Universiteit Leiden

Master Computer Science

LLaMEA-BO: A Large Language Model Evolutionary Algorithm for Automatically Generating Bayesian Optimization Algorithms

Name:	Wenhu Li
Student ID:	s3978087
Date:	[05/08/2025]
Specialisation:	Artificial Intelligence
1st supervisor:	Elena Raponi
2nd supervisor:	Niki van Stein

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The automated design of complex algorithms is a grand challenge in computer science. Bayesian Optimization (BO), a critical framework for optimizing expensive black-box functions, often requires hand-crafting algorithms that may not generalize well. This work explores a new paradigm using Large Language Models (LLMs) for automated algorithm discovery. We introduce LLaMEA-BO, which embeds an LLM within an evolutionary loop to generate novel, end-to-end BO algorithms. Through carefully engineered prompts for mutation, crossover, and initialization, the LLM proposes new Python implementations, which are then rigorously evaluated on the BBOB benchmark suite. The performance feedback drives the evolutionary search toward more effective solutions. Our results show that LLaMEA-BO can discover a portfolio of high-performing algorithms that are competitive with, and sometimes superior to, human-designed baselines. This research demonstrates a significant step forward, showing that LLMs can serve not just as code synthesizers, but as creative partners in solving complex algorithm design problems.

Contents

Chapter 1

Introduction

Optimizing expensive black-box functions—problems where the objective's analytical form is unknown and each evaluation is costly—represents a fundamental challenge across scientific and engineering disciplines. This class of problems, known as black-box optimization (BBO), is ubiquitous in critical applications, including hyperparameter tuning for machine learning models [48] and automated experimental design [46]. Bayesian Optimization (BO) has emerged as a dominant paradigm for such problems, offering a sample-efficient, model-based approach to navigate vast search spaces by intelligently balancing exploration and exploitation [13, 12]. While powerful, established Bayesian Optimization algorithms are built from a fixed set of components (e.g., a specific surrogate model and acquisition function). The performance of these algorithms is highly dependent on the choice of these components, a selection process that often demands significant expertise and may not yield a universally optimal solution across different problem landscapes [46].

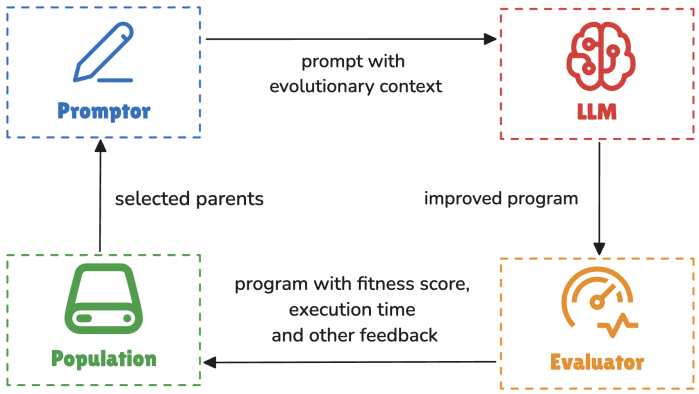


Figure 1.1: LLaMEA-BO Overview

Concurrently, the rapid advancement of Large Language Models (LLMs) has revolutionized numerous domains, showcasing remarkable capabilities in natural language understanding, reasoning, and code generation [29]. Their potential to assist in, or even automate, complex algorithm design tasks is increasingly being recognized. Initial efforts have seen LLMs integrated into evolutionary algorithms [30, 44], and within Bayesian Optimization itself to act as surrogate models [42], aid in acquisition function design [2], or orchestrate high-level BO phases [35]. These pioneering works highlight the promise of LLMs in augmenting specific components or workflows within existing optimization paradigms.

However, the prospect of leveraging LLMs to generate **entire, executable optimization algorithms** from high-level specifications, particularly for sophisticated frameworks like Bayesian Optimization, remains a largely nascent area of research. While frameworks like LLaMEA [51] have demonstrated success in evolving complete evolutionary algorithm

implementations using LLMs, a comparable system for the automated discovery of Bayesian Optimization algorithms has yet to be fully realized. This gap represents a significant opportunity: to harness the generative power of LLMs within an evolutionary search paradigm to not just assist, but to invent novel and effective BO algorithms.

This project addresses this gap by introducing **LLaMEA-BO**, an extension of the LLaMEA specifically tailored for the automated design and generation of complete, end-to-end Python implementations of Bayesian Optimization algorithms (a high-level overview is provided in Figure 1.1).

LLaMEA-BO is driven by an evolutionary strategy where an LLM acts as the core generative engine. The process begins with the LLM creating an initial population of diverse BO algorithms from scratch, guided by carefully engineered prompts that leverage its internal heuristics and understanding of optimization principles. Subsequently, the LLM functions as the primary variation operator, iteratively improving the population by performing sophisticated mutations on existing solutions and executing crossover to recombine the features of promising candidates. These generated algorithms are then rigorously evaluated on standard black-box optimization benchmarks, with performance feedback guiding the evolutionary search towards increasingly effective solutions. By iteratively refining candidate algorithms based on empirical performance, LLaMEA-BO explores the vast design space of BO algorithms, aiming to discover novel configurations that can rival or even surpass established, human-designed methods.

The contributions of this work:

- We present LLaMEA-BO, a novel framework that integrates LLMs with evolutionary search to automatically generate complete Bayesian Optimization algorithms.
- We design and implement specific prompt engineering strategies, including diversity-focused initialization and structured feedback mechanisms, to effectively guide the LLM in the complex task of BO algorithm synthesis.
- We conduct extensive ablation studies to identify best practices for configuring the evolutionary strategy and LLM hyperparameters, providing practical guidelines for effective LLM-driven algorithm design.
- We demonstrate through comprehensive experiments on the BBOB benchmark and Bayesmark benchmark that LLaMEA-BO can discover BO algorithms that achieve competitive, and in some cases superior, performance compared to state-of-the-art baselines, showcasing the potential of LLM-driven automated algorithm discovery in this domain.

Thesis Overview

- **Chapter 2 (Background)** provides a comprehensive review of the foundational concepts underpinning this research. It delves into Bayesian Optimization, detailing its core components, and discusses Evolution Strategies. Furthermore, it surveys the landscape of LLMs in code generation, and their emerging roles within Bayesian Optimization.
- **Chapter 3 (Methodology)** details the LLaMEA-BO framework. It elaborates on the evolutionary algorithm driving the search, the specific prompt engineering techniques employed to interface with the LLM, and the evaluation protocols used to assess generated BO algorithms during and after the evolutionary search.

- **Chapter 4 (Experiment)** presents a series of ablation studies to dissect the LLaMEA-BO framework. We systematically investigate the impact of key evolutionary strategy parameters—including population sizes, selection mechanisms, and crossover rates—as well as the LLM's generative hyperparameters, such as temperature and sampling methods. This analysis provides insights into the optimal configuration of the framework.

- **Chapter 5 (Evaluation)** validates the performance of LLaMEA-BO. First, we analyze the evolutionary search dynamics under an optimized configuration, examining performance progression and error rates. Second, we benchmark the top-performing algorithms generated by the search against state-of-the-art methods on both the analytical BBOB suite and the practical Bayesmark HPO suite to assess their effectiveness and generalization capabilities.

- **Chapter 6 (Conclusion)** summarizes the key findings of this research, discusses the limitations of the current LLaMEA-BO framework, and outlines promising directions for future work in the automated discovery of optimization algorithms using LLMs.

Chapter 2

Background

2.1 Bayesian Optimization

Bayesian Optimization (BO) is a powerful, model-based framework designed for the sample-efficient optimization of expensive-to-evaluate black-box functions, where each function evaluation incurs significant computational or experimental cost [13]. The BO pipeline is typically structured around four core components: the Design of Experiments (DoE), the surrogate model, the acquisition function(AF), and the AF optimizer. These components work in concert to iteratively refine the search for the global optimum while balancing exploration and exploitation.

2.1.1 Design of Experiments

The optimization process begins with the DoE, which involves selecting an initial set of points to evaluate the objective function. The choice of initial points is critical, as it influences the surrogate model's ability to capture the underlying structure of the target function early in the process [10]. Traditional approaches for DoE often rely on simple random sampling or more sophisticated static space-filling designs, such as Latin Hypercube Sampling (LHS), which aim to uniformly cover the search space. Building on this, more advanced static space-filling methods like Sobol sequences or low-discrepancy quasi-Monte Carlo (QMC) methods have been explored to further improve coverage and reduce redundancy in initial designs. Beyond purely uniform coverage, some modern approaches for initial sampling also incorporate prior domain knowledge [15] to guide the selection of early samples towards promising regions and accelerate optimization.

2.1.2 Surrogate Model

The surrogate model serves as a probabilistic approximation of the black-box function, enabling efficient predictions and uncertainty quantification across the search space. The Gaussian Process (GP) is the most commonly used surrogate due to its flexibility in modeling smooth, continuous functions and its ability to provide well-calibrated uncertainty estimates [43]. GPs rely on kernel functions, such as the Matérn kernels, to encode assumptions about the function's smoothness and variability. However, recent research has explored alternative surrogate models to address scalability and complexity challenges. For instance, Bayesian neural networks (BNNs)[31] and deep Gaussian processes[49] offer greater expressiveness for high-dimensional or non-stationary functions.

2.1.3 Acquisition Function

The acquisition function (AF) quantifies the utility of evaluating a point in the search space, guiding the trade-off between exploration (sampling uncertain regions) and exploitation (focusing on regions likely to yield high objective values). Popular AFs include Expected Improvement (EI), which measures the expected gain over the current best observation, and Probability of Improvement (PI), which prioritizes points likely to surpass the best-known value [28]. The Upper Confidence Bound (UCB) function, which combines the predicted mean and uncertainty, is another widely used choice due to its simplicity and robustness [50]. Recent developments have introduced more sophisticated AFs, such as Knowledge Gradient (KG) for optimizing expected final outcomes [11] and Entropy Search (ES) or Predictive Entropy Search (PES), which aim to maximize information gain about the global optimum [21, 22].

2.1.4 Acquisition Function Optimizer

The AF Optimizer determines the next point to evaluate by optimizing the acquisition function. This is often a non-convex optimization problem, requiring efficient numerical methods. Standard approaches include gradient-based optimizers like L-BFGS-B [63] or global optimization techniques such as CMA-ES [19]. Recent advances have focused on improving scalability and robustness, particularly in high-dimensional spaces. For example, multi-point selection strategies, such as batch BO, enable parallel evaluations by selecting multiple points per iteration, using techniques like q-EI or local penalization to promote diversity [5, 14, 58].

The iterative interplay of these four components enables BO to efficiently navigate complex search spaces. The process continues until a predefined budget (e.g., number of evaluations) is exhausted or a satisfactory solution is identified. This modular, component-based structure of BO, while powerful, presents a complex design space where the optimal combination of surrogate models, acquisition functions, and optimizers is highly problem-dependent. This complexity makes it an ideal candidate for automated algorithm discovery.

2.2 Evolution Strategies

Evolution Strategies (ESs) are powerful derivative-free optimization algorithms inspired by the principles of natural evolution, specifically focusing on the processes of mutation, recombination, and selection [3]. ESs operate on a population of candidate solutions, iteratively improving them through the continuous adaptation of their parameters. Unlike many gradient-based methods, ESs do not require information about the objective function's gradient, making it exceptionally well-suited for complex, non-differentiable, non-convex, or noisy optimization problems.

At their core, ESs maintain a population of individuals, each representing a potential solution to the optimization problem. The primary search mechanism is mutation, where random perturbations are applied to an individual's parameters. Optionally, recombination (or crossover) can be used to combine multiple parents to create new offspring. Classic ES schemes include the (μ, λ)-ES (where λ offspring are generated from μ parents, and the μ best offspring are selected) and the $(\mu + \lambda)$-ES (where the μ best individuals are selected from the combined pool of μ parents and λ offspring).

In recent decades, ESs have seen significant advancements and renewed interest, particularly with the development of sophisticated variants like the *Covariance Matrix Adaptation Evolutionary Strategy* (CMA-ES) [19]. CMA-ES is renowned for its ability to efficiently handle high-dimensional, non-separable, and ill-conditioned problems by adaptively learning the covariance matrix of the Gaussian mutation distribution. This allows CMA-ES to capture interdependencies between parameters and align its search distribution with the landscape of the objective function, making it one of the state-of-the-art black-box optimization algorithms.

In our work, we extend this evolutionary paradigm from optimizing numerical parameter vectors to searching through a space of entire programs. This requires redefining the mutation and crossover operators, moving beyond simple numerical perturbations to sophisticated, semantics-aware code transformations. This is precisely the role that a LLM can fulfill, acting as a generative engine for algorithmic variation.

2.3 LLMs for Code Generation and Algorithm Design

2.3.1 Prompt Engineering

Prompt engineering is a pivotal technique for eliciting the full potential of LLMs, particularly in complex domains like code generation and algorithm design. It involves carefully crafting input prompts to guide LLMs towards desired outputs, structured reasoning, and improved performance.

Several advanced prompting techniques have emerged, each designed to enhance LLMs' reasoning abilities and robustness. A foundational approach is **Few-Shot Learning**, where the prompt includes a small number of input-output examples to demonstrate the desired task or format. This technique implicitly teaches the model the pattern and style of the expected response, significantly improving its performance on similar, unseen inputs without explicit fine-tuning. Building upon this, **Chain-of-Thought (CoT)** [57] significantly improves LLMs' complex reasoning by extending few-shot prompting to include step-by-step demonstrations of the reasoning process itself. This explicit reasoning path not only makes the LLM's internal process more transparent but also guides it toward more accurate and logical conclusions. Building further, **Self-Consistency** [55] samples multiple reasoning paths and aggregates their results to enhance robustness, effectively reducing errors by leveraging the wisdom of diverse "thought processes."

More elaborate reasoning structures extend these core ideas. **Tree-of-Thought (ToT)** [60] extends CoT by allowing LLMs to explore multiple reasoning branches, enabling a more exhaustive search for solutions, similar to how humans might explore different problem-solving avenues.

Beyond structured reasoning paths, interactive and iterative prompting paradigms have gained prominence. **Reasoning via Planning (RAP)** [20] enables LLMs to generate a plan before execution, allowing for more strategic and goal-oriented problem-solving. Iterative refinement techniques such as **Self-Refine** [38] and **Reflexion** [47] enable LLMs to iteratively improve their outputs by reflecting on their initial responses and incorporating feedback, either self-generated or external.

Despite these advancements, prompt engineering faces significant challenges. The performance of prompts, especially for novel and complex problems, remains highly sensitive to exact phrasing, keyword choice, subtle variations in instructions and LLM. Furthermore, crafting an effective prompt often requires substantial human intuition and

iterative trial-and-error. This manual tuning process is time-consuming and difficult to scale, making it impractical for large-scale or dynamic development environments.

2.3.2 Coding Agent

The challenges inherent in relying solely on prompt engineering for complex software development tasks, such as maintaining context over long interactions and ensuring robustness across diverse scenarios, have driven the exploration of more sophisticated architectural paradigms. Inspired by the efficacy of human software development teams, where specialized roles contribute collaboratively towards a common goal, multi-agent systems for code generation leverage multiple roles, each often assigned distinct responsibilities and endowed with access to external tools, knowledge bases, and internal memory mechanisms.

One pioneering system in this domain is **MetaGPT** [23], which draws inspiration from human Standard Operating Procedures (SOPs). It assigns specific, well-defined roles such as Product Manager, Architect, Project Manager, Engineer, and Quality Assurance (QA) Engineer. This organizational structure effectively mimics a streamlined software development lifecycle (SDLC), allowing for complex projects to be broken down and managed by specialized agents.

Further enhancing validation and quality control, **AgentCoder** [25] places particular emphasis on robust testing. It introduces a dedicated *Tester* agent, uniquely prompted to generate a comprehensive suite of tests—including basic, edge, and large-scale scenarios to the generated code.

To emulate more sophisticated human cognitive processes in problem-solving, **MapCoder** [26] orchestrates four specialized LLM agents: Retrieval, Planning, Coding, and Debugging. The Retrieval Agent functions as an external memory module, efficiently identifying and recalling relevant past problems and their solutions from a knowledge base, thereby enriching the current problem-solving context. Subsequently, the Planning Agent devises a detailed, step-by-step strategic plan for the target problem, critically informed by these retrieved examples. These crucial memory and planning stages provide structured guidance and contextual awareness, enabling the subsequent Coding and Debugging agents to generate, refine, and validate accurate and efficient code, mirroring how human developers leverage past experiences and systematic approaches.

A key innovation in self-correction and quality assurance is presented by **QualityFlow** [24]. This system introduces a *Quality Checker* agent, employing a novel method called *Imagined Execution*. In this approach, an LLM performs reasoning to mentally emulate the execution of a synthesized program against provided unit tests. By meticulously comparing the emulated output with the expected output based on its reasoning, the Quality Checker can predict the correctness of the code. This prediction mechanism then determines whether to submit the code, trigger further debugging steps or initiate clarification queries.

Despite their notable advancements, the inherent complexity of multi-agent systems often leads to increased computational costs due to multiple LLM calls and intricate inter-agent communication protocols. Furthermore, while mimicking human team dynamics, these systems still lack true human-like intuition, common sense reasoning, or the ability to generalize robustly beyond their training data in truly novel scenarios, inheriting some of the fundamental limitations of the underlying LLMs. That make them more suitable for the normal code generation tasks, rather than the algorithm design tasks. In contrast, our approach deliberately avoids the overhead of a multi-agent framework. By

embedding a single LLM directly into a classic evolutionary loop, we aim to harness its creative generation capabilities more directly for the singular goal of algorithmic innovation, prioritizing a focused, iterative search over a complex, role-based workflow.

2.3.3 Evolutionary Search

Evolutionary Search have gained significant attention for their efficacy and robustness in navigating complex search spaces with the assistance of LLMs [40, 59].

A pivotal shift occurred with the introduction of the **Evolution through Large Models (ELM)** [30]. This work innovatively integrated LLMs directly into the evolutionary loop, tasking the LLM with performing mutation operations on code. This marked a departure from traditional, often random or rule-based, mutation operators by leveraging the LLM's learned understanding of code structure and semantics to propose more contextually relevant and potentially more effective modifications.

Building upon this foundation, **EvolLLM** [52] further explored the synergy between LLMs and evolution strategies. EvolLLM introduced two key innovations: a warm-start mechanism to initialize the population with potentially high-quality candidates generated by the LLM, and a specialized "least-to-most" selection strategy. This selection prioritizes simpler, more promising solutions early on, gradually increasing complexity, thereby aiming for more efficient and guided exploration of the search space. Concurrently, the **Evolution of Heuristics (EoH)** [32], focused on refining the evolutionary operators themselves. EoH defined more specialized mutation and crossover operations to achieve a better balance between exploration of novel solution areas and exploitation within the search space.

Addressing the challenge of maintaining population diversity, **FunSearch** [44] adopted a multi-island evolutionary algorithm. In this paradigm, multiple populations evolve in parallel, with occasional migration of individuals between islands. The LLM's role in FunSearch was to generate new programs or functions within these distinct populations. This architectural innovation facilitated the exploration of a more diverse set of candidate solutions, proving particularly effective in domains like mathematical discovery by preventing the entire search from collapsing into a single local optimum. **AlphaEvolve** [39] further extended this concept of diverse exploration by building upon the multi-island approach. Its key innovations included the use of multiple LLMs and multi-objective optimization. This allowed AlphaEvolve to maintain diversity not only in the solution space but also in terms of the objectives being optimized, guiding the search towards a richer set of Pareto-optimal solutions.

A distinct line of work, exemplified by the **LLaMEA** [51, 54], has embraced Evolution Strategies more holistically to tackle continuous black-box optimisation problems. LLaMEA innovatively uses an LLM to generate complete Python classes, including stateful components, that represent metaheuristics. These LLM-generated optimizers are then rigorously evaluated on benchmark problems using tools like IOHexperimenter [7]. The framework establishes a closed evolutionary loop where selection mechanisms, LLM-driven mutation prompts tailored for heuristic design, and automatic error handling collaborate. This sophisticated integration has led to the discovery of novel metaheuristics that outperform established state-of-the-art optimization algorithms such as CMA-ES and Differential Evolution on the BBOB testbed [18], showcasing the potential of LLM-guided evolution to not just generate code snippets but entire algorithmic strategies.

The randomness introduced by evolutionary search methods forces LLMs to explore solutions that may not be directly suggested by the training data, thereby encouraging

the generation of genuinely novel solutions. This is particularly beneficial in algorithm design tasks, where the goal is to innovate and discover new algorithms or heuristics that can outperform existing methods.

Despite these significant advancements, several challenges persist in the application of LLMs to evolutionary code generation. The computational overhead of repeatedly querying powerful LLMs within an evolutionary loop can be substantial. Furthermore, ensuring genuine novelty and preventing the evolutionary process from merely rediscovering solutions heavily influenced by the LLM's vast but potentially common training data remains an active area of research. Scalability to very large or highly complex codebases and the interpretability of the evolutionary trajectory guided by opaque LLM decisions also pose ongoing difficulties.

2.4 LLMs in Bayesian Optimization

Recent advancements have seen LLMs increasingly integrated into various facets of BO, moving beyond simple task automation to fundamentally innovate components of the BO pipeline.

BO-ICL [42], a zero-shot BO method, innovatively bypassed the need for traditional model training or explicit feature engineering, with the LLM serving as an in-context surrogate regressor that also provided mean and uncertainty estimates crucial for defining an acquisition function.

Building on the idea of LLMs assisting specific BO components, **ADO-LLM** [62] uniquely addressed complex multi-objective optimization in analog circuit sizing by employing LLMs in a dual capacity: first, for a zero-shot initialization phase to suggest viable design points, and second, for generating new candidate solutions during the acquisition phase through in-context learning.

A distinct line of inquiry focuses on automating the design of core BO components themselves, particularly acquisition functions (AFs). **EvolCAF** [61] automatically design cost-aware AFs through iterative crossover and mutation operations performed directly in the algorithmic space. Furthering the theme of algorithmic discovery for AFs, **FunBO** [2], builted upon the FunSearch [44], distinctively frames AF design as an algorithm discovery problem. In this paradigm, an LLM is tasked with generating Python code for novel AFs.

While the aforementioned works utilized LLMs to improve or automate specific BO components, **LLAMBO** [35] presented a more holistic integration. They demonstrated that LLMs can orchestrate all phases of a modular BO framework, where interactions are framed in natural language. This comprehensive approach proved particularly effective in improving algorithmic performance compared to established methods, especially during the early stages of optimization when observational data is sparse.

Chapter 3

Methodology

In this chapter, we present our LLaMEA-BO algorithm. We begin by detailing two primary obstacles in using LLMs for evolutionary search: ensuring the feasibility of generated code and maintaining sufficient population diversity against the LLM's inherent biases. We then introduce the LLaMEA-BO framework, explaining how it is structured to manage the evolutionary process. Subsequently, we delve into the specifics of our prompt design, demonstrating how carefully engineered prompts serve as the primary mechanism for mitigating the identified challenges. Finally, we describe the evaluation protocols used to assess the performance of the generated algorithms during the evolutionary search.

3.1 Challenges of LLM-based ES

Using an LLM to generate entire algorithms within an evolutionary loop presents a paradigm shift from traditional ES, where operators manipulate well-defined representations like numerical vectors or permutations. This shift introduces two fundamental challenges: generating feasible solutions and maintaining population diversity.

Generating Feasible Solutions In conventional optimization, mutation and crossover on a solution vector almost always produce a syntactically valid new solution. Constraints might be violated (e.g., a parameter value exceeds its bounds), but these issues are typically easy to detect and rectify with simple heuristics like clipping or resampling[8].

In contrast, when the solution is a complete block of code, ensuring feasibility is far more complex. The LLM, despite its proficiency, is prone to several failure modes that can render a generated algorithm unusable:

- **API Hallucination:** The model may invent functions, methods, or parameters that do not exist in the specified libraries[27].
- **Compatibility Issues:** The model can inadvertently mix APIs from different library versions or programming environments, leading to code that is syntactically plausible but fails at runtime[33].

Rectifying these errors automatically is non-trivial and highlights the need for strategies that proactively guide the LLM towards producing correct and executable code.

Maintaining Diversity Population diversity is the engine of evolutionary search, preventing premature convergence and enabling the discovery of novel solutions. Traditional ES maintains diversity through high-entropy stochastic operators like Gaussian mutation or uniform crossover[8]. An LLM, however, is not a source of pure randomness; it is a highly biased model, which is both its strength, for generating plausible code, and its weakness, for lacking true originality. This bias manifests in several ways that threaten diversity:

- **Inherent Model Biases:** An LLM is trained to predict the most probable next token, that pulls the population towards common, canonical implementations seen in its training data. This tendency is amplified by fine-tuning methods. This process actively discourages the generation of unconventional solutions that are crucial for the long-term exploratory power of evolution.
- **Prompt-Induced Biases:** A generic prompt like "Mutate this code to be more efficient" will likely cause the LLM to apply a limited set of common optimization techniques to every individual, making the population more similar, not more diverse.

Addressing the diversity challenge could involve several advanced strategies, each with its own trade-offs:

- **Advanced Prompting**: One could employ sophisticated prompt engineering, such as multi-agent setups where LLMs adopt different "roles" (e.g., an "explorer" vs. an "exploiter"), though this risks introducing new, uncontrolled biases.
- **Parameter Manipulation**: Another approach is to manipulate the LLM's generation parameters, like temperature, to increase randomness. This often comes at the cost of generating more infeasible solutions.
- **Structured Population Management**: For larger populations, structured approaches like Quality-Diversity algorithms[41] or island models[44] could be effective.
- **Explicit Diversity Metrics**: One could define and optimize for explicit diversity metrics. These range from superficial text-based measures (e.g., Levenshtein distance) to more semantically meaningful but complex embedding-based metrics. However, defining a truly meaningful and computationally tractable code diversity metric remains an open research problem, making its direct application in the optimization loop impractical.

Given these complexities, we recognize that such intricate solutions can be a double-edged sword, potentially introducing new, uncontrolled biases and reducing the robustness or reproducibility of the search process. In light of this, our methodology opts for a more direct two-pronged approach. First, we leverage the robust structure of the ES framework itself to impose selective pressure and guide the search. Second, we complement this structural foundation with meticulous prompt engineering, designed to directly steer the LLM's behavior without being overly prescriptive. The following sections detail these two pillars of our methodology.

Algorithm 1 LLaMEA-BO

Input:

T: Total number of iterations, μ: Size of parent population
λ: Size of offspring population, p_{cr}: Crossover rate
top_k: Top-k parameter for LLM, $temperature$: Temperature parameter for LLM
$isElitism$: Boolean indicating whether to use elitism or not.

$t \leftarrow 0$, $P_t \leftarrow \emptyset$
▷ *Initialization Phase*
while $t < \mu$ **do**
    $prompt \leftarrow \text{getInitialPrompt}(P_t)$
    $solution \leftarrow \text{LLM}(prompt, top\_k, temperature)$
    $result \leftarrow \text{Evaluate}(solution)$, $t \leftarrow t + 1$
    $P_t \leftarrow P_t \cup (solution, result)$
▷ *Evolutionary Loop*
while $t < T$ **do**
    $prompts \leftarrow \text{getPrompts}(P_t, \min(\lambda, T - t), p_{cr})$
    $solutions \leftarrow \text{LLM}(prompts, top\_k, temperature)$
    $results \leftarrow \text{Evaluate}(solutions)$
    if $isElitism$ **then**
        $P_t \leftarrow \text{Select}(P_t \cup solutions, results)$ ▷ Elitist Selection
    else
        $P_t \leftarrow \text{Select}(solutions, results)$ ▷ Non-Elitist Selection
    $t \leftarrow t + |P_t|$

function GETPROMPTS(P, $size$, p_{cr})
    $sortedP \leftarrow \text{sort}(P)$ ▷ Sort population by fitness
    $Pa_{cr} \leftarrow \text{comb}(sortedP)$ ▷ Create combinations for crossover
    $Pa_{\mu} \leftarrow sortedP$
    $prompts \leftarrow \emptyset$
    for $i \leftarrow 0$ **to** $size - 1$ **do**
        if $\text{rand}() < p_{cr}$ **then**
            $parents \leftarrow \text{dequeue}(Pa_{cr})$ ▷ Perform crossover
        else
            $parents \leftarrow \text{dequeue}(Pa_{\mu})$ ▷ Perform mutation
        $prompts \leftarrow prompts \cup \text{generatePrompt}(P, parents)$
        if $|Pa_{cr}| = 0$ **then**
            $Pa_{cr} \leftarrow \text{comb}(sortedP)$
        if $|Pa_{\mu}| = 0$ **then**
            $Pa_{\mu} \leftarrow sortedP$
    return $prompts$

3.2 LLaMEA-BO

The LLaMEA-BO algorithm, outlined in Algorithm 1, is architecturally an ES, but with its core components specifically redesigned to address the unique challenges of using an LLM as a variation operator.

The process begins with an **initialization phase** (lines 2-6 in Algorithm 1). Here, an initial parent population P_0 of size μ is established. To counteract the LLM's inherent bias towards generating canonical solutions—a key diversity challenge identified in Section 3.1—LLaMEA-BO employs a **diversity-driven initialization**. When generating each of the μ initial solutions, the prompt includes the source code of any previously generated solutions from the initial batch. This instruction explicitly encourages the LLM to produce solutions distinct from those already in the initial set, fostering a diverse starting population critical for effective exploration. Each generated solution is evaluated, and establish the initial population P_0 with their fitness values.

Following initialization, LLaMEA-BO enters its main **evolutionary loop** (lines 7-15). In each iteration of this loop, a new offspring solution is created through genetic variation operators—either **mutation** or **crossover**—applied to selected parent solutions. The specific operator is chosen stochastically based on a predefined crossover probability, p_{cr}.

A second significant departure from conventional ES mechanisms is LLaMEA-BO's **deterministic parent selection strategy** (lines 16-31) for these variation operators. Instead of relying on purely probabilistic methods (e.g., fitness-proportional selection), LLaMEA-BO first sorts the current parent population P_t by fitness. The individuals or pairs with high performance will be selected first. The algorithm then deterministically cycles through these pre-selected parent individuals or combinations when generating prompts for the LLM. This strategy is motivated by the need to reduce the generation of infeasible offspring, which is assigned a fitness of 0. However, this deterministic focus on top performers is an explicit trade-off, as it reduces population diversity and favors exploitation over exploration.

Once the λ (or fewer) offspring solutions are generated and their fitness subsequently evaluated, a **selection mechanism** determines the composition of the parent population for the next generation.

This evolutionary cycle of parent selection, variation (mutation/crossover via LLM prompting), offspring generation, evaluation, and population selection repeats. The algorithm terminates when a predefined stopping criterion is met, specifically, when a total of T solutions (i.e., T fitness evaluations) have been performed (as indicated by the loop condition $t < T$, where t accumulates the count of evaluated solutions from initialization and subsequent generations).

3.3 Prompt Design

Careful prompt engineering is the second pillar of our methodology, where each component of the prompt is designed to counteract the failure modes identified in Section 3.1. The prompt is the direct interface to the LLM, making it our primary tool for controlling code generation. A comprehensive example of our base prompt template is presented in Prompt Template.

Prompt Template

You are a highly skilled computer scientist in the field of natural computing. Your task is to design novel metaheuristic algorithms to solve black box optimization problems

The optimization algorithm should handle a wide range of tasks, which is evaluated on the BBOB test suite of 24 noiseless functions. Your task is to write the optimization algorithm in Python code. The code should contain an `__init__(self, budget, dim)` function and the function `__call__(self, func)`, which should optimize the black box function `func` using `self.budget` function evaluations.

The `func()` can only be called as many times as the budget allows, not more. Each of the optimization functions has a search space between -5.0 (lower bound) and 5.0 (upper bound). The dimensionality can be varied.

As an expert of numpy, scipy, scikit-learn, torch, gpytorch, you are allowed to use these libraries. Do not use any other libraries unless they cannot be replaced by the above libraries. Do not remove the comments from the code.

Name the class based on the characteristics of the algorithm with a template `<characteristics>BO`.

Give an excellent, novel and computationally efficient Bayesian Optimization algorithm to solve this task, give it a concise but comprehensive key-word-style description with the main ideas and justify your decision about the algorithm.

<*Mutation*> or
<*Crossover*> or
<*Diversity Initialization*> or
<*Code Template (Appendix B)*>

Give the response in the format:
Description
<description>
Justification
<justification for the key components of the algorithm or the changes made>
Code
<code>

Role Setting Role prompting is a well-established technique for eliciting desired behaviors, particularly for complex tasks [45]. By assigning the LLM the persona of a *"highly skilled computer scientist in the field of natural computing,"* we prime it to access its knowledge base related to algorithm design and optimization.

Task Instructions and Constraints By providing explicit and detailed instructions, we heavily constrain the LLM's vast output space. This includes:

- The programming language (Python).
- A specific class structure, requiring an `__init__(self, budget, dim)` constructor and a `__call__(self, func)` method to standardize the evaluation interface.
- Constraints on resources, such as the `budget` for function evaluations and the search space bounds ([-5.0, 5.0] per dimension).
- A whitelist of permitted libraries (`numpy`, `scipy`, `scikit-learn`, `torch`, `gpytorch`). This is a critical step to prevent API hallucination and compatibility issues. Restricting the model to a known, stable set of libraries that are likely well-represented in its training data increases the probability of generating correct code.
- A naming convention for the generated class (`<characteristics>BO`) to reflect the algorithm's nature.
- An explicit directive for *excellent, novel and computationally efficient* algorithms to guide the LLM towards high-quality solutions.

Diversity Initialization Prompt To directly combat the LLM's tendency towards homogenization, the prompt for generating the initial population is augmented with an explicit instruction for novelty.

> **Diversity Initialization**
>
> n algorithms have been designed. The new algorithm should be as **diverse** as possible from the previous ones on every aspect.
> If errors from the previous algorithms are provided, analyze them. The new algorithm should be designed to avoid these errors.
> *<list of n algorithms>*

This prompt steers the LLM to explore different regions of the algorithm design space from the outset, mitigating the risk of the population converging prematurely around a single canonical design.

Mutation and Crossover Operators The mutation operator follows a methodology similar to that described in vanilla LLaMEA [51]. For a selected parent, the mutation prompt appends the parent's source code, its performance score, and a directive to improve it.

> **Mutation**
>
> The selected solution to update is: *<parent a>*
> Refine the strategy of the selected solution to improve it.

In the crossover operation, two parents are provided, and the LLM is instructed to synthesize a novel algorithm by combining their beneficial features.

Crossover

The selected solutions to update are: <*parent* a_i> <*parent* a_j>
Combine the selected solutions to create a new solution. Then refine the strategy of the new solution to improve it. If the errors from the previous algorithms are provided, analyze them. The new algorithm should be designed to avoid these errors.

These prompts are intentionally minimalist. Rather than providing complex instructions that might over-constrain the LLM and cause it to ignore the rich context of the parent solutions, they offer high-level directives ("refine," "combine"). This approach is designed to leverage the LLM's emergent reasoning and code synthesis capabilities, using the parent code and performance data as a catalyst. It thereby encourages a context-aware exploration of the algorithmic search space.

Code Template A partial code template (see Appendix B) is provided within the prompt to modularize the Bayesian Optimization process into distinct logical components: (1) sampling strategy in `_sample_points`, (2) surrogate model fitting in `_fit_model`, (3) acquisition function logic in `_acquisition_function`, (4) optimization of the acquisition function in `_select_next_points`, and (5) the main optimization loop in `__call__`. This structured approach serves a threefold purpose. First, it acts as a structural exemplar, a form of in-context learning, demonstrating the expected structure. Second, it ensures all generated algorithms have a consistent, composable structure, which is vital for the evolutionary process. Third, by enforcing a rigid skeleton, the template heavily constrains the LLM's output, directly addressing the feasibility challenge. However, this rigidity is a deliberate trade-off: it confines the search to the family of BO-like algorithms, gaining feasibility and consistency at the cost of potentially limiting the LLM's ability to discover radically different algorithmic paradigms.

Output Format A structured output format is mandated, requiring the LLM to provide:

- `# Description`: A concise, keyword-style summary of the algorithm's main ideas.
- `# Justification`: An explanation for the design choices or modifications. This component is crucial for enabling a Reflexion-like process [47], where the LLM analyzes and rationalizes its own output, potentially leading to self-correction.
- `# Code`: The generated Python code for the algorithm.

This structured output facilitates automated parsing and subsequent use of the LLM's response.

Solution Feedback for Iterative Refinement A critical component of the evolutionary loop is the feedback provided to the LLM after a solution is evaluated. This feedback includes the **fitness score**, the sole criterion for the evolutionary selection mechanism, which quantifies performance on the target benchmark. Additionally, the **execution time** is reported back to guide the LLM towards more computationally efficient solutions. Finally, any **error messages** from failed executions are also fed back, allowing the LLM to identify and rectify bugs, thereby enabling an iterative debugging process. Thus, while fitness, execution time, and error messages all inform the LLM's generation process, only fitness dictates survival in the evolutionary selection step.

3.4 Evaluation during Evolutionary Search

The fitness score that drives the evolutionary search is derived from performance on Black-Box Optimization Benchmarking (BBOB) test suite from the COmparing Continuous Optimizers (COCO) platform[17]. To maintain computational tractability during evolution, we use a representative subset of 10 functions (specifically, IDs 2, 4, 6, 8, 12, 14, 15, 18, 21, and 23) from the 24 noiseless functions. For each of these functions, only instance 1 is used to guide the evolutionary search, reserving the other instances for out-of-sample evaluation. This selection ensures coverage of two functions per high-level problem class (e.g., separable, multimodal). Each algorithm's performance is measured in a dimensionality of $d = 5$ with a strict budget of $B = 100$ evaluations, with each experimental run repeated 3 times. The primary performance metric is the **Area Over the Convergence Curve** (AOCC), which integrates the improvement in solution quality over the evaluation budget, thus quantifying the overall efficiency of the convergence trajectory. Formal definitions of the AOCC performance metric and specifics on its aggregation across different settings are detailed in Appendix A.

Chapter 4

Experiment

To systematically assess the performance of the LLaMEA-BO algorithm and understand the influence of its core components, we performed a series of comprehensive experiments. The LLaMEA-BO framework is built upon two fundamental pillars: the design of the instructional prompts given to the LLM, and the configuration of the evolutionary strategy and LLM. While the prompt structures were carefully crafted based on heuristics and extensive preliminary testing, this chapter focuses on the latter pillar, systematically analyzing the impact of key ES parameters—such as parent and offspring sizes (μ and λ), the selection mechanism, and the crossover rate—as well as the generative hyperparameters of the LLM.

The LLM at the core of LLaMEA-BO is Gemini-2.0-Flash[1]. This model was chosen for its strong performance in preliminary tests and its widespread free API availability, which enhances the reproducibility of our results. Unless otherwise specified, all experiments were conducted using a default configuration: an LLM temperature of 1.0, Top-K sampling with K=40, and an ES crossover rate of 0.6. The search budget was fixed at 40 evaluation units, where one unit entails the generation and evaluation of a single candidate BO algorithm. Performance is primarily evaluated using the Area Over the Convergence Curve (AOCC) and the error rate, which is defined as the proportion of generated solutions that fail to compile or execute: $\textit{Error Rate} = (\textit{Number of Failed Solutions})/(\textit{Total Number of Runs})$. All experiments were benchmarked against 10 functions from the BBOB suite, adhering to the methodology detailed in Section 3.4. Specifically, for each function, we used the instance 1 in 5 dimensions, with an evaluation budget of 100, and averaged the results over 5 independent repetitions.

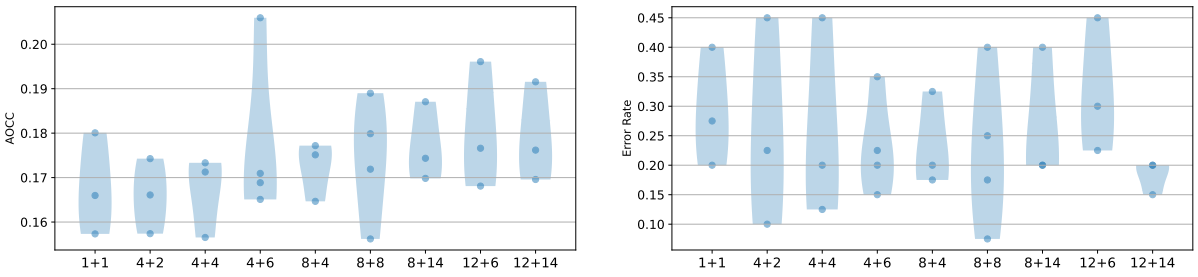
4.1 ES Configuration

The Ratio of μ and λ

A fundamental parameter in ES is the offspring-to-parent ratio, λ/μ, which governs the selection pressure. A higher ratio signifies a stronger selection pressure: a larger pool of offspring is generated from a smaller parent population, meaning a greater proportion of individuals will be discarded. This intensified pressure increases the likelihood of finding an individual that surpasses the current parents in a single generation, thereby accelerating the search for high-quality solutions. However, it also carries the risk of reducing population diversity too quickly, potentially leading to premature convergence. To investigate this effect, we conducted experiments with various combinations of μ and λ. The configurations

tested were (1+1), (4+2), (4+4), (4+8), (8+4), (8+8), (8+14), (12+6), and (12+14). Performance was measured using AOCC and the error rate.

The results, presented in Figure 4.1, offer two key insights. First, for a fixed parent population size μ, increasing the offspring population size λ (i.e., a higher λ/μ ratio) consistently leads to better performance. For example, (4+8)-ES outperforms both (4+4)-ES and (4+2)-ES. This suggests that a higher selection pressure, afforded by a larger pool of offspring, provides a more effective balance between exploration and exploitation, enabling the search to discover better solutions. Second, increasing the overall population size (both μ and λ) also tends to yield improved AOCC. However, this comes at the cost of stability, as larger populations appear to have higher variance in their final performance. The error rate, in contrast, does not exhibit a clear trend across the different configurations, indicating that these population parameters primarily influence the quality of the evolutionary search rather than the syntactic correctness of the LLM's generated code.

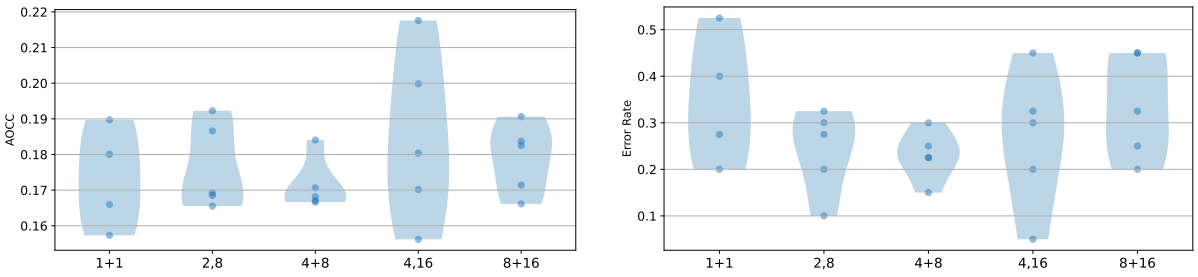


(a) AOCC (b) Error Rate

Figure 4.1: Results from different μ and λ configurations averaged over 10 BBOB functions.

Selection Mechanism

Based on the findings from the section 4.1, which highlighted the benefit of a significant selection pressure, we established specific offspring-to-parent ratios for comparing elitist (plus) and non-elitist (comma) selection strategies. For the non-elitist (μ, λ) strategy, a higher ratio of $\lambda = 4\mu$ was chosen to increase the likelihood of generating an offspring superior to any parent, thereby compensating for the absence of elitism. For the elitist (μ + λ) strategy, a more moderate ratio of $\lambda = 2\mu$ was selected, as the preservation of the best parent already guarantees non-degradation of the best-so-far solution.



(a) AOCC (b) Error Rates

Figure 4.2: Results from different population sizes and elitism configurations averaged over 10 BBOB functions and 5 repetitions per function.

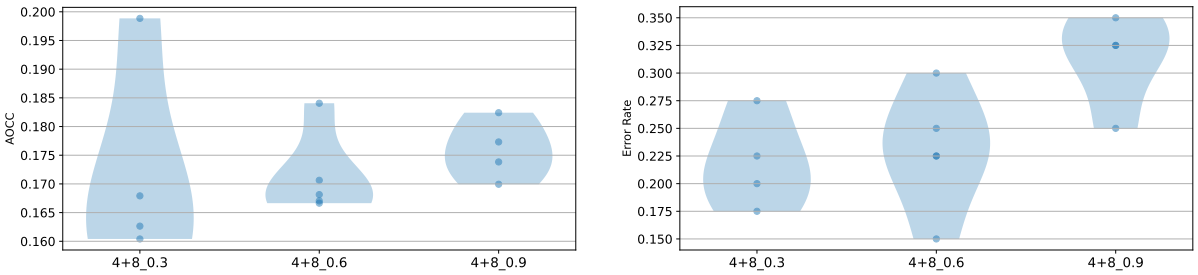
This investigation into population size and elitism employed a (1+1)-ES as a baseline configuration. This baseline was compared against selected elitist strategies, namely (4+8)-ES and (8+16)-ES, and non-elitist strategies, specifically (2,8)-ES and (4,16)-ES. This selection allows for an examination of varying parent (μ) and offspring (λ) population sizes, as well as the direct impact of preserving the best parent(s) via elitism.

The results, presented in Figure 4.2a, reaffirm the findings from the previous section: an increase in population size generally correlates with improved performance, as measured by higher AOCC values. This trend holds across both selection strategies; within their respective categories, the configurations with larger populations demonstrated enhanced performance, with (8+16)-ES outperforming (4+8)-ES and (4,16)-ES surpassing (2,8)-ES. When comparing the two strategies, a clear trade-off emerges. For a given offspring size λ, the non-elitist (comma) strategy tends to find better solutions, but at the cost of higher instability. For example, (4,16)-ES achieves a better median AOCC than (8+16)-ES but exhibits a much wider performance distribution. This volatility is inherent to the non-elitist approach, which encourages aggressive exploration by discarding the parent population but also risks losing high-quality solutions. The instability of the non-elitist strategy is also evident in the error rates shown in Figure 4.2b. The comma strategies consistently yield higher and more variable error rates, suggesting that the complete replacement of parents may disrupt the evolutionary context provided to the LLM, making it more difficult to generate syntactically and semantically valid offspring.

Crossover Rate

To assess the influence of the crossover rate, experiments were conducted using crossover probabilities of 0.3, 0.6, and 0.9. These rates were applied consistently to the (4+8)-ES strategy, chosen as a representative configuration.

Figure 4.3a reveals that both excessively high and excessively low crossover rates resulted in a discernible degradation of performance, again measured by AOCC. The rationale behind this observation can be elucidated by examining the error rates presented in Figure 4.3b. Specifically, a higher crossover rate (e.g., 0.9) was correlated with an increased error rate, possibly due to overly disruptive recombination leading to the frequent loss of beneficial schemata. Conversely, a very low crossover rate (e.g., 0.3) appeared to diminish the algorithm's exploratory capabilities by limiting the generation of novel solutions through effective recombination, thereby potentially hindering its ability to escape local optima and discover more promising regions of the search space.



(a) AOCC (b) Error Rates

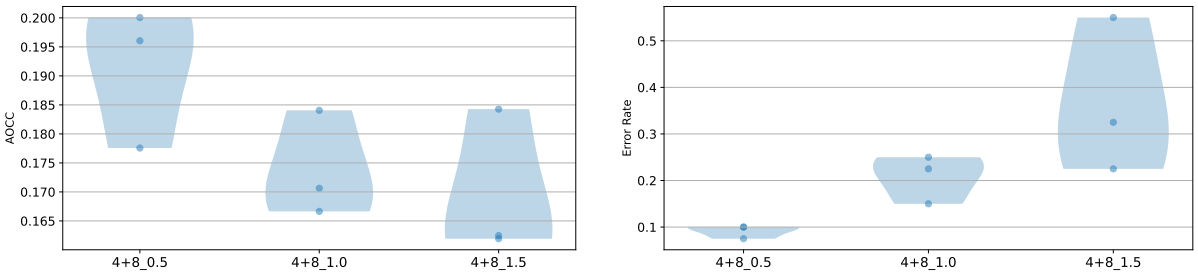
Figure 4.3: Results from different crossover rate $[0.3, 0.6, 0.9]$ configurations (using a $(4+8)$ ES strategy) averaged over 10 BBOB functions and 4 repetitions per function.

4.2 LLM Configuration

This section details the configuration of the Large Language Model (LLM) employed in our experiments and presents an ablation study investigating the impact of three key hyperparameters: temperature, top-k sampling, and top-p sampling. These hyperparameters critically influence the LLM's generation process. The study was conducted within an Evolutionary Strategy (4+8). Each optimization process was constrained to a budget of 40 fitness evaluations.

- **Temperature**: Controls the randomness of token selection. Lower temperatures (e.g., <1.0) yield more deterministic, focused, and often syntactically conventional code, favoring high-probability (common) constructs. Higher temperatures (e.g., >1.0) encourage more diverse, novel, or even unexpected code structures, but increase the risk of syntactical errors or unconventional (potentially less efficient or incorrect) logic.
- **Top-k Sampling**: Restricts sampling to the k most probable next tokens. Smaller k values lead to more predictable and conservative code, as the model is limited to a narrow set of common next tokens. Larger k values allow for greater diversity in token choice, potentially leading to more varied or creative code solutions, but can also introduce less relevant or erroneous tokens if k is too large.
- **Top-p Sampling**: Selects tokens from the smallest set whose cumulative probability exceeds a threshold p. Lower p values result in more focused and deterministic code, similar to low temperature, as only the most probable tokens are considered. Higher p values allow for more diversity, especially when the model is less certain about the next token (i.e., a flatter probability distribution). This adaptability can be useful for code, allowing conservative choices for syntax-critical parts and more explorative choices for semantic content or algorithmic variations.

Temperature



(a) AOCC (b) Error Rate

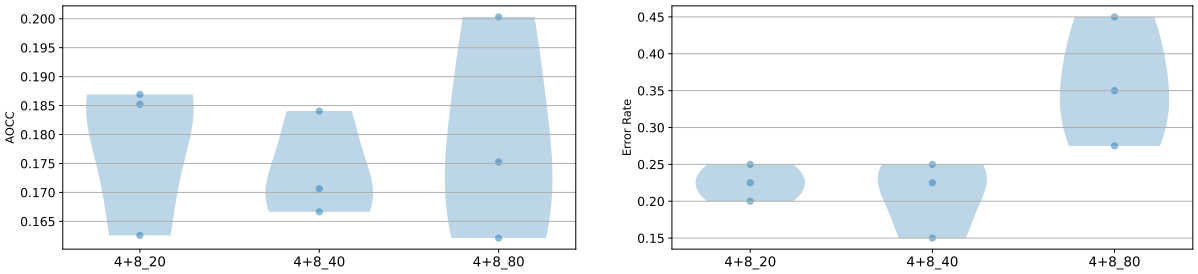
Figure 4.4: Results from different LLM temperature $[0.5, 1.0, 1.5]$ configurations (using a $(4+8)$ ES strategy) averaged over 10 BBOB functions and 3 repetitions per function.

In our ablation study on temperature, we evaluated three distinct values: 0.5, 1.0, and 1.5, while keeping other hyperparameters constant. The results, illustrated in Figure 4.4, reveal a clear and significant pattern. Specifically, lower temperature settings (e.g., 0.5) demonstrably reduce the error rate of the generated outputs. This reduction in errors

translates directly to improved overall performance in the context of our experimental task.

Top-K

For the investigation into top-k sampling, we experimented with values of 20, 40, and 80. The findings, presented in Figure 4.5, indicate that a top-k value of 80 achieved the best performance as measured by AOCC. Interestingly, this same configuration (top-k=80) also exhibited the highest observed error rate. This intriguing outcome suggests that a larger top-k, by broadening the pool of potential tokens for selection, may enhance the LLM's capacity for generating more creative or explorative responses. While this increased exploration can elevate the error rate, it appears to simultaneously facilitate the discovery of superior solutions, thereby improving the AOCC metric.

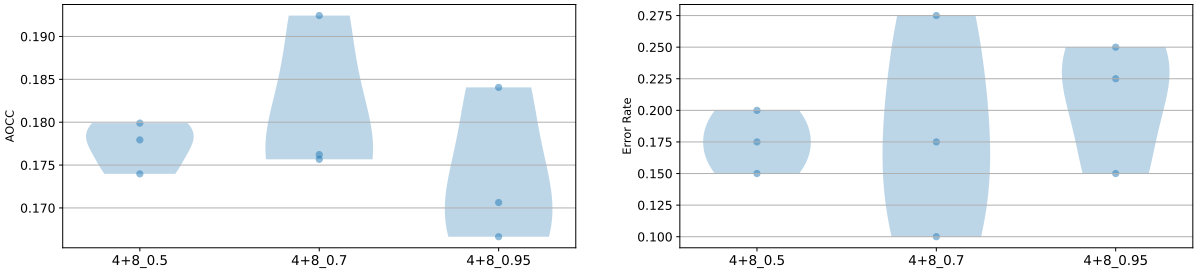


(a) AOCC (b) Error Rate

Figure 4.5: Results from different LLM top-K $[20, 40, 80]$ configurations (using a $(4 + 8)$ ES strategy) averaged over 10 BBOB functions and 3 repetitions per function.

Top-P

The effect of top-p (nucleus) sampling was investigated by testing values of 0.5, 0.7, and 0.95. Similar to the other studies, the AOCC and error rate data from three independent runs for each top-p setting are presented in Figure 4.6.



(a) AOCC (b) Error Rate

Figure 4.6: Results from different LLM top-P $[0.5, 0.7, 0.95]$ configurations (using a $(4 + 8)$ ES strategy) averaged over 10 BBOB functions and 3 repetitions per function.

The results show an interesting interplay between AOCC and error rate. While top-p=0.5 achieved the lowest average error rate, its AOCC was surpassed by top-p=0.7. The top-p value of 0.7 appears to strike a beneficial balance, achieving the best AOCC despite

a marginally higher error rate than top-p=0.5. Increasing top-p further to 0.95 resulted in the highest error rate and a decrease in AOCC compared to top-p=0.7, suggesting that while higher top-p values increase token diversity, excessively high values might introduce too much noise or irrelevant exploration, thereby hindering the discovery of optimal solutions.

The variability observed across the independent runs for each setting also highlights the stochastic nature of the generation and optimization process. Another possible explanation is that the number of runs was too low to reduce the variance of the results to a level that would allow for a more reliable comparison of the top-p values.

Chapter 5

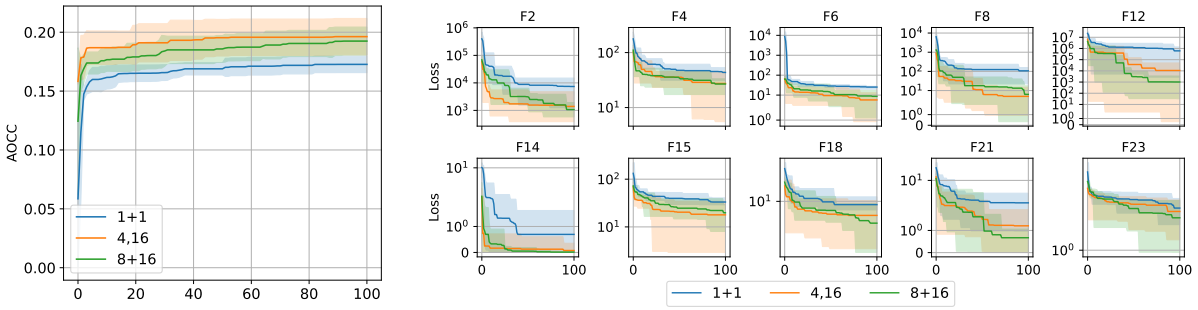
Evaluation

This chapter presents a comprehensive evaluation of our proposed approach. We first analyze the evolutionary search process itself, comparing different configurations of the evolution strategy to understand their impact on performance. Subsequently, we validate the generalization capabilities of the algorithms produced by this search, benchmarking them against state-of-the-art methods on a comprehensive test suite.

5.1 Evolutionary Search Evaluation

Experimental Setup Based on preliminary ablation studies (see Chapter 4), which indicated that a decoding temperature of 0.5 significantly improves performance, we adopted this setting for our main experiments. To counteract the potential reduction in diversity associated with lower temperatures, we employed a high Top-K sampling value(60). The evolutionary search was configured using three common evolution strategy (ES) setups to assess the impact of population size and selection strategy: an elitist $(1+1)$–ES, a large-population elitist $(8+16)$–ES, and a large-population non-elitist $(4, 16)$–ES. Each search was conducted for a budget of 100 evaluation units with a 30-minute timeout and repeated five times for statistical robustness. In this more detailed analysis, we supplement the AOCC metric with the *loss*, defined as the absolute difference between the best function value identified by the algorithm and the true global minimum: $\text{loss} = \min_{h \in \mathcal{H}_t} |f(h) - f^*_{\text{opt}}|$. Here, $\mathcal{H}_t$ denotes the set of all points evaluated up to trial t, and f^*_{opt} represents the known global optimum value.

ES Configuration on Search Performance The performance of these three ES configurations is visualized in Figure 5.1. Figure 5.1a presents the Area Over the Convergence Curve (AOCC) scores, while Figure 5.1b tracks the loss of the best-found algorithm over time. A clear trend emerges: strategies with larger populations, namely $(8+16)$-ES and $(4, 16)$-ES, significantly outperform the simple (1+1)-ES. The (1+1)-ES not only starts with poorer performance but also shows minimal improvement over the search, suggesting that its limited population size prevents it from effectively escaping local optima. This pattern is mirrored in the loss plot (Figure 5.1b). When comparing the two larger-population strategies, the non-elitist $(4, 16)$-ES demonstrates superior overall performance. It converges to high-quality solutions more rapidly, likely due to a greater emphasis on exploration in the early stages. However, this aggressive exploration may also lead to earlier stagnation. In contrast, the elitist $(8+16)$-ES shows more sustained improvement later in the search (e.g., after 40 evaluations), presumably by carefully refining the best solutions found.



(a) AOCC for different ES configurations.

(b) Loss of best generated algorithm at each evaluation of LLaMEA-BO.

Figure 5.1: LLaMEA-BO's performance on selected BBOB functions in terms of AOCC and generated algorithm loss over time.

Table 5.1: Error rates of the LLM operator across different ES configurations. An "error" is defined as that the algorithm fails during compilation or execution. $P_{total}(err)$ is the total error rate. $P(err|init)$ is the error rate for initial population generation. $P_{cr/mu}(err|err_{pa})$ and $P_{cr/mu}(err|non\_err_{pa})$ are the conditional probabilities of a crossover/mutation operation producing an error, given that the parent(s) were erroneous or non-erroneous, respectively.

Error Type	1+1	8+16	4,16
$P_{total}(err)$	0.26 (132/500)	0.26 (132/500)	0.14 (69/500)
$P(err\|init)$	0.60 (3/5)	0.30 (12/40)	0.20 (4/20)
$P_{cr}(err\|err_{pa})$	-	0.43 (6/14)	0.20 (2/10)
$P_{mu}(err\|err_{pa})$	0.25 (1/4)	0.00 (0/1)	0.50 (4/8)
$P_{cr}(err\|non\_err_{pa})$	-	0.20 (54/273)	0.11 (31/278)
$P_{mu}(err\|non\_err_{pa})$	0.26 (128/491)	0.35 (60/172)	0.15 (28/184)

Error Rate The error rates detailed in Table 5.1 provide further insight into the performance differences. The high initial error rate, $P(err|init)$, for the (1+1)-ES (0.60) partly explains its poor starting performance. Conversely, the best-performing $(4, 16)$-ES configuration exhibits the lowest total error rate ($P_{total}(err) = 0.14$), suggesting a correlation between generation reliability and search efficiency. Although the (1+1)-ES and (8+16)-ES have similar total error rates, the superior performance of the latter highlights how a larger population can effectively mitigate the impact of failed generations. The high probability of generating an error from an erroneous parent, $P(err|err_{pa})$, indicates that the LLM is not proficient at fixing faulty code, which validates our design choice of using a deterministic parent selection strategy (see Section 3.2) that can prioritize valid parents. Finally, a comparison between operators reveals that mutation is more prone to introducing errors from valid parents than crossover (e.g., for (8+16)-ES, $P_{mu}(err|non\_err_{pa}) = 0.35$ vs. $P_{cr}(err|non\_err_{pa}) = 0.20$). This is intuitive, as crossover primarily recombines existing, likely valid, code segments, whereas mutation introduces novel tokens that have a higher chance of breaking the algorithm's syntax or logic.

5.2 Evaluation of Generated Algorithms

To assess the quality and generalization capabilities of the algorithms produced by our evolutionary search, we conduct a rigorous validation study. This involves selecting the top-performing algorithms generated by LLaMEA-BO and benchmarking them against several state-of-the-art (SOTA) optimization methods. The evaluation is performed on two distinct and comprehensive benchmark suites: the BBOB suite, which consists of analytical test functions, to test performance on classical optimization problems; and the Bayesmark suite, which contains practical hyperparameter optimization tasks, to evaluate performance on real-world challenges. This two-pronged approach allows us to thoroughly validate the effectiveness and robustness of the automatically generated algorithms.

Selected Algorithms Table 5.2 provides an overview of the selected algorithms automatically generated by LLaMEA-BO. These algorithms were selected based on their superior overall AOCC performance during the generation phase and collectively exhibit a diverse range of algorithmic characteristics.

Table 5.2: Generated algorithms from LLaMEA-BO. Each algorithm is labeled with its name, short description, population configuration (μ+λ), AOCC(Search) score, and its origin within the evolutionary process (Initialization, Mutation, or Crossover).

Algorithm	Name	ES Configuration	AOCC (Search)	Type
ATRBO	Adaptive Trust Region Bayesian Optimization	(4,16)	0.2091	Initialization
TREvol	Adaptive Trust Region Evolutionary BO with Dynamic Kernel, Acquisition Blending, Adaptive DE, Gradient-Enhanced Trust Region Adjustment, and Variance-Aware Exploration (ATREBO-DKA-BDE-GE-VAE)	(8+16)	0.2138	Crossover
TROpt	Adaptive Trust Region Optimistic Hybrid BO	(4,16)	0.2043	Crossover
TRPareto	Adaptive Evolutionary Pareto Trust Region BO	(8+16)	0.1827	Mutation
ARM	Adaptive Batch Ensemble with Thompson Sampling, Density-Aware Exploration, Uncertainty-Aware Local Search with Adaptive Radius and Momentum Bayesian Optimization (ABETSALSED_ARM_MBO)	(8+16)	0.1813	Crossover

- **ATRBO** employs a Gaussian Process (GP) surrogate with a Lower Confidence Bound (LCB) acquisition function. It dynamically adjusts its trust region and utilizes sequential evaluation with adaptive trust region shrinking, proving particularly effective in lower-dimensional problems.
- **TREvol** synergistically combines a trust region framework with Differential Evolution (DE). Its effectiveness is further enhanced by dynamic kernel selection for the GP, acquisition function blending, gradient-informed trust region adjustments, and variance-aware exploration. It also incorporates mechanisms to prevent premature trust region shrinkage and sets a lower bound for the trust region radius.
- **TROpt** is another trust-region-based algorithm that distinguishes itself by using a K-Nearest Neighbors approach for efficient GP hyperparameter (lengthscale) estimation. It leverages a GP-informed local search and balances exploration and exploitation through a hybrid of Expected Improvement (EI) and Upper Confidence Bound (UCB) acquisition functions, with adaptive adjustments to the trust region size and UCB's exploration parameter.

Other distinct approaches among the top performers include:

- **TRPareto** employs DE within a trust region framework to identify candidate solutions along a Pareto front, simultaneously optimizing for both Expected Improvement (EI) and a diversity metric, thereby encouraging a balanced exploration of the search space.
- **ARM** utilizes an adaptive ensemble of GP models with Thompson Sampling for acquisition. It further incorporates a sophisticated hybrid acquisition strategy that integrates EI, a distance-based exploration term, and Kernel Density Estimation (KDE) to guide the search.

Baselines To contextualize the performance of our LLaMEA-BO generated algorithms, we establish a comparative benchmark against several state-of-the-art (SOTA) optimization methods. These established baselines include the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [16], and prominent Bayesian Optimization approaches such as Heteroscedastic and Evolutionary Bayesian Optimization (HEBO) [6], Trust-Region Bayesian Optimization (TuRBO1) [9], and the standard Vanilla BO implementation from the `BoTorch` library [4].

BBOB

Experimental Setup Our experimental framework is built upon the all 24 analytical test functions of BBOB. To ensure a robust evaluation, each of these functions is subjected to 5 independent repetitions. These repetitions are performed across 3 distinct problem instances, specifically instances $\{4, 5, 6\}$. Furthermore, our investigation spans multiple problem complexities by considering dimensions $d \in \{5, 10, 20, 40\}$. Algorithm performance is meticulously quantified using convergence curves, tracked over a total budget of $10d + 50$ function evaluations for each experimental run.

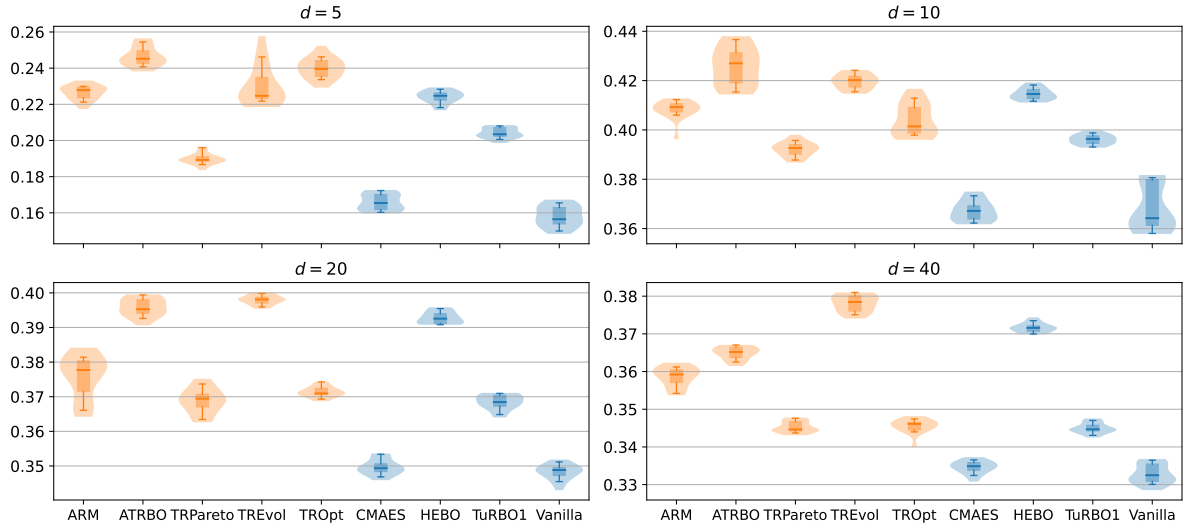


Figure 5.2: Best algorithm evaluation based on AOCC: Violin plots aggregating over 24 functions, 3 instances, 5 runs.

Table 5.3: Average Area Over Convergence Curve over all 24 BBOB functions, 3 instances and 5 random seeds. **Boldface** indicates the best value per dimension setting, underlined text indicated statistical significance using a Paired T-test with $\alpha = 0.05$ of the best generated BO algorithm versus the best BO baseline. Algorithms are split in two groups: the ones generated by LLaMEA-BO on the left, the state-of-the-art baselines on the right.

Dim	ARM	ATRBO	TREvol	TROpt	TRPareto	CMAES	HEBO	TuRBO1	Vanilla
5	0.2264	**0.2464**	0.2304	0.2401	0.1899	0.1659	0.2241	0.2041	0.1581
10	0.4083	**0.4257**	0.4197	0.4043	0.3922	0.3673	0.4146	0.3961	0.3680
20	0.3758	0.3958	**0.3980**	0.3714	0.3688	0.3497	0.3928	0.3685	0.3483
40	0.3586	0.3649	**0.3781**	0.3455	0.3453	0.3346	0.3716	0.3450	0.3328

AOCC Analysis Figure 5.2 illustrates the performance of the LLaMEA-BO-generated algorithms against SOTA baselines, measured by the average Area Over the Convergence Curve (AOCC) on the full BBOB suite. A key finding is the strong generalization capability of the discovered algorithms. Despite being evolved on a subset of only 10 BBOB functions, the algorithms maintain or even improve their AOCC scores when tested on all 24, indicating that the evolutionary search did not overfit to the training problems.

The generated algorithms are highly competitive with established methods. In the 5-dimensional setting, both ATRBO and TROpt outperform all baselines, including the specialized BO methods HEBO and TuRBO1. As dimensionality increases, the diversity of the generated solutions becomes apparent. ATRBO maintains strong performance, while TREvol's performance scales favorably, becoming the top-performing algorithm in 20 and 40 dimensions. Conversely, TROpt's performance degrades in higher dimensions. This variety demonstrates that LLaMEA-BO discovers a portfolio of algorithms with different strengths, suitable for different problem characteristics. Furthermore, the search discovered fundamentally different strategies. While four of the top five algorithms are based on trust regions, ARM—which relies on an ensemble and Thompson sampling—remains stable across all dimensions, showcasing that the LLM operator is capable of producing effective and distinct algorithmic designs beyond a single, dominant paradigm.

Table 5.3 quantitatively corroborates the trends observed in the violin plots. The table confirms ATRBO's dominance in lower dimensions, where it achieves the highest AOCC scores for both $d = 5$ (0.2464) and $d = 10$ (0.4257). A paired t-test ($\alpha = 0.05$) validates that this lead over the best-performing BO baseline, HEBO, is statistically significant. As dimensionality increases, the performance leadership transitions to TREvol, which secures the top rank at $d = 20$ (0.3980) and $d = 40$ (0.3781), surpassing all baselines, including the strong-scaling HEBO.

Convergence Analysis To delve deeper into the generality of the algorithms, we analyze the convergence behavior using the *loss*, defined as the absolute difference between the best-found value and the true global optimum. Figure 5.3 and Figure 5.4 display the average loss over time for the 5D and 40D BBOB functions. Solid lines represent LLaMEA-BO's generated algorithms, while dashed lines denote the SOTA baselines. The plot highlights a key difference in initialization: baselines use a standard fixed-size design ($min(2 \times dim, budget/2)$), whereas our generated algorithms employ their own evolved initialization strategies, marked by circles. An overline on a function's title (e.g., $\overline{\text{F2}}$) indicates it was part of the training set for the evolutionary search (F2, F4, F6, F8, F12, F14, F15, F18, F21 and F23). The number in brackets in each title, e.g., (1), denotes

the BBOB problem group (1: separable, 2: low condition, 3: high condition/unimodal, 4: multimodal/structured, 5: multimodal/weak structure)[17].

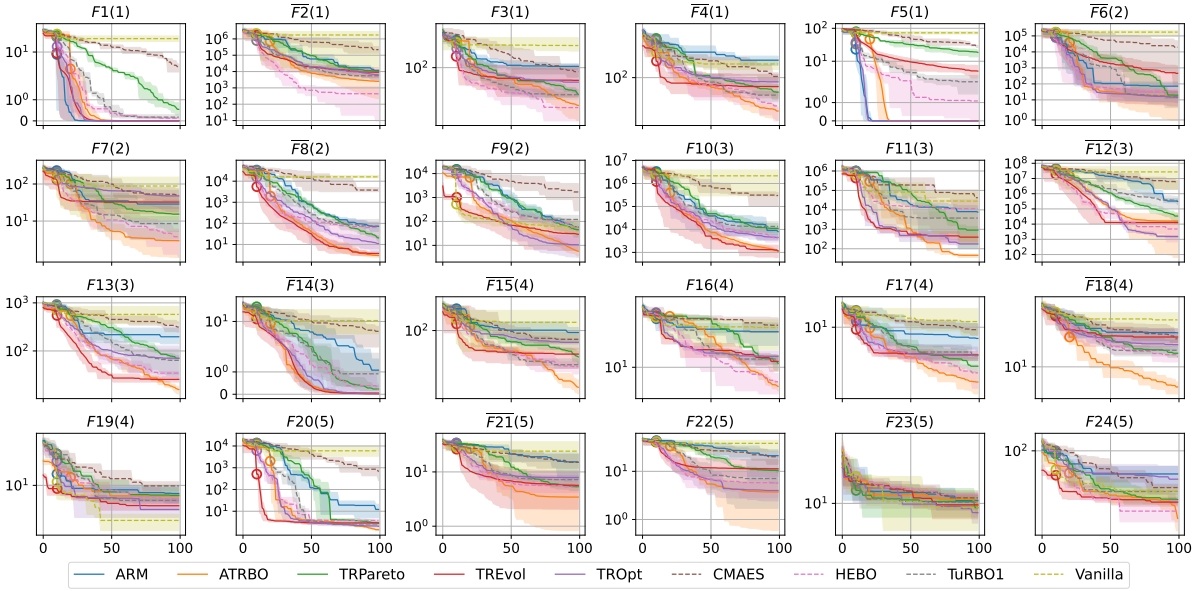


Figure 5.3: Convergence curves on the BBOB suite in 5D. The curves show the loss at each evaluation, averaged over 5 independent runs.

The results from Figure 5.3 reaffirm the strong generalization of the discovered algorithms. There is no evidence of overfitting to the training set. In fact, the algorithms often perform just as well, or better, on unseen functions. For instance, in the high-condition, unimodal problems (Group 3), ATRBO, TREvol, and TROpt demonstrate competitive performance on training functions F12 and F14, and this advantage is consistently maintained on the unseen functions F10, F11, and F13. This indicates that the evolutionary search successfully identified and encoded general principles for tackling this class of problems, rather than merely memorizing solutions.

Individual algorithm strengths become apparent on specific function classes. ATRBO, for instance, confirms its status as a robust all-rounder. Its performance is particularly dominant on multimodal problems with adequate global structure (Group 4), where it outperforms all other methods on four out of the five functions in this category. This suggests its adaptive trust region mechanism is highly effective at navigating complex but structured search spaces to locate the global optimum. Specific adaptations, however, prove highly effective in certain contexts. ARM, for example, performs exceptionally well on F5, a simple separable function. Its ensemble-based Thompson Sampling approach likely allows it to quickly model and exploit the function's straightforward structure. Conversely, most algorithms, both generated and baselines, struggle on F23, a highly rugged function with weak global structure (Group 5). The difficulty of building an accurate surrogate model for such a chaotic landscape leads to minimal improvement after initialization, demonstrating a known challenge for model-based optimization methods.

Figure 5.4 presents the convergence curves for the 40-dimensional BBOB functions, revealing how algorithm performance characteristics evolve in high-dimensional spaces. In general, the generated algorithms maintain their strong performance. As expected, the SOTA baseline HEBO, renowned for its robustness in high dimensions, showcases its strength by regaining the lead on several functions (F3, F7, F17, and F18). Against

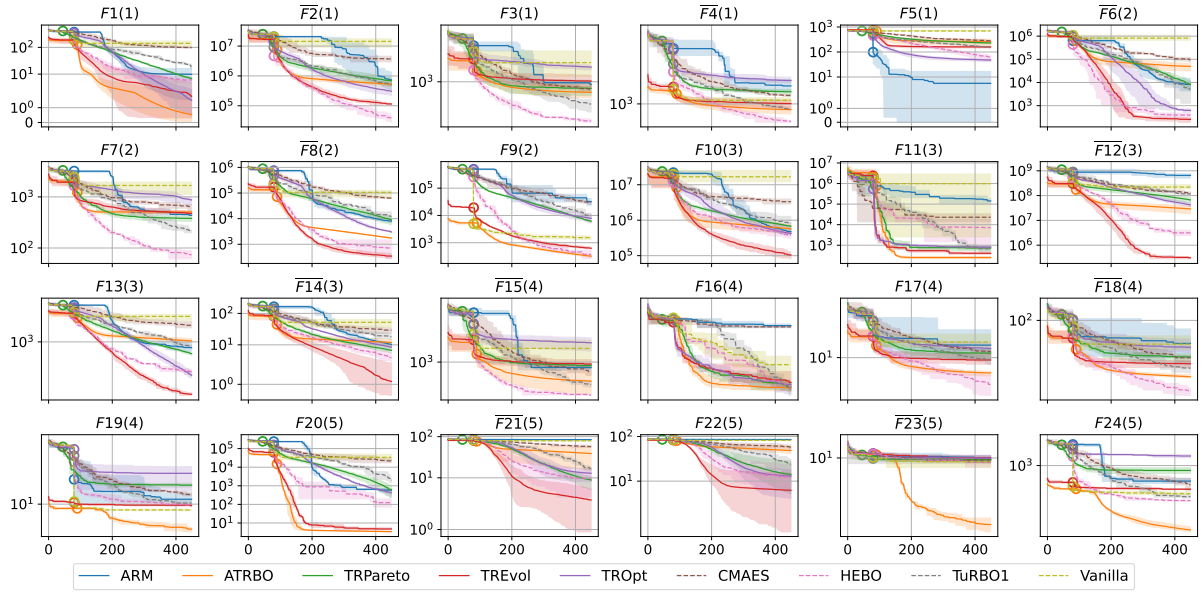


Figure 5.4: Convergence curves on the BBOB suite in 40D. The curves show the loss at each evaluation, averaged over 5 independent runs.

this strong competitor, TREvol emerges as a particularly powerful generated algorithm, surpassing other methods, especially on the high-condition, unimodal problems (Group 3). Its hybrid design, which synergistically combines Differential Evolution with an adaptive trust region, appears to scale more effectively, likely by building more robust surrogate models and exploring the larger search space more efficiently.

In contrast, ATRBO, the top performer in lower dimensions, shows signs of instability. While it remains competitive on multimodal problems with adequate global structure (Group 4) and even dominates on F19, F23, and F24, its performance degrades significantly on functions like F5, F6, F13, and F22. This may be due to its aggressive trust region shrinking, a strategy that, while effective in low-dimensional spaces, can lead to premature convergence in high-dimensional landscapes. Notably, ARM defends its strong performance on the separable F5 function, suggesting its ensemble-based Thompson Sampling is adept at exploiting simple structures regardless of dimensionality.

Another notable observation is the impact of the initial Design of Experiments (DOE). On functions F9 and F19, where the global optimum is known to be near the origin [36], algorithms such as ATRBO, TREvol, and Vanilla BO gain a substantial initial advantage. Their initialization strategies happen to place early samples in this favorable region, an effect that is greatly amplified in high dimensions and provides a pronounced head start.

Bayesmark

Experimental Setup To assess the performance of the generated algorithms on practical, real-world hyperparameter optimization (HPO) challenges, we extend our evaluation to the Bayesmark benchmark suite [53, 34]. This suite comprises 40 distinct HPO tasks, created by pairing five standard machine learning models (AdaBoost, Decision Tree, MLP with SGD, Random Forest, and SVM) with eight different datasets. The datasets include five well-known public benchmarks (breast, digits, wine, iris, diabetes) and three challenging synthetic datasets derived from complex multimodal functions: Rosenbrock, Griewank, and KTablet [56]. For each task, performance is measured using the standard corresponding

metric: accuracy for classification and Mean Squared Error (MSE) for regression.

For each task within the Bayesmark suite, we conducted five independent repetitions to ensure statistical robustness. To establish a fair comparison, all algorithms were initialized with the same five pre-determined samples. The optimization process for each run was constrained to a total budget of 30 function evaluations.

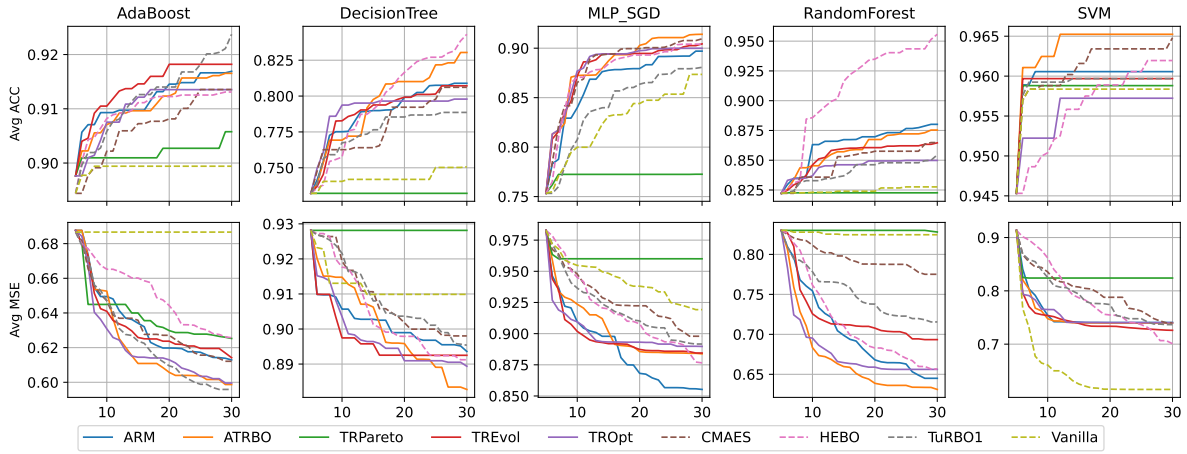


Figure 5.5: Performance comparison on the Bayesmark benchmark. The plots show convergence curves aggregated across all classification tasks (top) and regression tasks (bottom).

Analysis The aggregated results on the Bayesmark suite, shown in Figure 5.5, confirm the strong practical performance and generalization capabilities of the algorithms discovered by LLaMEA-BO. When compared against the SOTA baselines, our generated algorithms remain highly competitive. On a per-dataset basis, our generated algorithms secured the top-performing spot on five of the eight datasets, demonstrating their practical utility. A notable trend among the four best-performing generated algorithms (ATRBO, TREvol, TROpt, and ARM) is their rapid initial improvement; they generally exhibit a very steep learning curve, identifying high-quality solutions within the first 10-20 iterations. This is a crucial advantage in real-world HPO, where evaluation budgets are often tight.

In particular, ATRBO again showcases its remarkable generality, delivering consistently strong performance across both classification and regression tasks, reinforcing its status as a robust all-rounder. Conversely, TRPareto failed to generalize to this new benchmark, showing minimal improvement in most tasks. This disparity in generalization performance reinforces the notion that there is no "silver bullet" algorithm, thereby underscoring the value of our LLM-based method for discovering a diverse portfolio of specialized optimizers. Rather than being limited to finding one-size-fits-all solutions, LLaMEA-BO can be readily adapted to search for specialized algorithms tailored to specific problem classes or domains.

Analysis of ATRBO

Algorithm 2 ATRBO

Require: Budget B, Dimension d

Initialize:
  $X \leftarrow \emptyset$, $y \leftarrow \emptyset$ ▷ Observed points and values
  $n_{init} \leftarrow \min(10d, B/5)$ ▷ Number of initial exploration points
  $x_{best} \leftarrow$ None, $y_{best} \leftarrow \infty$ ▷ Best point and value found
  $r \leftarrow 2.5$ ▷ Trust region radius
  $\rho \leftarrow 0.95$ ▷ Radius adjustment factor
  $\kappa \leftarrow 2.0$ ▷ Exploration-exploitation parameter
  $n_{evals} \leftarrow 0$
  Bounds $\leftarrow [[-5, ..., -5], [5, ..., 5]]$
function SamplePoints(n, c, r) ▷ Sample n points around center c within radius r
    Sample n points from Sobol sequence scaled to $[-1, 1]^d$
    Project points to hypersphere of radius 1
    Scale points to radius r and center at c
    Clip points to within Bounds
    return Scaled points
▷ Initialization
$X_{init} \leftarrow$ SamplePoints(n_{init}, Bounds.mean(), Bounds.range())
$y_{init} \leftarrow$ Evaluate(f, X_{init}), $n_{evals} \leftarrow n_{evals} + X_{init}$
$x_{best} \leftarrow X[\arg\min y]$
▷ Optimization Loop
while $n_{evals} < B$ **do**
  $GP \leftarrow$ Fit(X, y)
  $X_{samples} \leftarrow$ SamplePoints($100d$, x_{best}, r) ▷ Sample points for acquisition function
  $acq \leftarrow$ LCB($X_{samples}$, GP, κ) ▷ Lower Confidence Bound
  $x_{next} \leftarrow X_{samples}[\arg\min acq]$
  $y_{next} \leftarrow$ Evaluate(f, x_{next}), $n_{evals} \leftarrow n_{evals} + 1$
  $x_{best} \leftarrow X[\arg\min y]$
  $y_{best} \leftarrow \min y$
  ▷ Adjust Trust Region
  $r \leftarrow r * \rho$
  $\kappa \leftarrow \kappa/\rho$
  $r \leftarrow \text{clip}(r, 10^{-2}, \max(Bounds.range())/2)$
  $\kappa \leftarrow \text{clip}(\kappa, 0.1, 10.0)$
return y_{best}, x_{best}

Algorithm Overview

ATRBO is a Bayesian optimization algorithm that operates within an adaptive trust region framework (Algorithm 2). The process begins with an initial sampling, where a set of `n_init` points are evaluated. The number of these initial points is determined by a heuristic, $\min(10d, B/5)$, where d is the problem dimension and B is the total evaluation budget. This strategy ensures the initial sampling scales with problem complexity while

being capped at 20% of the budget, providing a robust start to the optimization process.

Following this initialization, the algorithm iteratively builds and refines a Gaussian Process(GP) surrogate model of the objective function. To enhance the robustness of the surrogate, the GP's hyperparameters are tuned via optimization, and multiple restarts are used to mitigate the risk of converging to suboptimal hyperparameter values. To guide the search for the optimum, ATRBO uses the Lower Confidence Bound (LCB) as its acquisition function. Instead of relying on a simple random search to find the next evaluation point, ATRBO employs hyperspherical Sobol sequence to generate candidate points within the current trust region.

Sampling Strategy The sampling strategy in ATRBO is designed to generate a sobol sequence from a hypersphere for both initialization and iterative optimization. As the sobol sequence are designed to cover a hypercube, ATRBO employs a projection to adapt the sequence for a hypersphere.

The sampling procedure is implemented as follows: 1. A scrambled Sobol sequence is generated within a hypercube; 2. These samples are scaled to $[-1, 1]$ and then projected onto the surface of a hypersphere; 3. The projected samples are scaled by the radius to generate candidate points centered by a certain point. (More details can be found in Appendix C.) For initialization, the sampling is centered at the geometric center of the problem's bounds. The radius is set to half the length of the longest axis of the problem's bounding box, ensuring the initial search covers the largest possible spherical area within the defined space. During optimization, the sampling is centered at the current best-found solution, and the radius corresponds to the current trust region size, focusing the search around the most promising area.

The choice of this additional projection from a hypercube to a hypersphere might be based on the assumption that for many real-world problems, the optimal solution is not located at the corners of the search space. Here, corners refer to the space of non-inscribed hyperspheres in the hypercube. Standard Sobol sequence cover a hypercube. However, this poses a challenge, particularly in high dimensions where the vast majority of a hypercube's volume is concentrated in its corners. A standard hypercube-based sampling strategy would therefore has a high chance to explore these corner regions, which are assumed to be less promising. After the projection, the new sequence preserve the low-discrepancy property of the original Sobol sequence but in the hypersphere space.

Comparison with TuRBO's Sampling Strategy The sampling strategy within TuRBO's trust region is executed as follows: 1. Define the the hypercube trust region centered at the current best point. The side lengths of this hypercube are decided by the lengthscales of the GP and a certain length; 2. Generate a Sobol sequence within this hypercube; 3. In lower-dimensional spaces (when the dimension $d < 20$), the points generated by the Sobol sequence are used directly as the candidates; 4. In higher-dimensional spaces, a sparse perturbation strategy is applied. The final candidate points are formed by combining the center point with the Sobol points. For each candidate, a random binary mask determines wheather the value on the certain dimension is taken from the Sobol point or the center point.

The primary distinction between ATRBO and TuRBO lies in the geometry of their sampling trust regions and search behavior.

- Trust Region Geometry: ATRBO uses a hypersphere defined by a single radius, treating all search directions from the center point equally. TuRBO uses a hypercube,

with dimensions scaled by the GP's lengthscales. Consequently, ATRBO's search is agnostic to the GP model, making it potentially more robust. TuRBO, in contrast, explicitly relies on the GP to guide the shape of the search space, which can accelerate convergence but makes it more sensitive to the quality of the learned lengthscales.

- High-Dimensional Search: In high-dimensional spaces, ATRBO performs a more explorative search with Sobol sequences within its hypersphere (Appendix C). TuRBO, however, transitions to a more exploitative search by applying sparse, dimension-wise perturbations.

Adaptive Kappa and Trust Region Radius A key innovation in ATRBO is the dynamic coupling between the trust region radius (r) and the LCB acquisition function's exploration parameter (κ). This coupling is governed by a shrinking factor $\rho \in (0, 1)$. After each iteration, the parameters are updated according to the rules: $r_{t+1} = \rho \cdot r_t$ and $\kappa_{t+1} = \kappa_t / \rho$

This inverse relationship creates a sophisticated exploration-exploitation schedule. As the trust region radius r systematically shrinks, the algorithm focuses its search on a more localized area, signaling a transition towards exploitation. Simultaneously, the exploration parameter κ is increased. This counterintuitively makes the acquisition function more exploratory within the shrinking region. This mechanism acts as a safeguard against premature convergence; by thoroughly exploring the narrowing region of interest, the algorithm is less likely to be misled by local optima or inaccuracies in the surrogate model.

The hyperparameter ρ controls the rate of this transition. A value of ρ close to 1 (e.g., 0.95) results in a slow, conservative convergence, allowing ample time for local exploration at each step. Conversely, a smaller ρ (e.g., 0.80) enforces an aggressive schedule, rapidly shrinking the trust region to accelerate convergence. While potentially faster, this aggressive approach heightens the risk of converging prematurely if the initial search phase fails to identify the basin of attraction containing the global optimum.

Empirical Study

To validate the effectiveness of ATRBO and analyze the impact of its key hyperparameters, we conducted a comprehensive empirical study. The experiments were performed on all 24 BBOB functions in 5 dimensions ($d = 5$), with a total evaluation budget of 100 evaluations per run. To ensure statistical robustness of the results, we executed 5 independent runs for each of the first 3 instances of every function. Performance is measured by loss convergence curves, which plot the best-found objective value against the number of function evaluations. In the following analysis, we compare different ATRBO configurations against a baseline configuration with default parameters set to $\rho = 0.95$, initial $\kappa = 2.0$, and initial trust region radius $r = 2.5$.

Radius Shrinking Factor ρ Figure 5.6 illustrates the performance of ATRBO with different trust region shrinking factors $\rho \in \{0.65, 0.8, 0.95\}$. Theoretically, a smaller ρ enforces a more aggressive optimization schedule, which should accelerate convergence. However, our empirical results reveal a more nuanced reality.

For a majority of the functions, the more conservative configuration ($\rho = 0.95$) achieve superior final performance. The aggressive schedules of $\rho = 0.65$ and $\rho = 0.8$ often lead to premature convergence, where the trust region shrinks too quickly around a suboptimal point before the global optimum's surface is identified.

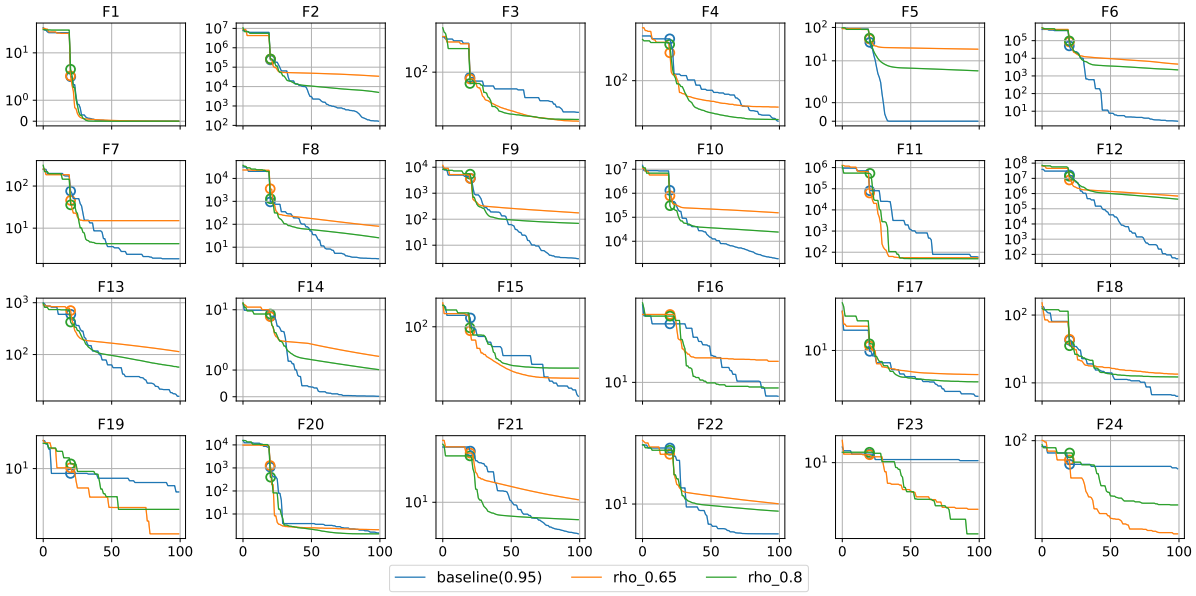


Figure 5.6: Loss convergence curves for ATRBO with different ρ values on the BBOB suite. The curves show the loss at each evaluation, averaged over 5 independent runs.

Interestingly, this trend is reversed on highly multimodal functions such as the F9, F23, and F24. On these functions, the aggressive configurations outperform their more conservative counterparts. This phenomenon can be attributed to the specific landscape properties of these functions, which are characterized by a global optimum surrounded by many close local optima. In such landscapes, the more aggressive shrinking allows ATRBO to commit to a reasonably good local optimum, thereby preventing it from wasting resources on exploring the function's complex global structure.

Sampling Projection Figure 5.7 compares the performance of ATRBO with and without the projection strategy. The projection strategy, which maps candidate points onto the surface of a unit hypersphere before scaling, is designed to ensure uniform sampling across the search space. The results indicate that the projection strategy significantly enhances performance, both during initialization and in subsequent optimization stages.

The projected sampling yields a better initial set of evaluated points, which can be attributed to two factors. First, the strategy's primary goal is achieved: it ensures a more uniform volumetric distribution of samples, providing a comprehensive initial survey of the search space. Second, many BBOB functions have their global optimum near the origin; the hypersphere-based sampling is coincidentally biased towards this promising region, unlike standard hypercube sampling where volume is concentrated in the corners, far from the center.

Furthermore, the sustained performance advantage in later iterations suggests a deeper compatibility. As ATRBO's search is confined to a shrinking spherical trust region, the hyperspherical projection sampling is inherently more efficient at exploring this region to minimize the acquisition function. This alignment between the sampling geometry and the trust region shape allows the algorithm to more effectively locate promising candidates in each step, leading to faster and more reliable convergence.

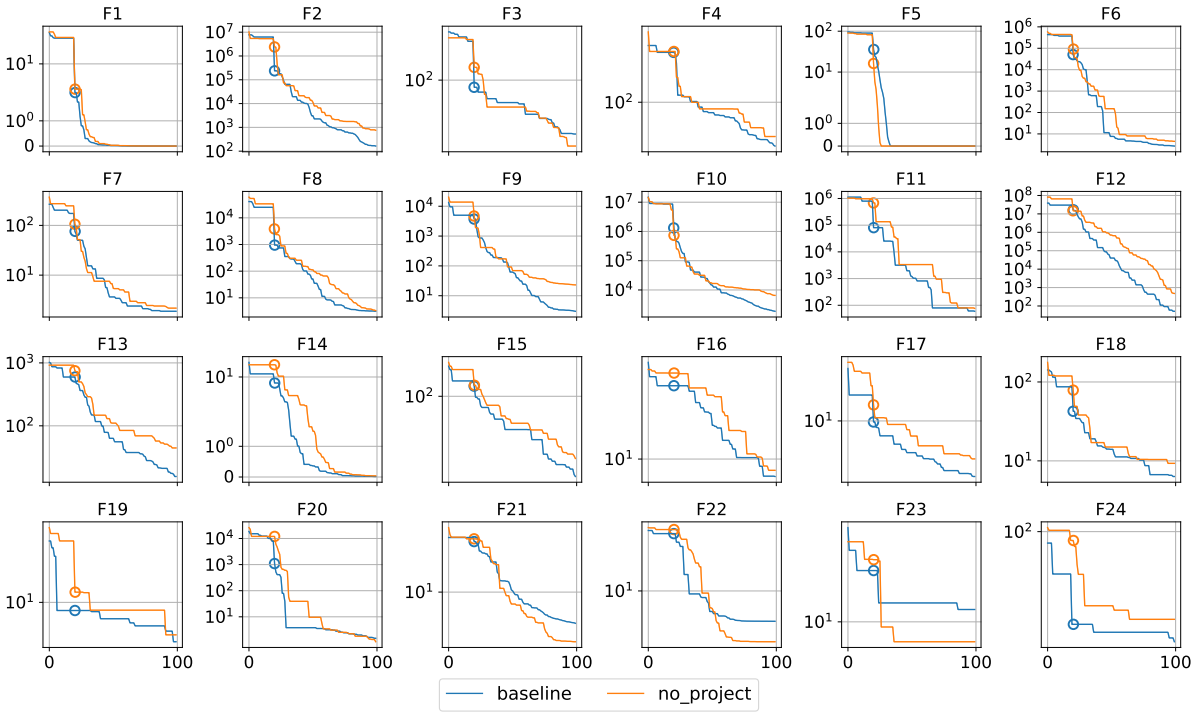


Figure 5.7: Loss convergence curves for ATRBO with and without projection strategies on the BBOB suite. The curves show the loss at each evaluation, averaged over 5 independent runs.

Chapter 6

Conclusion

In this paper, we introduced LLaMEA-BO, a framework that employs a Large Language Model as a core operator within an evolutionary strategy to automate the design of Bayesian Optimization algorithms. By integrating the LLM into a closed evolutionary loop, LLaMEA-BO iteratively generates, evaluates, and refines complete BO algorithms. The evolutionary search is guided by performance feedback from a diverse suite of benchmark functions from the BBOB suite, steering the LLM towards more effective solutions. Our methodology centered on tackling two primary challenges inherent in LLM-based evolution: ensuring the feasibility of generated code and maintaining population diversity against the LLM's intrinsic biases. These were addressed through a combination of a structured evolutionary framework and meticulous prompt engineering.

Comprehensive ablation studies demonstrated the impact of key evolutionary and LLM parameters, revealing that higher selection pressure and a balanced crossover rate enhance the search process, while lower LLM temperature improves stability and performance. The final algorithms generated by LLaMEA-BO achieve performance comparable to, and in some cases exceeding, established state-of-the-art BO baselines across various tasks and dimensionalities. While not intended to produce a single universally dominant algorithm, this work validates LLaMEA-BO as a robust and effective generator for continuous optimization algorithms. It underscores the potential of LLMs to evolve from mere code synthesizers into powerful co-design partners for complex algorithmic discovery.

Limitations

While LLaMEA-BO demonstrates strong performance in generating competitive algorithms, it is important to acknowledge several limitations inherent in its current design and experimental setup.

- Our work is primarily empirical, and we have not formally analyzed the theoretical properties of using an LLM as an evolutionary operator. This introduces a disconnect from the traditional ES theory, as key concepts like the self-adaptation of strategy parameters (e.g., mutation strengths) lack a direct analog in a prompt-based generative process. Furthermore, we have not formally investigated the novelty of the solutions generated by the LLM. Consequently, it is unclear whether the LLM is merely interpolating between known algorithmic concepts from its training data or genuinely extrapolating to discover novel solutions.
- The evolutionary search remains susceptible to premature convergence. This susceptibility stems from multiple factors: the LLM's intrinsic bias towards common

or canonical solutions present in its training data, the potential for prompts to inadvertently steer the population towards a homogeneous set of solutions, and the possibility that certain ES configurations (e.g., high selection pressure) may too aggressively eliminate novel but initially uncompetitive individuals. A systematic study is needed to disentangle these factors and develop more robust diversity-preservation mechanisms.

- The significant error rate in code generation, as observed in our studies, represents a substantial computational inefficiency. Each failed generation is a wasted evaluation in the evolutionary budget, which not only slows down the search but may also disrupt selection pressure by introducing zero-fitness individuals, thereby undermining the full potential of the evolutionary strategy.

- The reliance on a fixed code template and specific prompt structures, while necessary for ensuring feasibility, may inadvertently constrain the LLM's creative potential. This structured approach could prevent the discovery of radically different algorithmic paradigms that do not fit the predefined BO-like template, effectively limiting the search to a specific family of algorithms.

- Our experiments were conducted using a single LLM backend (`gemini-2.0-flash`). A comprehensive evaluation across a diverse range of models, particularly those with strong reasoning capabilities, was not performed. Therefore, the generalizability of our findings to other LLM architectures remains an open question.

Future Work

The identified limitations naturally pave the way for several promising avenues of future research.

- **Theoretical Foundations:**
  - **Integrating LLM Operators into Classical ES Theory:** A foundational research direction is to develop a theoretical framework that connects our LLM-based approach with classical ES principles. This involves creating formal analogies for key ES concepts, such as strategy parameter adaptation. For instance, research could investigate how LLM parameters like temperature or targeted prompt modifications can be conceptualized and controlled as analogs to mutation strength or step-size adaptation, thereby allowing for a more principled and self-adaptive evolutionary process.
  - **Novelty of Generated Solutions:** Another important area for future research is to study whether evolutionary pressure can guide the LLM to produce genuinely novel solutions. This could be formally analyzed by examining the token-level log-probabilities of generated code. We hypothesize that truly novel solutions would manifest as low-probability sequences, indicating a departure from the common patterns in the LLM's learned distribution. Such an analysis is essential to determine if the LLM is functioning as a partner in innovation or merely as a sophisticated retrieval system.

- **Enhancing Search Dynamics and Efficiency:** To combat premature convergence and computational inefficiency, research could explore more advanced diversity-

maintenance techniques. Additionally, investigating methods to reduce code generation errors could significantly improve the search budget's efficiency.

- **Flexible Algorithmic Representations:** To unlock the LLM's full creative potential, future iterations could move beyond fixed code templates. Research into more flexible instructions, such as allowing the LLM to define its own modular components, could enable the discovery of truly novel and unconventional optimization paradigms.
- **Generalization Across LLM Architectures:** A comprehensive comparative study across a diverse suite of LLMs (including both proprietary and open-source models) is essential. Such a study would test the generalizability of the LLaMEA-BO framework and provide insights into how different model architectures, training data, and fine-tuning methods influence the outcomes of the evolutionary design process.

Bibliography

[1] Gemini flash 2.0: A next-generation coding assistant. `https://ai.google.dev/gemini-api/docs/models#gemini-2.0-flash`, 2024.

[2] Virginia Aglietti, Ira Ktena, Jessica Schrouff, Eleni Sgouritsa, Francisco J. R. Ruiz, Alan Malek, Alexis Bellot, and Silvia Chiappa. Funbo: Discovering acquisition functions for bayesian optimization with funsearch, 2024.

[3] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[4] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*, 2020.

[5] Clément Chevalier and David Ginsbourger. Fast computation of the multi-points expected improvement with applications in batch selection. In *International conference on learning and intelligent optimization*, pages 59–69. Springer, 2013.

[6] Alexander I. Cowen-Rivers, Wenlong Lyu, Rasul Tutunov, Zhi Wang, Antoine Grosnit, Ryan Rhys Griffiths, Alexandre Max Maraval, Hao Jianye, Jun Wang, Jan Peters, and Haitham Bou Ammar. HEBO Pushing The Limits of Sample-Efficient Hyperparameter Optimisation, May 2022.

[7] Jacob de Nobel, Furong Ye, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. Iohexperimenter: Benchmarking platform for iterative optimization heuristics. *Evol. Comput.*, 32(3):205–210, 2024.

[8] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[9] David Eriksson, Michael Pearce, Jacob R. Gardner, Ryan Turner, and Matthias Poloczek. Scalable Global Optimization via Local Bayesian Optimization, February 2020.

[10] Alexander I. J. Forrester, András Sóbester, and Andy J. Keane. *Engineering Design via Surrogate Modelling - A Practical Guide*. John Wiley & Sons Ltd., 2008.

[11] Peter Frazier, Warren Powell, and Savas Dayanik. The knowledge-gradient policy for correlated normal beliefs. *INFORMS journal on Computing*, 21(4):599–613, 2009.

[12] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

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

[14] Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. Batch bayesian optimization via local penalization. In *Artificial intelligence and statistics*, pages 648–657. PMLR, 2016.

[15] Stewart Greenhill, Santu Rana, Sunil Gupta, Pratibha Vellanki, and Svetha Venkatesh. Bayesian optimization for adaptive experimental design: A review. *IEEE access*, 8:13937–13948, 2020.

[16] Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[17] Nikolaus Hansen, Anne Auger, Raymond Ros, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 36:114–144, 2021.

[18] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions. Technical Report RR6829, INRIA, 2009.

[19] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[20] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model, 2023.

[21] Philipp Hennig and Christian J Schuler. Entropy search for information-efficient global optimization. *Journal of Machine Learning Research*, 13(1):1809–1837, 2012.

[22] José Miguel Hernández-Lobato, Matthew W Hoffman, and Zoubin Ghahramani. Predictive entropy search for efficient global optimization of black-box functions. *Advances in neural information processing systems*, 27, 2014.

[23] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024.

[24] Yaojie Hu, Qiang Zhou, Qihong Chen, Xiaopeng Li, Linbo Liu, Dejiao Zhang, Amit Kachroo, Talha Oz, and Omer Tripp. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. *arXiv preprint arXiv:2501.17167*, 2025.

[25] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024.

[26] Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving, 2024.

[27] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, March 2023.

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

[29] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*, 2023.

[30] J. Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. 2022.

[31] Yucen Lily Li, Tim GJ Rudner, and Andrew Gordon Wilson. A study of bayesian neural network surrogates for bayesian optimization. *arXiv preprint arXiv:2305.20028*, 2023.

[32] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. 2024.

[33] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.

[34] Tennison Liu. Tennisonliu/LLAMBO, May 2025. MIT license.

[35] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large Language Models to Enhance Bayesian Optimization, March 2024.

[36] Fu Xing Long, Diederick Vermetten, Bas van Stein, and Anna V. Kononova. Bbob instance analysis: Landscape properties and algorithm performance across problem instances, 2022.

[37] Manuel López-Ibáñez and Thomas Stützle. Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582, 2014.

[38] Aman Madaan, Niket Tandon, Peter R. Clark, Yiming Ma, and Gurusharan Havasi. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. Technical report, Google DeepMind, 05 2025.

[40] Una-May O'Reilly and Erik Hemberg. Using large language models for evolutionary search. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 973–983, 2024.

[41] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

[42] Mayk Caldas Ramos, Shane S. Michtavy, Marc D. Porosoff, and Andrew D. White. Bayesian Optimization of Catalysts With In-context Learning, April 2023.

[43] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[44] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 01 2024.

[45] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, H Han, Sevien Schulhoff, et al. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 5, 2024.

[46] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[47] Noah Shinn, Simon Labash, and Ashwin Gopinath. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

[48] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[49] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180. PMLR, 2015.

[50] Niranjan Srinivas, Andreas Krause, Sham M Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. *arXiv preprint arXiv:0912.3995*, 2009.

[51] Niki van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2):331–345, 2025.

[52] Yujin Tang, Robert Tjarko Lange, and Yingtao Tian. Large language models as evolution strategies, 2024.

[53] Uber. Uber/bayesmark. Uber Open Source, April 2025. Apache-2.0 license.

[54] Niki van Stein, Diederick Vermetten, and Thomas Bäck. In-the-loop hyper-parameter optimization for llm-based automated design of heuristics. *ACM Trans. Evol. Learn. Optim.*, April 2025. Just Accepted.

[55] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Denny Chi, Sharan Narang, Aakanksha Mishra, and Yi Zhou. Self-consistency improves chain of thought reasoning in large language models. *arXiv preprint arXiv:2203.11171*, 2022.

[56] Shuhei Watanabe. Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance, May 2023.

[57] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xiong, Quoc V Le Huang, Denny Chi, Huong Le, Yifen Mokotoff, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[58] James Wilson, Frank Hutter, and Marc Deisenroth. Maximizing acquisition functions for bayesian optimization. *Advances in neural information processing systems*, 31, 2018.

[59] Xingyu Wu, Sheng-hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap. *arXiv preprint arXiv:2401.10034*, 2024.

[60] Shunyu Yao, Dianjie Cui, Haizuyu Li, Yuanhan Li, Ziwei Hao, Enduo Ma, Jinwen Du, Lu Ping, Leyang Ding, Xinyu Wen, et al. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[61] Yiming Yao, Fei Liu, Ji Cheng, and Qingfu Zhang. Evolve Cost-aware Acquisition Functions Using Large Language Models, June 2024.

[62] Yuxuan Yin, Yu Wang, Boxun Xu, and Peng Li. ADO-LLM: Analog Design Bayesian Optimization with In-Context Learning of Large Language Models. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, October 2024.

[63] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, December 1997.

Appendix A

AOCC Performance Metric Definition

To evaluate and compare algorithm performance, we employ the Area Over the Convergence Curve (AOCC) metric. AOCC is calculated as the area under the convergence curve when precision is plotted on a logarithmic scale. This formulation offers two key advantages. First, it naturally rewards algorithms that make rapid progress towards the optimum. Second, it uses tight lower and upper bounds (lb/ub) to cap precision values, making the metric robust to pathological outliers. After aggregating scores across an entire benchmark suite, AOCC yields a single, well-behaved scalar fitness. This final score enables a clear and unambiguous ranking of algorithms.

AOCC Definition. The calculation begins with $\mathbf{y}_{a,f}$, the sequence of *best-seen* objective function values recorded by an algorithm a on a function instance f after each evaluation. Following [37], we first convert these objective values to log-precisions. The log-precision after the i-th evaluation is defined as $y_i = \log\big(f(x_i) - f^\star\big)$, where $f^\star$ is the global optimum.

These precision values are then normalized according to the formula:

$$AOCC(\mathbf{y}_{a,f}) = \frac{1}{B}\sum_{i=1}^{B}\Big(1 - \frac{\min\big(\max(y_i, lb), ub\big) - lb}{ub - lb}\Big)\,. \tag{A.1}$$

This process involves first clamping each y_i to lie within the interval $[lb, ub]$ and then scaling it to $[0, 1]$. We used bounds of $lb = 10^{-8}$ and $ub = 10^4$ for 5D BBOB functions, increasing the upper bound to 10^9 for higher-dimensional (10D, 20D, 40D) problems. This entire calculation can be interpreted as finding the area under the empirical CDF of the algorithm's performance across all possible precision targets.

Aggregation across functions, instances and runs. For each algorithm a we compute the mean $AOCC$ over the N_f functions and N_j instances per function:

$$AOCC(a) = \frac{1}{N_i \cdot N_f}\sum_{j=1}^{N_f}\sum_{k=1}^{N_i} AOCC\left(\mathbf{y}_{a,f_{jk}}\right)\,. \tag{A.2}$$

We then average over N_s independent repetitions to obtain the scalar fitness feedback to the LLM:

$$f(a) = \frac{1}{N_s}\sum_{s=1}^{N_s} AOCC^{(s)}(a)\,. \tag{A.3}$$

Any runtime, compilation or import error during evaluation is assigned an $AOCC$ of zero, propagating a minimum fitness to the evolutionary loop.

Appendix B

Code Template

Listing 1 Code template used in the initialisation prompt of LLaMEA-BO.

```python
from collections.abc import Callable
from scipy.stats import qmc #If you are using QMC sampling.
from scipy.stats import norm
import numpy as np
class <AlgorithmName>:
    def __init__(self, budget:int, dim:int):
        self.budget = budget
        self.dim = dim
        # bounds has shape (2,<dimension>), bounds[0]: lower bound, bounds[1]: upper bound
        self.bounds = np.array([[-5.0]*dim, [5.0]*dim])
        # X has shape (n_points, n_dims), y has shape (n_points, 1)
        self.X: np.ndarray = None
        self.y: np.ndarray = None
        self.n_evals = 0 # the number of function evaluations
        self.n_init = <your_strategy>

        # Do not add any other arguments without a default value

    def _sample_points(self, n_points):
        # sample points
        # return array of shape (n_points, n_dims)

    def _fit_model(self, X, y):
        # Fit and tune surrogate model
        # return the model
        # Do not change the function signature

    def _acquisition_function(self, X):
        # Implement acquisition function
        # calculate the acquisition function value for each point in X
        # return array of shape (n_points, 1)

    def _select_next_points(self, batch_size):
        # Select the next points to evaluate
        # Use a selection strategy to optimize/leverage the acquisition function
        # The selection strategy can be any heuristic/evolutionary/mathematical/hybrid methods.
        # Your decision should consider the problem characteristics, acquisition function, and the computational efficiency.
        # return array of shape (batch_size, n_dims)

    def _evaluate_points(self, func, X):
        # Evaluate the points in X
        # func: takes array of shape (n_dims,) and returns np.float64.
        # return array of shape (n_points, 1)

        self.n_evals += len(X)

    def _update_eval_points(self, new_X, new_y):
        # Update self.X and self.y
        # Do not change the function signature

    def __call__(self, func:Callable[[np.ndarray], np.float64]) -> tuple[np.float64, np.array]:
        # Main minimize optimization loop
        # func: takes array of shape (n_dims,) and returns np.float64.
        # !!! Do not call func directly. Use _evaluate_points instead and be aware of the budget when calling it. !!!
        # Return a tuple (best_y, best_x)

        self._evaluate_points()
        self._update_eval_points()
        while self.n_evals < budget:
            # Optimization

            # select points by acquisition function
            self._evaluate_points()
            self._update_eval_points()

        return best_y, best_x
```

Appendix C

Sampling in ATRBO

Given an initial point $\mathbf{x} \in [-1, 1]^d \setminus \{\mathbf{0}\}$ used to define a direction, a uniform random number $u \sim U(0, 1)$, and a maximum radius R, the projected point $\mathbf{p}$ is calculated as:

$$\mathbf{p} = \underbrace{\left(R \cdot u^{1/d}\right)}_{\text{Magnitude}} \cdot \underbrace{\left(\frac{\mathbf{x}}{\|\mathbf{x}\|_2}\right)}_{\text{Direction}}$$

In the context of a trust region, the vector $\mathbf{p}$ represents a random step sampled in a hypersphere, which is then scaled by the trust region's radius.

$$\hat{\mathbf{p}} = \mathbf{p} * \mathit{radius} + \mathit{center}$$

where $\hat{\mathbf{p}}$ is the candidate in the search space. It is generated by scaling the step vector $\mathbf{p}$ by the current *radius* and translating it by the *center*, which represents the best solution found so far.

Notably, An important property of this sampling method is that for a fixed $u \in (0, 1)$, the magnitude factor $u^{1/d}$ increases as the dimensionality d increases. This implies that the algorithm becomes more explorative in high-dimensional spaces.

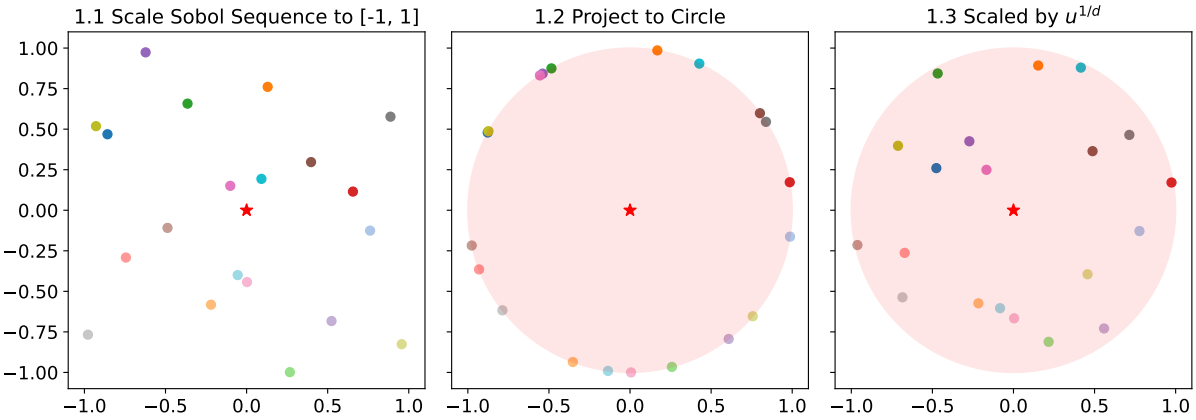


Figure C.1: Projection of 20 distinct points from a Sobol sequence onto a hypersphere, with each point denoted by a unique color.

Visualization of the Sampling Process The sampling process is visualized in a two-dimensional space ($d = 2$) in the subsequent figures. These illustrations depict the projection of points from a Sobol sequence onto a hypersphere, contrasting it with a standard uniform sampling approach. In all visualizations, a red star marks the origin, and the term u represents a uniform random variable sampled from $[0, 1]$.

Figure C.1 illustrates the projection process for 20 distinct points, each identified by a unique color. While the initial Sobol points set the direction vector from the origin, the magnitude of the final projected point is scaled by a factor derived from a uniform random variable, u. Consequently, a point's final distance from the origin is randomized, meaning it is not systematically pushed towards the periphery or pulled towards the center.

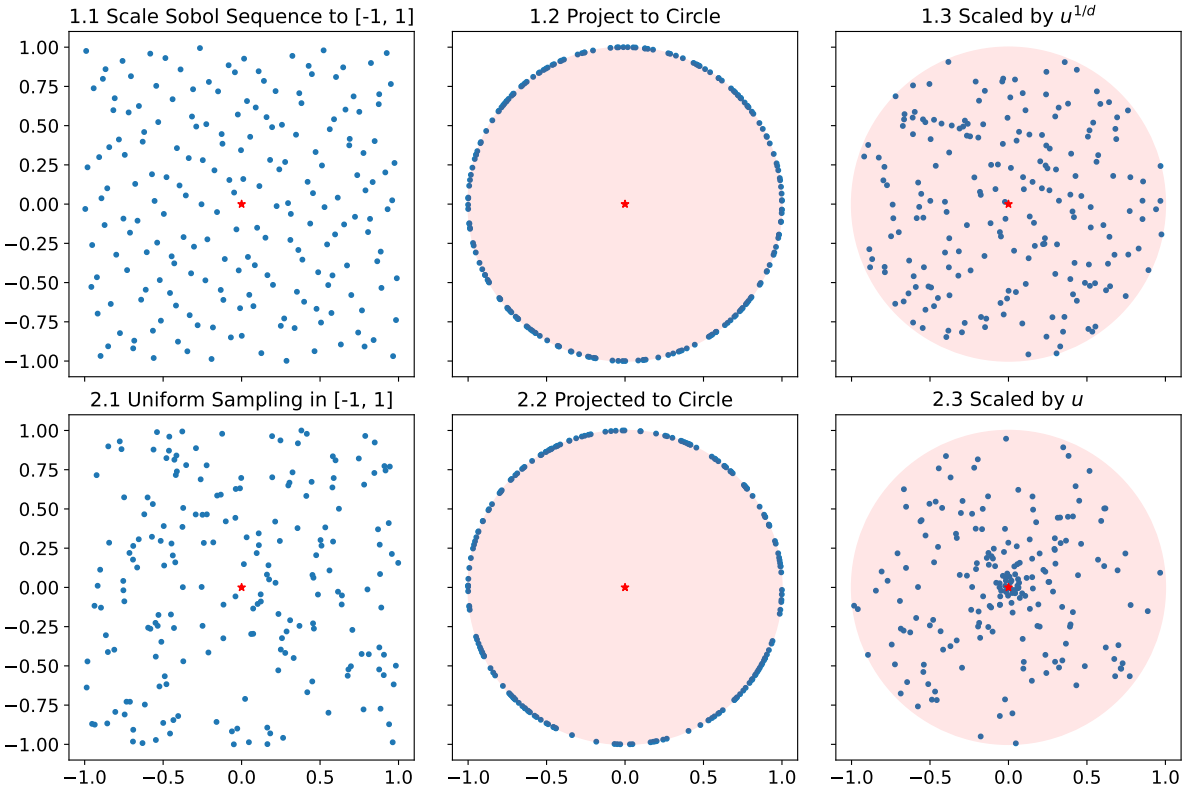


Figure C.2: Projection of 200 points from Sobol sequence to the hypersphere. The top row shows the projection of Sobol points, while the bottom row shows the projection of uniform points.

Figure C.2 extends this visualization to 200 points to better illustrate the distributional properties of the sampling methods. The top row of subplots shows projections where the direction is determined by Sobol points, while the bottom row uses standard uniform sampling for direction. The middle subplots reveal that both methods provide comprehensive angular coverage, suggesting that the use of Sobol sequences for generating directions may not offer a significant advantage. The rightmost subplots highlight the effect of the magnitude scaling: using the dimension-aware factor $u^{1/d}$ results in points that are distributed more uniformly within the hypersphere's volume. In contrast, scaling by a simple uniform variable u leads to a distribution heavily concentrated near the origin.

Derivation of The Magnitude Let point $\mathbf{p} \in \mathbb{R}^d$ is drawn from a uniform probability distribution over the volume of a d-dimensional hypersphere of radius R centered at the origin.

The probability density function $f(\mathbf{p})$ is defined as:

$$f(\mathbf{p}) = \begin{cases} \frac{1}{V_d(R)} & \text{if } \|\mathbf{p}\| \leq R \\ 0 & \text{otherwise} \end{cases}$$

where $V_d(R)$ is the volume of the d-hypersphere of radius R.

Let $F_r(\hat{r})$ be the Cumulative Distribution Function (CDF) of the radius $r = \|\mathbf{p}\|_2$. This function gives the probability that the magnitude of our sampled point is less than or equal to some value $\hat{r}$:

$$\begin{aligned} F_r(\hat{r}) &= P(r \leq \hat{r}) \\ &= \frac{V_d(\hat{r})}{V_d(R)} \\ &= \frac{C_d \hat{r}^d}{C_d R^d} \\ &= \left(\frac{\hat{r}}{R}\right)^d \end{aligned}$$

where C_d is a dimension-dependent constant. (e.g., for $d = 2$, $C_2 = \pi$; for $d = 3$, $C_3 = 4/3\pi$)

Let u be a random variable sampled from a uniform distribution on the interval $[0, 1]$, i.e., $u \sim U(0, 1)$. We set $u = F_r(\hat{r})$ and solve for $\hat{r}$:

$$u = \left(\frac{\hat{r}}{R}\right)^d \implies u^{1/d} = \frac{\hat{r}}{R} \implies \hat{r} = R \cdot u^{1/d}$$