance properties as the currently implemented algorithms.

Lastly, we want to discuss the increased usability that the MATLAB/C++ interface provides. First, during our testing process, we were able to exchange optimization algorithms very quickly and managed to use most of MATLAB's *Global Optimization Toolbox* algorithms within a few lines of code (e.g., Listing 5.1). Additionally, we could easily write multiple function files for the same optimization algorithm that differed by small implementation details and were able to switch the used implementation within seconds. Further, we were also able to use MATLAB's plotting features to enhance *Tamashii's* GUI with real-time plots. An example of this can be seen in Figure 5.4, where we used the built-in plotting function from MATLAB's *surrogateopt* function

Table 5.4: Overview of the results using L-BFGS optimization. Columns: testing scene, L-BFGS implementation, best achieved objective value of the optimization, number of iterations of the algorithm, total wall clock time for the entire optimization $t[s]$ in seconds. average time per iteration $t_\Delta[ms]$ in milliseconds.

Scene	Implementation	Best Objective	Iterations	t	t_Δ
Small Office	C++	1.5804	62	17.0	274
Small Office	MATLAB	1.581	27	8.4	311
David Statue	C++	0.4031	84	11.8	140
David Statue	MATLAB	0.4035	46	7.4	160

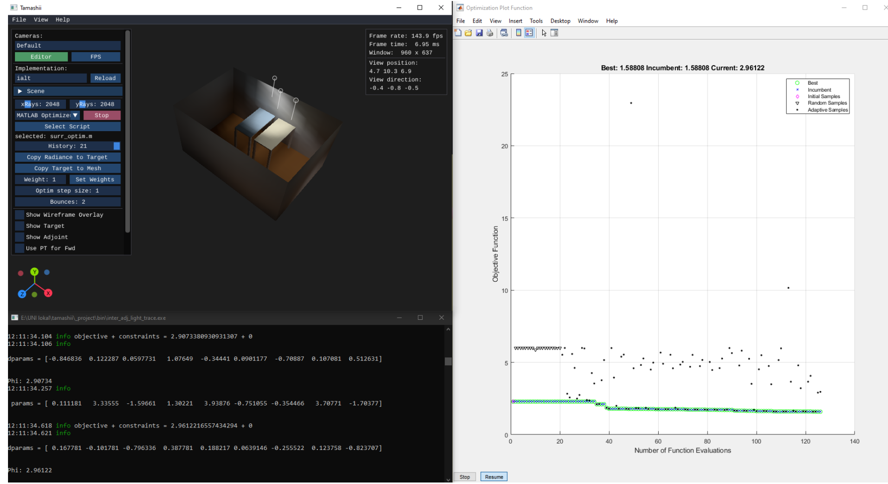


Figure 5.4: Real-Time Plotting using *surrogateopt*

to get a real-time insight into the optimization process. It was also possible to use the *Pause*, *Resume*, and *Stop* buttons provided by the figure. Note that with the implemented interface, we are also able to save the optimization data from optimization runs with the algorithms that are currently implemented in plain C++ within *Tamashii*. This can be done by collecting optimization data during the optimization process in C++ and then sending it to a MATLAB function once the optimization is completed. This data can then either be analysed instantly in MATLAB or saved as a MATLAB workspace file, which makes it very easy to create plots and metrics from optimization processes in an automated way, even if the algorithm is completely implemented in C++. This does not only support the implementation of new optimizers but also gives more insight into the currently implemented algorithms. Consequently, we conclude that the implemented MATLAB/C++ interface offers a significant increase in *Tamashii's* usability and, moreover, offers many possibilities for future research.

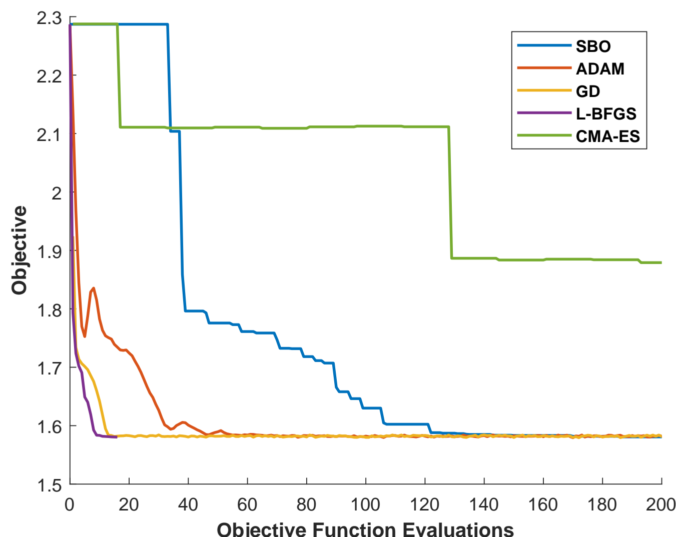


Figure 5.5: Small Office Testing Scene: Here we show the currently best-found objective value (i.e., lighting configuration) at each iteration of the optimization algorithm on the *Small Office* testing scene. Note that CMA-ES and SBO exceed the 200 iterations that are shown in this plot but were cut for better visibility.

5.2 Comparison of Optimization Algorithms

After having evaluated the performance of optimization using our MATLAB/C++ interface, we will now discuss MATLAB's Surrogate-Based Optimization (SBO) in the context of solving the *Tamashii* optimization problem as one example of a MATLAB black-box optimizer. We do this by comparing the performance of each algorithm presented in Chapter 3 on the *Small Office* and *David Statue* test scenes. Note that in general, the choice of the optimization algorithm used very much depends on the specific optimization problem that has to be solved, but with this section, we still want to show that SBO can work as an alternative approach to the currently implemented and more conventional optimization algorithms. For simplicity, we again only optimize the position parameters of each light source.

Note that in Figures 5.5 and 5.6, we each plot the currently best-found objective value against the number of objective function evaluations. Further, more metrics of the optimization process, such as the optimization time, are shown in Table 5.5 and Table 5.6, respectively. What becomes immediately observable when looking at the graphs is that SBO and CMA-ES tend to need multiple objective function evaluations in order to find a better minimum, which also comes down to the implementation of these optimization methods. Also note that MATLAB's SBO implementation and CMA-ES are gradient-free optimization methods, whereas GD, ADAM, and L-BFGS all use gradient information for the optimization process. Therefore, we see that gradient-based methods tend to converge faster on this type of optimization problem. However, if we look at Tables

Table 5.5: Overview of the results on the *Small Office* Test Scene. Columns: optimization algorithm, best achieved objective value of the optimization, number of objective function valuations, step size α, total wall clock time for the entire optimization $t[s]$. average time per iteration $t_\Delta[ms]$ in milliseconds.

Optim.	Best Objectvive	Evaluations	α	t	t_Δ
Gradient Descent	1.5793	202	0.5	55.2	273
ADAM	1.5793	201	0.1	56.8	282
ADAM (MATLAB)	1.5793	201	0.1	59.4	295
L-BFGS	1.5804	62	1.0	17.0	274
L-BFGS (MATLAB)	1.5810	27	1.0	8.4	311
CMA-ES	1.5786	3202	-	893.5	279
SBO	1.5807	221	-	88.8	401

Table 5.6: Overview of the results on the *David Statue* Test Scene. Columns: optimization algorithm, best achieved objective value of the optimization, number of objective function valuations, step size α, total wall clock time for the entire optimization $t[s]$. average time per iteration $t_\Delta[ms]$ in milliseconds.

Optim.	Best Objectvive	Evaluations	α	t	t_Δ
Gradient Descent	0.4032	202	0.5	27.8	137
ADAM	0.4032	201	0.1	28.5	141
ADAM (MATLAB)	0.4031	201	0.1	29.1	144
L-BFGS	0.4031	84	1.0	11.8	140
L-BFGS (MATLAB)	0.4035	46	1.0	7.4	160
CMA-ES	0.4027	3202	-	463.3	144
SBO	0.5792	300	-	57.3	191

5.5 and 5.6, we see that the needed function evaluations of SBO are approximately of the same magnitude as Gradient Descent and ADAM. This is one upside of SBO compared to CMA-ES, which needs by far the most objective function evaluations of all tested optimization methods. This number of objective function evaluations is also visible in the duration of the optimization process, which exceeds the duration of all other optimization methods by far. Therefore, when again comparing the gradient-free optimization methods SBO and CMA-ES, we can conclude that SBO brings far better performance (i.e., fewer objective function evaluations and faster termination) to the optimization problem of these testing scenes. However, this increase in performance comes with a drawback in solution quality since CMA-ES is able to find better solutions in both testing scenes (see Tables 5.5 and 5.6). Nevertheless, we want to stress that even if CMA-ES is finding slightly better solutions, it does take an disproportionate

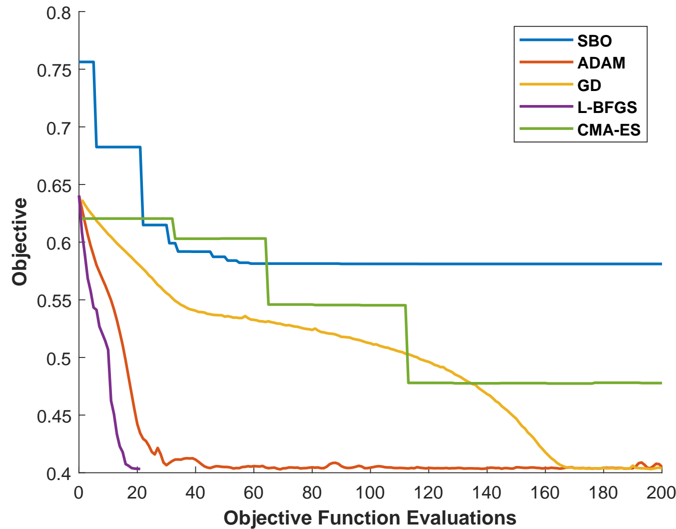


Figure 5.6: David Statue Testing Scene: Here we show the currently best-found objective value (i.e., lighting configuration) at each iteration of the optimization algorithm on the *David Statue* testing scene. Note that CMA-ES and SBO exceed the 200 iterations that are shown in this plot but were cut for better visibility.

amount of time and objective function evaluations to do so. Further, we have to note that all other optimization algorithms solve unconstrained optimization problems, whereas MATLAB's *surrogateopt* does need boundaries for the optimization variables. This can lead to drastically different performance and solutions since building a surrogate for a larger space does take more time and might also change the set of local minima that are reachable. In our testing, we try to manually adjust these bounds to reasonable values; however, we cannot exclude the possibility that there may be a configuration where the optimization results for these particular scenes may be better. Another point that becomes visible when checking the graphs and the tables is that the function values of the best-found solutions are similar not only for SBO and CMA-ES but also for all other optimizers. One outlier here is again SBO in the David Statue Scene, which fails to build an accurate surrogate within the 300 iterations. In general, we can say that SBO can achieve similar or possibly even better performance on the *Tamashii* optimization problem; however, as with any other optimization method, it very much depends on the specific problem (i.e., scene and lighting configuration). Consequently, in order to find the best-performing optimization algorithm for a particular problem, there is still no other way than to test all the available optimizers and pick the best one.

During our testing we also tried to extend the optimization problem to a mixed-integer problem where the intensity of certain light sources is an additional optimization variable. More precisely, by adding the intensity of a light source as a variable to the optimization vector, the MATLAB script changed the intensity to a integer variable restricted to the values 0 and 1 that was then marked as an integer variable when passed to the

optimization algorithm. When the optimization process was then evaluating the objective function, the integer variable is first multiplied by its original intensity. This means the light sources intensity can either be 0 or the original intensity value, which again corresponds to turning the light source on or off. MATLAB's *surrogateopt* function allows for input parameters to be integer valued only and does have a very efficient tree search implementation for these problems, however, during our testing we were not able to achieve good results with black box testing. We leave this field open for future research.

CHAPTER 6

Conclusions

In this work, we were able to show an approach for integrating MATLAB functionality into an existing C++ environment, in particular the *Tamashii* renderer developed at Vienna University of Technology (TU Wien), in order to make use of MATLAB's different toolboxes and File Exchange code base. We did this by using the MATLAB Engine API for C++ in order to start and call MATLAB functions from C++ and the C++ MEX API in order to call C++ functions from MATLAB. Further, we implemented a protocol for communication between the MATLAB and C++ processes using Windows Named Pipes. In order to then use and evaluate this bidirectional interface, we first discussed different approaches for finding (local) minima that were currently implemented in C++ in *Tamashii* and then focused on Surrogate-Based Optimization (SBO) as a less conventional optimization method. Since there are currently hardly any C++ libraries that offer SBO but many MATLAB options, we proposed a way to provide the possibility of SBO to *Tamashii* via the MATLAB/C++ interface.

In Chapter 4 we then gave a detailed description of how our interface is implemented and how it can be used to run MATLAB functions. Our tests then evaluated the performance of the bidirectional interface by running identical implementations of a rerendering function and an ADAM optimization in plain C++ and via our MATLAB interface. The tests showed great performance of the interface and further demonstrated the effectiveness of the inter-process communication method that was implemented. Both the rerendering runs and the ADAM optimization runs had only negligible performance differences between the two implementations. Further, we also tested the use of MATLAB black-box optimizers, in particular L-BFGS and SBO, and compared their performance to the current C++ implementations. Again, we found that the MATLAB implementations were at least similar in performance. With L-BFGS, we were even able to achieve better results on some testing scenes compared to the C++ implementation. We also found that SBO was slightly behind the performance of other gradient-based optimization

algorithms on all scenes. However, we again stress that the current SBO implementation of MATLAB does not use gradients, and when therefore comparing it with CMA-ES, we found that SBO was able to achieve far better performance on one scene and similar performance on the other. Further, there are also SBO implementations for MATLAB that include gradients for the construction phase. Since our interface is technically able to provide this data, we suggest comparing the performance of gradient-based optimization methods with the performance of a SBO implementation that uses gradients for future research. However, we also stress that the performance evaluation on our testing scenes is not generalizable since the ideal choice of optimization algorithm used always comes down to the specific optimization problem (i.e., the scene and lighting configuration). Further, we only manually adjusted the hyper-parameters and did not conduct any more sophisticated hyper-parameter tuning. Nevertheless, SBO was able to find optima that were similar to other conventional algorithms and also used about the same amount of time and objective function evaluations. Still, in contrast to the C++ implementations, we were able to use the MATLAB optimization functions within minutes and therefore achieved great results with the interface.

In conclusion, we have presented a workflow for the *Tamashii* renderer that allows for very fast adaptation of the used optimization algorithm and the incorporation of MATLAB's full functionality. Further, MATLAB's plotting tools make it possible to augment the GUI of *Tamashii* with real-time plots that provide a better insight into the optimization processes. Thus, we believe that our implementation opens up a number of different fields for future research. These include further testing of possible optimization algorithms via the interface, analyzing the already implemented C++ algorithms with MATLAB tools, and testing optimization on non-position parameters, such as light intensity or color.

Acronyms

CPU central processing unit. 4

CSV comma-separated values. 30, 34

DLL Dynamic Link Library. 29, 38

EAs Evolutionary Algorithms. 18

ES Evolution Strategies. 18

GD Gradient Descent. 12–14, 47

GPU graphics processing unit. 1, 2, 4, 41

GUI graphical user interface. 28, 31, 37, 45, 52

IALT interactive adjoint light tracing. 1, 3, 5, 38, 42

IPC inter-process communication. xiii, 28, 30, 31, 34, 42, 51

RBF Radial Basis Function. 21–23

SBO Surrogate-Based Optimization. xi, xiii, 2, 4, 20, 21, 28, 36, 47–49, 51, 52

SGD Stochastic Gradient Descent. 14

Bibliography

[1] LBFGSpp. Accessed October 2, 2023. https://github.com/yixuan/LBFGSpp.

[2] C-CMAES. Accessed October 2, 2023. https://github.com/cma-es/c-cmaes.

[3] MATLAB Engine API for C++. Accessed August 3, 2023. https://de.mathworks.com/help/matlab/cpp-engine-api.html.

[4] C++ MEX API. Accessed August 3, 2023. https://de.mathworks.com/help/matlab/cpp-mex-file-applications.html.

[5] MATLAB Data API for C++. Accessed August 3, 2023. https://de.mathworks.com/help/matlab/matlab-data-array.html.

[6] MATLAB Scripts vs. Functions. Accessed September 19, 2023. https://de.mathworks.com/help/matlab/matlab_prog/scripts-and-functions.html.

[7] CMake MEX Support. Accessed August 28, 2023. https://cmake.org/cmake/help/latest/modul/FindMatlab.html#command:matlab_add_mex.

[8] MATLAB Surrogate Optimization Algorithm. Accessed August 20, 2023. https://de.mathworks.com/help/gads/surrogate-optimization-algorithm.html.

[9] Windows Named Pipes. Accessed August 4, 2023. https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes.

[10] Andrew M. Bradley. Pde-constrained optimization and the adjoint method. 2010.

[11] C. G. Broyden. A new double-rank minimization algorithm. *Notices American Math. Soc*, 16:670, 1969.

[12] Yann Dauphin, Harm Vries, Junyoung Chung, and Y. Bengio. Rmsprop and equilibrated adaptive learning rates for non-convex optimization. *arXiv*, 35, 2015. doi:10.48550/arXiv.1502.04390.

[13] R. Fletcher. A new approach to variable metric algorithms. *Comput. J.*, 13:317–322, 1970. doi:10.1093/comjnl/13.3.317.

[14] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24:23–26, 1970. doi:10.2307/2004873.

[15] H.-M. Gutmann. A radial basis function method for global optimization. *J. of Global Optimization*, 19(3):201–227, 2001. doi:10.1023/A:1011255519438.

[16] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 312–317, 1996. doi:10.1109/ICEC.1996.542381.

[17] Nikolaus Hansen. The cma evolution strategy: A tutorial. 2023. doi:10.48550/arXiv.1604.00772.

[18] Elad Hazan John Duchi and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2011. doi:10.5555/1953048.2021068.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2017. doi:10.48550/arXiv.1412.6980.

[20] Lukas Lipp, David Hahn, Pierre Ecormier-Nocca, Florian Rist, and Michael Wimmer. View-independent adjoint light tracing for lighting design optimization, 2023. doi:10.48550/arXiv.2310.02043.

[21] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979. doi:10.2307/1268522.

[22] Michael Muehlebach and Michael I. Jordan. Optimization with momentum: Dynamical, control-theoretic, and symplectic perspectives, 2021. doi:10.48550/arXiv.2002.12493.

[23] Jorge Nocedal. Updating quasi newton matrices with limited storage. *Mathematics of Computation*, 35(151):951–958, 1980. doi:10.1090/S0025-5718-1980-0572855-7.

[24] Michael JD Powell. The theory of radial basis function approximation in 1990. *Advances in numerical analysis*, 2:105–210, 1992.

[25] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, USA, 3 edition, 2007.

[26] Néstor V. Queipo, Raphael T. Haftka, Wei Shyy, Tushar Goel, Rajkumar Vaidyanathan, and P. Kevin Tucker. Surrogate-based analysis and optimization. *Progress in Aerospace Sciences*, 41:1–28, 2005. doi:10.1016/J.PAEROSCI.2005.02.001.

[27] Rommel G. Regis and Christine Annette Shoemaker. A stochastic radial basis function method for the global optimization of expensive functions. *INFORMS J. Comput.*, 19:497–509, 2007. doi:10.1287/ijoc.1060.0182.

[28] Robert Schaback. A practical guide to radial basis functions. 2007.

[29] David F. Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of Computation*, 24:647–656, 1970. doi:10.1090/S0025-5718-1970-0274029-X.

[30] Andrew N. Sloss and Steven Gustafson. 2019 evolutionary algorithms review, 2019. doi:10.48550/arXiv.1906.08870.

[31] Philip Wolfe. Convergence conditions for ascent methods. *SIAM Rev.*, 11(2):226–235, 1969. doi:10.1137/1011036.

[32] Philip Wolfe. Convergence conditions for ascent methods. ii. *SIAM Rev.*, 13(2):185–188, 1971. doi:10.1137/1013035.