TU WIEN Informatics

Optimierungsverfahren zur Extremalokalisierung in einem 5G Mobilfunktransceiver-Modell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Thomas Fromherz, BSc.
Matrikelnummer 11909438

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Mitwirkung: Projektass. Jie He, MSc.

Wien, 19. März 2025

Thomas Fromherz	Radu Grosu

TU WIEN Informatics

Optimization Strategies for Locating Extrema in a 5G Cellular Transceiver Model

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Thomas Fromherz, BSc.
Registration Number 11909438

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Assistance: Projektass. Jie He, MSc.

Vienna, March 19, 2025

Thomas Fromherz	Radu Grosu

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

Erklärung zur Verfassung der Arbeit

Thomas Fromherz, BSc.

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

Wien, 19. März 2025

Thomas Fromherz

Kurzfassung

Die zunehmende Komplexität im Design von integrierten Schaltkreisen erfordert innovative Methoden im Systems Engineering. Modellierung und Simulation sind entscheidend für die frühzeitige Identifikation von Architekturfehlern. Diese Arbeit formuliert die Identifikation solcher Fehler als Optimierungsproblem, da es im Kern darum geht, Extrema in einem Simulationsmodell zu finden. Insbesondere wird das Problem auf das Optimierungsproblem von teuren Black-Box Funktionen reduziert, da die Evaluierung der Zielfunktion, welche einem Simulationsdurchlauf entspricht, zeitaufwändig ist und keine mathematische Formulierung des Modells vorliegt. Optimierungstechniken aus verschiedenen Forschungsbereichen werden untersucht, wobei sich Partikelschwarm-Optimierung, evolutionäre Algorithmen und mehrere Varianten der bayesschen Optimierung aus dem Bereich des maschinellen Lernens als am Vielversprechendsten erweisen. Vier Algorithmen werden implementiert und sowohl auf künstlichen Benchmark-Funktionen als auch auf einem ereignisbasierten Simulationsmodell eines 5G-Mobilfunktrasnceivers evaluiert. Zusätzlich wird ein Parallelisierungsumgebung vorgestellt, um die Ausführung der Algorithmen auf parallelen Rechenressourcen zu ermöglichen. Die Ergebnisse zeigen, dass drei der vier implementierten Algorithmen das Optimierungsproblem effektiv lösen. Algorithmen basierend auf maschinellem Lernen weisen eine schnelle Konvergenz auf, während populationsbasierte Algorithmen hohe Zuverlässigkeit zeigen. Das Parallelisierungsumgebung erweist sich als ein wesentlicher Bestandteil zur Erreichung kurzer Konvergenzzeiten. Sie ermöglicht einen annährend linearen Speed-up auf bis zu 100 CPUs und bietet großes Potenzial für weitere Skalierbarkeit.

Abstract

The increasing complexity in integrated circuit (IC) design necessitates innovative methods in systems engineering. Modeling and simulation are crucial for early identification of architectural flaws. This thesis formulates this flaw identification as an optimization problem, as it refers to locating extrema in a simulation model. Specifically, the problem is reduced to the expensive black-box optimization problem, due to the time-consuming nature of objective function evaluations corresponding to simulation runs and the lack of a mathematical model. Optimization techniques from various research fields are investigated, with Particle Swarm Optimization, Evolutionary Algorithms, and several Bayesian Optimization methods from the field of machine learning identified as the most promising ones. Four algorithms are implemented and evaluated on both artificial benchmark functions and an event-based simulation model of a 5G cellular transceiver. Additionally, a parallelization framework is introduced to enable execution of the algorithms on parallel computing ressources. Results indicate that three out of the four implemented algorithms effectively solve the optimization problem. Machine learning-based algorithms exhibit fast convergence, while population-based algorithms demonstrate high reliability. The parallelization framework proves to be a vital part for achieving reasonable convergence times. It achieves near-linear speed-up on up to 100 CPUs, with huge potential for further scalability.

Contents

CHAPTER 1

Introduction

Over the past decades we have seen rapid advancements and continuously rising complexity in integrated circuit (IC) design. This development poses entirely new challenges to electronic systems engineering, demanding continuous innovation in systems engineering methods and approaches. Simulation and modelling have become a vital part of many systems engineering processes. These tools enable engineers to estimate the consequences of design decisions early in the development process, thus preventing costly iterations and ensuring a more efficient design phase. In the first chapter of this thesis it is outlined how different use cases of such simulation models lead to a common, difficult optimization problem. This optimization problem is characterized by a black-box objective function that is expensive to evaluate, as evaluation corresponds to simulation. These insights lead to a formal problem statement, describing the core challenges addressed in this thesis. What follows is a rigorous investigation of methods to solve this expensive black-box optimization problem. Different approaches from various fields of computer science and engineering are examined, with a focus on machine learning based methods and heuristic methods. Particle Swarm Optimization, an Evolutionary algorithm and two different approaches of Bayesian Optimization turn out most promising and are chosen for implementation. Difficulties that need to be solved are adapting these general algorithms to the particular problem as well as parallelizing the algorithms and interfacing with external parallel computing infrastructure. This setup enables simultaneous execution of hundreds of model evaluations, resulting in significant speed-up and improved scalability. The details of how these challenges are solved are described in the second chapter of the thesis. Finally, a set of experiments is conducted on both artificial and real-world use cases. In particular, the real-world experiments are conducted using an event-based simulation model of a 5G cellular transceiver. The results, which are described in the third chapter, show that three out of the four implementations solve the black-box optimization problem both for the artificial as well as the real-world usecases. The architecture proves of being scalable and achieves convergence times of only a few minutes on expensive

black-box optimization tasks. Finally, in the last two chapters the results are discussed and compared in detail, providing valuable insights for further implementations and future research.

CHAPTER 2

Background and Problem Statement

2.1 Mobile Communications

Although this thesis clearly focuses on optimization topics, a very basic understanding of mobile communications supports comprehension of this work. Mobile communications have transformed the way individuals and societies connect and interact from ground up. A central part of this development is the user equipment (UE), commonly known as the mobile phone. The communications takes place between the UE and a network of basestations ("cell phone towers"). This bidirectional communication involves the UE sending data in the uplink (UL) and receiving data in the downlink (DL) direction. The network of basestations provides comprehensive area coverage and acts as an interface to the backend of the mobile network of the cellular provider which in turn provides access to the internet. [Plo24]

A series of mobile communication standards has been developed over more than four decades. The first generation (1G) dates back to the early 1980s and comprises a set of different specifications and services with strong regional differences and complete lack of interoperability. A first attempt to unify these services was the Global System for Mobile Communications (GSM) taking over in the early 1990s. This second generation (2G) standard allowed interoperability between devices across borders. The third generation (3G) communication standards were specified by the Third Generation Partnership Project (3GPP) and enabled high speed data rates for the first time. The first of this generation was the Universial Mobile Telecommunications System (UMTS), advancements thereof were High Speed Packet Access (HSPA) and HSPA+. In 2008 3GPP introduced a fourth generation (4G) standard called Long Term Evolution (LTE). LTE and its successor Long Term Evolution Advanced (LTE-A) solved many problems and limitations introduced in 3G standards. For the first time data rates of up to 100 Mbit/s became realistic. Today,

still more than 60% of mobile communication takes place via 4G standards. In 2016 3GPP introduced the fifth generation (5G) standard New Radio (NR) which nowadays is the latest release that has been commercially deployed at large scale. It's core promises are ultra-low latency, ultra-reliability and massive connection density enabling new fields of applications requiring these capabilities such as autonomous vehicles, smart cities or remote surgery [Plo24].

Resource management plays a critical role in mobile communications. The principles are quite similar in LTE and NR. Generally speaking, in radio frequency (RF) communications, both time and frequency are used as resources, resulting in a two-dimensional resource grid. The largest time unit in this grid is a frame, lasting 10ms. Each frame is divided into 10 subframes, each 1ms long. Subframes are further segmented into slots, which can vary from 1ms to 0.0625ms, and each slot consists of 14 OFDM symbols. In the frequency domain, the basic unit is the subcarrier, typically 15kHz or 30kHz wide. The smallest unit in the resource grid, the resource element (RE), comprises one subcarrier and one OFDM symbol. Twelve consecutive resource elements in the frequency domain form a resource block [3GP24]. Figure 2.1 illustrates the resource grid for the duration of one subframe [Ryu20].

The 5G NR protocol defines several communication channels. Some of these channels are

- Physical Downlink Shared Channel (PDSCH)
- Physical Uplink Shared Channel (PUSCH)
- Physical Downlink Control Channel (PDCCH)
- Physical Uplink Control Channel (PUCCH)
- Physical Random Access Channle (PRACH)
- Physical Broadcast Channel (PBCH)

The base station dynamically schedules these channels based on factors such as user demands, signal quality, and UE speed. Each channel serves different purposes, such as transmitting user data, control information, or signal quality measurements. For instance, the PDCCH communicates the scheduling information to the UE, indicating where and when data can be expected in the resource grid. Thus, to transmit or receive the correct RF signals, the transceiver must be in a specific state at a precise time. This precision requires a 5G cellular transceiver to be tightly controlled in terms of time, ensuring that data is correctly sampled or transmitted.

2.2 Cellular Transceiver

The signal processsing chain of the UE that enables 4G and 5G standards is typically implemented by an integrated radio frequency (RF) transceiver and a powerful baseband

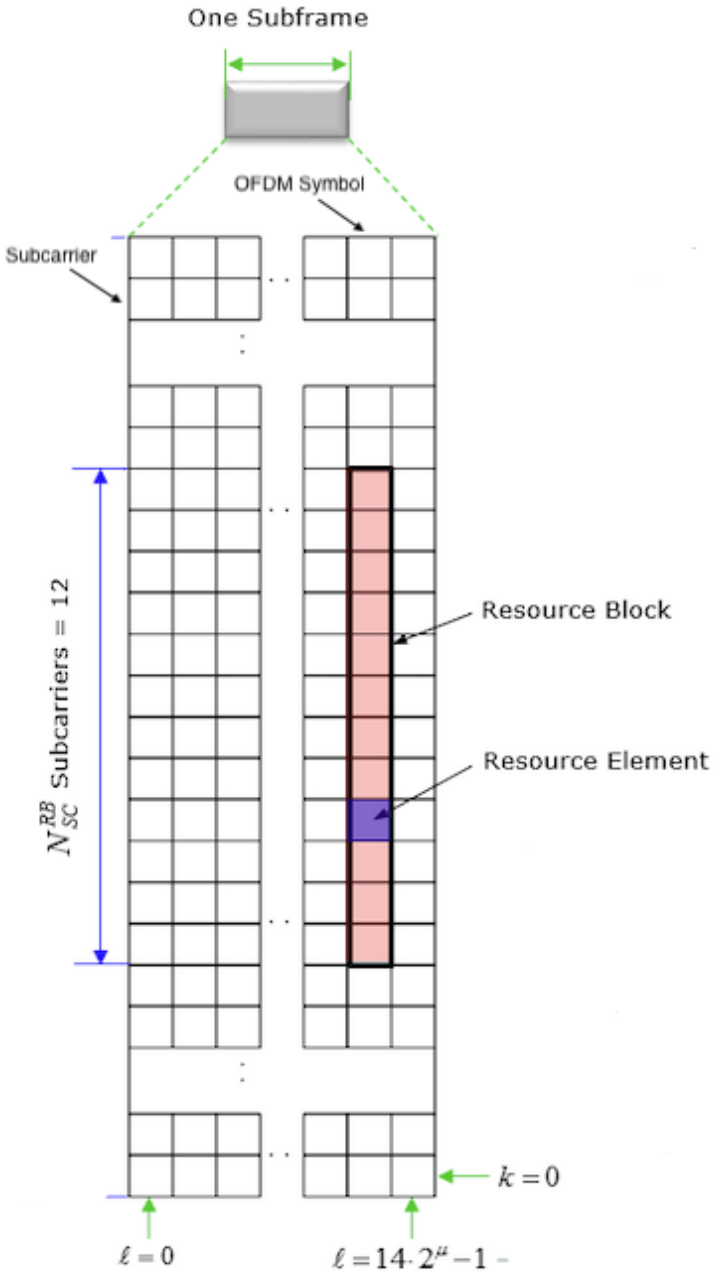


Figure 2.1: 5G NR Resource Grid [Ryu20]

(BB) processor. The baseband processor, equipped with many CPUs, RAM, and digital signal processing (DSP) cores, operates a real-time operating system to manage the cellular connection. It processes connection requests from the application processor and controls the transceiver. The transceiver is a mixed signal IC, which provides the RF

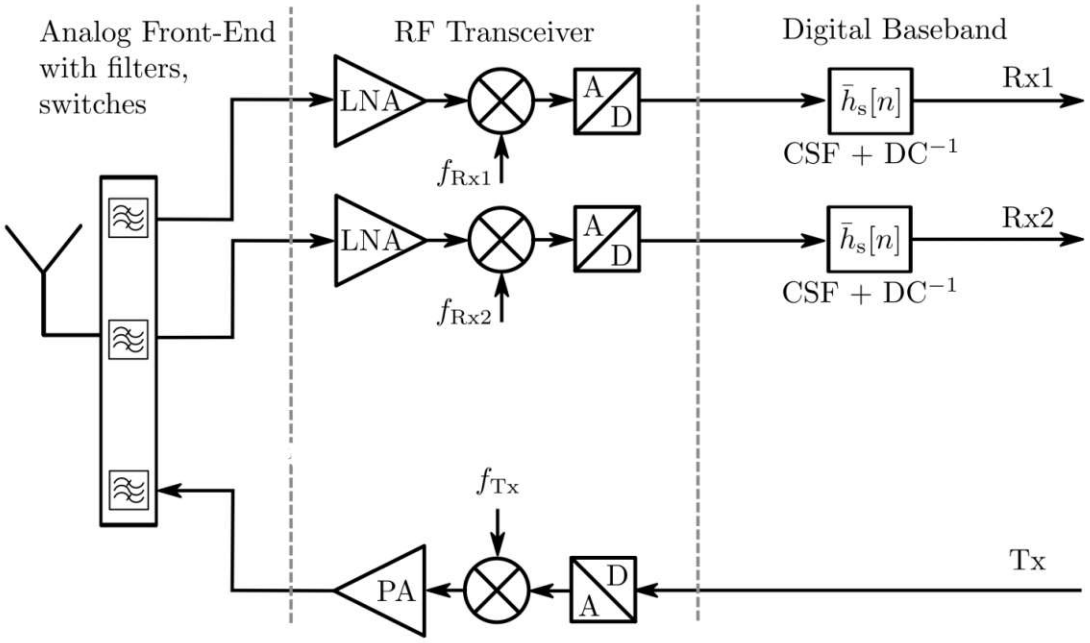


Figure 2.2: The signal processing chain implemented by the transceiver and the baseband processor. Modified from [Plo24].

interface in both transmit and receive direction, converting digital data to RF signals (transmit) or vice versa (receive).

The transceiver includes analog and digital elements for frequency conversion and amplification. Key components include the analog-to-digital converter (ADC), which converts analog signals to digital form, and the upconversion mixer, which shifts the baseband transmit (Tx) signal to an RF signal centered around the transmit frequency (fTx). This RF signal is amplified by the power amplifier (PA) to ensure enough transmission power is reached.

On the receive (Rx) side, the low-noise amplifier (LNA) amplifies the incoming RF signal before it is downconverted by mixers to baseband for further processing. The channel-select filter (CSF) and direct current (DC) filter limit the received signal's bandwidth and remove any DC offset. All components of this signal processsing chain implemented by the transceiver and the baseband are illustrated in figure 2.2

A critical challenge in this process is controlling the transceiver components to ensure precise timing of data transmission, as required by cellular protocols. The timing is crucial to meet the strict requirements of 4G and 5G standards, which demand accurate synchronization for reliable communication [Plo24]. This time-sensitive control is particularly hard due to the transceiver being a mixed-signal integrated circuit, which introduces various analog timing requirements.

2.3 Computer Aided Systems Engineering

In recent years electronic systems design has been shaped by the increasing complexity of both hardware and software and by aggressive time-to-market constraints, constituting major challenges for system design engineers. According to Moore's Law which was first stated in 1969, transistor count per chip doubles every 18 months due to evolving silicon fabrication technologies [DER97] - a law many believe is still intact today. The rising number of transistors also leads to an exponential increase in the functionality that an IC can possibly implement. However, design teams are hardly able to keep up with that pace - leading to a divergence between technology capabilities and design capabilities, the well-known *design gap* [MS13]. The systematic approach to tackle these design challenges is what the discipline of systems engineering is all about. Systems engineering is a multidisciplinary field, that is dedicated to the systematic design and management of complex systems throughout their life cycles. The increasing complexity of deployed technologies together with ever-rising demands for performance, reliability, and integration constitute a challenge that cannot be solved by expert domain knowledge alone anymore.

Therefore, the modern systems engineering process is augmented by a variety of technological and model-based tools, leading to the term Computer-Aided Systems Engineering. This approach leverages advanced computer models, simulations and artificial intelligence/machine learning (AI/ML) techniques to enhance the design, analysis, and management of complex systems. This integration not only improves the efficiency and effectiveness of the design process but also helps in avoiding costly errors and delays. By using computer-aided and model-based tools, engineering teams can explore design alternatives earlier in the design phase, when changing designs is easier and less disruptive.

The section begins with subsection 2.3.1 Systems Engineering describing the fundamental principles of systems engineering and its theoretic foundations. The goal is to provide a comprehensive understanding of the underlying challenges and the relevance for every modern systems design process. We also outline the necessity of computational aid through modelling, simulation and optimization. Modelling and simulation is dealt with in the separate subsection 2.3.2 Model Based Systems Engineering. This subsection will cover the types of simulation models commonly deployed and their development processes. This thesis illustrates their practical utility by demonstrating how a simulation model may enhance the systems engineering process of a cellular transceiver.

As soon as an expressive simulation model is at hand, naturally the question arises on how to find ideal design parameters, whatever ideal means in the given context. It is shown that this question always leads to a complex optimization problem. Model-based optimization is a valuable tool enabling systems engineers to analyse a designed system under possibly millions of scenarios without the need for physical prototypes. The subsection 2.3.3 Optimization Oriented Systems Engineering shows how the results of such an optimization may be utilized for both design and verification/validation purposes. A detailed description of both classical and machine learning based strategies solving

this complex optimization problem is subject of the next section, 2.7 Modern approaches to Optimization.

2.3.1 Systems Engineering

In the subsequent sections the discipline of Systems Engineering is explained in detail creating the foundation for a comprehensive understanding of the problem that this thesis deals with.

What is Systems Engineering?

In order to get an understanding what the discipline of Systems Engineering (SE) actually is, it's vital to define the term system first. According to [Wei07], a system is an artifact created by humans that consists of components that work together to achieve a goal that cannot be achieved by one of the components alone. The components themselves are not further specified - they might consist of hardware, software or mechanical parts or might even be of human nature. This very general definition - intentionally - covers an incredible variety of things and also allows for things that are initially unexpected as systems, e.g. a country's public transport system or a product's supply chain. The physical size of a system however does not imply anything about the complexity. A physically large system like a country's public transport system might be quite easy to model, whereas a tiny thing such as an IC is way more complex to model due to the level of detail necessary. Complexity of a system therefore always depends on the level of detail in the underlying model.

The complexity and interconnectivity of our world and therefore the systems we create are continuously growing. This is were the term engineering comes into play. Often times engineering describes a discipline that uses methods and tools in a structured way to develop a product. Now Systems Engineering is an engineering discipline based on systems thinking, which in turn is a way of thinking that considers both the system as a whole and the interaction of its parts. According to the definition by the *International Council on Systems Engineering* (INCOSE), Systems Engineering is:

> *"... a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods." [INC23]*

Systems Engineering describes a unified and structured process that aims at developing a complex technical system. Development includes concept development, implementation, operation and if applicable also deconstruction and recycling. Due to the generality of the system term and the variety of development steps it follows that Systems Engineering describes a multidisciplinary approach, including but not limited to all kinds of engineering disciplines - from computer science, electrical engineering, mechanical engineering all the

way to material engineering or chemical and biological engineering. Most of the times also economic aspects must be considered and appropriate expertise is required. The overall result must be system that adheres to the requirements, solves a given problem and meets the user's needs.

According to [Wei07] Systems Engineering can also be seen as a discipline that stands above all other, more specific disciplines, such as software or hardware engineering. It is often times also described as a meta-discipline and as such provides a holistic way of thinking. This way of thinking enables engineers to solve problems on system level and therefore provides solutions of elegance that would otherwise not be possible.

Why is Systems Engineering difficult?

To understand the core difficulties that arise in Systems Engineering it's essential to understand the single steps that the Systems Engineering process consists of. Although there is no standardized process model, there exist several principles that Systems Engineering is built upon. To the derive the single steps of Systems Engineering let's first delve into the most important ones of these principles. As described in [INC23] Systems Engineering is a rather young discipline that evolved of the past 30 years. During this time a set of principles have emerged that define how Systems Engineering is practiced. Important consistencies among all those principles are that they

- outlast a system's particular life cycle,
- are not specific to a system type,
- are not 'how to' statements
- and are supported by literature or have proven successful across a wide range of system types and organizations.

In 1993 INCOSE put together a set of Systems Engineering principles for the first time. Over time some of these initial principles were found not to adhere to the foundations of Systems Engineering principles as itemized above. Consequently, the set of Systems Engineering principles was updated in 2022 and now comprises 15 principles, the most important of which are listed in Table 2.1.

Based on those principles INCOSE tries to break down the Systems Engineering process into single steps. Although a term 'step' often implies sequential execution INCOSE stresses that this is not the intended understanding of their Systems Engineering process. Rather those steps are to be understood as stages that can be executed arbitrarily often and in parallel. Moreover, stages have decision gates for entry and exit. The entry decision gate ensures that certain criteria are met, e.g. that all resources needed for the particular stage are available. The exit decision gate ensures that the targets of the concerning stage are met and the risk of proceeding is reasonable. These abstract definitions become more clear when we take a look at the actual stages. INCOSE defines

1	SE in application is specific to stakeholder needs, solution space, resulting system solution(s), and context throughout the system life cycle.
2	SE has a holistic system view that includes the system elements and the interactions amongst themselves, the enabling systems, and the system environment.
5	The real system is the perfect representation of the system.
6	A focus of SE is a progressively deeper understanding of the interactions, sensitivities, and behaviors of the system, stakeholder needs, and its operational environment.
8	SE addresses stakeholder needs, taking into consideration budget, schedule, and technical needs, along with other expectations and constraints.
9	SE decisions are made under uncertainty accounting for risk.
10	Decision quality depends on knowledge of the system, enabling system(s), and interoperating system(s) present in the decision making process.
13	SE integrates engineering and scientific disciplines in an effective manner.

Table 2.1: Excerpt of the 15 principles of Systems Engineering as described by INCOSE in 2022.

the stages as *Concept*, *Development*, *Production*, *Utilization*, *Support* and *Retirement*. Figure 2.3 illustrates the order in which those stages might be passed. It follows naturally that stages are passed in parallel and arbitrary many times: the development stages overlaps with production (e.g. for prototyping) and production overlaps with utilization because not all items are produced before selling starts. After some times there might be feedback available from the utilization stage and the system enters the concept stage again and receives improvements.

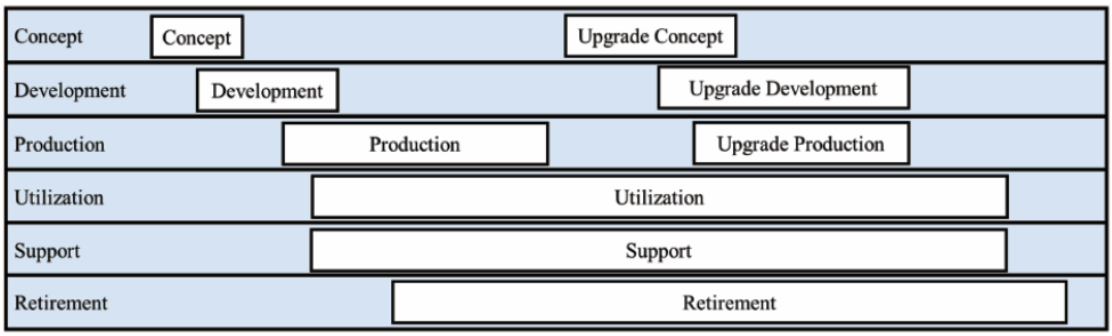


Figure 2.3: The stages of the Systems Engineering process by [INC23]. The stages can be passed several times and in parallel.

In order to better understand in which of these stages the present work provides assistance, we will first take a closer look at how INCOSE defines them and why this makes Systems Engineering a difficult discipline.

Concept. The concept stage consists of exploratory research to identify needs and (potentially existing) solutions, assess feasibility, and define preliminary requirements

and estimates. The output of this stage significantly shapes the project's path.

Development. The development stage refines the system concept into an engineering baseline producing detailed plans and requirements to ensure the system can proceed to production and other succeeding stages. Thereby modeling, simulation and prototyping takes place achieving system balance and optimization for key parameters.

Production. The production stage implements the baselines from the development stage and produces and an actual system along with necessary documentation for future stages.

Utilization. In the utilization stage the system transitions into use. Often modifications are introduced throughout the stage to address deficiencies, which have been remedied by a concept or development stage running in parallel.

Support. The support stage provides necessary support for the system's utilization. Planning of this stage has to take place before it starts. Results include deficiency notes being taken into account in future concept and development phases.

Retirement. Throughout this stage the system or a system element and its related services are removed from operation. SE activities in the support stage primarily focus on ensuring that extensive disposal requirements are satisfied.

Let's now take a look at what makes systems engineering so difficult at its core. In addition to the complexity of the system itself, many difficulties in SE originate in the temporal and logical distance between conception, development and utilisation.

The consequences of the decisions that have to be made during the concept and development phase will only become effective in later phases. However at the time they do become effective, even small changes to the system properties may be extremely costly in regards of financial, human or time resources. Depending on when an undesired effect or behavior becomes visible, it might be not even be possible anymore to adapt the system specifications accordingly and an entire reiteration is necessary. This work aims to address this issue by deploying appropriate technology to estimate the effects of decisions made in the concept and early development stages. We demonstrate that through the development of custom machine learning and optimization techniques, it is possible to predict decision effects during any phase of the conception and development stages, given a simulation model that provides the necessary level of detail. This approach, which falls under the term of Computer-Aided Systems Engineering, supports Systems Engineering at various stages and aligns with the 15 core principles of Systems Engineering outlined in Table 2.1. Specifically, Computer Aided Systems Engineering addresses the challenges associated with principles 9 and 10, which describe the uncertainty involved in Systems Engineering decision-making.

2.3.2 Model Based Systems Engineering

Modelling has been an important pillar of Systems Engineering ever since. However, increasing scale and complexity of modern systems necessitate continuous innovation

and rethinking of the Systems Engineering process. These efforts led to a new paradigm called Model-Based Systems Engineering (MBSE). MBSE is an approach to Systems Engineering that uses formalized modeling to support Systems Engineering activities such as requirements engineering, design, analysis, verification, and validation. MBSE shifts the focus from traditional document-based Systems Engineering to a more integrated, model-centered approach. This transition promises greater rigor and effectiveness for developing complex systems. MBSE introduces the model as single source of truth that reflects the entire state of the system development process [MS18].

The motivation behind MBSE is manifold. Research has shown that Systems Engineers waste lots of time searching for information and writing reports, a problem which is only reinforced by the growing complexity of modern systems. Document centric approaches are becoming more and more infeasible as they hold the risks of missing critical information. MBSE addresses these challenges by maintaining a shared context that contains key system requirements, usage scenarios, high-level architecture and so on in a compact, manageable model. In large projects where systems may grow immensely complex or may be nested (systems of systems), the document centric approach also poses challenges on keeping the entire project library consisting of single documents for each view (e.g. hardware, firmware) of each subsystem up-to-date. This bears the risk of introducing incompleteness and inconsistencies. To circumvent this problem MBSE provides the ability to automatically generate these documents - if even necessary - from the model. The model as the sole source of truth reliably reflects the most up-to-date state of the system development and so the documents generate from it do as well. Another problem that MBSE tackles is related to block diagrams. Block diagrams are often used for communication purposes among Systems Engineers. However they tend to cause confusion and ambiguity as their semantics are often not standardized and vary from team to team. This makes them inconsistent and hard to verify. According to the literature often times these issues hinder clear communication among stakeholders with different backgrounds. MBSE promises to overcome these challenges by providing clear, consistent, and formalized representations of needs and designs [MS18].

Over the past two decades, during which MBSE has mainly been developed, some key principles have emerged that define MBSE at its core. Among those are [MS18]:

Centralized Model Repository: MBSE utilizes a centralized model repository that serves as a single source of truth. This repository serves as *the* reference for all stakeholders, preventing misunderstandings.

Formal Languages and Notations: MBSE employs standardized modeling languages such as the Systems Modeling Language (SysML) and Unified Modeling Language (UML). These languages provide a common notation to describe system architecture, behavior, and data flow comprehensively.

Lifecycle Integration: MBSE spans the entire system lifecycle (or the 'stages' from section 2.3.1 Why is Systems Engineering difficult?, from conceptual design through to system deployment and maintenance. It enables continuous verification and validation

via up-to-date models that reflect the current state of the system.

Automation and Tool Support: MBSE leverages powerful modeling tools like IBM Rational Rhapsody [1], Cameo Systems Modeler [2], and others. These tools offer capabilities for simulation, analysis, and automatic documentation generation.

Enhanced Collaboration: By providing a visual representation of the system, MBSE facilitates better communication among multidisciplinary teams. This collaboration is crucial for maintaining a common understanding of different system aspects among teams of different engineering domains

Enhancing the Classical Approach with Simulation

Model-bases Systems Engineering in large parts is still subject to ongoing research [MS18] and therefore far from being fully adopted within the industry. What companies came up with is some kind of a hybrid approach. This means the Systems Engineering process is still document-based however models are built to complement the classical approach. These models don't serve as a single source of truth but are rather used for simulation purposes. Simulation is then used to estimate upfront how design decisions effect the overall system regarding a certain aspect. The observed system aspect, along with factors like the level of detail and the simulation framework used, is entirely flexible and can be tailored by the model developer. This flexibility is a feature that MBSE does not provide, at least if common modelling languages such as SysML/UML are utilized, as suggested in the literature [Wei07].

A well suited simulation framework for many real-world applications, such as SoC or IC design is an event-oriented approach. The event-oriented simulation paradigm is based on the assumption that the entire system behavior is based on discrete events (e.g. signal transition with the rising clock edge) rather then continuous processes (e.g. die temperature curve). Event based simulation works on a set of events that is stored together with the time of occurrence. The simulation maintains an efficient datastructure (e.g. a priority queue) of such events. The simulation now occurs in a reversed manner compared to what one might expect. Instead of simulating the passage of time and checking which events are currently occurring, the events are sorted and the simulation time is set to the time of the next scheduled event. The time until this event is skipped, as it does not bring any changes to the system state (because it's discrete) [Mat08].

Unfortunately purely event-based simulation tends to result in hard-to-read, messy code. To address this shortcoming process-oriented simulation was invented. Process-based simulation introduces stateful processes that can be seen as a collection of events. Now instead of processing events one after another, processes hand over the control flow from one to another. A process models an enclosed entity or resource of the simulated system, e.g. a CPU or a write operation to a bus. Process oriented simulation produces modular

[1] https://www.ibm.com/products/systems-design-rhapsody
[2] https://www.3ds.com/products/catia/no-magic/cameo-systems-modeler

code that is easier to read, understand and maintain. Several open source frameworks implementing process oriented simulation have been developed, with SimPy being the most popular one among them [Sim23].

SimPy SimPy is a Python framework that provides an easy-to-use environment for writing process-oriented simulations. It makes use of Pythons generator functions, where the `yield` statement is used to return a value but keep the state of the function (local variables and position of execution). This functionality is perfectly suited to implement a process. The value returned by `yield` is an event, that is scheduled by the SimPy engine. Internally the event is stored together with a callback to the event-`yield`ding function. Once an event is triggered, SimPy will execute this callback to resume the process and continue its execution. This mechanism allows processes to interact with each other and with shared resources, enabling complex coordination and synchronization.

For example, when a process requests a resource, it yields a resource request event. SimPy schedules this event and, once the resource becomes available, the callback associated with the event resumes the process. This approach allows for the simulation of complex systems with interactions between multiple entities and resources, making SimPy a powerful tool for modeling dynamic systems [Sim23] [Mat08].

2.3.3 Optimization Oriented Systems Engineering

With the availability of detailed system models the question arises how they can effectively be utilized to enhance the systems engineering process. There are two main strategies to utilize a simulation model for Systems Engineering.

The first one is to find design parameters that lead to ideal design properties (e.g. in terms of power consumption, area, ...). This can be formalized as the following optimization problem:

$$\underset{\mathbf{P_d} \in D_1 \times \cdots \times D_n}{\operatorname{argmin}} \; f(\mathbf{P_{fix}}, \mathbf{P_d}), \qquad \mathbf{P_{fix}} \in F_1 \times \cdots \times F_m \tag{2.1}$$

where $\mathbf{P_d}$ is an n-dimensional vector where each entry $P_{d,i}$ corresponds to the value of the ith variable design parameter with domain D_i and $\mathbf{P_{fix}}$ is an m-dimensional vector where each entry $P_{fix,i}$ corresponds to some fixed design parameter with domain F_i. f is a scalar function that evaluates the quality of the design. Note that depending on the nature of f our goal may be to either maximize or minimize it. Without loss of generality, it is assumed that optimization refers to the minimization of an objective function throughout this thesis. This assumption is reasonable due to the equality of minimizing f and maximizing $-f$, for any given function f.

The second use-case of a simulation model in the systems engineering domain arises from the fact that systems are becoming increasingly complex. In order to evaluate a given design decision it is necessary to estimate its worst case consequences. Traditionally, expert domain knowledge was sufficient to identify which system use-cases could lead

to worst-case scenarios. However, due to the rising systems complexity it is no longer possible for humans to reliably predict these scenarios. For example modern cellular protocols such as LTE and NR introduce a variety of new technologies leading to a vast amount of new use-cases compared to simple protocols like GSM or UMTS. This causes an immense spike in the complexity of state-of-the-art cellular transceivers. The overlapping of scenarios that were previously known to be worst-cases may now lead to entirely new and unforeseen worst-cases. This effectively renders the traditional approach, which solely relies on the knowledge experts with decades of industry experience, infeasible. Fortunately, simulation can help in this case as well. The core idea is to computationally evaluate a potentially critical design decision with regards to its worst-case consequences. In previously used terms, for a given design, defined by fixed parameters $\mathbf{P_{fix}}$ and design parameters $\mathbf{P_d}$ (now constituting the design decision of interest), we now seek input parameters $\mathbf{I}$ (constituting the worst-case) that minimize the score of a simulated system run with those inputs. Formally this is expressed through the following optimization problem:

$$\underset{\mathbf{I} \in I_1 \times \cdots \times I_n}{\operatorname{argmin}} f(\mathbf{P_{fix}}, \mathbf{P_d}, \mathbf{I}), \qquad \mathbf{P_{fix}} \in F_1 \times \cdots \times F_m, \quad \mathbf{P_d} \in D_1 \times \cdots \times D_n \tag{2.2}$$

where now f is a score calculated from a set of interesting parameters observed during the simulated system run. These parameters must effectively characterise the potentially adversarial effects of the design decision $\mathbf{P_d}$. However, using the transformation $\tilde{\mathbf{P}}_{\mathbf{fix}} = [\mathbf{P_{fix}}, \mathbf{P_d}]$ the second optimization problem 2.2 can also be written as

$$\underset{\mathbf{I} \in I_1 \times \cdots \times I_n}{\operatorname{argmin}} f(\tilde{\mathbf{P}}_{\mathbf{fix}}, \mathbf{I}), \qquad \tilde{\mathbf{P}}_{\mathbf{fix}} \in F_1 \times \cdots \times F_m \times D_1 \times \cdots \times D_n \tag{2.3}$$

which yields essentially the same problem as stated in equation 2.1. It shows that both equations 2.1 and 2.2 are actually a different view of the identical problem.

This justifies and encourages the usage of methods in the systems engineering domain that have previously been reserved for design optimization, leading to the term of Optimization Oriented Systems Engineering. This approach, not really documented in the literature, aims to enhance the decision-making quality during the Concept and Development phase of Systems Engineering. This is particularly important as bad decisions made in these stages might stay unnoticed for a very long time leading to costly and hard to resolve bugs. Although some similarities exist, this approach differs fundamentally from Model Based Systems Engineering in that it incorporates modeling and optimization but not making the model *the* central part of Systems Engineering.

2.4 Formalization of Optimization

Optimization is a fundamental problem in many scientific disciplines such as Mathematics, Engineering, Economics and more. At its core, it involves selecting the best solution

among a set of alternatives. The solutions are typically given by the so-called *objective function* which is to be *minimized* or *maximized*. The set of feasible solutions is called the *search space*. Formally the problem may be stated as

$$\max_{\mathbf{x} \in S} f(\mathbf{x}) \tag{2.4}$$

or

$$\min_{\mathbf{x} \in S} f(\mathbf{x}) \tag{2.5}$$

where $f : S \mapsto \mathbb{R}$ is the objective function and S is the search space. Without loss of generality it is assumed throughout this thesis that optimization refers to the problem of minimizing an objective function. This is a reasonable assumption due to the equality

$$\min_{\mathbf{x} \in S} f(\mathbf{x}) = \max_{\mathbf{x} \in S} -f(\mathbf{x}) \tag{2.6}$$

The elements $\mathbf{x} \in S$ are points in the search space. A point $\mathbf{x}$ in the search space is a vector with entries $x_j, 1 \leq j \leq D$, where D is called the dimensionality of the search space. Depending on the nature of the elements of S optimization can be categorized into

- *continuous* problems, i.e. $\forall \mathbf{x} \in S : \mathbf{x} \in \mathbb{R}^D$
- *discrete* problems, i.e. $\forall \mathbf{x} \in S : \mathbf{x} \in \mathbb{Z}^D$
- *mixed* problems consisting of n continuous and $D - n$ discrete dimensions, i.e. $\forall \mathbf{x} \in S : \mathbf{x} \in (\mathbb{R} \cup \mathbb{Z})^D$

The problem may be subject to *constrains*, meaning that not all solutions in the target set of f are feasible. Only such $\mathbf{x} \in S$ where $g(\mathbf{x})$ for some function $g : S \mapsto \mathbb{R}$ adheres to specific constraints are desired. Formally

$$\min_{x \in S} f(\mathbf{x}) \tag{2.7}$$

subject to

$$\begin{aligned} g_i(\mathbf{x}) &= 0, \quad i = 1, \ldots, p \\ h_j(\mathbf{x}) &\leq 0, \quad j = 1, \ldots, q \end{aligned} \tag{2.8}$$

is an optimization problem with p *equality* and q *inequality* constraints.

The objective function can take a wide variety of forms. For some problems it might be a mathematical expression, for other problems it could be a physical experiment. The nature of f has a big impact on the set of feasible optimization strategies for the problem. As many heuristic strategies rely on repeated evaluation of the objective function, the evaluation cost of f plays a crucial role when choosing a strategy.

2.5 Expensive Black-Box Optimization

Many engineering problems require to solve optimization of objective functions that are not given by mathematical equations or any other formal description. Instead the objective functions is given as a so-called black-box. A black-box is characterized by its hidden internal structure. The only thing it does is providing outputs given inputs. Expensive black-box optimization refers to the optimization of black-box objective functions that are particularly expensive to evaluate, either in terms of time or computing resources or both. Such situations occur when the objective function is given by the results of e.g. a computer simulation or physical experiments. In such cases the development of a mathematical model is simply not feasible, at least under reasonable effort [AH17]. Expensive black-box optimization has been a very active field of research for more than two decades already [JSW98]. Successful applications range from multi-disciplinary engineering problems such as automated drug discovery [TSTT21] and the development of aircraft wings [VSS06] to highly specialized problems, such as configuring a cooking robot to make optimal omelettes [JHTI20]. The evaluation of the objective function in the latter case involves tasting by human testers.

2.6 Problem Statement

The particular problem that this work deals with is solving the problem

$$\underset{\mathbf{I} \in I_1 \times \cdots \times I_n}{\operatorname{argmin}} \; f(\tilde{\mathbf{P}}_{\mathbf{fix}}, \mathbf{I}), \qquad \tilde{\mathbf{P}}_{\mathbf{fix}} \in F_1 \times \cdots \times F_m \times D_1 \times \cdots \times D_n \tag{2.9}$$

which was established in section 2.3.3 Optimization Oriented Systems Engineering. f is an expensive black-box function given through an event-based model of an integrated circuit (see section 2.3.2 Enhancing the Classical Approach with Simulation for details). The value of f is a score determined through simulation and based on the determined system property that changes depending on the inputs $\mathbf{I}$. Possible optimization strategies should be researched and compared and, if promising, implemented. These solutions should be independent of the specific search space and the implementation of the model itself. The search space is assumed to be mixed and to have between 1 and 100 dimensions. In order to tackle this search space the solution must be parallelizable and scale well on several hundreds of cores. The developed methods should be benchmarked with a model of the control architecture of a state-of-the-art cellular transceiver (see sections 2.1 Mobile Communications and 2.2 Cellular Transceiver for details).

2.7 Modern approaches to Optimization

An incredible amount of methods that aim to solve optimization problems exist. Over the time different scientific disciplines came up with different approaches that all have their strengths and weaknesses. In this context the "no free lunch theorem" of optimization and search plays an important rule. It states, that in black-box optimization no algorithm is

clearly superior to all others when considering all problems [WM97]. However, that does not mean that algorithm performance does not vary for specific problems - it clearly does. Therefore, the core challenge in optimization is to find the best method for a specific problem. To do so, the following section introduces various concepts in optimization such as mathematical, heuristic and approximate optimization.

Mathematical Optimization

The availability of a mathematical description of the objective functions allows the application of mathematical optimization methods. The benefits of such an approach are at apparent. These methods offer rigorous mathematical guarantees on both solution optimality and runtime efficiency. However formulating problems into precise mathematical models can be challenging and time-consuming, particularly for complex real-world scenarios. What follows is a short overview of two well-known mathematical methods, illustrating the difference to heuristic approaches.

Gradient descent Gradient descent makes use of the fact that given a differentiable function $f(\mathbf{x})$, moving from a point $\mathbf{x_0}$ into the opposite direction of the gradient, $-\nabla f(\mathbf{x})$, will decrease the function the fastest way possible. It follows that if we construct the sequence

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \gamma_k \nabla f(\mathbf{x^k}) \tag{2.10}$$

with a reasonably small step size $\gamma_k \geq 0$, the function values associated to the sequence decrease, i.e. it holds that

$$f(\mathbf{x}^{k+1}) \leq f(\mathbf{x}^k) \quad \forall k. \tag{2.11}$$

[Pol20]

From this observation it's possible to construct an optimization algorithm by starting with an initial guess $\mathbf{x_0}$ and updating the n-th solution by computing $\mathbf{x}^{n+1}$ according to equation 2.10. This algorithm is guaranteed to converge under certain conditions. First, the gradient $\nabla f(\mathbf{x})$ must satisfy the Lipschitz-condition

$$||\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})|| \leq L||x - y|| \tag{2.12}$$

and γ_k has to satisfy

$$0 < \gamma_k < \frac{2}{L} \tag{2.13}$$

where L is the Lipschitz-constant from 2.12 [Pol20]. Introduced in 1847 by Augustin-Louis Cauchy, this algorithm gained massive popularity in recent decades as it is the foundation of *Backpropagation*, the training algorithm for most modern Deep Neural Networks.

Linear Programming Linear programming is a method for optimizing linear functions $\mathbf{c}^T\mathbf{x}$ (commonly referred to as the cost function with cost vector $\mathbf{c}$) subject to inequality constraints $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$. The constraints form a feasible region in the shape of a convex polytope, a geometric object with flat faces enclosing a convex sets of points. Georg Dantzig recognized that optimal solutions always occur along the edges of this polytope and constructed an algorithm from this insight. It walks along the edges until reaching a guaranteed optimum. The algorithm is known as the Simplex algorithm and was introduced in 1946 [Pol20].

It quickly becomes clear that these methods require a differentiable function (e.g. for gradient descent) or even a linear function (for linear programming) as a mathematical model of the problem. Therefore they are not suitable to solve the given problem of optimizing an objective function given by a simulation model.

2.7.1 Optimization as a machine learning problem

In order to further clarify the problem of optimizing a black-box function this section gives a differentiation between such an optimization problem and problems that are usually solved with reinforcement learning. For better understanding a short overview of reinforcement learning follows.

Reinforcement Learning

Learning through interaction with our environment is a very fundamental principle of human intelligence. Reinforcement learning (RL) is the attempt to mimic this type of learning in computer programs. It involves interactions between the computer program (the learner, typically referred to as the agent) and an environment, which responds to the agent's actions and rewards it upon reaching a desirable state (some intermediate or final goal). Unlike supervised learning, where an agent learns from labelled examples provided by an external supervisor, in RL the agent learns to map situations to actions that maximizes the lifetime reward through trial and error. This process is illustrated in figure 2.4. [SB18]

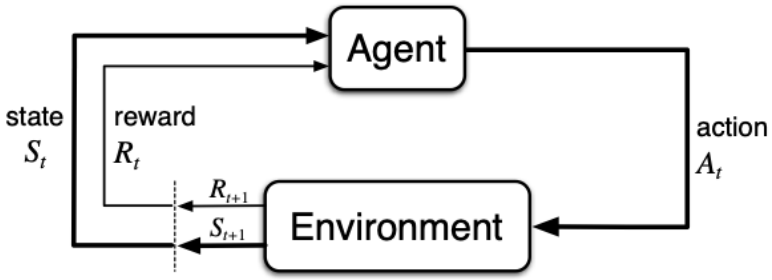


Figure 2.4: The interaction between agent and environment in reinforcement learning [SB18]

The essence of RL is captured in the Markov decision process (MDP). An MDP is a 4-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$ where

- $\mathcal{S}$ is a set of possible states s that the environment can take on. The state at time t is denoted as S_t.
- $\mathcal{A}$ is a set of possible actions a that the agent can take. A_t is the action at time t.
- $\mathcal{R} \subset \mathbb{R}$ is the reward probability $R(s, a, s')$. It represents the probability that the agent receives a reward depending on the action a taken in s upon transitioning into successor state s'.
- $p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t, A_t\}$ represents the probability of transitioning to state s' and receiving reward r upon taking action a in state s.

The definition of p is called the Markov property. It means, that the environment's response S_{t+1}, R_{t+1} only depends on S_t and A_t but is independent of the previous history $S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t$. The Markov property ensures that future states and rewards can be predicted with the knowledge of the current state just as good as with knowledge of the entire history.

The goal in RL is to learn a policy $\pi(a|s)$ that expresses the probability of taking action a in state s. The policy can be seen as a mapping from state to actions, that includes a certain degree of uncertainty. At each time t it should guide the agents behavior to maximize the expected cumulative future reward G_t

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.14}$$

where $\gamma < 1$ is the discount factor. Future rewards are discounted to incentivize the agent to collect rewards and therefore reach its goal as fast as possible. Furthermore γ ensures that G_t is a finite number. Typically the environment will be designed to provide a positive reward when the goal is reached and a negative reward when an undesired state is reached (e.g. a robot crashes).

In order for the agent being able to learn which action to take in state it needs a notion of the utility of a state. Therefore we first need to define the total expected reward of a state s as

$$R(s) = \mathbb{E}[R_{t+1}|s] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}, a \in \mathcal{A}} p(s', r|s, a) \tag{2.15}$$

Now we can define the Utility U of a state s as

$$U(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} U(s') p(s'|s, a) \tag{2.16}$$

where $p(s'|s,a)$ refers to the marginalization $\sum_{r\in\mathcal{R}} p(s',r|s,a)$. The Utility represents the expected discounted sum of rewards until termination assuming optimal actions. Under the assumption that the Utility function is known, it's easy to define an optimal policy π^*:

$$\pi^*(s) = \underset{a\in\mathcal{A}}{\operatorname{argmax}} \sum_{s' in \mathcal{S}} U(s')p(s'|s,a) \quad \forall s \in \mathcal{S} \tag{2.17}$$

The big challenge in RL is that p and therefore neither R or U are known but must be learned through interaction with the environment. This involves among others the challenge of balancing exploration and exploitation during training. Exploring the environment instead of exploiting knowledge gained so far bears the risk of not collecting the maximum reward but potentially let's the agent discover even more rewarding states [SB18]. Important milestones in RL include the introduction of algorithms such as Temporal-Difference (TD) Learning in 1989 [Sut88], Deep Q-Learning (DQN) in 2013 [MKS+13], Deterministic Policy Gradient (DPG) in 2014 [SLH+14], Twin Delayed Deep Deterministic Policy Gradient (TD3) in 2018 [FvHM18], Deep Deterministic Policy (DDPG) Gradient (DDPG) in 2019 and the Dreamer in 2020 [HLBN20]. Built on the DQN algorithm, Google's AlphaGo was the first computer program to beat a professional Go player in tournament conditions in 2015. Nowadays RL is successfully applied in countless real-world tasks, such as autonomous driving, robotics, biology and many more.

Applicability of Reinforcement Learning

The successful application of reinforcement learning on many difficult real-world problems in recent years raises the question whether this technique is applicable for the present problem of expensive black-box optimization. Applying RL to a problem involves formulating the problem as an MDP (see section 2.7.1 Reinforcement Learning). To define an MDP the notion of a state, an action, a transition and a reward must be defined. Bengio et al. [BLP20] point out several works successfully formulating combinatorial optimization problems as MDPs and solving them with reinforcement learning approaches. Khalil et al. [DKZ+18] for example effectively learn algorithms to solve the NP-hard graph problems *Minimum Vertex Cover*, *Maximum Cut* and the *Travelling Salesman Problem*. They apply a greedy algorithm where the selection of the next vertex to add to the solution is learned. Their MDP formulation looks as follows: as a state they define the set of currently added vertices, the action space is defined by vertices that are still available for adding and the reward is the change of the cost of the current solution compared to the previous solution. The transition probabilities are negligible in this case as there is no stochasticity involved in adding a vertex to a graph. The given problem has both combinatorial as well as continuous and integer parts. This makes it harder but an MDP formulation still seems feasible. On a high level, one would define a current candidate solution vector as the state, possible modifications to this vector as actions and as rewards the change in value of the objective function at the current candidate solution compared to the previous solution. It is obvious from the nature of the reward

function that after T time steps the cumulative reward $R = \sum_{t=0}^{T} R_t$ for time $0, 1, \ldots, T$ corresponds to the value of the objective function $f(S_T)$ for the terminal state S_T. Hence, a reinforcement learning agent aiming to maximize its lifetime reward will indeed find a maximum of f (or a minimum of $-f$).

However, while this approach works in theory, there are several caveats to consider when it comes to practical implementation. The first concern is that the reinforcement learning approach does not provide the necessary flexibility. The given problem requires a solution that has the flexibility to work on objective functions as well as search spaces of almost arbitrary structure. It's not clear in advance if the search space renders the problem purely combinatorial, continuous or mixed. The objective function might be oscillating, non-convex or even mostly constant. One assumption of most machine learning approaches is that problem instances a drawn from a common distribution. The better the learned algorithm or model works on these instances, the poorer it usually performs on instances that are less probable or impossible under that distribution. Also the authors of the aforementioned work on reinforcement learning approaches for combinatorial optimization ([DKZ+18]) point out that their work is mostly relevant for use cases where the problem structure stays the same and only the data varies. This however cannot be guaranteed for the given problem. This could necessitate regular retraining or restructuring, making a reinforcement learning approach too cumbersome in this regard. In this case a more general and robust approach as provided by e.g. evolutionary algorithms is expected to be a better fit in the long term. Furthermore Bengio et al. in [BLP20] point out that at least for many combinatorial optimization problems a classical, hand-crafted heuristic performs better by several orders of magnitudes when compared to reinforcement learning approaches.

Another important aspect in regards of performance is parallelizability. With the availability of massively parallel computing resources parallelizability becomes the dominant factor when choosing a solution. A sequential solution that performs even several magnitudes better on a single core will not be able to keep up with a less performant but inherently parallelizable solution. This is another point against a reinforcement learning approach, as it is fundamentally not easily parallelizable.

For the sake of completeness it's worth noting that there still are machine learning approaches that are able to handle the given problem. Some of them are described in the subsequent sections and their implementation is reviewed in section 3.2.1 Bayesian Optimization. The distinguishing fact about this approach is, that it is not trained on predefined problems but learns from scratch every time during execution. This enables it to adapt to diverse objective functions and search space structures, including scenarios where the problem's nature—whether combinatorial, continuous, or mixed—is not initially clear.

Supervised Learning

Supervised learning refers to learning from examples. The examples are pairs of inputs and outputs of a function. A set of such pairs is also called *labelled data*, data refers to the inputs and the label is the corresponding output. The goal is to predict the outputs for unseen inputs, i.e. to learn the function. An example for a function to be learned is a mapping from pixel values to strings, where the string is a description of the image formed by the pixels (image recognition). The labelled data used for training will then be a labelled image data set. Besides reinforcement learning, where the agent learns good actions from rewards, and unsupervised learning, where structures in unlabelled data are learned, supervised learning is one of the three paradigms in machine learning [RN10]. After a short introduction to supervised learning the subsequent sections describe how a less well-known form of supervised learning can be applied to optimization.

Given is a *training set* of labelled data points $\{(x_1, y_1), \ldots, (x_N, y_N)\}$ which are samples of a function f, i.e. $f(x_i) = y_i \quad \forall i$. The task in supervised learning is to learn a function h that approximates the true function f. h is called the *hypothesis*. The quality of h is measured by how well it generalizes, i.e. how well it performs on previously unseen examples. Sometimes the definition of f is relaxed and instead of a function $f(x) = y$ the goal is to learn a conditional probability distribution $P(y|x)$. Depending on the nature of the sets over which x and y are defined, the learning task might also be referred to as *classification* (y is an element of a finite set) or *regression* (y is a number).

The hypothesis h is selected from a *hypothesis space* $\mathcal{H}$ which is defined before learning starts. If $\mathcal{H}$ contains the true function f, then the learning problem is *realizable*. A simple way to make every learning problem realizable is to define $\mathcal{H}$ such that it contains the set of all computable functions. However, this would render the problem computationally infeasible, showing that there is a tradeoff between the expressiveness of the hypothesis space and the complexity of the learning problem. Subfields of machine learning research efforts focus on particular types of hypothesis spaces. Deep Learning, for example, focuses on solving learning problems within the hypothesis spaces defined by multi-layer neural networks. In contrast, Decision Tree Learning concentrates on hypothesis spaces over decision trees.

To illustrate how learning is conducted let's assume a learning problem over a very simple hypothesis space. Let $\mathcal{H}$ be the class of univariate linear functions. The functions in $\mathcal{H}$ are parameterized by a vector $\mathbf{w} = [w_1, w_2]$ (referred to as weights) and are of the form $y = w_1 x + w_0$. The learning problem now refers to learning w_1 and w_2 and is known as *linear regression*. For each $\mathbf{w}$ we define a hypothesis

$$h_{\mathbf{w}} = w_1 x + w_0 \tag{2.18}$$

where the quality of $h_{\mathbf{w}}$ is measured by a loss function, typically the squared loss:

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(x_j))^2 = \sum_{j=1}^{n} (y_j - (w_1 x_j + w_0))^2 \tag{2.19}$$

We are looking for weights $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} Loss(h_{\mathbf{w}})$. The simplicity of $\mathcal{H}$ admits solving this problem analytically: to minimize the loss function, we take partial derivatives with respect to w_0 and w_1 and set them to zero. The derivatives can be computed as

$$\frac{\partial Loss}{\partial w_0} = \sum_j -2\,(y_j - (w_1 x_j + w_0)) = 0 \iff \sum_j y_j = w_1 \sum_j x_j + w_0 N \tag{2.20}$$

and

$$\frac{\partial \text{Loss}}{\partial w_1} = \sum_j -2 x_j\,(y_j - (w_1 x_j + w_0)) = 0 \iff \sum_j x_j y_j = w_1 \sum_j x_j^2 + w_0 \sum_j x_j \tag{2.21}$$

Solving this system of linear equations gives the optimal weights w_0 and w_1:

$$w_1 = \frac{N \sum_{j=1}^{N} y_j x_j - \sum_{j=1}^{N} x_j \sum_{j=1}^{N} y_j}{N \sum_{j=1}^{N} x_j^2 - (\sum_{j=1}^{N} x_j)^2} \tag{2.22}$$

$$w_0 = \frac{\sum_{j=1}^{N} y_j - w_1 \sum_{j=1}^{N} x_j}{N} \tag{2.23}$$

These weights minimize the squared loss, providing the best-fit linear model for the given data.

However in most cases the hypothesis space will be much more complex (up to hundreds of billions of weights [BMR+20]) and no closed-form solution will exist. In such cases the minimization problem over the highly dimensional weight space must be solved iteratively e.g. via gradient descent (see **?? ??**). Therefore after each example the learning algorithm sees, it updates each weight w_i of the model according to

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \alpha \frac{\partial}{\partial w_i} Loss(h_{\mathbf{w}}) \tag{2.24}$$

Intuitively, $\frac{\partial}{\partial w_i} Loss(h_{\mathbf{w}})$ quantifies the influence of weight w_i on the output error. Note that in the case of a multi-layer perceptron with many layers this derivative can be tedious to calculate, especially when the neuron that the weight belongs to, is buried deep within the network. However, it can be computed efficiently via a chain-rule based dynamic programming approach known as *backpropagation*, which is perfectly suited for massive parallelization on GPUs [RN10].

Sequential Model-Based Optimization

Sequential Model-Based Optimization (SMBO) is a class of optimization techniques that are based on the principles of supervised learning. SMBO algorithms learn a regression model predicting the objective function based on previously seen objective function values. Then they use this model to determine the next, most promising point for evaluation and update the model based on the newly observed value. In SMBO the learned model is called *surrogate* or *response surface model*. The training data set now consists of pairs $\{(\mathbf{x_1}, f(\mathbf{x_1}), \ldots, (\mathbf{x_n}, f(\mathbf{x_n})\}$ of observed function evaluations [HHLB11]. The algorithm is listed in listing 2.1. In line 1 the set of initial observations is defined. These observations are evaluations of the objective functions for a set of randomly drawn points. The next step in line 3 is to fit a model to these observations in a supervised learning manner. There exist many different approaches for the selection of the next evaluation (line 4) based on the model. The choice is between evaluating in areas of high uncertainty (potentially a new optimum is hidden there) and areas where the model predicts a high value (exploitation of known information). This trade-off between exploration and exploitation is central to SMBO algorithms. One specific form of SMBO is Bayesian Optimization, which is discussed in the subsequent chapter [BBBK11] [HHLB11].

Algorithm 2.1: Sequential Model-Based Optimization (SMBO) [HHLB11]

Input: objective function f
Output: best solution
1 $Obs \leftarrow$ set of initial observations $\{(\mathbf{x_1}, f(\mathbf{x_1})), \ldots, (\mathbf{x_n}, f(\mathbf{x_n}))\}$;
2 **while** time budget left **do**
3 $\quad \mathcal{M} \leftarrow FitModel(Obs)$;
4 $\quad \mathbf{x_{n+1}} \leftarrow SelectNextEvaluation(\mathcal{M})$;
5 $\quad y_{n+1} \leftarrow f(\mathbf{x_{n+1}})$;
6 $\quad Obs \leftarrow Obs \cup \{(\mathbf{x_{n+1}}, y_{n+1})\}$;
7 **end**
8 **return** $\text{argmin}_{(\mathbf{x},y)\in Obs}\, y$;

Bayesian Optimization

Bayesian Optimization (BO) is a specific instance of Sequential Model-Based Optimization (SMBO) that leverages Bayesian inference to guide the search for the optimal solution. In Bayesian Optimization, the surrogate model is typically a Gaussian Process (GP) but also other approaches such as Tree-structured Parzen Estimator (TPE) exist [BBBK11]. These approaches mainly deal with the $FitModel$ and $SelectNextEvaluation$ parts of the SMBO algorithm (2.1). Bayesian Optimization has become a popular method for numerous problems in a variety of research fields. Garnett et al. describe how they solved the problem of placing a small set of sensors to make predictions about a spatial field, such as air temperature. They managed to find good solutions using Bayesian Optimization [GOR10]. Lorenz et al. successfully applied Bayesian Optimization to find

parameters for a physical experiment in the domain of neuroscience [LSM+19]. Further applications in the fields of materials sciences, financial industry and machine learning are described in the literature [WJSO22].

These successful applications make Bayesian Optimization an interesting candidate and it was chosen for implementation. Section 3.2.1 Bayesian Optimization gives details about the algorithm using both Gaussian Processes and Tree-structured Parzen Estimators and shows how it is implemented.

2.7.2 Classical Approach: Heuristic Algorithms

Computing truly optimal solutions is infeasible for a lot of real world optimization problems. The unavailability of a mathematical model - as discussed in section 2.7 Mathematical Optimization, the computational intractability of the problem or other reasons may prevent us from efficiently computing exact solutions. However, it turned out that in many cases exact solutions are not necessary. In such cases, we turn to heuristic approaches, which offer practical means to find near-optimal solutions within a reasonable timeframe. A lot of times a "sufficiently good" solution obtained under reasonable cost (in terms of time, computing power, ...) is more useful then an exact solution under possibly exponentially rising cost. This chapter will explore several such heuristic algorithms in detail, providing insights into the pros and cons of applying them and highlighting considerations in the context of the given optimization problem.

Approximation versus approximate

One thing not to be mixed up in the terminology of heuristic algorithms are approximate methods and approximation algorithms. Approximate methods are a class of algorithms that consists of two subclasses, which are approximation algorithms and heuristic algorithms. To fully grasp the essence of heuristic algorithms, let's first contrast them with approximation algorithms. Approximation algorithms are algorithms that don't necessarily produce optimal solutions, too. However, they come with a guarantee on how "bad" the solutions gets in the worst case. Formally [Tal09],

Definition 1 (ϵ-approximation algorithm) *An algorithm is an ϵ-approximation algorithm with ϵ-approximation factor $\epsilon > 1$if it runs in polynomial time and for any input instance it computes a solution a such that*

$$a \leq \epsilon \cdot \ s \tag{2.25}$$

where s is the optimal solution.

An ϵ-approximation algorithm gives a performance guarantee, whereas a heuristic algorithm might possibly produce an arbitrarily bad solution. That guarantee of course comes with a trade-off between ϵ and computational effort. For example Christofides showed in 1976 that there exists a 2-approximation algorithm for the NP-hard euclidean

Travelling Salesman Problem (TSP where all vertices have euclidean distance) [Chr76]. However, there exists no ϵ such that there exists an ϵ-approximation for the general TSP. Such results are valuable from a theoretic point of view but introduce too much overhead to yield practically relevant algorithms.

Heuristic algorithms in contrast are designed to produce good solutions or even optimal solutions with reasonable computational efforts. However there is no guarantee that such a solution will *always* be found. There might exist input instances or entire classes of input instances where a heuristic algorithm performs really bad. The quality of a heuristic algorithm is entirely defined by how well it works in practice on real-world instances. Heuristic algorithms have gained popularity in the past 20 years and showed incredible good results on many large instances of NP-hard problems across multiple disciplines, ranging from structural optimization in VLSI, machine learning, bioinformatics, robot planning to scheduling problems [Tal09].

Throughout the subsequent sections several heuristic algorithms relevant to the given optimization problem are introduced. The first two, random search and local search are intended to demonstrate problems that might occur with heuristic search strategies. Their practical relevance is very limited. The subsequent methods, Simulated Annealing, Evolutionary Algorithms and Particle Swarm Optimization overcome these issues and are highly effective in exploring complex search spaces, finding near-optimal solutions, and avoiding local optima.

Random Search

In random search, randomly generated solutions are evaluated for a predefined number of iterations and the best seen solution is returned. For a sufficiently large number of operations the expectancy value of the distance between the found solution to the true global optimum will become arbitrarily small. Nowadays random search is used in as a simple form of hyperparameter optimization in machine learning [LL19]. However, the motivation keep exploring further heuristics is the idea that it should be possible to achieve better results than naive random search by exploiting advanced and problem-specific knowledge.

Local Search

In local search, the algorithm starts with an (potentially arbitrary) initial solution. It then generates a new candidate solution in the neighborhood of the current solution. This might happen through random perturbation or other problem specific operations. Next, the objective function is evaluated at the new candidate solution. If the function value of the candidate solution is better (smaller) then the previously seen optimum, otherwise the candidate is discarded and the procedure repeats. The big drawback of local search is it's susceptibility to get stuck at local optima.

Markov Chain Monte Carlo

As the previous two naive approaches show, heuristic optimization is all about collecting samples from the objective function. The difficulty is to determine where to sample next. Sampling according to a uniform distribution as in random search results in sampling in bad areas most of the time. It would be better to employ a more sophisticated method that samples more frequently in regions of the parameter space that are likely to yield better results. An example for a distribution that would immediately improve results is for example the softmax distribution

$$\text{softmax}(x) = \frac{\exp\left(\frac{f(x)}{T}\right)}{\sum_{\mathbf{x}\in S}\exp\left(\frac{f(\mathbf{x})}{T}\right)}$$

However, sampling from this distribution is non-trivial because the denominator cannot be easily computed. Markov Chain Monte Carlo (MCMC) methods provide a way for sampling from such complex distributions without explicitly computing them, which can be particularly useful for optimization problems. The idea behind MCMC is to construct a Markov chain that has the desired distribution as its equilibrium distribution. The equilibrium distribution (or stationary distribution) of a Markov Chain is a distribution of states that remains unchanged if a transition step is applied. Such an equilibrium distribution exists under certain assumptions. By running such a chain for a sufficient amount of time (the "burn in"), the collected samples approximate the desired distribution.

Although very different in detail, the following optimization alogrithms incorporate the MCMC sampling idea in some form. In particular, Simulated Annealing is an optimization algorithm based on the Metropolis-Hastings sampling algorithm, one of the most commonly known MCMC method.

Simulated Annealing

Simulated annealing (SA) is a heuristic optimization strategy based on the metal processing technique annealing. Annealing refers to a process where metal is cooled down in a controlled manner. If liquid metal is cooled down slowly enough, according to the laws of thermodynamics its atoms will find a state of minimal energy - which is a perfect crystal. Cooling it down quicker will raise the energetic state of the atoms of the final product. Researchers found, that the way in which the atoms find an energetically (near) optimal state can be used to build a heuristic optimization algorithm. Metropolis and Teller [MRR+53] used this concept in 1953 already to compute complex integrals in the context of the development of nuclear weapons at Los Alamos. Kirkpatrick et al. in 1983 [KGV83] and Cerny in 1985 [Čer85] were the first to use these insights to develop a general heuristic optimization method and called it Simulated Annealing. SA is a simple and robust algorithm suited for mixed and continuous optimization problems. Among its first application is the optimization of IC layout, a combinatorial optimization problem. Further applications include optimization of antenna arrays, warehouse logistics optimization and project scheduling problems [SK06].

Although these successful applications of Simulated Annealing indicate that the algorithm is a promising candidate for the problem of this thesis it was found out in the implementation part that it lacks efficient possibilities to be parallelized. This is a crucial disadvantage of SA. The details of the algorithm and the struggles with its parallelization are described in sections 3.2.2 and 3.2.5.

Evolutionary Algorithms

Evolutionary algorithms (EAs) are a class of algorithms that simulate the principles of biological evolution. By repeatedly applying fundamental evolutionary operations such as selection, reproduction and mutation, a population of candidate solutions evolves over time, progressively converging towards more optimal solutions.

At their core, EAs are based on Charles Darwin's theory of biological evolution, "The Origin of Species" [Dar59]. This theoretical foundation inspires the process where the fittest individuals are more likely to survive and reproduce, thereby passing their advantageous traits to the next generation. New traits arise during reproduction through a process called *mutation*. Mutations are random changes to the genome that occur due to the presence of radiation or unavoidable errors in the genetic copying process. These genetic changes result in either positive or negative changes in the ability of the concerned individual to live within its environment. This ability is also referred to as the fitness of the individual. Individuals with high fitness might be better in finding food, defending theirselves against predators or attracting partners. All in all, fitter individuals have higher probability to survive, reproduce and pass on their beneficial properties than individuals with lower fitness, that tend to fail to survive and reproduce, thus causing their disadvantageous genes to disappear. This process is called *selection*. During reproduction another process called *crossover* takes place. Crossover refers to a scenario during reproduction on a cellular level. During crossover, the genetic material of two parent organisms is recombined to produce offspring with a mixture of traits from both parents. This recombination of genetic material potentially results in individuals that are fitter then their parents. Given the enormously complex and intelligent forms of live that evolution has created it's reasonable to believe that simulating this process yields good solutions for an optimization problem as well [KMB+22].

And indeed, a literature review shows that EAs are applied successfully across diverse domains. For example in manufacturing, they are used to optimize processes like assembly line balancing and facility layout design. For energy management, they address power system optimization and renewable energy integration, improving utilization and grid stability. In finance, EAs support tasks such as stock market prediction and portfolio optimization, yielding accurate forecasts and effective risk management strategies [CDH+24].

These successful applications make Evolutionary Algorithms an interesting candidate and it was chosen for implementation. The section 3.2.3 Evolutionary Algorithm gives a detailed description of the specific algorithm applied for the problem of this thesis.

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization algorithm that was first proposed by Kennedy and Eberhart in 1995 and has grown immensely popular since [EK95]. It is similar to evolutionary algorithms (see Evolutionary Algorithms) in that it is both inspired by nature and based on a population of candidate solutions. In contrast to evolutionary algorithms, PSO maintains all population members throughout the entire run without selection. The interactions among these members lead to iterative enhancements in the quality of the solutions over time. In order to compute these interactions, PSO leverages the principles of swarm intelligence. The core principle of swarm intelligence is that all individuals maintain independent local knowledge while also sharing the most important information with the entire group.

A survey paper by Gad demonstrates the extensive success of PSO in various applications, particularly in black box optimization. These applications include disease detection and classification in healthcare, agricultural monitoring or flood control and routing problems [Gad22].

These successful applications make Particle Swarm Optimization an interesting candidate and it was chosen for implementation. The section 3.2.4 Particle Swarm Optimization gives a detailed description of the specific algorithm applied for the problem of this thesis.

CHAPTER 3

Methodology and Implementation

The previous chapter outlined the problem in detail and provided a comprehensive overview of methods to tackle the problem. However, the transition from theory to practice involves solving many unforeseen problems and thinking about many details. Four algorithms have been chosen for implementation. Those are Particle Swarm Optimization, the Evolutionary Algorithm, Bayesian Optimization using a Gaussian Process and Bayesian Optimization using Tree-structured Parzen Estimators. The details of these implementations and the implementation of the framework around them are provided in this section.

3.1 Evaluation and Parallelization Framework

The framework is built around the implementations of the different algorithms to support usage and benchmarking of the entire application. It's purpose is to create a layer of independence between the algorithms and the computation platform as well as the benchmarking and statistics tools.

The core component of the framework is a scheduler that abstracts over the computation platform. During execution any black box optimization will get to a point where the evaluation of the objective function is necessary. For parallel implementations (see 3.2.5 Parallelization of Optimization) batches of points must be evaluated. The evaluation of the objective function means running a simulation with a given set of parameters in the context of this work. To keep the algorithm implementations neat, the coding and algorithmic overhead of these function evaluations must be outsourced to a separate module, in this case the scheduler. Furthermore the scheduler supports single-core, multi-core and cloud computation of the specified function evaluations.

The scheduler exhibits a quite simple interface. For common usage it offers functions for scheduling evaluations, starting evaluations and retrieving the available results. When

scheduling evaluations, either a single point in the parameter space or a batch of points or a range can be specified. Before the evaluations starts, a platform must be specified (either locally single-core, locally multi-core or cloud computation). All evaluation results available are retrieved as a batch via a single (optionally blocking) call. The scheduler may also be used in a debug mode where it will not schedule the a simulation run but instead evaluate the given parameters for a mathematical function (e.g. the sphere or the Rastrigin function) in order to test convergence of the algorithm and for benchmarking purposes.

Internally the scheduler maintains two queues, one for pending evaluations and one for the results. Upon start of evaluation, it will take all pending evaluations and process them. The scheduler processes evaluations one after another (local single-core), in batches (local multi-core) or by distributing batches as cloud computing jobs. For batch evaluation the scheduler will process the pending evaluations in batches that are multiples of the number of available CPUs. This approach enables a simple implementation with little overhead. An approach supporting asynchronous parallel evaluations would offer performance improvement via a reduction in idle time. However, as most of the simulation runs take an equal amount of time no severe drawbacks in performance must be expected for the synchronous approach. Furthermore, the algorithms are aware of the number of CPUs available and will always schedule a number of evaluations that is a multiple of the CPU count. This ensures that no CPUs will idle in the last evaluated batch. The evaluations results are stored in a multiprocessing queue and made available to the algorithm via the scheduler interface.

The framework also offers tools for benchmarking and statistics. Therefore it collects the necessary data (e.g. runtime, convergence data, best results, etc.) during an optimization run and offers plotting functionality. Furthermore it's possible to collect and export checkpoints during an optimization run, ensuring no data is lost upon unexpected termination.

3.2 Implementation of Algorithms

In the sections Optimization as a machine learning problem and Classical Approach: Heuristic Algorithms we gave an overview of promising algorithms to solve the given problem. In this section we are giving a more detailed description and all the details of implementation as well as the hyperparameters used. For most hyperparameters there are standard values that can be found in the literature. For implementation, at some points open source libraries are used, which is also indicated in the subsequent paragraphs.

3.2.1 Bayesian Optimization

Bayesian Optimization (BO) is a specific instance of Sequential Model-Based Optimization (SMBO) that leverages Bayesian inference to guide the search for the optimal solution. The model is either a surrogate for the objective function (Bayesian Optimization via a

Gaussian Process) or it can be derived by maintaining distributions for good and bad values (Bayesian Optimization via Tree-structured Parzen Estimators).

The following section gives a detailed decription of how Bayesian Optimization work as well as the specific adaption to serve the needs of this thesis. To begin, we want to examine the process of model fitting in Bayesian Optimization. As the name implies, this approach is rooted in Bayes' theorem.

$$P[\mathcal{M}|Obs] \propto P[Obs|\mathcal{M}]P[Obs] \tag{3.1}$$

where $\mathcal{M}$ denotes the model and Obs is the set of observations as introduced in SMBO. However, as discussed in 2.7.1 we still need to define a hypothesis space $\mathcal{H}$ that defines the character of the model. In the Bayesian context this is usually known as the prior, in this case a prior distribution over functions, given the previous observations $\{(\mathbf{x_1}, f(\mathbf{x_1}), \ldots, (\mathbf{x_n}, f(\mathbf{x_n})\}$. One such model for a prior over functions is the Gaussian Process (GP).

Gaussian Process A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution. It is specified by its mean function $\mu(\mathbf{x})$ and covariance function $k(\mathbf{x}, \mathbf{x}')$:

$$\mu(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})]$$
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - \mu(\mathbf{x}))(f(\mathbf{x}' - \mu(\mathbf{x}'))]$$

To model a function we assume the Gaussian process contains an infinite number of random variables, each representing the function at a certain point. A sample drawn from a Gaussian Process corresponds to a function. The definition of $k(\mathbf{x}, \mathbf{x}')$ defines the nature of these samples, it is also referred to as the kernel function. A simple kernel function is the linear kernel where

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T\mathbf{x}' \tag{3.2}$$

Samples from this Gaussian Process with this kernel have the form $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$. A more expressive and therefore often used kernel is the squared exponential kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^T \mathrm{diag}(\theta)^{-2}(\mathbf{x} - \mathbf{x}')\right) \tag{3.3}$$

with a parameter vector θ. Now all that that the $FitModel$ function of the SMBO algorithm (2.1) does in Bayesian Optimization, is to fit a Gaussian Process to the set of observations.

In order understand what the *SelectNextEvaluation* of the SMBO algorithm (2.1) does in Bayesian Optimization, first the term of an *acquisition function* needs to be defined. The acquisition function is a function that indicates where it pays off most to evaluate the objective function next (maximizing the acquisition function will yield this point). Thereby it is the central aspect in Bayesian Optimization that trade off exploration against exploitation. Several ways to define this acquisition function have been introduced in the literature. The first one is known as *Probability of Improvement* (PI) and does exactly what its name suggests, i.e. calculating the probability that an evaluation of the objective function at a point $\mathbf{x}$ leads to an improvement of the previously known best point. PI is defined as

$$PI(\mathbf{x}) = P(f(\mathbf{x}) \geq f(\mathbf{x}^+) + \xi) \tag{3.4}$$

$$= \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})}\right) \tag{3.5}$$

where Φ denotes the normal cumulative distribution function and $\mathbf{x}^+ = \text{argmax}_{\mathbf{x} \in Obs} f(\mathbf{x})$ is the best known observation. ξ is a parameter trading off exploration against exploitation. Maximizing the $PI(\mathbf{x})$ yields the suggestions for the next objective function evaluation [Gar23].

While PI considers only the probability that an improvement over the current best is achieved, it does not pay attention to the amount of improvement. To improve on this shortcoming, an acquisition function named Expected Improvement (EI) has been defined. It is based on the improvement function $I(\mathbf{x})$:

$$I(\mathbf{x}) = \max(0, f(\mathbf{x}) - f(\mathbf{x}^+)) \tag{3.6}$$

Now EI is defined as

$$EI(\mathbf{x}) = \mathbb{E}[I(\mathbf{x})] \tag{3.7}$$

and has the analytical expression

$$EI(\mathbf{x}) = (\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi)\Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})}\right) + \sigma(\mathbf{x})\phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+) - \xi}{\sigma(\mathbf{x})}\right) \tag{3.8}$$

EI not only indicates whether to expect improvement but also what magnitude of improvement to expect.

Finally, the probably simplest acquisition function: Upper Confidence Bound (UCB). UCB at a point $\mathbf{x}$ is simply the sum of the expected value of the objective function at $\mathbf{x}$ and the uncertainty for this point, weighted by a exploration-exploitation tradeoff parameter λ:

$$UCB(\mathbf{x}) = \mu(\mathbf{x}) + \lambda\sigma(\mathbf{x}) \tag{3.9}$$

Lower Confidence Bound is the pendant of UCB for minimization problems and defined as $LCB(\mathbf{x}) = \mu(\mathbf{x}) + \lambda\sigma(\mathbf{x})$.

Note that the acquisition function is substantially easier to optimize then the objective functions, that GP-bases Bayesian optimization is applied to. Otherwise there would be no point in using the Gaussian Process. However, as pointed out by Bergstra et al. in [BBBK11] the runtime of fitting the GP model is in $\mathcal{O}(|Obs|^3)$, i.e. cubic in the number of available samples. Bergstra et al. claim that usually this cubic runtime is dominated by the objective function evaluations. This might be true for purely sequential implementations, however as through parallelization a significantly higher number of samples is generated, the cubic runtime is a significant drawback. During the implementation of this algorithm it was found that the GP approach essentially becomes infeasible when applied at large scale (see section 4 Experiments and Results for the details).

Tree-structured Parzen Estimator Using tree-structured Parzen estimators as a model in Bayesian Optimization was suggested by Bergstra et al. in 2011 for the first time [BBBK11]. This method was specifically developed to address the hyperparameter optimization problem for machine learning algorithms, aiming to find the set of hyperparameters that optimize the performance of a given model. Evaluating a set of hyperparameters involves training the machine learning model with them, which is computationally expensive. As such, hyperparameter optimization closely parallels the optimization problem addressed in this thesis, making the application of TPE-based algorithms a promising approach.

The TPE approach is different to the GP approach in that the goal is not to directly learn a surrogate for the objective function, $P(y|x)$, which is then used to suggest the next evaluation. Instead, TPE aims to learn directly where good and where bad values $\mathbf{x}$ of $f(\mathbf{x})$ are. Therefore the authors of [BBBK11] propose to maintain two probability densities:

$$l(\mathbf{x}) = p(\mathbf{x}|y \leq y^{\gamma}) \tag{3.10}$$

$$g(\mathbf{x}) = p(\mathbf{x}|y > y^{\gamma}) \tag{3.11}$$

where y^{γ} is the top γ-quantile of the observed objective function evaluations Obs. $l(\mathbf{x})$ is the density of points that yielded better (smaller) objective function values, i.e. it is computed from observations $\mathbf{x_i}$ with $f(\mathbf{x_i}) \leq y^{\gamma}$ and $g(\mathbf{x})$ is the density of points that yielded worse values. To compute these density functions so-called Parzen estimators, which are a form of kernel density estimators (KDE), are used. The authors of TPE claim that building the densities scales linearly in the number of observations $|Obs|$ in contrast to the cubic runtime when using the Gaussian Process [BBBK11].

In order to compute suggestions for the next evaluations, the TPE based algorithm now

simply samples from $l(\mathbf{x})$ and computes a score α for these samples:

$$\alpha(\mathbf{x}) = \frac{l(\mathbf{x})}{g(\mathbf{x})} \tag{3.12}$$

Samples that have a high probability under the "good" distribution $l(\mathbf{x})$ and low probability under the "bad" distribution $g(\mathbf{x})$ will achieve high scores. The sample with the highest score is suggested. TPE showed superior results and became popular in the hyperparameter tuning community [Wat23].

Implementation Details Gaussian Process (GP) based Bayesian Optimization consists of several building blocks: the surrogate model fitting, the acquisition function computation and optimization and the evaluation of the suggested points. For the surrogate model implementation as well as for the acquisition function the open source implementation of GPyOpt was used ([aut16]). In order to accelerate the fitting of the model, a sparse GP is used. While fitting a GP to n data points usually scales with $O(n^3)$, a sparse GP reduces this time complexity to $O(nm^2)$, where m is the size of a set of inducing variables [LDJD22]. As acquisition functions both Expected Improvement and Lower Confidence Bound are available. For batch sampling, which is necessary for parallelizing this algorithm, Pure Exploration and Local Penalization are available.

The implementation of TPE is based on the popular Hyperopt library by the group of Bergstra et al. which also invented the TPE algorithm [BKE+15]. For the parameter γ that defines the quantile used to separate the observations into the two sets which are used to construct the model, it was determined to set it to 0.25, which aligns with the standard recommendation provided by the literature. Parallelization of this algorithm is done by sampling batches from the acquisition function as described in the section Bayesian Optimization.

3.2.2 Simulated Annealing

Simulated Annealing (SA) is a heuristic optimization algorithm inspired by the annealing process in metalworking, where controlled cooling allows atoms to settle into a minimal-energy state. SA is a simple and robust method suitable for mixed and continuous optimization problems, with applications ranging from IC layout design to logistics and scheduling. It should be noted upfront that due to the lack of efficient parallelization possibilities for this algorithm, it has not been chosen for implementation. Nevertheless, it is such a crucial heuristic algorithm that it is still described here to maintain the completeness of this thesis. The core problems of parallelizing SA are described in section 3.2.5 Simulated Annealing.

The core principle of the algorithm follows local search. However, it overcomes its limitations by probabilistically accepting worse solutions, allowing the algorithm to escape local optima. In particular, the algorithm chooses a worse solution with probability $e^{\frac{-\delta}{T}}$, where δ is a measurement for how much worse the candidate solution is and T is a control

parameter referred to as temperature. The role of the temperature in SA is analogous to the role of the temperature in annealing, hence the naming. Just like the temperature in annealing controls how probable it is for atoms to end up in a state of higher energy, the temperature in SA controls, how likely a move to a less optimal solution is. A probability function is used such that a lower temperature leads to exponentially decreasing chances a suboptimal solution is accepted.

The complete algorithm is listed in listing 3.1. The procedure starts by initializing the temperature and the candidate solution (lines 1-2). The energy of the solution is the value of the objective function (line 3). Within the main loop repeatedly candidate solutions are generated (line 5). The exact strategy for generating a new candidate is strongly dependant on the specific problem and should ensure that the new solution is somehow related to the current solution. In the original paper by Cerny, where SA was applied to the Traveling Salesman Problem (TSP), a new candidate tour is generated by selecting two vertices in the current tour and reversing the order of traversal of the vertices between them. The next step in the algorithm is to evaluate the objective for the new candidate and observe the difference in energy (line 7). Better solutions are accepted immediately, worse solutions probabilistically as described previously with δ being the difference in energy (line 8-17). Finally after each iteration the temperature is updated according to a cooling schedule (line 18).

The initial temperature and the cooling schedule play a vital role: a fast cooling rate or a small initial temperature may lead to under-exploration of states, while a slow cooling rate or a high initial temperature increases runtime and computing costs. An important insight about the initial temperature by van Laarhoven is that it should be just high enough that initially almost all candidates are accepted. Let the desired initial acceptance rate be $0 < \chi_0 \leq 1$ and let the average difference in energy between two candidates be Δe_0 [van88].

$$T_0 = -\frac{\Delta e_0}{\ln(\chi_0)} \tag{3.13}$$

ensures that the probability p of accepting a solution in the first iterations is on average

$$p = e^{-\frac{\Delta E}{T}} = e^{-\frac{\Delta E}{-\frac{\Delta e_0}{\ln(\chi_0)}}} = e^{\frac{\Delta E \ln(\chi_0)}{\Delta e_0}} \approx e^{\ln \chi_0} = \chi_0 \tag{3.14}$$

Thus, choosing $\chi_0 \approx 1$ will ensure all possible transitions are considered initially. It's worth noting that this approach also aligns with the physical analogy: accepting all transitions can be seen as heating the metal until it is completely liquid.

Several cooling schedules have been proposed. Important schedules are the logarithmic and the Cauchy schedule. The temperature T_i at time i is given by $T_i = \frac{T_0}{\log i}$ for the logarithmic and by $T_i = \frac{T_0}{i}$ for the Cauchy schedule. It can be shown that for these schedules, the algorithm converges to the true global optimum when candidates are

generated according to a Gaussian distribution for the logarithmic schedule and a Cauchy distribution for the Cauchy schedule, respectively. A faster cooling schedule is given by $T_i = T_0 e^{-C_i}$ and a constant C. For this schedule, known as the exponential schedule, no convergence guarantees can be given. Some authors propose adaptive cooling schedules that update the temperature based on information gathered during the run [SK06]. There exists no cooling schedule that clearly outperforms any other. Hence, it must be decided by the algorithm engineer based on the specific problem. A good rule of thumb is to slow down cooling during phases where the evaluations improve rapidly.

Algorithm 3.1: Simulated Annealing [KGV83] [Čer85]

Input: objective function f
Output: best solution
$T \leftarrow T_{max}$;
$\mathbf{x} \leftarrow$ initial candidate solution;
$E \leftarrow f(\mathbf{x})$;
while T and E above threshold **do**
  $\mathbf{x}_{new} \leftarrow$ new candidate solution;
  $E_{new} \leftarrow f(\mathbf{x}_{new})$;
  $\Delta E \leftarrow E_{new} - E$;
  $p \leftarrow e^{\frac{-\Delta E}{T}}$;
  if $\Delta E < 0$ **then**
    $\mathbf{x} \leftarrow \mathbf{x}_{new}$;
    $E \leftarrow E_{new}$;
  else
    if $random < p$ **then**
      $\mathbf{x} \leftarrow \mathbf{x}_{new}$;
      $E \leftarrow E_{new}$;
    end
  end
  $T \leftarrow$ cooling schedule(T);
end
return $\mathbf{x}$;

3.2.3 Evolutionary Algorithm

Evolutionary algorithms (EAs) simulate natural evolution, using selection, reproduction, and mutation to optimize solutions. Inspired by Darwin's theory, they evolve populations by favoring fitter individuals, introducing variation through mutation, and combining traits via crossover. This approach effectively solves complex optimization problems.

The following section gives a detailed description of how EAs work as well as the specific adaptions made to serve the needs of this thesis.

Listing 3.2 shows how evolutionary algorithms work on a high level. Initially a population is generated randomly (line 2). Now the principle of evolution is applied to this population for a given number of iterations (line 3). First, the fitness of each individual is evaluated. This corresponds to the evaluation of the objective function for the parametrization that each individual represents. To identify the set of individuals that qualifies for reproduction the *selection* algorithm is applied (line 5). Next, the genetic operations of *mutation* and *crossover* are applied to obtain the offspring, i.e. the next generation (line 6-7). After termination of the loop the fittest individual is returned as the solution (line 9).

Algorithm 3.2: Evolutionary algorithms on a high level [KMB+22]

Input: objective function f
Output: best solution
1 $t \leftarrow 0$;
2 $\text{population}_t \leftarrow$ initialize population;
3 **while** t below threshold **do**
4 $\quad t \leftarrow t + 1$;
5 $\quad \text{fitness\_vals} \leftarrow$ evaluate f for population_{t-1};
6 $\quad \text{offspring}_t \leftarrow \text{selection}(\text{population}_{t-1}, \text{fitness\_vals})$;
7 $\quad \text{offspring}_t \leftarrow \text{variation}(\text{offspring}_t)$;
8 $\quad \text{population}_t \leftarrow \text{offspring}_t$;
9 **end**
10 **return** best individual from (population_t);

While the algorithm on a high level is fairly simple, there are a ton of different implementations, when it comes to the details. The variables are mainly the way of encoding a solution, the selection mechanism as well as the genetic operations (mutation and crossover).

Encoding. The encoding of candidate solutions plays a vital role when engineering an evolutionary algorithm. It defines the way the algorithm represents a candidate solution. This representation must strike a balance between simplicity and expressive power, allowing the algorithm to navigate efficiently through the search space. Common encoding schemes include binary strings, integer arrays, real-valued vectors, permutations, and more. The choice of encoding impacts different aspects of the algorithm, including the diversity of candidate solutions, the ease of applying genetic operators and the interpretability of the final solutions obtained. To illustrate the importance of the encoding, consider the *n-queens* problem. This problem refers to the question how to place n queens on an $n \times n$ chess board such that no two queens can attack each other. One possibility of encoding a candidate would be a binary matrix S with $s_{i,j} \in \{0, 1\}$, where the 1s indicate the positions of the queen. However, this would be a poor choice as the encoding allows solutions that are known to be invalid a priori (e.g. with several queens in a row). One would have to make sure to exclude such solutions when applying

the genetic operations. A smarter way is to structure the encoding such that it narrows down the search space in advance as far as possible. Using a simple permutation $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$, where $\pi(i)$ defines the position of the queen in row i satisfies this requirement [KMB+22].

Selection. The selection mechanism chooses the individuals best suited for reproduction, allowing them to pass on their genes to the next generation. It introduces so-called *selective pressure*, which quantifies how strongly fitter individuals are preferred over less fit individuals. Selection creates an exploration-exploitation trade-off where low selective pressure results in random search and high selective pressure tends to drive the population into local optima. A straightforward selection mechanism is *roulette-wheel selection*. It computes the relative fitness f_{rel} of each individual s based on its absolute fitness f_{abs}:

$$f_{rel}(s) = \frac{f_{abs}(s)}{\sum_{s' \in \text{population}_t} f_{abs}(s')} \tag{3.15}$$

Next, the subsequent population is drawn from the current population, where the probability of drawing s is $f_{rel}(s)$. Roulette-based selection suffers from two drawbacks. First, individuals with a very high fitness *dominate* the selection, resulting in an undesired small variance in the population. Second, too small distances in fitness values are also not desired as they might lead to *vanishing selective pressure*, where the selection probabilities are more or less equal among all individuals, rendering the selection random. Scaling the fitness function helps tackling both problems. Updating the fitness values according to

$$f_{lds}(s) = \alpha \cdot f(s) - \min\{f(s') | s' \in \text{population}_t\}, \quad \alpha > 0 \tag{3.16}$$

is known as *linear dynamic scaling*. Another popular scaling scheme is *σ-scaling*:

$$f_\sigma(s) = f(s) - (\mu_f(t) - \beta \cdot \sigma_f(t)), \quad \beta > 0 \tag{3.17}$$

where $\mu_f(t) = \frac{1}{|\text{population}_t|} \sum_{s \in \text{population}_t} f(s)$ and $\sigma_f(t) = \frac{1}{|\text{population}_t|} \sqrt{\sum_{s \in \text{population}_t} (f(s) - \mu_f(t))^2}$ are mean value and standard deviation of the individuals' fitness scores, respectively [KMB+22].

Applying the principles of a tournament yields another interesting selection mechanism, the so-called *tournament selection*. In tournament selection $k \in \{2, \ldots, |\text{population}_t|\}$ individuals are drawn from the population randomly and compete in a tournament against each other. The fittest individual wins a spot in the offspring generation. Note that each individual has the same chance to compete in a tournament, independently of its fitness. After the tournament the competing individuals are put back into the population and are allowed to compete again. The parameter k controls the selective pressure. Assume $k = |\text{population}_t|$, then any variance is removed and the offspring generation will consist of the fittest individual only. The smaller k becomes, the higher the chances are for weak individuals to establish themselves against the other $k - 1$ individuals in their tournament. Note that the populations weakest $k - 1$ individuals will

never win and their genetics are lost for sure. The number of tournaments conducted is equal to the size of the population. Tournament selection also addresses the problem of dominance as the probability of an individual to be selected does not directly correspond to its fitness [KMB+22]. The expected number of spots in the offspring generation for the fittest individual ind_{max} is the expected number of tournaments it competes in:

$$\mathbb{E}[\text{spots}(ind_{max})] = \frac{k}{|\text{population}_t|} \cdot |\text{population}_t| = k \tag{3.18}$$

Thus the dominance problem can be tackled by varying k. In particular, the expected number of spots for the fittest individual does not depend on it's fitness as it does with roulette-wheel selection.

The selection mechanisms introduced so far bear the risk that, although unlikely, the best individual of a generation may be discarded, resulting in the best individual of the next generation being less fit than the current one. To overcome this issue, a technique called *elitism* is applied. Elitism ensures that the k fittest individuals are guaranteed a spot in the next generation without undergoing selection or genetic operations such as mutation and crossover. In many practical applications this technique has proven to accelerate convergence towards the global optimum [KMB+22].

Genetic operations. Once the individuals selected for reproduction are chosen, the genetic processes that occur in nature during cell separation are simulated. These processes primarily include mutation and crossover. All individuals of the next generation (except the k fittest ones when applying elitism) undergo these genetic operations, even if there is a high probability of decreasing their fitness.

Mutation refers to a process that alters a single individual. Its biological background lies in the fact that during the process of cell division, random modifications to the genetic material can occur, influenced by external factors such as the presence of radiation or internal factors such as unavoidable errors during this complex process. To simulate mutation, evolutionary algorithms apply *mutation operators*. The nature of the mutation operator applied depends on the encoding of the candidate solutions. If a bit vector is used for encoding, a popular method to mutate it is to flip each bit with a given probability. The corresponding algorithm is given in listing 3.3. In practical applications choosing $p_m = \frac{1}{\text{length}(x)}$ has shown good results.

Algorithm 3.3: Mutation of a bit vector [KMB+22]

```
   Input: bit vector x, probability p_m
1  for i ← 1 to len(x) do
2  |  if random() ≤ p_m then
3  |  |  x_i ← 1 − x_i;
4  |  end
5  end
```

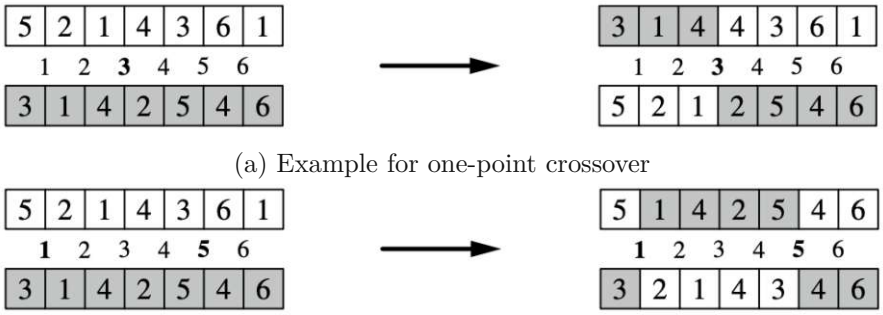
For solution encodings that contain continuous components, Gaussian mutation usually delivers good results [KMB+22]. Each continuous component is summed with a random number sampled from a Gaussian with $\mu = 0$. The value of σ is search space dependent and offers a way to control the balance between exploration and exploitation. Listing 3.4 describes Gaussian mutation in detail.

Algorithm 3.4: Mutation of a continuous-valued vector [KMB+22]

Input: continuous-valued vector $\mathbf{x}$, standard deviation σ

```
for i ← 1 to len(x) do
    u ← sample from N(μ, σ);
    x_i ← x_i + u;
end
```

Crossover refers to a process during cell separation where the genetic material of both parents overlaps and exchanges parts with each other. This exchange, that may or may not take place, can create new combinations of genes, potentially combining advantageous traits from both parents. This process is reflected in evolutionary algorithms by the application of *crossover operators*. Numerous crossover operators have been described in the literature with *one-point crossover* and *two-point crossover* being the most popular ones. In one-point crossover the candidate solution vectors of two individuals are cut at one randomly chosen point and the parts of the vector on one side of this point are exchanged between the two individuals. In two-point crossover two points of both vectors are cut at two random points and the part in between them is exchanged. Figure 3.1 illustrate both one-point and two-point crossover for a integer vector solution encoding.



(a) Example for one-point crossover

(b) Example for two-point crossover

Figure 3.1: Examples for the crossover operation.

A crossover operation that works well for continuous search spaces is *blend-crossover*. The blend-crossover operation defines for each component of the solution vector a space where it samples a new value from. The space depends on the distance of the component values of the parents, extended by some factor α. Then the new values are sampled randomly from this space. Listing 3.5 illustrates the algorithm in detail.

Algorithm 3.5: Blend-crossover operation [AXCS12]

Input: continuous-valued vectors $\mathbf{x_1}, \mathbf{x_2}$, parameter α

for $x_{1,i}, x_{2,i} \in \mathbf{x_1}, \mathbf{x_2}$ **do**
  $d \leftarrow ||x_{1,i} - x_{2,i}||$;
  $lo \leftarrow \min(x_{1,i}, x_{2,i}) - \alpha d$;
  $hi \leftarrow \max(x_{1,i}, x_{2,i}) + \alpha d$;
  $x_{1,i} \leftarrow \text{random}(lo, hi)$;
  $x_{2,i} \leftarrow \text{random}(lo, hi)$;
end

Implementation Details For Evolutionary Algorithms primarily the genetic operations mutation and mating, the selection algorithm and the encoding need to be defined. Regarding the encoding, a vector of float and integer values is suitable, where float values represent the continuous dimensions and integers represent the categorical dimensions. As a selection algorithm tournament selection was chosen with tournament size $k = 3$. Tournament is computationally very efficient and effectively tackles the dominance problem. The small tournament size favors exploration and prevents converging too fast. The selection mechanism is combined with elitism in order not to allow degradation of the solution. Elitism is implemented such that the top 3 % of individuals are always passed to the next generation. When it comes to genetic operations, Gaussian mutation is used for continuous variables and random mutation is applied on categorical variables. While Gaussian mutation is standard in the literature, the random mutation approach might surprise. However, it is the most natural way to do it as there is no real ordering or structure on categorical variables that allows to favor the selection of values "close" to each other. As a mating algorithm Two Point Crossover was chosen. The probability that crossover happens was set to 0.7, the probability for mutation to 0.2. Parallelization of this algorithm is done by evaluating all individuals in parallel, thus the population size is set to a multiple of the number of available CPUs.

3.2.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a nature-inspired optimization algorithm based on swarm intelligence. Unlike evolutionary algorithms, PSO retains all population members throughout its run, with individuals sharing key information while maintaining local knowledge to iteratively improve solution quality.

The following section gives a detailed description of how PSO works as well as the specific adaptions to serve the needs of this thesis.

The swarm is modelled as a set of N particles: $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_N}\}$. Each particle represents a candidate solution, i.e. a vector in the D-dimensional search space. In addition to the location each particle also gets an individual velocity. This velocity is then used to move the particle through the search space. Furthermore each particle keeps track of the

coordinates corresponding to the best solution (objective function value) achieved so far, which is called *pbest* in the literature. In some implementation each particle also knows the globally best solution found so far, named *gbest*.

The algorithm consists basically of two simple update rules for each particle. First the velocity is updated, afterwards each particle is moved with respect to its new velocity. The update rule itself varies from implementation to implementation, however the principle is always to move the velocity into the direction of regions of the search space that seem most promising. In natural swarms each individual decides locally in which direction it moves next based on it's own observations and on information received from its direct neighbors. Assume one individual finds a new global best. First its neighbors will notice and they start moving towards the optimum. Now also the neighbors of these neighbors notice, and so on. The information flows through the swarm and causes the individuals to converge towards an optimal point. During this process however also new areas are explored, potentially discovering even better points [Ken10].

In it's simplest form PSO computes the velocity updates such that - conceptionally - every particle is in the neighborhood of every other particle. This means that the update of the velocity is performed with respect to *pbest* as well as *gbest*. In the literature this PSO version is known as *global best* and was also the original form of PSO [EK95]. In detail, every dimension d of the velocity $\mathbf{v_i}$ of particle $\mathbf{x_i}$ is updated according to:

$$v_{i,d}^{(t+1)} \leftarrow \alpha v_{i,d}^{(t)} + r_1\beta\left(pbest_{i,d} - x_{i,d}^{(t)}\right) + r_2\gamma\left(gbest_{i,d} - x_{i,d}^{(t)}\right) \tag{3.19}$$

where α, β, γ are constants and r_1, r_2 are random numbers uniformly distributed on $[0, 1]$.

After that every dimension $x_{i,d}$ of particle $\mathbf{x_i}$ is updated according to:

$$x_{i,d}^{(t+1)} \leftarrow x_{i,d}^{(t)} + v_{i,d}^{(t+1)} \tag{3.20}$$

The previously mentioned version of PSO, where only the neighboring particles influence each other, is known as *local best*. The update rules are almost identical to global best, but instead of *gbest* another metric called *lbest* is used. *lbest* corresponds to the best particle within the neighborhood of the updated particle. The definition of the neighborhood is problem specific, for a euclidean search space it's common defining all particles within a certain radius as neighbors.

Extending PSO, originally designed for continuous search spaces, to mixed search spaces is non-trivial and remains an active field of research. Several approaches are described in the literature, among them *relaxation* and *hybridization*. Relaxation simply treats the discrete variables as continuous for the motion updates and then transfers them back into discrete space by rounding. While this approach works well for integer valued variables, it performs quite poorly on categorical variables. Wang et al. suggested hybridization, an approach where discrete and continuous variables are treated independently [WZZ21]. As an update rule for the continuous portion they suggest an adapted version of *local best* and for the discrete portion they suggest learning a distribution of well-performing

values and sampling from it. This approach works merely for discrete values with limited domain. For search spaces where the discrete variables only have little influence a random updating rule is reasonable.

Implementation Details Implementing Particle Swarm Optimization for a mixed search space was done via the hybridization approach, meaning that discrete and continuous dimension of the solution vector are independently updated. While the continuous part is updated according to the velocity of this dimension, the discrete part is treated differently. Here the *global best* strategy is applied, also described in the section Particle Swarm Optimization. Parameter values for the velocity update are $\alpha = 0.5, \beta = 1.5, \gamma = 1.5$. The discrete part represents categorical variables with small, bounded domains only, therefore velocity does not really make sense. Instead, a randomly drawn value is used for update. Parallelization for this algorithm is done by evaluating all individuals in parallel, thus the population size is set to a multiple of the number of available CPUs.

3.2.5 Parallelization of Optimization

Parallelizability is a highly desired property of any optimization strategy. The widespread availability of parallel computing infrastructure makes parallelization and scalability one of the most important aspects when choosing an optimization strategy. Optimization algorithms that perform suboptimal on a single CPU core can still achieve superior results when run on several hundreds of CPUs, provided they scale well enough. In this section we will go through several previously described algorithms and evaluate their potential in terms of parallelizability and scalability.

Bayesian Optimization

Parallelization of Bayesian optimization is an open and very active field of research. The basic concept is clear and identical for all approaches: instead of generating just a single suggestion $\mathbf{x}$ for evaluation in each iteration we want to generate an entire batch $B = \{\mathbf{x}_1, \ldots, \mathbf{x}_{n_b}\}$ of n_b suggestions and run all evaluations in parallel. That's the reason why parallel Bayesian optimization is also referred to as Batch Bayesian optimization (BBO). While generating batches from a Gaussian process is very cumbersome, there exist very simple approaches for TPE.

As described in section 3.2.1 Gaussian Process, generating suggestions using a Gaussian process model involves maximizing a dedicated acquisition function. However, in order to generate an entire batch of suggestions without duplicates, we cannot maximize it several times. A suitable method must consider the correlations between the samples within the batch. Several methods have been proposed. The first one introduced here is *multi-point expected improvement* (qEI) by Ginsbourger et al. [GLRC08]. Analog to expected improvement (EI) for the sequential version of Bayesian optimization they

suggest considering the EI for the entire batch. qEI is defined as

$$qEI(B) = \mathbb{E}\left[\max\left(0, f(\mathbf{x}^+) - \min_{i=1,\dots,n_b} f(\mathbf{x}_i)\right)\right] \tag{3.21}$$

where $\mathbf{x}^+ = \text{argmax}_{\mathbf{x}\in Obs}\, f(\mathbf{x})$ is the best known observation so far. qEI aims at finding a set B of size n_b that maximizes this expression. This maximization problem becomes extremely difficult to solve as the dimensionality of the problem grows, as Ginsbourger et al. point out [GLRC08]. Frazier et al. came up with an efficient algorithm based on a gradient estimation of the qEI function [WCLF19].

Another approach is *Local Penalization* (LP) which has been suggested by Gonzalez et al. in 2015 [GDHL15]. The intuition behind LP is to modify the acquisition function after one suggestion has been generated in order to discourage the selection of points that are close by (the acquisition function is "penalized locally"). This approach rests on the assumption that the objective function is Lipschitz continuous, i.e. it must hold that

$$|f(\mathbf{x}_1) - f(\mathbf{x}_2)| \leq L||\mathbf{x}_1 - \mathbf{x}_2|| \tag{3.22}$$

for a global positive constant L, which is a reasonable and common assumption in optimization according to the authors. This Lipschitzian assumption enables to establish a bound on how far the optimum of f is from a certain location. This bound is essential to LP as it allows to choose the penalization such that the true optimum of f is not excluded unintentionally. The bound depends on the constant L which can be estimated from the underlying Gaussian process. Figure 3.2 illustrates this process. After the first suggestion has been generated my maximizing the acquisition function $\alpha(x)$, a local penalization function $\varphi(x)$, $0 \leq \varphi(x) \leq 1$ is calculated and multiplied to $\alpha(x)$. It discourages the selection of suggestions that are close to the first suggestion. This process iterates until the last suggestion of the batch has been selected by maximizing $\alpha(x)\Pi_{i=1}^{n_b}\varphi_i$.

Contal et al. introduced another different version of batch selection known as Pure Exploration (PE) [CBRV13]. They suggest generating the first batch element by maximizing the acquisition function analogous to the sequential Bayesian optimization. The remaining $n_b - 1$ samples are generated in a purely random manner - hence the name. They show that PE is efficient in reducing overall uncertainty which in turn results in a more accurate acquisition function in each iteration.

The TPE approach offers a very natural way of generating a batch of suggestions. As described in the section Tree-structured Parzen Estimator a sample is generated by sampling from the distribution $l(\mathbf{x})$. Since $l(\mathbf{x})$ is a distribution rather than a deterministic function, it facilitates the generation of multiple suggestions by iterative sampling. After the scoring of the samples not just the single best sample but the top n_b samples are selected to form the suggestions of the batch. This advantage was also recognized by the TPE authors and introduced already in the original TPE paper [BBBK11].

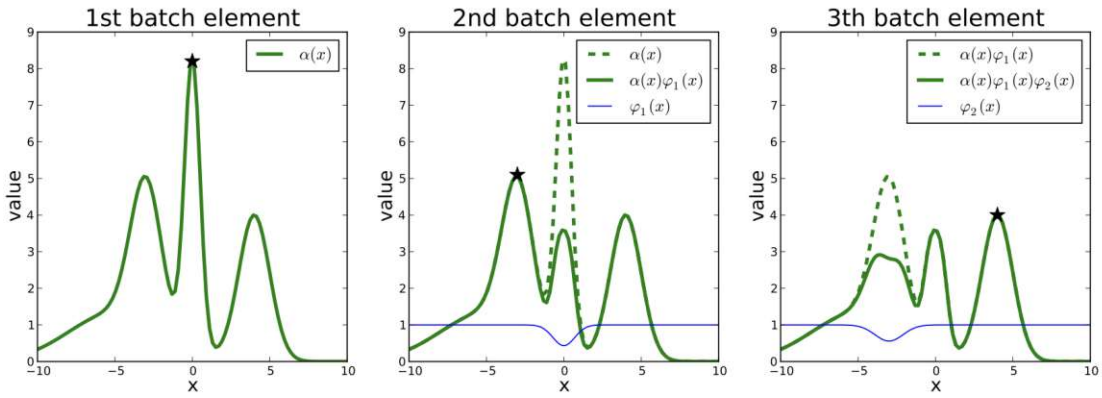


Figure 3.2: Local penalization applied to an acquisition function $\alpha(x)$. After the first suggestion is generated (left), $\alpha(x)$ is multiplied with the penalization function $\varphi_1(x)$ that discourages the selection of points near the first suggestion (middle). After the second suggestion is generated, $\varphi_2(x)$ is multiplied to $\alpha(x)\varphi_1(x)$ to prepare the acquisition function for the third iteration (right).

Simulated Annealing

The parallelization of Simulated Annealing (SA) is difficult due to its inherently sequential nature. SA is inherently sequential because each move, i.e. whether or not to accept a solution, depends on the temperature (which decreases after every move) and on the previously found best solution. First attempts to parallelize SA focused on running several instances of the algorithm at the same time. However, this approach is inefficient as in n parallel instances, each instance neglects the information that has already been collected by the other $n-1$ instances. Some authors developed ways to parallelize a single run of SA. They proposed adapting the cooling schedule so that a number τ of moves can be completed simultaneously. However, these methods were found to provide a significant speedup only for up to 16 or 32 processors. This is a fairly small number, since the average clusters used in the industry have hundreds or thousands of CPUs [LR16]. Due to these concerns and the insufficient options for parallelizing Simulated Annealing, it will not be further considered for implementation.

Evolutionary Algorithms

As a population-based algorithm Evolutionary Algorithms (EAs) rely on the evaluation of a set of individuals. These evaluations are completely independent of each other, making EAs a perfect candidate for parallelization. Other operations of EAs are mutation, crossover and selection. Mutation and crossover depend solely on one and two individuals, respectively, making them highly parallelizable as well. The parallelizablity of the selection mechanism varies from implementation to implementation. Most mechanisms are well parallelizable but depend on information about the entire population making communication the bottleneck [SO22]. However, for expensive black box optimization the

evaluation of the objective function (the fitness calculation in EAs) dominates all other operations by several orders of magnitude. Therefore all concerns about parallelizability of the genetic operations and selection mechanism can be neglected in this case, making EA a promising candidate for implementation.

Particle Swarm Optimization

Particle Swarm Optimization is also a population based algorithm and therefore provides promising possibilities for optimization. The evaluations of the individuals can be done in a perfectly parallel manner. Only the selection of the values of *gbest* or *lbest* needs to be done sequentially before each iteration. The selection of *gbest* can be done in time linear in the population size, the selection of *lbest* runs in quadratic time in the worst case. However, similar to the evolutionary algorithm, this overhead is negligible when applied for expensive black box optimization.

CHAPTER 4

Experiments and Results

This chapter presents the results of several experiments conducted to compare the performance of different optimization algorithms on a set of benchmark problems and a real-world simulation model. The objective was to evaluate the entire implementation based on various interesting aspects such as convergence, solution quality, efficiency, reliability and scalability. First, the algorithms are compared to each other on different use cases. Afterwards, the scalability of the parallelization framework is assessed.

4.1 Algorithm comparison

Fist, experiments were conducted to assess the performance of the evolutionary algorithm, Particle Swarm Optimization (PSO), Bayesian Optimization with a Gaussian Process model (BayesOpt-GP), and Bayesian Optimization using the Tree-structured Parzen Estimator (BayesOpt-TPE). Each algorithm was tested on both artificial benchmark functions and a real-world simulation model, with multiple runs conducted to ensure robustness. The algorithms are then compared with each other and among the different use cases.

4.1.1 Benchmarking on artificial functions

For the first part of benchmarking a set of mathematical functions has been chosen. These functions are specifically developed as benchmarking problems for optimization algorithms and used extensively throughout the literature [PS22]. This set comprises:

- **Sphere** function: The simplest and most commonly used function for continuous optimization problems. It's convex and unimodal.
- **Rastrigin** function: A highly multimodal function with many, regularly distributed local optima.

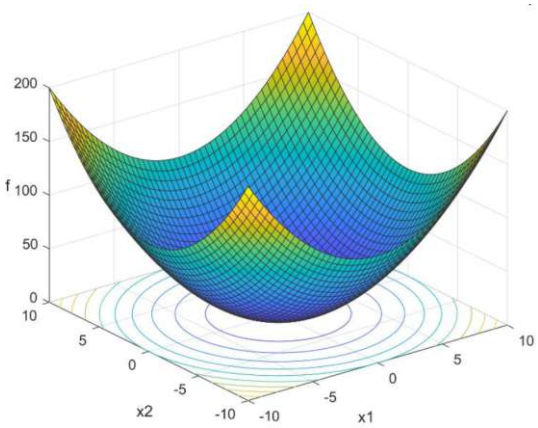
- **Rosenbrock** function: Function with a narrow, parabola-shaped valley that contains the global optimum. Finding the valley is easy, but converging to the optimum difficult.
- **Schwefel** function: Similar to Rastrigin, a highly multimodal function with many, regularly distributed local optima.
- **Branin** function: A relatively smooth function with an easy to find global optimum and some local optima.

Figure 4.1 depicts the 2-dimensional versions of these functions. For benchmarking these functions were extended to the 10-dimensional real space.

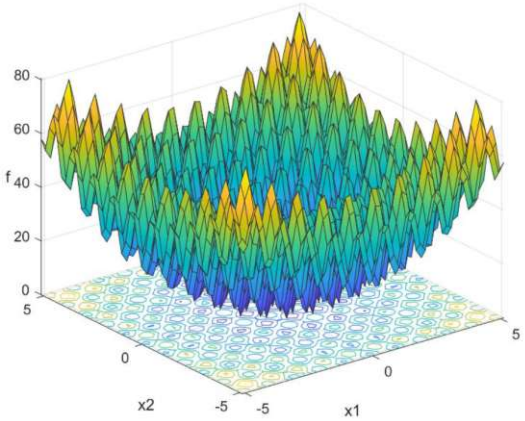
Benchmarking is conducted by running each optimization algorithm for each benchmark problem $n = 20$ times. During each run data is collected and afterwards plots of the interesting aspects were generated. Interesting aspects include convergence speed, quality of solution, reliability and efficiency. The parameters of the algorithms are chosen as described in the section 3.2 Implementation of Algorithms. For the benchmarking on the artificial functions, each evaluation is run locally on a single core. The reason is that in this section we only want evaluate how well the algorithms move towards the optimum of a function and how much overhead the algorithm introduces. The cost of evaluating these functions is negligible and no different results can be expected for running the evaluations on multiple cores or in the cloud. What does influence the performance however is how often the model is updated when using the model-based Bayesian Optimization. To simulate parallel execution (which is the main usecase of all algorithms), the model is updated in batches of 200 evaluations. This a reasonable batch size for actually parallel execution and therefore should accurately simulate the scenario. This approach allows us to measure the performance characteristics of the algorithms in a way that reflects their intended use in parallel environments. For the population based algorithms a population size of 200 was chosen as well, making it easy to compare the collected data also on batch level.

Convergence

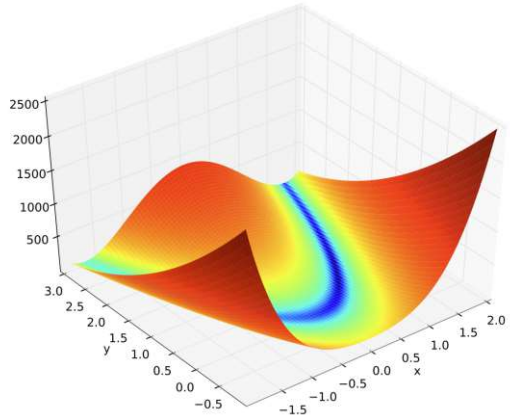
Convergence measures how fast and how well the optimization algorithm approaches the true optimum. To visualize the quality and speed of convergence, the best objective function value found so far at each iteration is tracked. To obtain the convergence graph, this value is plotted against the number of the according evaluation. The plots in figure 4.2 show the average across all runs of one algorithm on one problem together with the standard deviation. Note that the total number of function evaluations in each run was 4000, except for Bayesian Optimization using the Gaussian process model. This algorithm exceeded the time budget long before reaching the 4000 evaluations. This is due to the inefficient scaling of the fitting of the Gaussian process to observed data, which causes a significant overhead both when using a sparse Gaussian process and when using a Gaussian process. For the Sphere function and the Rosenbrock function only the



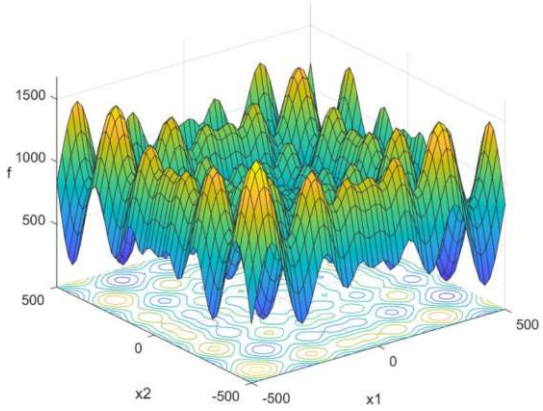
(a) Sphere function [PS22]



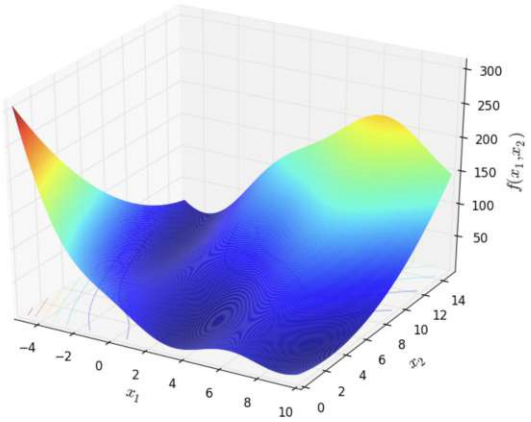
(b) Rastrigin function [PS22]



(c) Rosenbrock function [ros]



(d) Schwefel function [PS22]



(e) Branin function [bra]

Figure 4.1: The set of mathematically defined benchmark functions.

first 1600 evaluations are plotted because no significant improvement could be observed afterwards.

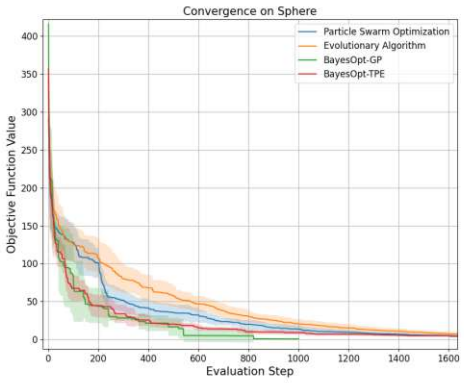
The evolutionary algorithm provides the most reliable convergence, finding optimal or near optimal solutions for all five problems. The variance vanishes when approaching the optimum indicating stable performance across all runs. For the Rastrigin and the Schwefel function it also outperformed all other algorithms significantly in terms of solution quality. It reliably converges towards the optimum on those functions while the other algorithms get stuck in suboptimal locations. However, in terms of convergence speed, the evolutionary algorithm consistently underperforms compared to the other algorithms, at least when ignoring the problems where the other algorithms struggled to find the optimum at all (Rastigin and Schwefel). This makes the evolutionary algorithm an interesting candidate for exact solutions provided enough budget is available.

Particle Swarm Optimization converges faster than the evolutionary algorithm on most problems, however it struggles to find the optimum on the Rastrigin and the Schwefel function. It does discover the second-best solution on Rastrigin, though this is still significantly inferior to the solution achieved by the evolutionary algorithm. However, it completely fails on the Schwefel function. Benchmarking on the real model will show if this lack in realiability disqualifies Particle Swarm Optimization entirely or if the Schwefel functions is simply a corner case, that Particle Swarm Optimization is not made for.

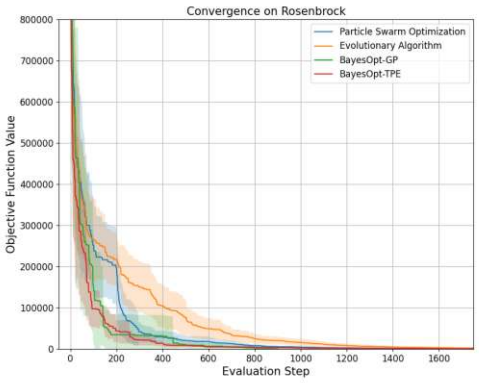
Bayesian Optimization with the Gaussian Process (BayesOpt-GP) model shows interesting results. It converges very fast on the Sphere, Rosenbrock, Rastrigin and average or worse than average on Schwefel and Branin. Until termination, it delivers the best results on the Sphere, Rosenbrock and Rastrigin function across all algorithms. However, as already mentioned, this algorithm scales so bad that it had to be terminated earlier, because it consumed magnitudes more time per iteration (see section on efficiency for details). It must be examined whether this overhead becomes negligible due to the simulation runtime on the actual model or if it remains significant even there. The lack in realiability however is a disadvantage to keep in mind.

Bayesian Optimization using the TPE model (BayesOpt-TPE) delivers consistent and fast performance across all problems. Although the quality of the solution is clearly suboptimal on Rastrigin and Schwefel, BayesOpt-TPE provides a very fair trade-off between convergence speed an solution quality. For Sphere, Rosenbrock and Branin, BayesOpt-TPE delivers the same solution quality as the evolutionary algorithm with significant faster convergence. All in all BayesOpt-TPE is a great candidate if fast convergence has priority while maintaining high reliability.

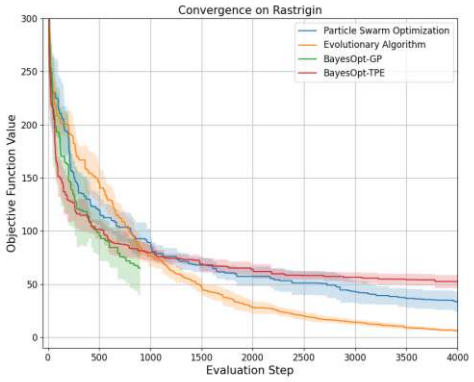
Summarizing, both the evolutionary algorithm as well as BayesOpt-TPE show very promising results on the artifical benchmarking functions. While the evolutionary algorithm delivers high quality and reliable solutions, BayesOpt-TPE trades off solution quality for convergence speed. Both algorithms are superior to the other two in most aspects.



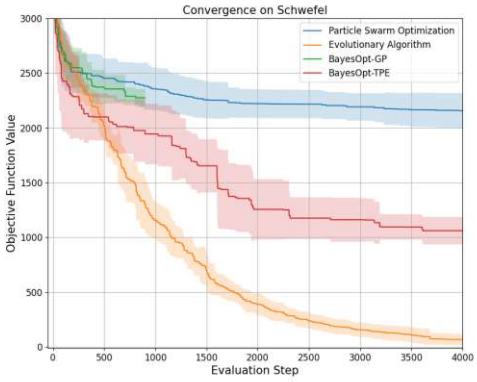
(a) Convergence on Sphere function



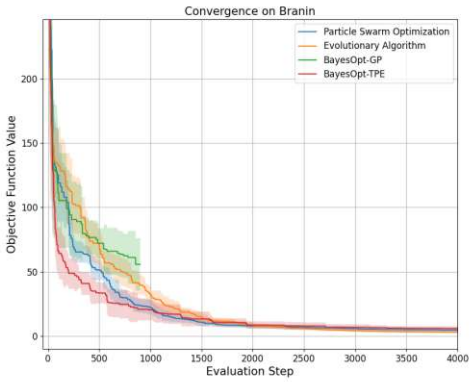
(b) Convergence on Rosenbrock function



(c) Convergence on Rastrigin function



(d) Convergence on Schwefel function



(e) Convergence on Branin function

Figure 4.2: Convergence plots of all four algorithms on different benchmark functions.

Batch diversity

Evaluating the diversity of each batch also provides valuable insights. Each batch consists of 200 evaluations as mentioned at the beginning of this section. Diversity refers to the difference in objective function values for each evaluation in a batch. From the development of this diversity during the run, for example, information can be derived about the trade-off between exploration and exploitation. In order to visualize this behavior for each batch a boxplot is plotted against the batch number. The boxplot indicates the distribution of the objective function values obtained in this batch. Note that for readability purposes only 20 batches out of the 200 of each run are plotted. For BayesOpt-GP, which has been terminated earlier due to excessive time consumption, only four out of 50 batches are plotted for comparability. The plots are depicted in figures **??**, 4.4 and 4.5.

The evolutionary algorithm shows significant decrease in solution diversity already after 10 batches across all problems. However the diversity does never completely vanish even until the last batch. This is exactly the idea of a reasonable exploration-exploitation trade-off. Even if most evaluations take place close to the assumed optimum, there are still some evaluations in unexplored regions that might contain even better solutions.

Particle Swarm Optimization shows very similar results, although the diversity almost completely disappears over time, indicating that there is a bit too much exploitation. However this does not seem to affect the solution quality, as can be derived from the performance of Particle Swarm Optimization on the Rastrigin function and the Schwefel function. On both functions, the algorithm struggles to find a good solution although maintaining a good diversity in each batch.

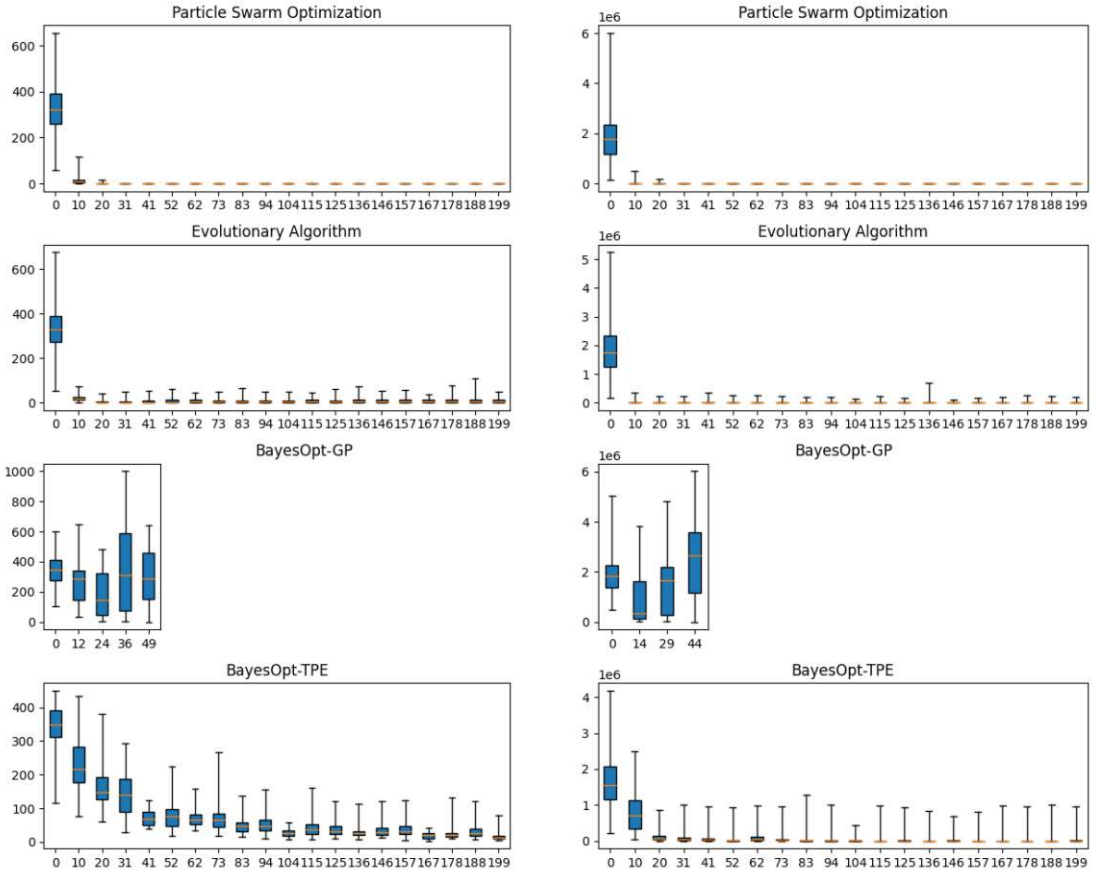
BayesOpt-GP maintains great diversity throughout every batch and there does not seem to be a correlation between solution quality and batch diversity.

BayesOpt-TPE reduces batch diversity slower than Particle Swarm Optimization or the evolutionary algorithm. This could explain why BayesOpt-TPE shows superior convergence speed across all problems, especially very early in the run. This algorithm keeps at least some diversity throughout the entire run, similar to the evolutionary algorithm.

Summarizing, reducing diversity too early could be a reason for slower convergence in the early phase of the run. Afterwards no clear correlation between batch diversity and solution quality can be observed.

Cost per major iteration

Another interesting aspect is the cost (time) per iteration of the optimization algorithm. For optimization problems on easy-to-evaluate functions (such as the benchmark functions in this chapter) this cost plays an essential role for the time it takes to find a good solution, because it will dominate the cost of evaluating the function itself. For expensive black-box optimization on the other hand, the expectation is, that the overhead introduced by the



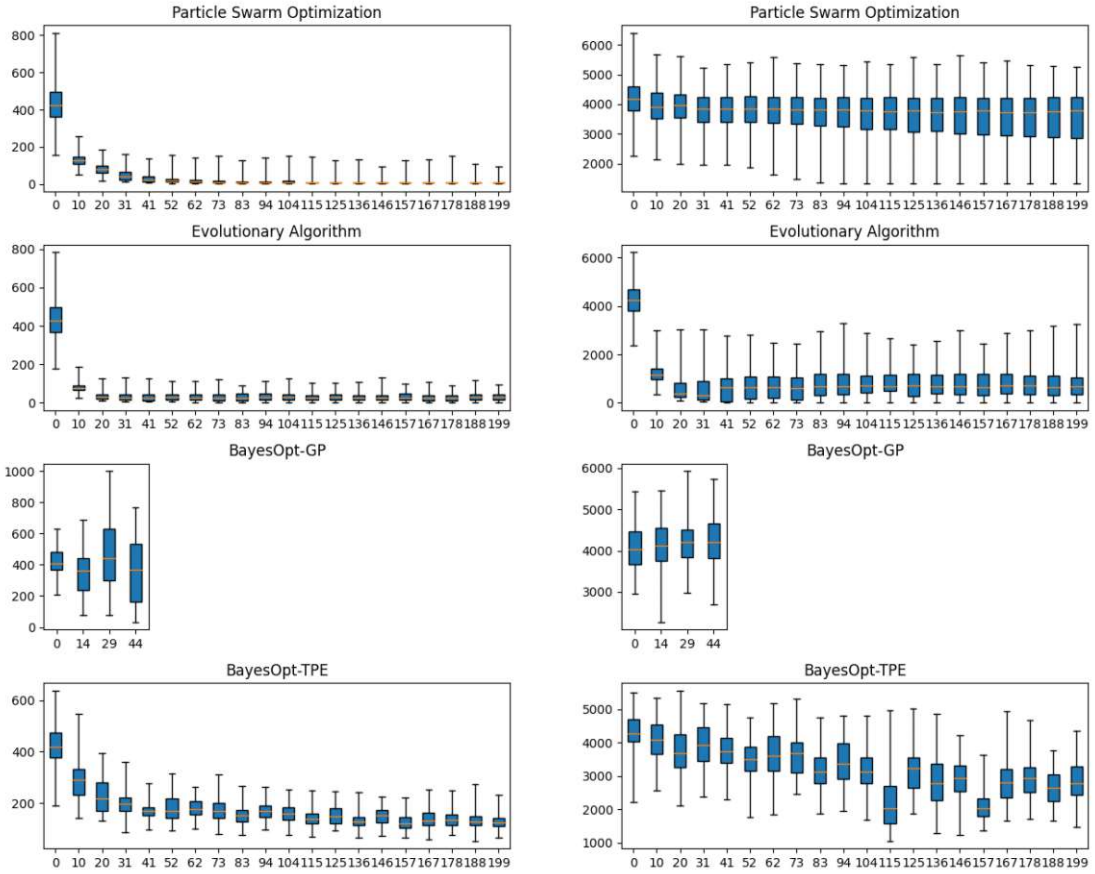
(a) Batch diversity on Sphere function

(b) Batch diversity on Rosenbrock function

Figure 4.3: Diversity of each evaluated batch. The x-axis shows the batch number, the y-axis shows the objective function value.

algorithm is negligible and dominated by the cost of evaluating the objective function. To verify this expectation, we determine the runtime of each major iteration and plot it against the number of the according iteration. A *major iteration* refers to an iteration where a new generation is introduced in population based algorithms or a new model is fitted in model-based algorithms. The plots are depicted in figure 4.6. This cost now also contains the time that it took to evaluate the objective function. This evaluation cost however is very small allowing a good estimation of the algorithmic overhead. The experiments have been conducted for two functions (Branin, Rastrigin) only, as the results should not significantly depend on the actual function.

The plots clearly shows two trends. First, the iteration cost for the population based algorithms stays constant. This is expected as each iteration processes a constant number of individuals. There is even a slight decrease in iteration cost for the evolutionary algorithm which is caused by a rising number of duplicate individuals throughout the run.



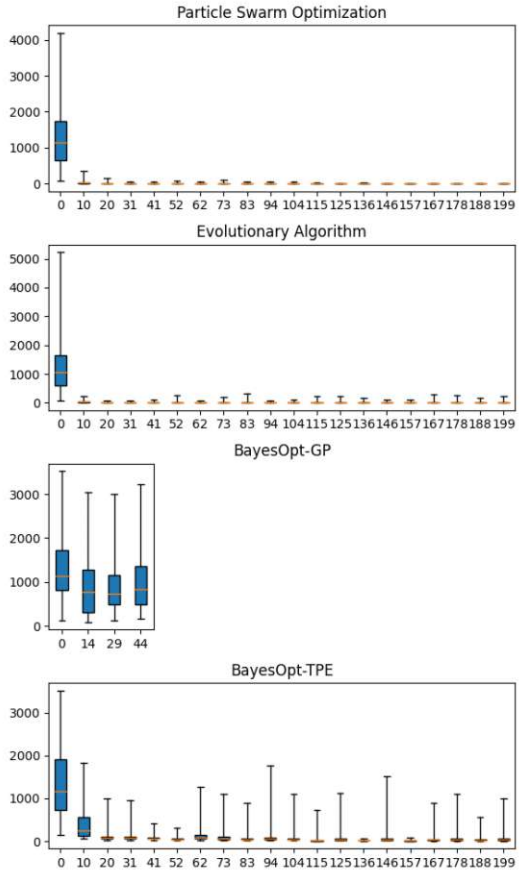
(a) Batch diversity on Rastrigin function

(b) Batch diversity on Schwefel function

Figure 4.4: Diversity of each evaluated batch. The x-axis shows the batch number, the y-axis shows the objective function value.

Duplicates evaluated only once. Second, the iteration cost for the model-based algorithms rises constantly. This is also expected due to the fact that in each major iteration a new model is fitted, taking into account a constantly rising number of observations.

The assumption of negligible algorithmic overhead is definitely true for the population based algorithms, where the time per iteration stays in the region of 0.1 seconds. For BayesOpt-TPE the cost stays below 1 second for at least 200 iterations. This number of iterations is already close to the expected limit for expensive black-box optimization, confirming the assumption also in this case. BayesOpt-GP in contrast introduces significant overhead from the first iteration on. After 40 iterations the runtime per iteration is already in the region of 1.5 to 3 seconds, which is quite significant. Although the plot suggests linear growth in iteration cost, it was observed during experimenting, that soon after the 40 iterations the cost jumped significantly to up to several minutes. This is also the reason why it was not possible to conduct experiments with more iterations



(a) Batch diversity on Branin function

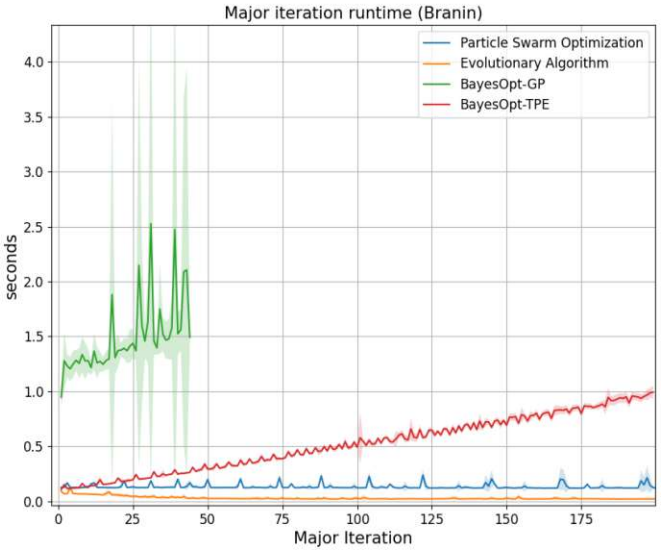
Figure 4.5: Diversity of each evaluated batch. The x-axis shows the batch number, the y-axis shows the objective function value.

for BayesOpt-GP. The assumption does clearly not hold in this case, indicating the BayesOpt-GP is a bad candidate.

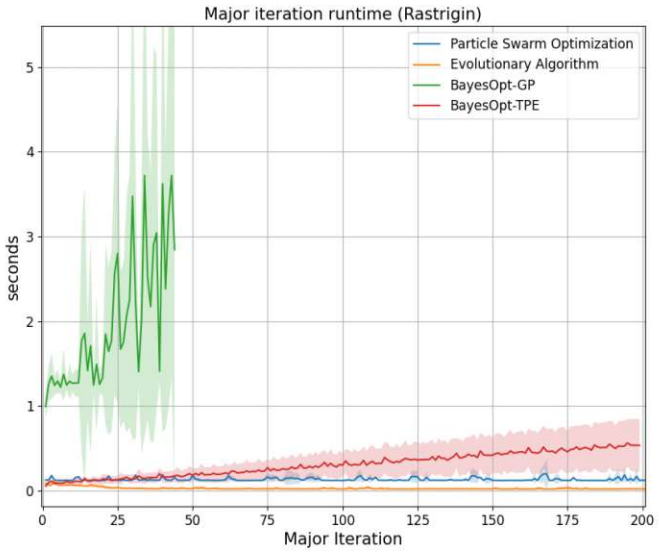
Summarizing, the data shows that population based algorithms introduce only insignificant overhead on top of the function evaluations. This also applies for the model-based algorithm BayesOpt-TPE (at least for up to 200 iterations) but does not hold for BayesOpt-GP.

4.1.2 Benchmarking on the real model

The second part of benchmarking took place on the actual simulation model. As a test case a dual connectivity scenario was chosen. Dual connectivity is a usecase where the user equipment (UE) is sending and receiving via two base stations at the same time, improving speed and reliability. The base station requires the signals of it's connected



(a) Major iteration cost for Branin



(b) Major iteration cost for Rastrigin

Figure 4.6: The cost for a major iteration of each algorithm throughout the run on two benchmark functions.

UEs to arrive precisely at certain points in time. Therefore, the UE must compensate for the signal run time by sending signals earlier according to the distance to the base station. This time delta is called the *timing advance* in 3GPP terms. In dual connectivity, each base station requires a distinct timing advance value, posing a challenge for the transceiver. Different timing advance combinations can lead to overlapping high load scenarios, potentially causing even higher load or overload.

For benchmarking, two timing advance values and 8 categorical design parameters were chosen as variables. The goal was to find a scenario causing maximum load for the transceiver. The load was measured by observing the delay of internal, time-critical messages, which are closely related.

With each algorithm again $n = 20$ runs have been performed. Each algorithm was configured to deliver batches of 200 evaluation suggestions in each iteration to match the settings of artifical function benchmarking. Those suggestions were then simulated on parallel computing infrastructure and the results were fed back to the algorithm. In total, one run of an algorithm consisted of 50 such iterations, except for Bayesian Optimization using the Gaussian process (BayesOpt-GP), which again consumed an excessive amount of time and had to be quit early (more on that below).

Again, the first aspect to be discussed is convergence. The convergence of all four algorithms on the experiment described above is depicted in figure **??**, the plots were generated as described in 4.1.1. All algorithms converge fast initially. All algorithms except for BayesOpt-GP find the same maximum. BayesOpt-GP shows high standard deviation, indicating that some runs found the maximum, but some did not. For all other algorithms, the variance among the experiments vanishes and at least after 2500 iterations they all found the same maximum. In terms of convergence speed only small differences are visible. Bayesian Optimization using TPE converges fastest on average, confirming the good results of the artificial function benchmarking. Among the population based algorithms Particle Swarm Optimization performs better then the Evolutionary algorithm on average which is also expected and in line with the artificial function benchmarks.

Next, batch diversity was evaluated, which gives an idea of how exploration is traded off against exploitation. The average plot diversity for each algorithm is depicted in figure 4.8, the plots were generated as described in 4.1.1. The Evolutionary algorithm reduces batch diversity fast, with a significant jump after about 20 iterations. From this point on the algorithm focuses heavily on exploiting and less on exploration. Particle Swarm Optimization reduces the batch diversity slower, which could be the reason for its faster convergence, similar as observed for the artificial function benchmarks. ByaesOpt-GP and BayesOpt-TPE both start with high batch diversity, and BayesOpt-TPE keeps almost the same high diversity throughout the entire run. BayesOpt-TPE also has the highest batch diversity of all algorithms. As it is also the algorithm converging fastest, this seems to confirm the thesis that high batch diversity results in fast convergence.

Finally, major iteration cost was evaluated in an equal manner as in 4.1.1. The goal is to confirm that the choice of the algorithm does not really impact the major iteration

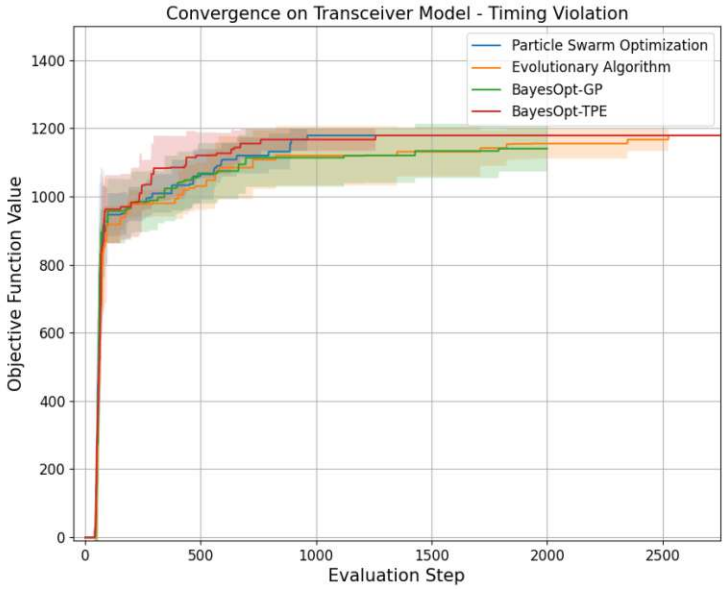


Figure 4.7: Convergence of all four algorithms on the simulation model optimization task.

time. Figure 4.9 shows the major iteration time for all four algorithms. It's immediately visible that the iteration time of BayesOpt-GP rises very fast with each iteration, as already observed with the artifical function benchmarking. After not even ten iterations the overhead of fitting the model has grown so much that an iteration takes around 10 minutes. The other algorithms maintain a major iteration time of 50 to 80 seconds throughout the entire run. Their plots are depicted separately in figure 4.10. No algorithm is significantly faster than the others, confirming that iteration time is majorly caused by the expensive model evaluation and not the algorithmic overhead. The bad results of BayesOpt-GP confirm, that this algorithm is not practically usable for this problem.

4.2 Scaling of parallelization framework

Finally, the scalability of the parallelization framework was investigated. Therefore $n = 20$ runs of the Particle Swarm Optimization algorithm were conducted on 5, 10, 20, 40, 60, 80 and 100 CPUs. Again, batches of 200 evaluations were generated and run on the given number of CPUs. The total time it took to complete all evaluations was measured. This time is independent of the algorithm and therefore the experiment was conducted with a single algorithm only. Finally, the time to complete all 200 evaluations was divided by 200 to obtain the average time per single simulation. These numbers are plotted in figure 4.11. The time that a single simulation takes on a single CPU is

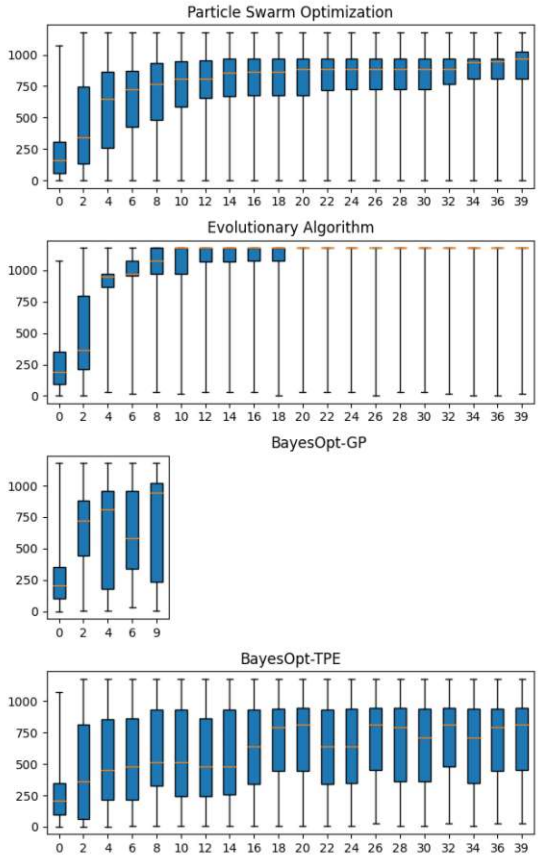


Figure 4.8: Diversity of each evaluated batch the simulation model optimization task. The x-axis shows the batch number, the y-axis shows the objective function value.

not plotted, but it is around 10 seconds. Up to 20 CPUs a perfect speed up can be observed as the average simulation time decreases to $\frac{10s}{20} = 0.5s$. To maintain perfect speed-up for up to 100 CPUs the average simulation time would need to decrease to $\frac{10s}{100} = 0.1s$. However, the actual average simulation time with 100 CPUs is approximately 0.15 seconds, slightly deviating from the ideal.

In summary, the parallelization framework introduces minimal overhead and achieves near-perfect speedup with up to 100 CPUs. Although the experiment was limited to 100 CPUS due to availability of computing resources, the results suggest significant potential for further scalability. Moreover, these results are crucial for the performance and rapid convergence of the optimization algorithms. Without parallelization, the algorithms would be impractical, as a 10-second evaluation time severely restricts the number of possible evaluations considering a reasonable time budget.

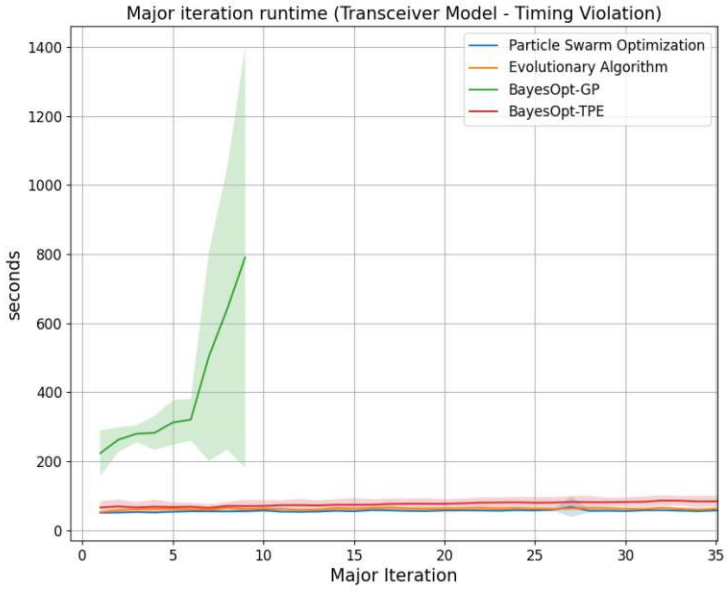


Figure 4.9: Major iteration run-time including BayesOpt-GP

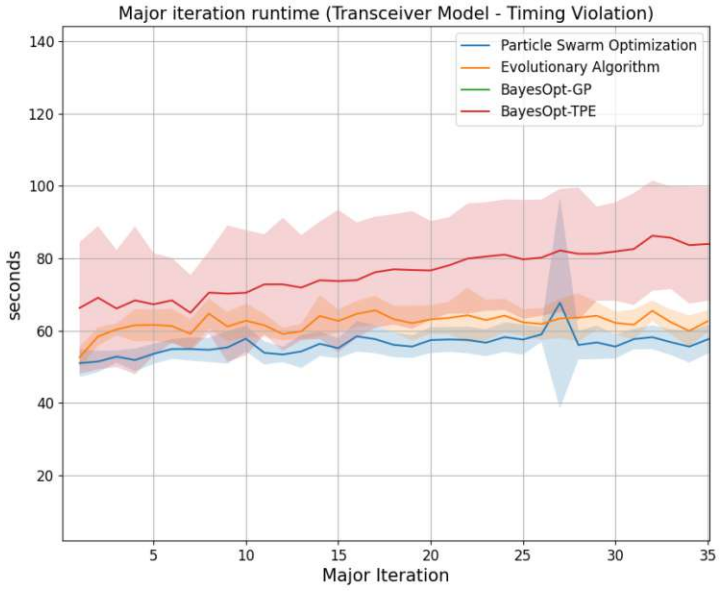


Figure 4.10: Major iteration run-time excluding BayesOpt-GP

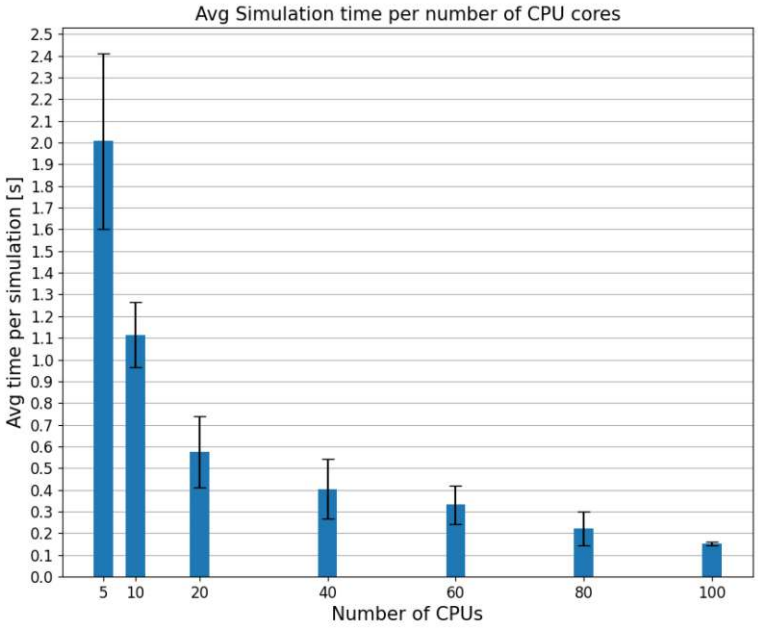


Figure 4.11: The average time per simulation depending on the number of CPU cores. Experiments conducted with batches of 200 simulations on the given number of CPUs. A single run on a single core takes approximately 10 seconds.

CHAPTER 5

Discussion and Conclusion

The goal of this thesis was to address the expensive black-box optimization problem in the context of a 5G cellular transceiver simulation model. Several algorithms introduced in the literature were investigated and the most promising candidates were implemented. The implementation furthermore required a framework for parallelization to interface with cloud computing infrastructure and to conduct experiments under scientific conditions. The set of algorithms implemented comprises Particle Swarm Optimization (PSO), an evolutionary algorithm (EA), as well as Bayesian Optimization using both Gaussian Process (BayesOpt-GP) and tree-structured Parzen estimator (BayesOpt-TPE) models.

It is shown with numerous experiments that three out of four implemented algorithms deliver very good results for a real-world usecase involving the transceiver model. Those algorithms are PSO, EA and BayesOpt-GP. Furthermore, several benchmarks on artificial benchmarking functions were conducted, yielding good results for all three mentioned algorithms. EA and PSO showed very reliable convergence, with EA showing slightly superior reliability. BayesOpt-TPE showed the fastest convergence on most tasks while still maintaining high reliability. BayesOpt-GP showed good convergence as well but the expensive fitting of the Gaussian process model makes this algorithm impractical. BayesOpt-GP would become interesting again if the simulation time increases by a factor of ten, making the algorithmic overhead relatively less significant.

The parallelization framework demonstrated good scalability properties and has proven to be the foundation of the algorithm implementations in terms of performance. It showed near-linear speed-up, meaning that evaluation times are up to 100 times faster on 100 CPUs compared to a single CPU, with potential for further scalability.

In conclusion the benchmarking results together with the results on the real simulation model provide confidence that the given implementation is well suited to solve a wide range of further real-world usecases related to the transceiver model in the future.

Several topics that are out of scope of this thesis are left to be addressed by future research. One thing that could further improve performance are optimizaiton algorithms that admit parallelizability at even larger scales. Although PSO, EA, and BayesOpt-TPE inherently support parallelization, it remains a limiting factor. During each major iteration, the information collected by one evaluation does not contribute to the all remaining suggestions in this batch. The larger the batches grow, the more random the search becomes therefore, regardless of the algorithm. Once this problem has been successfully addressed, a lot more samples are to be evaluated at once. Replacing the event-based approach of the current simulation model by a vectorized approach would enable the usage of GPUs. This would allow scaling parallel simulations from hundreds to hundreds of thousands, potentially providing speed ups of several orders of magnitudes. Both factors would guarantee an increase in convergence speed and capability of searching even more complex and high-dimensional search spaces.

List of Figures

List of Tables

List of Algorithms

Bibliography

[3GP24] 3GPP. 3rd generation partnership project; technical specification group radio access network; nr; physical channels and modulation (release 18). Technical Report TS 38.211 V18.2.0 (2024-03), 3GPP, 2024.

[AH17] Charles Audet and Warren Hare. *Derivative-Free and Blackbox Optimization*. Springer Berlin Heidelberg, 2017.

[aut16] The GPyOpt authors. Gpyopt: A bayesian optimization framework in python. `http://github.com/SheffieldML/GPyOpt`, 2016.

[AXCS12] Elisa Amorim, Carolina Xavier, Ricardo Campos, and Rodrigo Santos. *Comparison between Genetic Algorithms and Differential Evolution for Solving the History Matching Problem*, volume 7333 of *Lecture Notes in Computer Science*. Jun 2012.

[BBBK11] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[BKE+15] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, jul 2015.

[BLP20] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon, 2020.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[bra] N-D Test Functions B 0.1.0 documentation. http://infinity77.net/global_optimization/test_functions_nd_B.html. [Accessed 25-07-2024].

[CBRV13] Emile Contal, David Buffoni, Alexandre Robicquet, and Nicolas Vayatis. *Parallel Gaussian Process Optimization with Upper Confidence Bound and Pure Exploration*, page 225–240. Springer Berlin Heidelberg, 2013.

[CDH+24] Shu-Chuan Chu, Thi-Kien Dao, Thi-Minh-Phuong Ha, Truong-Giang Ngo, and Trong-The Nguyen. Recent evolutionary computing algorithms and industrial applications: A review. In Jerry Chun-Wei Lin, Chin-Shiuh Shieh, Mong-Fong Horng, and Shu-Chuan Chu, editors, *Genetic and Evolutionary Computing*, pages 489–499, Singapore, 2024. Springer Nature Singapore.

[Čer85] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, Jan 1985.

[Chr76] Nicos Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Carnegie Mellon University*, 3:10, Mar 1976.

[Dar59] 1809-1882 Darwin, Charles. *On the origin of species by means of natural selection, or preservation of favoured races in the struggle for life.* London : John Murray, 1859, 1859. With a half-title, binder's directions, and a 32-page list of works published by John Murray. - Colophon varies for the list. - With MS. inscription: H.M.Barton;Reproduction of original in British Library, London.;NSTC 2D3209.;Microfiche. Cambridge : Chadwyck-Healey Ltd., 1990. 6 fiches; 11x15 cm. The Nineteenth Century: General Collection: Science; Pos: Fiche N.1.1.4328.

[DER97] Alfred E. Dunlop, William J. Evans, and Lawrence A. Rigge. Managing complexity in ic design — past, present, and future. *Bell Labs Technical Journal*, 2(4):103–125, 1997.

[DKZ+18] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs, 2018.

[EK95] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[FvHM18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[Gad22] Ahmed G. Gad. Particle swarm optimization algorithm and its applications: A systematic review. *Archives of Computational Methods in Engineering*, 29(5):2531–2561, Aug 2022.

[Gar23] Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

[GDHL15] Javier González, Zhenwen Dai, Philipp Hennig, and Neil D. Lawrence. Batch bayesian optimization via local penalization, 2015.

[GLRC08] David Ginsbourger, Rodolphe Le Riche, and Laurent Carraro. A Multi-points Criterion for Deterministic Parallel Global Optimization based on Gaussian Processes. Technical report, March 2008.

[GOR10] R. Garnett, Michael A. Osborne, and Stephen J. Roberts. Bayesian optimization for sensor set selection. In *International Symposium on Information Processing in Sensor Networks*, 2010.

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

[HLBN20] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination, 2020.

[INC23] INCOSE, editor. *INCOSE Systems Engineering Handbook*. John Wiley & Sons, Nashville, TN, 5 edition, June 2023.

[JHTI20] Kai Junge, Josie Hughes, Thomas George Thuruthel, and Fumiya Iida. Improving robotic cooking using batch bayesian optimization. *IEEE Robotics and Automation Letters*, 5(2):760–765, 2020.

[JSW98] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[Ken10] James Kennedy. *Particle Swarm Optimization*, pages 760–766. Springer US, Boston, MA, 2010.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KMB+22] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Introduction to Evolutionary Algorithms*, pages 225–254. Springer International Publishing, Cham, 2022.

[LDJD22] Felix Leibfried, Vincent Dutordoir, ST John, and Nicolas Durrande. A tutorial on sparse gaussian processes and variational inference, 2022.

[LL19] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019.

[LR16] Zhihao Lou and John Reinitz. Parallel simulated annealing using an adaptive resampling interval. *Parallel Computing*, 53:23–31, 2016.

[LSM+19] Romy Lorenz, Laura E Simmons, Ricardo P Monti, Joy L Arthur, Severin Limal, Ilkka Laakso, Robert Leech, and Ines R Violante. Efficiently searching through large tACS parameter spaces using closed-loop bayesian optimization. *Brain Stimul*, 12(6):1484–1489, July 2019.

[Mat08] Norm Matloff, Feb 2008.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

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

[MS13] Benjamin Menhorn and Frank Slomka. Confirming the design gap. *Advances in Intelligent Systems and Computing*, 225:281–292, 01 2013.

[MS18] Azad M. Madni and Michael Sievers. Model-based systems engineering: Motivation, current status, and research opportunities. *Systems Engineering*, 21(3):172–190, 2018.

[Plo24] Oliver Ploder. *Machine Learning Aided Self-Interference Cancellation in 4G/5G Mobile Transceivers*. Phd thesis, Johannes Kepler University Linz, Linz, Austria, May 2024. Available at `https://epub.jku.at/obvulihs/content/titleinfo/9844155`.

[Pol20] Boris Polyak. *Introduction to Optimization*. 07 2020.

[PS22] Vagelis Plevris and German Solorzano. A collection of 30 multidimensional functions for global optimization benchmarking. *Data*, 7(4), 2022.

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[ros] File:Rosenbrock function.svg - Wikipedia — en.wikipedia.org. `https://en.wikipedia.org/wiki/File:Rosenbrock_function.svg`. [Accessed 25-07-2024].

[Ryu20] Jaeku Ryu. Reference, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[Sim23] Team SimPy. Simpy. `simpy.readthedocs.io`, Jun 2023. v4.1.1.

[SK06] B Suman and P Kumar. A survey of simulated annealing as a tool for single and multiobjective optimization. *Journal of the Operational Research Society*, 57(10):1143–1160, 2006.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. In *ICML*, Beijing, China, June 2014.

[SO22] Vladislav Skorpil and Vaclav Oujezsky. Parallel genetic algorithms' implementation using a scalable concurrent operation in python. *Sensors (Basel)*, 22(6):2389, March 2022.

[Sut88] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, Aug 1988.

[Tal09] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Series on Parallel and Distributed Computing. Wiley-Blackwell, Hoboken, NJ, June 2009.

[TSTT21] Kei Terayama, Masato Sumita, Ryo Tamura, and Koji Tsuda. Black-box optimization for automated discovery. *Accounts of Chemical Research*, 54(6):1334–1346, Mar 2021.

[van88] P.J.M. van Laarhoven. *Theoretical and computational aspects of simulated annealing*. Phd thesis 4 research not tu/e / graduation not tu/e), Erasmus University Rotterdam, Netherlands, 1988. Proefschrift.

[VSS06] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *Journal of Aerospace Computing Information and Communication - J AEROSP COMPUT INF COMMUN*, 3:123–137, 03 2006.

[Wat23] Shuhei Watanabe. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance, 2023.

[WCLF19] Jialei Wang, Scott C. Clark, Eric Liu, and Peter I. Frazier. Parallel bayesian global optimization of expensive functions, 2019.

[Wei07] Tim Weilkiens, editor. *Morgan Kaufmann OMG Press*. The MK/OMG Press. Morgan Kaufmann, Burlington, 2007.

[WJSO22] Xilu Wang, Yaochu Jin, Sebastian Schmitt, and Markus Olhofer. Recent advances in bayesian optimization, 2022.

[WM97] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[WZZ21] Feng Wang, Heng Zhang, and Aimin Zhou. A particle swarm optimization algorithm for mixed-variable optimization problems. *Swarm and Evolutionary Computation*, 60:100808, 2021.