

LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics

Niki van Stein, *Member, IEEE*, and Thomas Bäck, *Fellow, IEEE*

Abstract—Large Language Models (LLMs) such as GPT-4 have demonstrated their ability to understand natural language and generate complex code snippets. This paper introduces a novel Large Language Model Evolutionary Algorithm (LLaMEA) framework, leveraging GPT models for the automated generation and refinement of algorithms. Given a set of criteria and a task definition (the search space), LLaMEA iteratively generates, mutates and selects algorithms based on performance metrics and feedback from runtime evaluations. This framework offers a unique approach to generating optimized algorithms without requiring extensive prior expertise. We show how this framework can be used to generate novel black-box metaheuristic optimization algorithms automatically. LLaMEA generates multiple algorithms that outperform state-of-the-art optimization algorithms (Covariance Matrix Adaptation Evolution Strategy and Differential Evolution) on the five dimensional black box optimization benchmark (BBOB). The results demonstrate the feasibility of the framework and identify future directions for automated generation and optimization of algorithms via LLMs.

Index Terms—Large Language Models, Evolutionary Computation, Metaheuristics, Optimization, Automated Code Generation.

I. INTRODUCTION

FOR decades, algorithms for finding near-optimal solution candidates to global optimization problems have been developed based on inspirations gleaned from nature. Famous examples include the use of biological evolution in the field of Evolutionary Computation [1], [2] (with examples such as Genetic Algorithms and Evolutionary Strategies), the swarming behavior of bird-like objects in Particle Swarm Optimization [3], or the foraging behavior of ants in Ant Colony Optimization [4]. For the highly advanced variants of such algorithms, impressive results have been reported for solving real-world optimization problems [5]–[7].

However, the number of metaphor-inspired algorithms that have been proposed by researchers, often as relatively small variations of existing methods or claimed as a completely new branch, is very large (e.g., [8], and [9] mentions more than 500 methods). A systematic empirical benchmarking of such methods against the state-of-the-art, as exemplified in [10], is typically not performed.

Realizing that the laborious, expert driven approach for improving existing and inventing new algorithms has be-

Manuscript received May 31, 2024 (*Corresponding author: Niki van Stein*). Niki van Stein (n.van.stein@liacs.leidenuniv.nl), Niki van Stein and Thomas Bäck are with the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, The Netherlands.

come quite inefficient, researchers have recently started to develop modular frameworks for arbitrarily combining components (*modules*) from algorithm classes into new variants - thereby creating combinatorial algorithm design spaces in which thousands to millions of algorithms can be generated, benchmarked, and optimized. Examples include the modular Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [11], [12], modular Differential Evolution [13], Particle Swarm Optimization [14], and a recent overview that summarizes all such approaches towards automated design [15].

Although they often provide algorithm design spaces of millions of potential module combinations plus their hyper-parameter search spaces, modular frameworks also need to be created by experts in the field, by carefully selecting the included modules and providing the full infrastructure for configuring and searching through such algorithm spaces. Moreover, they are limited to the design space created by the choices that have been made by the experts in selecting modules for inclusion.

In this paper, we propose to overcome such limitations by using Large Language Models (LLMs) for automatically and iteratively evolving and optimizing the program code of such metaphor-based optimization algorithms. To test the approach, we use a specific optimization benchmarking tool to automatically evaluate the quality of the generated optimization algorithm and to provide a corresponding feedback to the LLM. In particular, the contributions of this work are as follows:

- We present the LLM-Evolutionary Algorithm (LLaMEA), an algorithm that uses an LLM to automatically generate and optimize high-quality metaphor-based optimization algorithms from scratch.
- We couple LLaMEA with a benchmarking tool, IOHexperimenter [16], to automatically evaluate the quality of the generated optimization algorithm, for providing feedback to the LLM.
- We demonstrate that LLaMEA can generate a new metaphor-based optimization algorithm that successfully competes with the state-of-the-art algorithms on a standard set of benchmark test problems.

In the remainder of the paper, we first discuss the state-of-the-art in LLM-based algorithm generation for direct, global optimization (Section II). We then introduce the newly proposed LLaMEA in Section III, and our experimental setup in Section IV. The experimental results are presented and

discussed in Section V. We also analyze the best algorithms found in Section VI, as we are striving to understand the resulting algorithms proposed by LLaMEA, and why they might perform so well. The conclusions are presented in Section VII.

II. RELATED WORK

The integration of Large Language Models (LLMs) into the optimization domain has recently received significant attention, resulting in various innovative methodologies.

Using optimization algorithms such as Genetic Algorithms for optimizing prompts (EVOPROMPT) [17] has shown impressive results in outperforming human-engineered prompts. This idea of directly evolving the prompts by an Evolutionary Algorithm has also recently been used to demonstrate an automated engineering design optimization loop for finding 3D car shapes with optimal aerodynamic performance [18].

These approaches are limited, however, as they are usually based on a fixed prompt template and a finite set of strings as building blocks and need evolutionary operators that work on such patterns and strings. To overcome such limitations, an LLM can be used to generate the prompts for an LLM, by asking the prompting LLM to propose and iteratively improve a prompt, based on the (quantifiable) feedback concerning the quality of the answer generated by the prompted LLM. The *Automated Prompt Engineer (APE)* [19] implements this loop by employing an iterative Monte Carlo search approach for prompt generation, in which the LLM is asked to generate new prompts similar to those with high scores. The authors illustrate the benefits of APE on a large set of instruction induction tasks [20] and by improving a "Chain-of-Thought" prompt ("Let's think step by step") from [21].

In the *Algorithm Evolution using Large Language Model (AEL)* [22] approach, ideas from Evolutionary Algorithms are used explicitly to design novel optimization algorithms. AEL treats each algorithm as a solution candidate in a population, evolving them by asking the LLM to perform crossover and mutation operators on the algorithms generated by the LLM. This approach has shown superior performance over traditional heuristic methods and domain-specific models in solving small instances of the Traveling Salesperson Problem (TSP).

The direct application of LLMs as Evolution Strategies for optimization tasks represents another innovative approach, named EvoLLM [23]. In this proposal, the LLM generates the means of uni-variate normal distributions which are then used as sampling distributions for proposing solution vectors for black-box optimization tasks. To guide the LLM towards improvements, the best m solution vectors from the best k generations, each, are provided to the LLM, and it is prompted to generate the new mean values of the sampling distribution. A method such as EvoLLM leverages the LLM to perform the sampling based on the information summarized above, and it shows reasonable performance on a subset of eight of the BBOB set of benchmarking functions for continuous optimization [24] for one 5-dimensional instance of each problem. EvoLLM uses the idea of *in-context learning* [25], [26], which works by providing a set of examples and their

corresponding scores to the LLM, which is then prompted to generate a score for a newly presented example.

The concept of recursive self-improvement, where systems iteratively improve their performance by refining their own processes, has been a focal point in AI research. The Self-Taught Optimizer (STOP) [27] framework exemplifies this by using LLMs to recursively enhance code generation. STOP begins with a seed improver that uses an LLM to generate and evaluate candidate solutions, iteratively refining its own scaffolding to improve performance across various tasks. This framework highlights the potential of LLMs to act as meta-optimizers, capable of self-improvement without altering their underlying model parameters.

When it comes to leveraging LLMs for the design of metaheuristics, results from a manual prompting approach with GPT-4 for selecting, explaining and combining components of such algorithms have illustrated the capabilities of LLMs for generating such algorithms [28]. Automated generation and evaluation of their performance was not performed, however.

Overall, these advancements illustrate the diverse use of LLMs in an optimization context. However, we observe that the application of LLMs for generating novel direct global optimization algorithms from scratch is still at a very early stage. Often, the performance evaluation of the LLM (e.g., in EvoLLM) or the generated algorithms (e.g., in AEL) focuses on small test problems or does not benchmark against state-of-the-art algorithms. In-context learning, as in EvoLLM, and algorithm generation, as in STOP or AEL, are usually not combined, and the generation of new algorithms is typically based on mutation and crossover requests to the LLM.

In our approach, described in Section III, we propose to overcome such limitations by presenting a framework that can be used to generate any kind of algorithm as long there is a way to assess its quality automatically and it stays within the token limits of current LLMs. To achieve this goal, we combine in-context learning and an Evolutionary Algorithm-like iteration loop that allows the LLM to either refine (i.e., *mutate*) the best-so-far algorithm or redesign it completely. We combine this approach with a sound performance evaluation of the generated algorithms, based on the well-established IOHprofiler benchmark platform.

III. LLAMEA

The proposed LLaMEA framework (Figure 1) consists of four primary steps, which are iteratively repeated, following a similar structure as an Evolutionary Algorithm with one parent and one child (like a (1 ± 1) -EA) with an *initialisation* step and a general optimization loop of *evaluation* (in this case, by using IOHexperimenter [29]) and *refinement or redesign* of the algorithm, depending on the *selection strategy* (in case of a $(1 + 1)$ -strategy, only improvements are accepted, while a $(1, 1)$ -strategy always accepts the newly generated algorithm).

Our approach, which we abbreviate as (1 ± 1) -LLMEA, is presented in more detail in Algorithm 1. Here, initialization is performed in lines 1-6 by prompting the LLM with a task description prompt S , evaluating the fitness $f(a_t)$ of the generated program code (text) $a_t \in \mathcal{A}$, and memorizing the

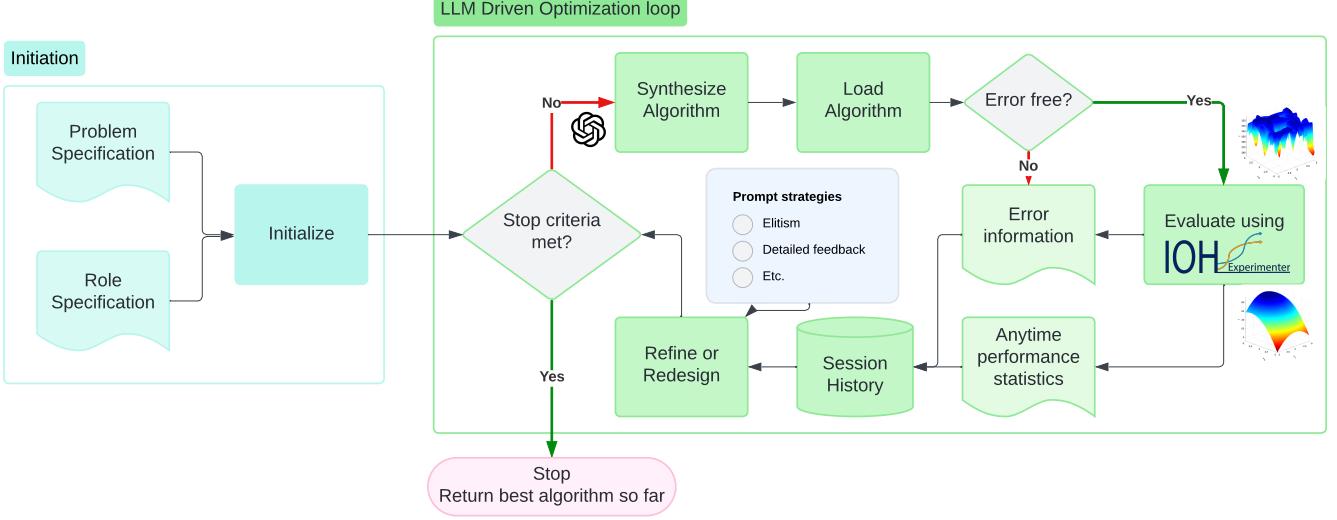


Fig. 1. The summary of the proposed LLM driven algorithm design framework LLaMEA. Full details of all steps are provided in the corresponding sections.

Algorithm 1 (1 + 1)-LLaMEA

```

1:  $S \leftarrow$  task-prompt                                ▷ Task description prompt
2:  $F_0 \leftarrow$  task-feedback-prompt                  ▷ Feedback prompt after each iteration
3:  $t \leftarrow 0$ 
4:  $a_t \leftarrow LLM(S)$                                 ▷ Initialize by generating first parent program
5:  $(y_t, \sigma_t, e_t) \leftarrow f(a_t)$                 ▷ Evaluate mean fitness and std.-dev. of first program and catch errors if occurring
6:  $a_b \leftarrow a_t; y_b \leftarrow y_t; \sigma_b \leftarrow \sigma_t; e_b \leftarrow e_t$  ▷ Remember best-so-far
7: while  $t < T$  do                                ▷ Budget not exhausted
8:   if (1 + 1) then                                ▷ Construct new prompt, using best-so-far algorithm
9:      $F \leftarrow (S, ((name(a_0), y_0, \sigma_0), \dots, (name(a_t), y_t, \sigma_t)), a_b, e_b, F_0)$  ▷ Construct new prompt, using latest parent algorithm
10:  else
11:     $F \leftarrow (S, ((name(a_0), y_0, \sigma_0), \dots, (name(a_t), y_t, \sigma_t)), a_t, e_t, F_0)$  ▷ Construct new prompt, using latest parent algorithm
12:  end if
13:   $a_{t+1} \leftarrow LLM(F)$                                 ▷ Generate offspring algorithm by mutation
14:   $(y_{t+1}, \sigma_{t+1}, e_{t+1}) \leftarrow f(a_{t+1})$           ▷ Evaluate offspring algorithm, catch errors
15:  if  $e_{t+1} \neq \emptyset$  then  $y_{t+1} = 0$            ▷ Errors occurred
16:  end if
17:  if  $y_{t+1} \geq y_t$  then  $a_b \leftarrow a_{t+1}; y_b \leftarrow y_{t+1}; \sigma_b \leftarrow \sigma_{t+1}; e_b \leftarrow e_{t+1}$  ▷ Update best
18:  end if
19:   $t \leftarrow t + 1$                                      ▷ Increase evaluation counter
20: end while
21: return  $a_b, y_b$                                     ▷ Return best algorithm and its fitness

```

initial program a_t , its mean fitness y_t and standard deviation of fitness σ_t over multiple runs, and potentially some runtime error information e_t as best solution (a_b, y_b, σ_b, e_b) (see Section III-C for a variation of this approach, in which more detailed information is provided to LLaMEA).

The while-loop (line 7-20) iterates through B (total budget) LLM calls to generate a new program each time (line 13). The prompt construction in lines 9 (in case of elitist selection) or line 11 (in case of non-elitist selection) is the essential step, generating a prompt F that consist of a concatenation¹

¹We use tuple notation here for clarity to show the components that compose the prompt string.

of the following information: (i) Task description S ; (ii) the algorithm names and mean fitness values and standard deviations of all previously generated algorithms; (iii) the current best algorithm's code a_b (in case of elitist selection) or the most recent algorithm's code a_t (in case of non-elitist selection); (iv) its execution error information e_t (which is empty if there are no execution errors) and (v) a short task prompt F_0 , with an instruction to the LLM. The new algorithm is generated by the LLM (line 13) and its fitness evaluated catching any runtime errors on the way (line 14), setting it explicitly to zero (worst possible fitness) if runtime errors occurred (line 15). The best-so-far algorithm is then updated if an improvement was found (line 17).

A. Starting Prompt

The initialisation of the optimization loop is crucial for the framework to work, as it sets the boundaries and rules that the LLM needs to operate with. Through experimentation, we found that including a small example code helps a lot in generating code without syntax and runtime errors. However, the example code could also bias the optimization search towards similar algorithms. In this work we therefore choose to use a simple implementation of random search as the example code to provide. The code of this random search algorithm is provided in our Zenodo repository [30]. The LLM receives an initial prompt with a specific set of criteria, domain expertise, and problem description. This starting point guides the LLM in generating an appropriate algorithm. The starting prompt has two main components, the specification of the role and general task e.g. *"Your task is to design novel metaheuristic algorithms to solve black box optimization problems."* and a detailed description of the problem to solve including a clear format of the expected response. An example problem description is given below:

```
Detailed task prompt  $S$ 
The optimization algorithm should handle a
→ wide range of tasks,
which is evaluated on a large test suite of
→ noiseless functions.
Your task is to write the optimization
→ algorithm in Python code.
The code should contain one function `def
→ __call__(self, f)`,
which should optimize the black box function
→ `f` using
`budget` function evaluations.
The f() can only be called as many times as
→ the budget allows.
An example of such code is as follows:
```
<initial example code>
```
Give a novel heuristic algorithm to solve
this task.
Give the response in the format:
# Name: <name of the algorithm>
# Code: <code>
```

B. Algorithm Synthesis (Initialization)

Using the prompt, the LLM generates the code for a new metaheuristic optimization algorithm, considering the constraints and guidance provided. The LLM should provide the answer in the format given, using MarkDown formatting for the code-block. The generated algorithm and name are extracted using regular expressions, including additional exception handling to capture small deviations from the asked format. In our experiments, 100% of the LLM responses lead to successfully extracted code. In addition, the LLM usually generates a small explanation in addition of what we ask, which we store for offline evaluation. In the case we cannot extract the code (which in practise never happened but can theoretically occur), we provide the LLM feedback that the response did not follow the provided format, trying to enforce the format for the next iteration.

Exception handling: Once the algorithm is extracted we dynamically load the generated Python code and instantiate a version of the algorithm for evaluation. The loading and instantiating of the code can lead to syntax errors, which we capture and store to be provided in the refinement prompt. When these errors occur, the evaluation run cannot commence, and is therefore skipped. Evaluation metrics are set to the lowest possible value (0) for the particular candidate and error messages e_t are stored for additional feedback to the LLM.

C. Evaluation

The generated algorithm is evaluated on the Black-Box Optimization Benchmark (BBOB) suite [24]. The suite consists of 24 noiseless functions, with an instance generation mechanic to generate a diverse set of different optimization landscapes that share particular function characteristics. The suite is divided into 5 function groups; separable functions, functions with low or moderate conditioning, high conditioning unimodal functions, multi-modal functions with strong global structure and multi-modal functions with weak global structure. The BBOB suite is considered a best-practise in the field of metaheuristic algorithm design as evaluation of newly proposed algorithms. For the evaluation feedback provided to the LLM, we summarize the performance on the whole BBOB suite by taking an average any-time performance metric, in our case the area over the convergence curve (AOCC) (see Equation 1). This results in one number y_t representing the performance of the proposed algorithm on average over the complete benchmark suite, and its standard deviation, σ_t . In addition to aggregating these values over all functions, we also experiment with a more detailed feedback mechanism where the average AOCC and its standard deviation are returned for each function group, resulting in ten performance values $(y_{t,1}, \dots, y_{t,5}), (\sigma_{t,1}, \dots, \sigma_{t,5})$. Theoretically, this approach should provide the LLM with information on what kind of functions the proposed solutions works well and on which it works less well.

D. Mutation, Selection and Feedback

The mutation and selection step in the LLaMEA framework consists mostly of the construction of the feedback prompt to the LLM in order to generate a new solution. Depending on the selection strategy, either the current-best algorithm a_b is given back (*plus* strategy) to the LLM including the score it had on the BBOB suite, or the last generated algorithm a_t (*comma* strategy) is provided to the LLM in the feedback prompt.

After the selection is made, a feedback prompt F is constructed using the following template:

```
Feedback prompt template  $F$ 
<role and task prompt>
<List of previously generated algorithm names
→ with mean AOCC score>
<selected algorithm to refine (full code) and
→ mean and std AOCC scores>
Either refine or redesign to improve the
→ algorithm.
```

The prompt includes the initial prompt S , a list $((\text{name}(a_0), y_0, \sigma_0), \dots, (\text{name}(a_t), y_t, \sigma_t))$ with previously

generated algorithm names and their mean scores and standard deviations, the selected algorithm a_b or a_t , including its score y and error information e , to mutate and a short task prompt $F_0 = \text{"Either refine or redesign to improve the algorithm"}$ telling the LLM to perform a mutation or redesign (restart) action. The list of previously tried algorithm names is included to make sure the LLM is not generating (almost) the same algorithm twice.

The construction of the feedback prompt includes a few choices, which in general can be seen in an evolutionary computation context as follows:

Restarts and Mutation Rate: We can ask the LLM to either make a (small) refinement of an algorithm or redesign it completely, where the latter is analogue to a restart or very large mutation rate in an ES optimization run, increasing its exploration behaviour. In the proposed framework we leave this choice to the LLM itself as our task is simply *"Either refine or redesign to improve the algorithm"*. In Section V-B, we analyze how the LLM makes this decision over time, in terms of the generated names as well as the code similarity .

Plus and Comma Strategy: In the proposed framework we support both $(1+1)$ - and $(1,1)$ -LLaMEA strategies, meaning the LLM either is asked to refine/redesign the best-so-far algorithm (in the elitist $(1+1)$ -case), or the last generated algorithm (in the $(1,1)$ -case). Only the full code of the selected (best or last) algorithm is provided to the LLM in every iteration. In our experiments we demonstrate the differences between both strategies per LLM model.

Detailed feedback: Instead of providing an overall score and standard deviation, we can provide the LLM with more performance details of the generated algorithms. In our case, we can provide additional metrics per BBOB function group, potentially giving the LLM information on what kind of functions the solution works well and on which ones it does not work well. This results in mean AOCC values and standard deviations for each of the five function groups, i.e., $(y_{t,1}, \dots, y_{t,5}), (\sigma_{t,1}, \dots, \sigma_{t,5})$. In our experiments we cover the inclusion of such details versus leaving this information out. When we provided more detailed feedback, we provide the mean AOCC and the standard deviation of the AOCC scores for the separable functions, functions with low or moderate conditioning, unimodal functions with high conditioning, multi-modal functions with adequate global structure and functions with weak global structure (the five BBOB function groups). In practise, it would even be possible to provide the full convergence plots, either as csv or image to the multi-modal LLMs, this option we leave for future research.

Session history: In the proposed framework we do not keep the entire run history in every iteration (including all codes (a_0, \dots, a_t)), as this becomes more and more expensive as the list grows.

However, the inclusion of a larger set of previous attempts by providing the previous solutions in code with their associated scores could in theory be beneficial. The LLM would be able to act as a kind of surrogate model in predicting the next solution. This would translate to evolutionary computation terms, as to keeping an archive of best solutions or the use

of machine learning as surrogate models in surrogate assisted optimization [31], [32].

Instead of providing all codes, we keep a condensed list of algorithm names (generated by the LLM) and their respective scores and standard deviations of scores, $((\text{name}(a_0), y_0, \sigma_0), \dots, (\text{name}(a_t), y_t, \sigma_t))$, to facilitate in-context learning of the LLM [25], [26]. The purpose of keeping this list and providing it to the LLM is two-fold, namely 1) the LLM could learn what kind of algorithms work well and which work less good (by analysing the algorithm names only), and 2) it makes it less likely that the LLM is generating the same algorithm twice.

IV. EXPERIMENTAL SETUP

To validate the proposed evolution framework for generating and optimizing metaheuristics, we designed a set of experiments to compare different LLMs, different ES strategies and different levels of detail in the provided evaluation feedback. In addition, the best generated algorithms are analysed and compared to several state-of-the-art baselines on the BBOB suite using additional instances and additional dimension settings.

A. Large Language Models

In our experiments we have limited the number of LLMs to the ChatGPT family of models, including GPT-3.5-turbo [33], GPT-4-turbo [34] and the recently released GPT-4o [35]. The LLaMEA framework leverages the OpenAI chat completion API call for querying the LLM. For abbreviating LLM-names, we drop the extension "turbo" in the following.

B. Benchmark Problems

To evaluate the proposed optimization algorithms we use the well-known Black-Box Optimization Benchmark suite (BBOB) [24] (See Table I in the Appendix for an overview of the functions). These benchmark functions can be separated into five groups with different high-level function characteristics. For a robust evaluation we use three different instances per function, where an instance of a BBOB function is defined by a series of random transformations that do not alter the global function characteristics. In addition, we perform 3 independent runs per function instance with different random seeds (giving a total of 9 runs per BBOB function). Each run has an evaluation budget of 10 000 function evaluations. In our experiments we set the dimensionality of the optimization problems to $d = 5$. We run the main $(1+1)$ -LLaMEA optimization loop in Algorithm 1 for $T = 100$ iterations.

C. Performance Metrics

To evaluate the generated algorithms effectively over a complete set of benchmark functions we use a so-called *anytime performance measure*, meaning that it quantifies performance of the optimization algorithm over the complete budget, instead of only looking at the final objective function value. For this we use the normalized Area Over the Convergence

Curve (AOCC), as introduced in [36]. The AOCC is given in Equation 1.

$$AOCC(\mathbf{y}) = \frac{1}{B} \sum_{i=1}^B \left(1 - \frac{\min(\max((y_i), lb), ub) - lb}{ub - lb} \right) \quad (1)$$

Here, \mathbf{y} is the sequence of best-so-far function values reached during the optimization run, $B = 10\,000$ is the budget, lb and ub are the lower and upper bound of the function value range of interest. The AOCC is equivalent to the area under the so-called *Empirical Cumulative Distribution Function* (ECDF) curve with infinite targets between the chosen bounds [37]. Following the conventions of the BBOB suite [38], the function values are log-scaled before calculating the AOCC, and we use the bounds 10^{-8} and 10^2 .

We aggregate the AOCC scores of all benchmark functions by taking the mean. The final mean AOCC determines the selected next parent (in a $(1 + 1)$ -strategy) and is given as feedback to the LLM in the next mutation step.

In addition to the mean AOCC score, any runtime or compile errors occurred during validation are also used to give feedback to the LLM. In case of fatal errors (no execution took place), the mean AOCC is set to the lowest possible score, zero.

V. RESULTS AND DISCUSSION

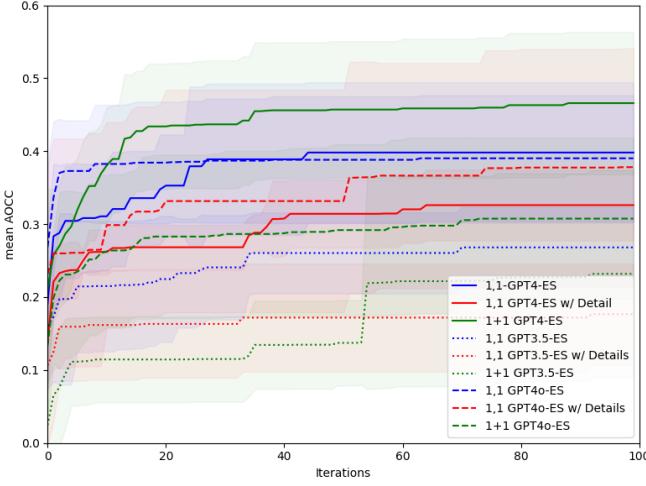


Fig. 2. Mean convergence curves (best-so-far algorithm scores) over the 5 different runs for each LLM and strategy. Shaded areas denote the standard deviation of the best-so-far. *w/Details* denotes that we use a feedback mechanism that provides not just the plain average AOCC but also the average AOCC per BBOB function group as feedback to the LLM.

In Figure 2, the mean best-so-far algorithm evaluation score (AOCC) per configuration (LLM and feedback strategy) is shown. The shaded region denotes the standard deviation over the 5 independent runs. Different strategies seem to have different effects depending on the model used. For example, the $(1 + 1)$ -selection strategy seems to be beneficial for GPT-4, but is having a deteriorating effect when using GPT-4o. Overall GPT-4 seems to be better suited for the task overall, and GPT-3.5 is clearly a less favorable option.

A. Novelty and diversity

Analysing the generated codes and their generates names, it is immediately obvious (and not surprising) that the LLM uses existing algorithms, algorithm components and search strategies in generating the proposed solutions.



Fig. 3. Word cloud of algorithm name parts generated over all different LLaMEA runs.

In Figure 3, a word-cloud is shown with all the sub-strings of generated algorithm names and their occurrence frequency visualized. The generated algorithm names were in Python Camel-case style, so it was easy to split the algorithm names in individual parts. In some cases, the algorithm name was just an abbreviation, these abbreviations are kept as words in this analysis. The most used parts in algorithm names are rather generic, such as “evolution”, “adaptive” and “dynamic”, some of the parts directly refer to existing algorithms such as “harmony” and “firework”, and other strings refer to existing strategies such as “gradient”, “local”, “elite” etc. In general when observing the different generated solutions, we see interesting and novel combinations of existing techniques, such as “surrogate assisted differential evolution combined with covariance matrix adaptation evolution strategies” and “dynamic firework optimizer with enhanced local search”. A full list of generated algorithms and their names is provided in our Zenodo repository [30].

B. Mutation Rates

To analyse how much the LLMs change (mutate) the algorithms over an entire optimization run, the code *diff* ratio (number of code lines that are different between a pair of programs divided by the length of the largest code) is calculated over each run between parent and offspring solutions. In Figure 4, the mean *diff* over 5 different runs per LLM is shown. There are a few interesting observations we can

make: The difference between parent and offspring for GPT-3.5 is on average much lower than for other models, the (1 + 1)-GPT3.5-LLaMEA shows much higher differences in the beginning of the run than the other strategies with the same model. GPT-4-LLaMEA in general shows the largest differences (exploration) over the runs. It is very interesting to observe that the ratio of code differences in most cases seems to converge, indicating more exploration in the beginning of the search and more exploitation during the final parts of the search. This is interesting as the LLM has no information on the search budget, and can therefore also not base its decision to make large or small refinements on the stage of the optimization run. We can furthermore see in the generated names of the algorithms that the LLM has either tried to refine the algorithm (adding “Improved” or “Refined” or “Enhanced” to the name, or names such as <algorithm>V1, etc.) or redesign the algorithm using a different strategy (based on different existing algorithm names such as Differential Evolution, Particle Swarm Optimization etc.). We can therefore also look at the similarity between parent and offspring algorithm names to visualize the refinement process of the optimization runs.

To do so, we use the *Jaro similarity* [39] as it gives a ratio between 0 (completely different names) and 1 (completely matching). The Jaro similarity score between two algorithm names s and t is calculated as:

$$\text{Jaro}(s, t) = \begin{cases} 0 & \text{if } m = 0, \\ \frac{1}{3} \left(\frac{m}{|s|} + \frac{m}{|t|} + \frac{m-t}{m} \right) & \text{otherwise,} \end{cases}$$

where

- s and t are the input strings.
- m is the number of matching characters. Two characters from s and t are considered matching if they are the same and not farther apart than $\max(|s|, |t|)/2 - 1$.
- t is half the number of transpositions. A transposition occurs when two matching characters are in a different order in s and t .
- $|s|$ and $|t|$ are the lengths of strings s and t , respectively.

Figure 5 illustrates the average algorithm name similarity over iterations for each optimization run and LLM. Figure 5 illustrates the Jaro similarity mean (five runs) of the subsequently generated algorithm names (i.e., $\text{Jaro}(\text{name}(a'), \text{name}(a_{t+1}))$ for $t = 0, \dots, B - 1$, where $a' = a_b$ for (1 + 1)-selection and $a' = a_t$ for (1, 1)-selection) over iterations t for each of the nine LLM-based optimization run configurations. In Figure 6, we show the Jaro similarity for just a single run selected for three of the nine configurations.

The Jaro similarity scores consistently show a behavior similar to the code difference ratios. Especially from the visualisation with individual runs it is clear that some steps only involve very small mutations (especially for GPT3.5) and sometimes a kind of restart occurs where the similarity score reaches zero.

VI. ANALYSIS OF BEST ALGORITHMS

The experiment above resulted in 3 657 algorithms (out of a maximum² of 4 500) that were at least able to get an AOCC

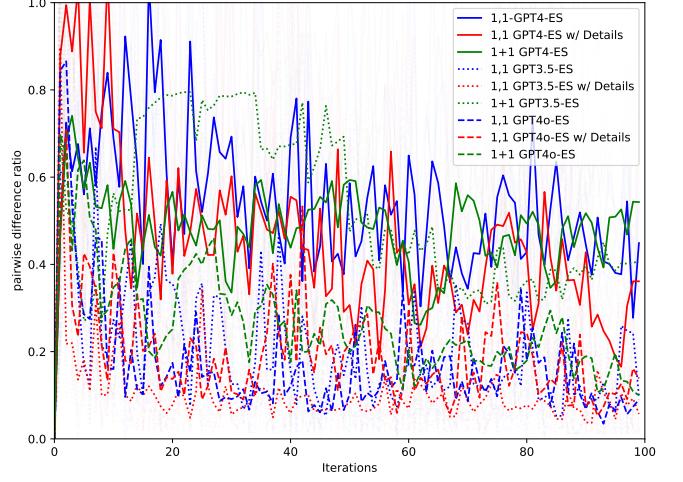


Fig. 4. Pairwise differences between parent and offspring for each iteration. Solid lines represent the mean over all runs per model and strategy, more transparent lines are individual runs. *w/ Details* denotes that we use a feedback mechanism that provided not just the plain average AOCC but also the average AOCC per BBOB function group as feedback to the LLM.

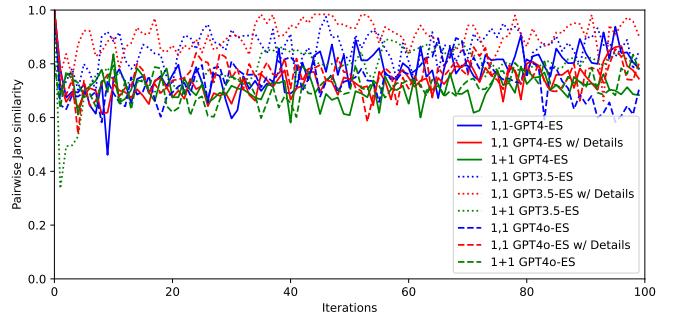


Fig. 5. Average Jaro similarity scores between parent and offspring over each optimization run for different models.

score larger than zero on the BBOB suite of optimization problems in 5 dimensions. In the next step to validate our proposed framework, we evaluate how much the best of these algorithms can generalize beyond the evaluation performed during the optimization loop, and subsequently we analyze the code and behaviour of the best-performing algorithms that beat the state-the-art baseline in-depth. The evaluation is done in 5, 10 and 20 dimensions, using 5 different instances and 5 independent runs per instance (so 25 runs per BBOB function to optimize). The state-of-the-art baselines include the covariance matrix adaptation evolution strategies (CMA-ES) [40], BIPOP-CMA-ES [41] and Differential Evolution (DE) [42].

In Figure 7 the *empirical attainment functions* (EAF) [37] are shown for the best algorithm generated per LLM and strategy, so 9 algorithms in total. The empirical attainment function estimates the percentage of runs that attain an arbitrary target value not later than a given runtime. Taking the partial integral of the EAF results in a more accurate version of the Empirical Cumulative Distribution Function, since it does not rely on discretization of the targets. The

²Resulting from 5 runs with $T = 100$ iterations, for 9 different models.

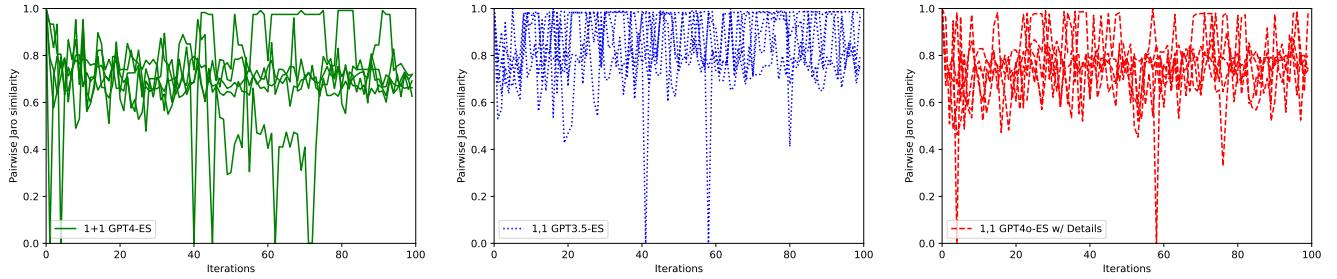


Fig. 6. Jaro similarity scores between parent and offspring for each run (5 independent runs) of different models and strategies. Left: (1 + 1)-GPT4-ES; middle: (1, 1)-GPT3.5-ES; right: (1, 1)-GPT4o-ES with Details.

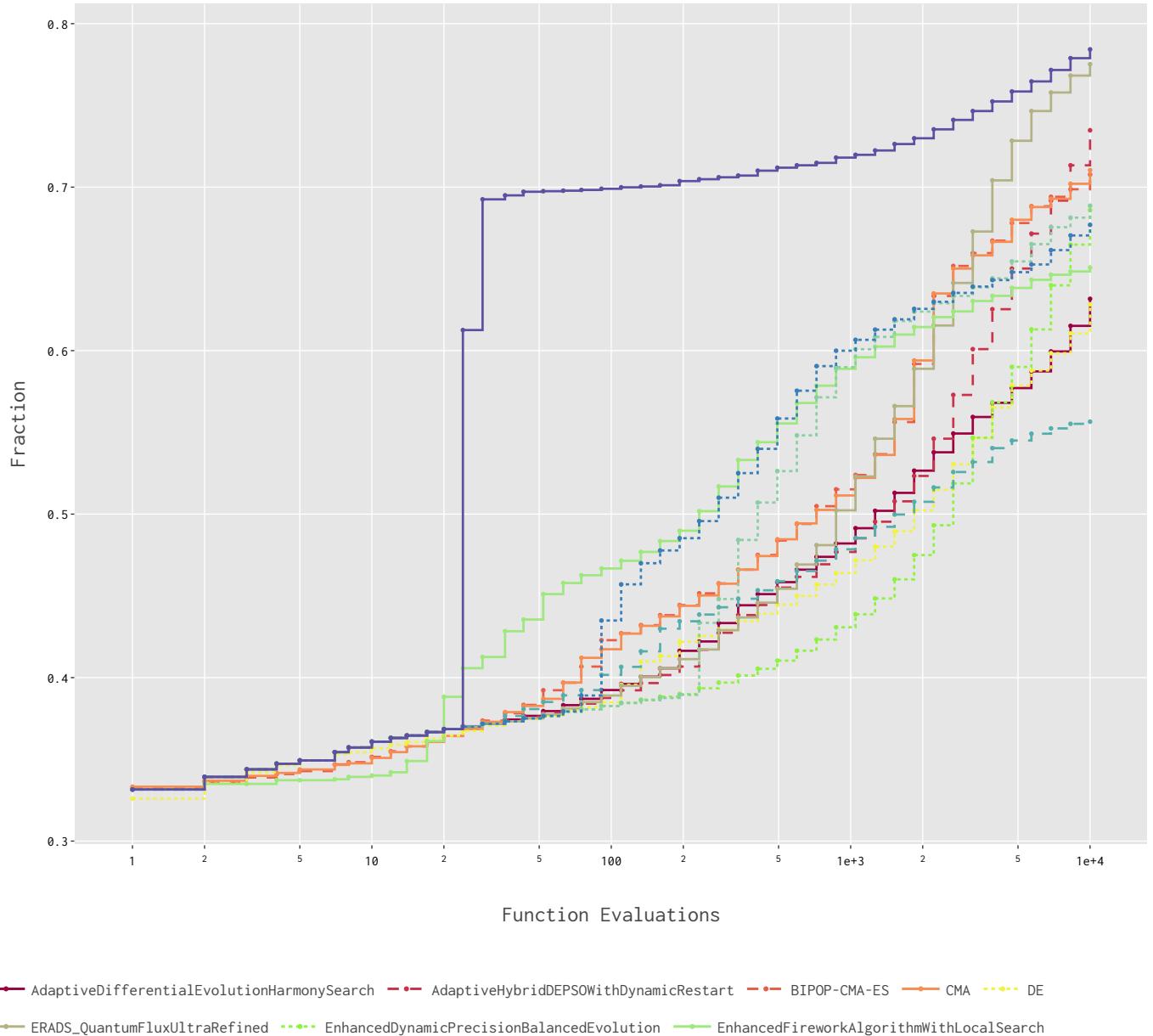


Fig. 7. The empirical attainment functions (EAF) estimate the percentage of runs that attain an arbitrary target value not later than a given runtime. EAF for the best algorithms per LLM configuration, averaged over all 24 BBOB functions in 5d

area under this curve is a similar measure as the AOCC used during the LLaMEA optimization runs, and denotes how well an optimization algorithm performed over the complete run (on average over all instances, independent runs and objective functions). From the EAF curves and the area under these curves (see Table II in the Appendix), we can observe that LLaMEA has found two algorithms with better performance for $d = 5$ than the state-of-the-art CMA-ES, BIPOP-CMA-ES and DE, with the algorithm called *RADEA* being the top performer and *ERADS_QuantumFluxUltraRefined* the second best. We therefore analyze both *RADEA* and the second best algorithm, *ERADS_QuantumFluxUltraRefined* (*ERADS*) in detail in the next sections. Detailed convergence curves per BBOB function (for $d = 5$) are shown in Figure 8.

A. Robust Adaptive Differential Evolution with Archive

One of the two top performing algorithms generated by the LLaMEA is "*Robust Adaptive Differential Evolution with Archive*" (*RADEA*). The key idea of *RADEA* is the utilization of an archive of best solutions, which are used in the mutation operator.

Algorithm 2 Robust Adaptive Differential Evolution with Archive (*RADEA*)

```

1:  $N \leftarrow 20$                                  $\triangleright$  Population size
2:  $N' \leftarrow 50$                                 $\triangleright$  Archive size
3:  $P \leftarrow$  u.a.r. population initialization within  $[low, high]$ 
4:  $A \leftarrow \emptyset$                              $\triangleright$  Initialize archive as empty
5:  $t \leftarrow 0$                                   $\triangleright$  Initialize iteration counter
6:  $F \leftarrow 0.8, CR \leftarrow 0.9$   $\triangleright$  Control parameters for mutation and crossover
7:  $fit_P \leftarrow f(P)$                           $\triangleright$  Evaluate initial population
8: while  $t \cdot N < B$  do                       $\triangleright$  While budget not exhausted
9:    $P' \leftarrow \text{Mutation}(P, A, F)$             $\triangleright$  Generate mutants
10:   $C \leftarrow \text{Crossover}(P, P', CR)$          $\triangleright$  Crossover to create trial population
11:   $fit_C \leftarrow f(C)$                           $\triangleright$  Evaluate trial population
12:   $P, fit_P \leftarrow \text{Selection}(P, fit_P, C, fit_C)$   $\triangleright$  Selection of the fittest
13:   $\text{Update\_Archive}(A, N', P, fit_P)$         $\triangleright$  Update archive with improved solutions
14:   $t \leftarrow t + 1$                              $\triangleright$  Increment iteration counter
15: end while
16:  $i_{best} \leftarrow \text{argmin}(fit_P)$            $\triangleright$  Index of the best solution
17: return  $P[i_{best}], fit_P[i_{best}]$          $\triangleright$  Return best solution and its fitness

```

The pseudocode of *RADEA*, representing the key ideas of the generated Python code, was extracted manually and is shown in Algorithm 2. The full Python code can be retrieved from our online repository [30]. The different functions of *RADEA* are summarized as follows:

- **Mutation**(P, A, F): Generates mutants for each individual in the population by selecting three distinct vectors from the union of P and A (if A is not empty), creating a new vector by adding the weighted difference of two vectors to the third. This method ensures that the archive can contribute to the diversity of the mutation process.

- **Crossover**(P, P', CR): Performs crossover between the population P and the mutant population P' with a crossover rate CR . Each dimension of each individual has a probability CR to be replaced by the corresponding dimension in the mutant.
- **Selection**(P, fit_P, C, fit_C): Compares each individual in P with its counterpart in C . An individual in C with better fitness replaces the corresponding individual in P .
- **Update_Archive**(A, N', P, fit_P): Adds improved solutions to the archive A and ensures that the size of the archive does not exceed its maximum capacity N' by removing the oldest entries.

The proposed *RADEA* algorithm seems most similar to the existing JADE [43] algorithm from literature. There are however some key differences which are, to the best of our knowledge, novel and have not been published before.

The key differences primarily revolve around the use of archives and parameter adaptation. *RADEA* emphasizes a robust use of an archive not only to store but actively use archived solutions in the mutation process, enhancing diversity and potentially aiding in overcoming local optima. This archive dynamically influences mutation strategies and contributes directly to the generation of new candidate solutions. In contrast, JADE's use of an archive is optional and primarily serves to maintain diversity without a direct integration into the mutation process. Furthermore, JADE features adaptive control parameters (F and CR) that adjust dynamically based on the success rates of previous generations, aiming to optimize these parameters in real time to improve performance. *RADEA* does not show adaptive control of parameters even though the name suggests otherwise.

B. *ERADS Quantum Flux Ultra Refined*

While the name sounds rather futuristic and very sophisticated, upon a close inspection of the code the "*ERADS Quantum Flux Ultra Refined*" algorithm looks very similar to the previously described *RADEA* algorithm, which is interesting as the *RADEA* algorithm was generated during a different optimization run (though with the same GPT model). This could indicate a bias towards Differential Evolution style algorithms in the search. According to the LLM, *ERADS* stands for "*Enhanced RADEDM with Strategic Mutation*", and *RADEDM* stands for "*Refined Adaptive Differential Evolution with Dynamic Memory*". The pseudocode of *ERADS* was again extracted manually and is shown in Algorithm 3. The full Python code can be retrieved from our online repository [30].

The main differences with *RADEA* are the use of a memory factor to guide the mutation in certain directions and an adaptive F and CR strategy. It is unclear, why the LLM generated the "Quantum Flux" component in the algorithm's name, but it can be assumed that LLM hallucinations [44] will also occur when applied to code generation tasks.

It is interesting to observe that both *RADEA* and *ERADS* are in principle very similar, with the key difference that *ERADS* uses adaptive parameter control and a memory factor for mutation, while *RADEA* uses an archive of best solutions.

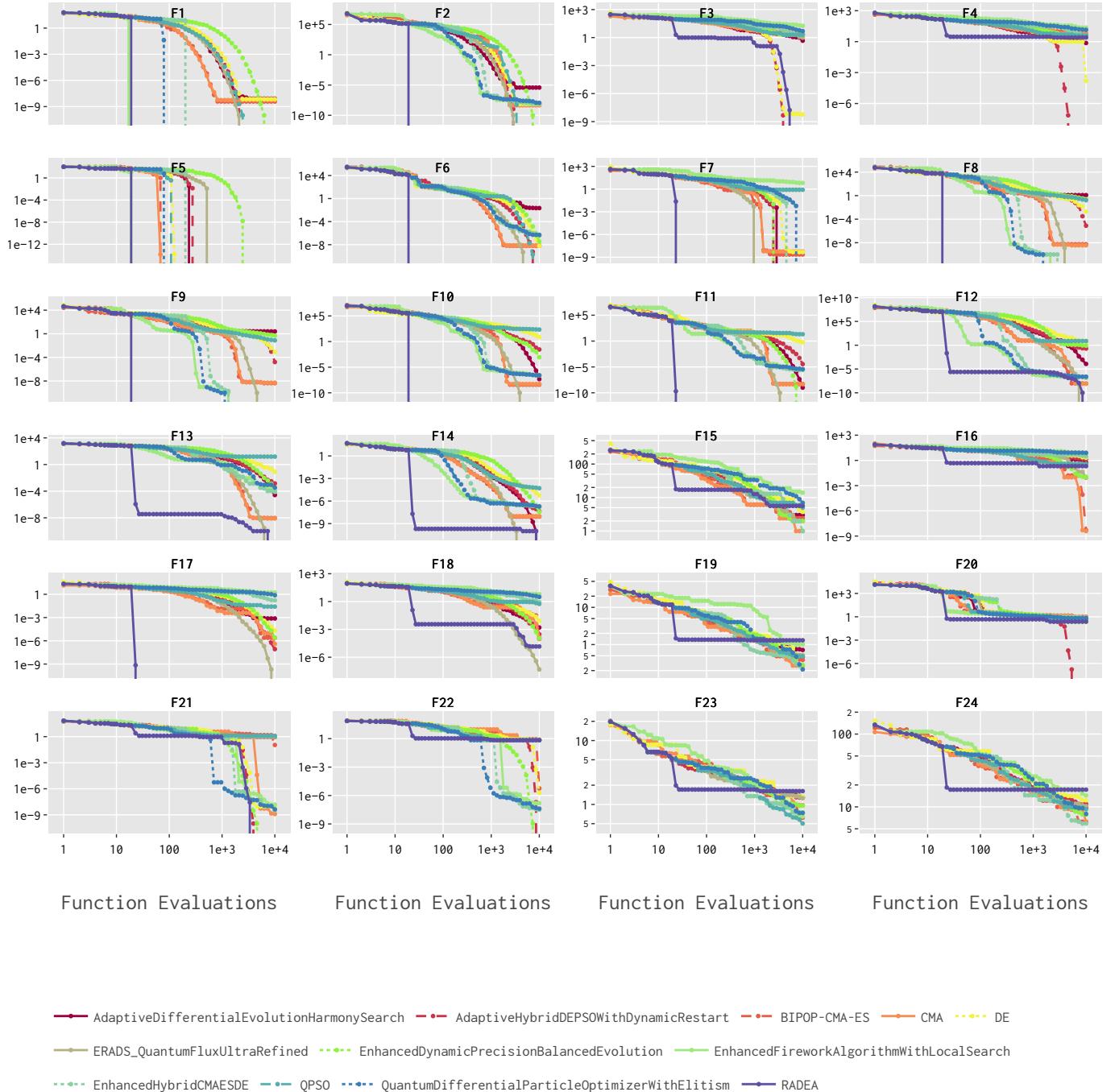


Fig. 8. Median best-so-far function value (y -axis) over the 10 000 function evaluations (x -axis) per BBOB function in 5d for the best algorithm per LLM configuration (9 algorithms) and the baselines; CMA-ES, DE and BIPOP-CMA-ES.

C. Performance Analysis in Higher Dimensions

The two best-found algorithms using the LLaMEA framework are evaluated against the most relevant state-of-the-art optimizers, namely the previously used baselines BIPOP-CMA-ES, CMA-ES and Differential Evolution (DE), and the JADE algorithm [43], all of which are using recommended hyper-parameter settings. The baselines all originate from the IOH analyzer benchmark data set [45]. In Figure 9, the empirical attainment function of the proposed algorithms and

baselines is shown for dimension $d \in \{10, 20\}$ using results from all BBOB functions, 5 instances per function and 5 random seeds (25 runs per BBOB function in total). It is interesting to observe that while ERADS performs well in 5 dimensional problems, it fails to generalize well to higher dimensions, while RADEA has very good performance in higher dimensions as well (in terms of anytime performance, which is the area below these curves). ERADS performs overall slightly better than JADE and standard DE and as good

Algorithm 3 ERADS_QuantumFluxUltraRefined

```

1:  $N \leftarrow 50$                                  $\triangleright$  Population size
2:  $F_{init} \leftarrow 0.55, F_{final} \leftarrow 0.85$   $\triangleright$  Initial and final mutation scaling factors
3:  $CR \leftarrow 0.95$                              $\triangleright$  Crossover probability
4:  $Memory\_factor \leftarrow 0.3$        $\triangleright$  Factor to integrate memory in mutation
5:  $P \leftarrow$  u.a.r. population initialization within  $(-5.0, 5.0)$ 
6:  $fit_P \leftarrow f(P)$ 
7:  $best\_index \leftarrow \text{argmin}(fit_P)$ 
8:  $f_{opt} \leftarrow fit_P[best\_index]$ 
9:  $\mathbf{x}_{opt} \leftarrow P[best\_index]$ 
10:  $Memory \leftarrow \mathbf{0}$   $\triangleright$  Initialize memory for mutation direction
11:  $t \leftarrow N$ 
12: while  $t < B$  do
13:    $F_{current} \leftarrow F_{init} + (F_{final} - F_{init}) \cdot \left(\frac{t}{B}\right)$ 
14:   for each  $i$  in  $[0, N - 1]$  do
15:      $indices \leftarrow$  select 3 distinct indices u.a.r. from  $[0, N - 1] \setminus \{i\}$ 
16:      $\mathbf{x1}, \mathbf{x2}, \mathbf{x3} \leftarrow P[indices]$ 
17:      $best \leftarrow P[best\_index]$ 
18:      $mutant \leftarrow \mathbf{x1} + F_{current} \cdot (best - \mathbf{x1} + \mathbf{x2} - \mathbf{x3} + Memory\_factor \cdot Memory)$ 
19:      $mutant \leftarrow \text{clip } mutant \text{ to bounds } (-5.0, 5.0)$ 
20:      $trial \leftarrow \text{crossover } (mutant, P[i]) \text{ w. prob. } CR$ 
21:      $f\_trial \leftarrow f(trial)$ 
22:      $t \leftarrow t + 1$ 
23:     if  $f\_trial < fit_P[i]$  then
24:        $P[i] \leftarrow trial$ 
25:        $fit_P[i] \leftarrow f\_trial$ 
26:       if  $f\_trial < f_{opt}$  then
27:          $f_{opt} \leftarrow f\_trial$ 
28:          $\mathbf{x}_{opt} \leftarrow trial$ 
29:          $best\_index \leftarrow i$ 
30:       end if
31:        $Memory \leftarrow (1 - Memory\_factor) \cdot Memory + Memory\_factor \cdot F_{current} \cdot (mutant - P[i])$ 
32:     end if
33:     if  $t \geq B$  then
34:       break
35:     end if
36:   end for
37: end while
38: return  $\mathbf{x}_{opt}, f_{opt}$   $\triangleright$  Return best solution and fitness

```

as the CMA-ES and BIPOP-CMA-ES.

See Appendix A for the detailed convergence curves per BBOB function in 5d, 10d and 20d.

VII. CONCLUSIONS AND OUTLOOK

This paper introduced LLaMEA, a novel framework leveraging Large Language Models (LLMs) for the automatic generation and optimization of metaheuristic algorithms. Our approach uniquely automates the evolution of algorithm design, enabling efficient exploration and optimization within a computationally feasible framework. Our findings demonstrate

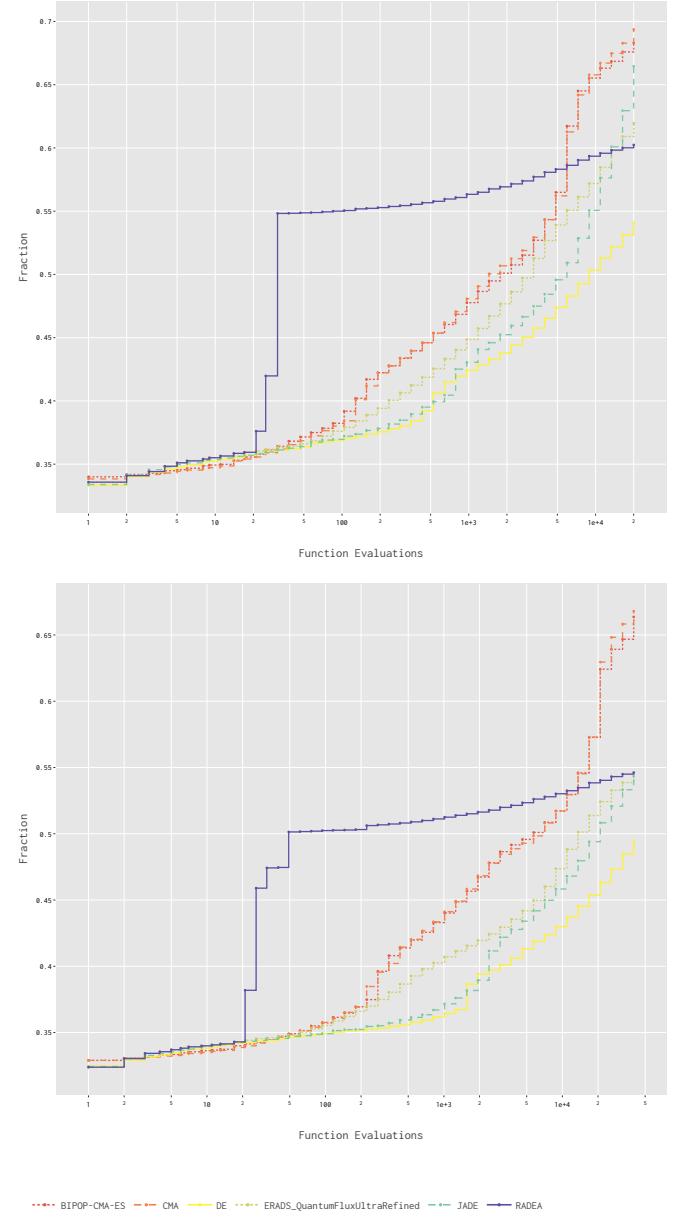


Fig. 9. The empirical attainment function (EAF) estimates the percentage of runs that attain an arbitrary target value not later than a given runtime. EAF for the best two algorithm (RADEA and ERADS) and the baselines; CMA-ES, DE, BIPOP-CMA-ES and JADE, averaged over all 24 BBOB functions in (from top to bottom) 10d and 20d respectively.

that LLaMEA can effectively generate high-performing algorithms that rival and sometimes surpass existing state-of-the-art techniques.

The LLaMEA framework proved capable of generating and evolving algorithms that perform comparably to traditional state-of-the-art metaheuristics, demonstrating the potential of LLMs to understand and innovate within the algorithmic design space effectively. Algorithms evolved through our framework successfully compete against established metaheuristics, highlighting the practical applicability of using LLMs for automated construction of optimization heuristics. Since LLMs have been trained on an exceedingly large code base, including

the currently available metaheuristics, we can explain their observed strong performance by interpreting them as a universal *modular framework for algorithm construction* (see also Section I) that have access to a huge number of "modules" that can be combined to form new algorithms.

Challenges and limitations: The proposed LLaMEA framework presents several limitations and challenges that can be addressed for further advancement. One significant challenge lies in the dependency on the quality and structure of the prompts used to guide the LLM, which can introduce biases or limit the diversity of the generated algorithms. Additionally, the execution reliability of the dynamically generated code can be problematic, as errors during runtime can affect the evaluation of algorithm performance. Moreover, the computational cost associated with training and querying large language models may pose scalability issues, particularly for extensive or multi-objective optimization problems. Addressing these challenges would enhance the robustness and applicability of the LLaMEA framework across different domains and more complex optimization scenarios.

Outlook: The promising results of the LLaMEA framework pave the way for several exciting directions for future research:

- Future work could explore the expansion of the LLaMEA framework to support a broader range of evolutionary strategies. While the proposed framework focuses on an (1 ± 1) -EA approach, where we have one parent and generate one solution at a time, it is possible to generalize the framework to a $(\mu \pm \lambda)$ -EA, keeping a larger population and generating multiple individuals. This would translate to generating multiple mutations with LLMs by leveraging multiple random seeds in the generation process.
- For generating diverse and innovative algorithm candidates, a population-based $(\mu \pm \lambda)$ -EA could use different LLMs in parallel and control the *temperature parameter* of the LLMs to behave more explorative in the beginning of the search [46].
- Applying the LLaMEA methodology to other domains such as automated machine learning (AutoML) and complex system design could further demonstrate the versatility and impact of this approach.

By continuing to develop and refine these approaches, we anticipate that the integration of LLMs in algorithmic design will significantly advance the field of evolutionary computation, leading to more intelligent, adaptable, and efficient systems.

DECLARATIONS

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Reproducibility statement. We provide an open-source documented implementation of our package at [30], including install and how-to-use guide.

Intermediate results, generated algorithms and BBOB evaluation results are available as well in subdirectories of the repository.

REFERENCES

- [1] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer, 2015.
- [2] T. Bäck, D. B. Fogel, and Z. Michalewicz, "Handbook of evolutionary computation," *Release*, vol. 97, no. 1, p. B1, 1997.
- [3] J. Kennedy and R. C. Eberhart, *Swarm Intelligence*. Morgan Kaufmann, 2001.
- [4] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.
- [5] D. J. Greiner Sánchez, A. Gaspar-Cunha, J. D. Hernández Sosa, E. Minisci, and A. Zamuda, *Evolutionary algorithms in engineering design optimization*. MDPI, 2022.
- [6] *Real-world Applications of Evolutionary Algorithms*, 2011, pp. 211–262. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9781848166820_0006
- [7] R. Chiong, T. Weise, and Z. Michalewicz, *Variants of evolutionary algorithms for real-world applications*. Springer, 2012, vol. 2.
- [8] Z. Ma, G. Wu, P. N. Suganthan, A. Song, and Q. Luo, "Performance assessment and exhaustive listing of 500+ nature-inspired metaheuristic algorithms," *Swarm and Evolutionary Computation*, vol. 77, p. 101248, 2023.
- [9] J. Del Ser, E. Osaba, A. D. Martinez, M. N. Bilbao, J. Poyatos, D. Molina, and F. Herrera, "More is not always better: insights from a massive comparison of meta-heuristic algorithms over real-parameter optimization problems," in *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2021, pp. 1–7.
- [10] D. Vermetten, C. Doerr, H. Wang, A. V. Kononova, and T. Bäck, "Large-scale benchmarking of metaphor-based optimization heuristics," 2024, arXiv:2402.09800.
- [11] S. van Rijn, H. Wang, M. van Leeuwen, and T. Bäck, "Evolving the structure of evolution strategies," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–8.
- [12] D. Vermetten, M. López-Ibáñez, O. Mersmann, R. Allmendinger, and A. V. Kononova, "Analysis of modular CMA-ES on strict box-constrained problems in the SBOX-COST benchmarking suite," in *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, ser. GECCO '23 Companion. New York, NY, USA: Association for Computing Machinery, 2023, p. 2346–2353. [Online]. Available: <https://doi.org/10.1145/3583133.3596419>
- [13] D. Vermetten, F. Caraffini, A. V. Kononova, and T. Bäck, "Modular differential evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 864–872. [Online]. Available: <https://doi.org/10.1145/3583131.3590417>
- [14] C. L. Camacho-Villalón, M. Dorigo, and T. Stützle, "Pso-x: A component-based framework for the automatic design of particle swarm optimization algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 26, no. 3, pp. 402–416, 2022.
- [15] C. L. Camacho-Villalón, T. Stützle, and M. Dorigo, "Designing new metaheuristics: Manual versus automatic approaches," *Intelligent Computing*, vol. 2, p. 0048, 2023. [Online]. Available: <https://sspi.science.org/doi/abs/10.34133/icomputing.0048>
- [16] J. de Nobel, F. Ye, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "IOHexperimenter: Benchmarking Platform for Iterative Optimization Heuristics," *Evolutionary Computation*, pp. 1–6, 02 2024. [Online]. Available: https://doi.org/10.1162/evco_a_00342
- [17] Q. Guo, R. Wang, J. Guo, B. Li, K. Song, X. Tan, G. Liu, J. Bian, and Y. Yang, "Connecting large language models with evolutionary algorithms yields powerful prompt optimizers," 2024, arXiv:2309.08532.
- [18] T. Rios, S. Menzel, and B. Sendhoff, "Large language and text-to-3d models for engineering design optimization," 2023, arXiv:2307.01230.
- [19] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, "Large language models are human-level prompt engineers," in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. [Online]. Available: <https://openreview.net/forum?id=YdqwNaCLCx>
- [20] O. Honovich, U. Shaham, S. R. Bowman, and O. Levy, "Instruction induction: From few examples to natural language task descriptions," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 1935–1952. [Online]. Available: <https://aclanthology.org/2023.acl-long.108>
- [21] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *ArXiv*, vol. abs/2205.11916, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:249017743>

- [22] F. Liu, X. Tong, M. Yuan, and Q. Zhang, "Algorithm evolution using large language model," 2023, arXiv:2311.15249.
- [23] R. T. Lange, Y. Tian, and Y. Tang, "Large language models as evolution strategies," 2024, arXiv:2402.18381.
- [24] N. Hansen, S. Finck, R. Ros, and A. Auger, "Real-parameter black-box optimization benchmarking 2009: Noiseless functions definitions," INRIA, Tech. Rep. RR6829, 2009.
- [25] M. Luo, X. Xu, Y. Liu, P. Pasupat, and M. Kazemi, "In-context learning with retrieved demonstrations for language models: A survey," 2024, arXiv:2401.11624.
- [26] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, "A survey on in-context learning," 2023, arXiv:2301.00234.
- [27] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, "Self-taught optimizer (STOP): Recursively self-improving code generation," 2024, arXiv:2310.02304.
- [28] M. Pluhacek, A. Kazikova, T. Kadavy, A. Viktorin, and R. Senkerik, "Leveraging large language models for the generation of novel metaheuristic optimization algorithms," in *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, ser. GECCO '23 Companion. New York, NY, USA: Association for Computing Machinery, 2023, p. 1812–1820. [Online]. Available: <https://doi.org/10.1145/3583133.3596401>
- [29] J. de Nobel, F. Ye, D. Vermetten, H. Wang, C. Doerr, and T. Bäck, "IOHxperimenter: Benchmarking platform for iterative optimization heuristics," *arXiv e-prints:2111.04077*, nov 2021. [Online]. Available: <https://arxiv.org/abs/2111.04077>
- [30] Anonymous and Anonymous, "LLaMEA," May 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.11358117>
- [31] R. Cheng, C. He, Y. Jin, and X. Yao, "Model-based evolutionary algorithms: a short survey," *Complex & Intelligent Systems*, vol. 4, no. 4, pp. 283–292, 2018.
- [32] M. I. E. Khaldi and A. Draa, "Surrogate-assisted evolutionary optimisation: a novel blueprint and a state of the art survey," *Evolutionary Intelligence*, pp. 1–31, 2023.
- [33] OpenAI, "Chatgpt-3.5-turbo," <https://platform.openai.com/docs/models/gpt-3-5-turbo>, 2022, version 0125, Accessed: 2024-05-01.
- [34] ———, "Chatgpt-4-turbo," <https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4>, 2023, version 2024-04-09, Accessed: 2024-05-01.
- [35] ———, "Chatgpt-4o," <https://platform.openai.com/docs/models/gpt-4o>, 2023, version: 2024-05-13, Accessed: 2024-05-14.
- [36] N. van Stein, D. Vermetten, A. V. Kononova, and T. Bäck, "Explainable benchmarking for iterative optimization heuristics," 2024, arXiv:2401.17842.
- [37] M. López-Ibáñez, D. Vermetten, J. Dreо, and C. Doerr, "Using the empirical attainment function for analyzing single-objective black-box optimization algorithms," 2024, arXiv:2404.02031.
- [38] N. Hansen, A. Auger, D. Brockhoff, and T. Tušar, "Anytime performance assessment in blackbox optimization benchmarking," *IEEE Transactions on Evolutionary Computation*, vol. 26, no. 6, pp. 1293–1305, 2022.
- [39] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/01621459.1989.10478785>
- [40] D. Brockhoff, A. Auger, and N. Hansen, "Comparing mirrored mutations and active covariance matrix adaptation in the ipop-cma-es on the noiseless bbob testbed," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, 2012, pp. 297–304.
- [41] N. Hansen and R. Ros, "Black-box optimization benchmarking of newuoа compared to bipop-cma-es: on the bbob noiseless testbed," in *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, 2010, pp. 1519–1526.
- [42] P. Pošík and V. Klemš, "Benchmarking the differential evolution with adaptive encoding on noiseless functions," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, 2012, pp. 189–196.
- [43] J. Zhang and A. C. Sanderson, "JADE: Adaptive differential evolution with optional external archive," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 945–958, 2009.
- [44] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," 2023, arXiv:2311.05232.
- [45] H. Wang, D. Vermetten, F. Ye, C. Doerr, and T. Bäck, "IOHanalyzer: Detailed performance analyses for iterative optimization heuristics," *ACM Transactions on Evolutionary Learning and Optimization*, vol. 2, no. 1, apr 2022. [Online]. Available: <https://doi.org/10.1145/3510426>
- [46] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, "Is temperature the creativity parameter of large language models?" 2024, arXiv:2405.00492.

APPENDIX
SUPPLEMENTARY MATERIAL

TABLE I
THE 24 BBOB NOISELESS FUNCTIONS GROUPED IN FIVE FUNCTION CATEGORIES, SEE [24] FOR DETAILS.

Group	Function ID	Function Name
1	f1	Sphere Function
	f2	Separable Ellipsoidal Function
	f3	Rastrigin Function
	f4	Büche-Rastrigin Function
	f5	Linear Slope
2	f6	Attractive Sector Function
	f7	Step Ellipsoidal Function
	f8	Rosenbrock Function, original
	f9	Rosenbrock Function, rotated
3	f10	Ellipsoidal Function
	f11	Discus Function
	f12	Bent Cigar Function
	f13	Sharp Ridge Function
	f14	Different Powers Function
4	f15	Rastrigin Function
	f16	Weierstrass Function
	f17	Schaffer's F7 Function
	f18	Schaffer's F7 Function, moderately ill-conditioned
	f19	Composite Griewank-Rosenbrock Function F8F2
5	f20	Schwefel Function
	f21	Gallagher's Gaussian 101-me Peaks Function
	f22	Gallagher's Gaussian 21-hi Peaks Function
	f23	Katsuura Function
	f24	Lunacek bi-Rastrigin Function

TABLE II
AREA UNDER THE EAF CURVE SCORES (AUC) FOR THE BEST 3 ALGORITHMS PER MODEL IN 5d, HIGHER AUC SCORES STANDS FOR A BETTER ANYTIME PERFORMANCE.

ID	AUC
RADEA	0.760556363533174
ERADS_QuantumFluxUltraRefined	0.693447883295973
CMA (baseline)	0.658860970620932
BIPOP-CMA-ES (baseline)	0.658331593610919
EnhancedHybridCMAESDE	0.651861506582319
QuantumDifferentialParticleOptimizerWithElitism	0.649117907096982
EnhancedFireworkAlgorithmWithLocalSearch	0.636800670169606
AdaptiveHybridDEPSOWithDynamicRestart	0.636048342802281
EnhancedDynamicPrecisionBalancedEvolution	0.586305304809909
AdaptiveDifferentialEvolutionHarmonySearch	0.578480363276591
DE (baseline)	0.571389863508737
QPSO	0.543594487232903

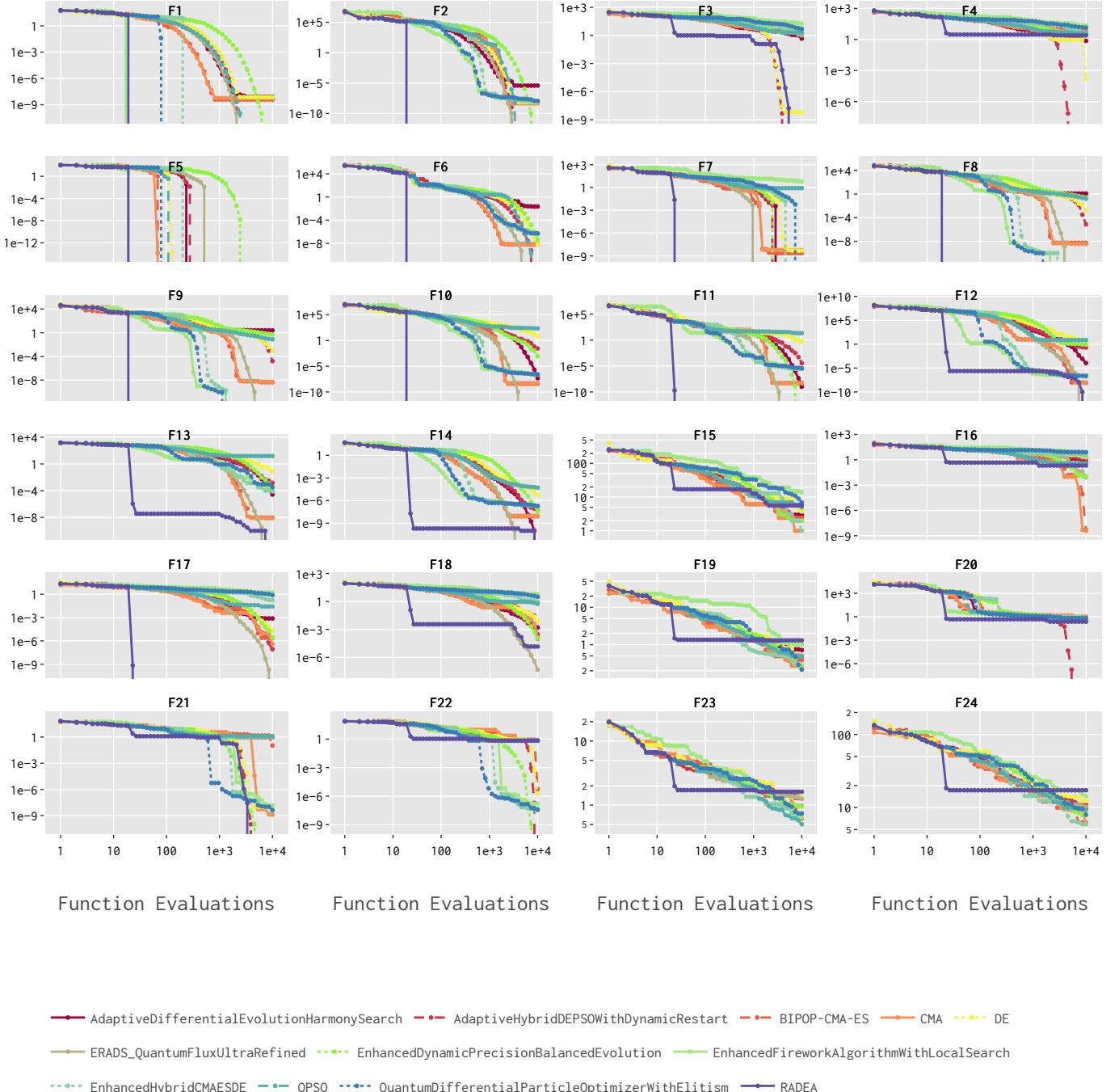


Fig. 10. Median best-so-far function value (y -axis) over the $2000 \times d$ function evaluations (x -axis) per BBOB function for the best algorithm per configuration and the baselines; CMA-ES, DE and BIPOP-CMA-ES (for $d = 5$).

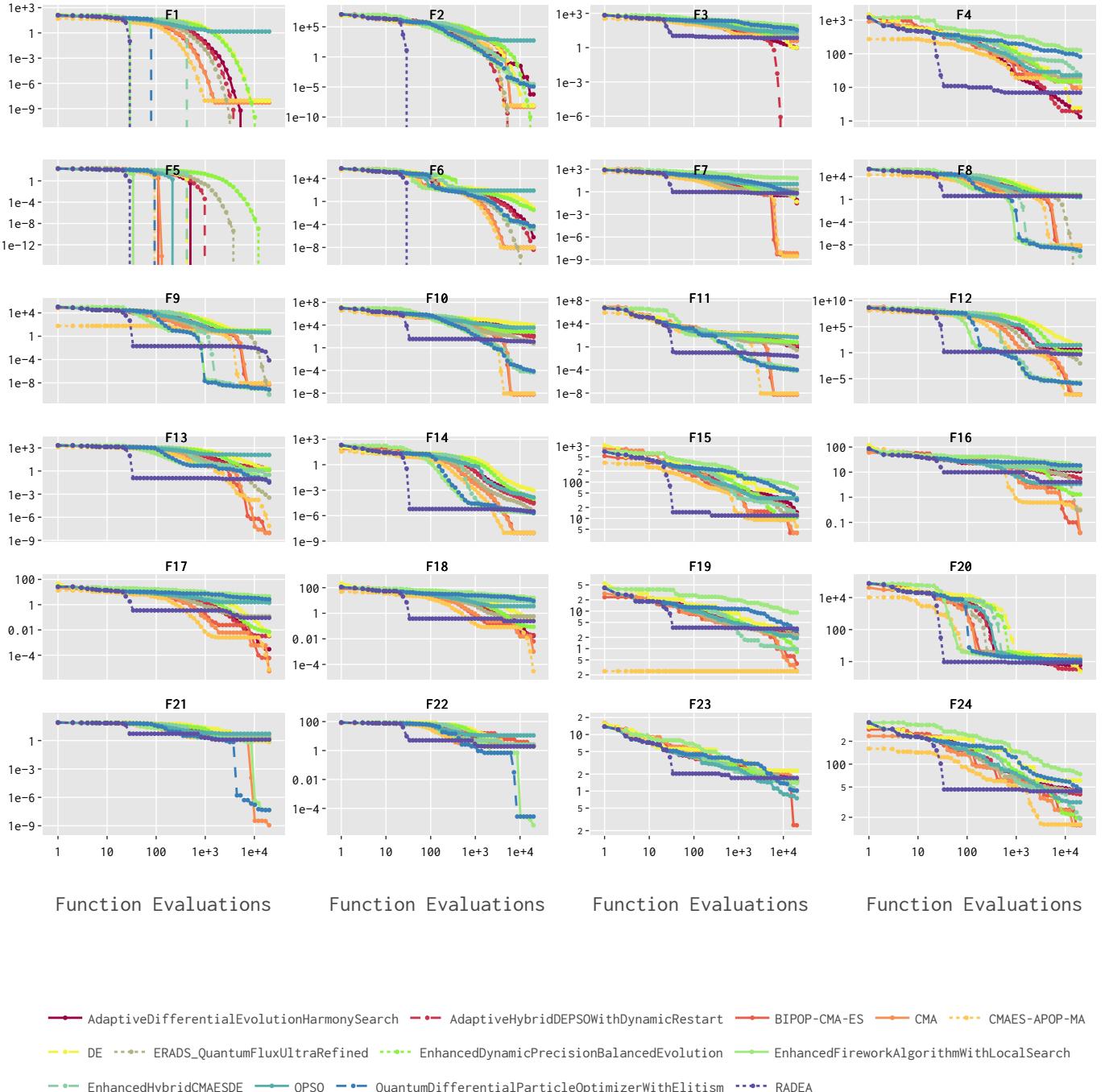


Fig. 11. Median best-so-far function value (y -axis) over the $2000 \times d$ function evaluations (x -axis) per BBOB function for the best algorithm per configuration and the baselines; CMA-ES, DE and BIPOP-CMA-ES (for $d = 10$).

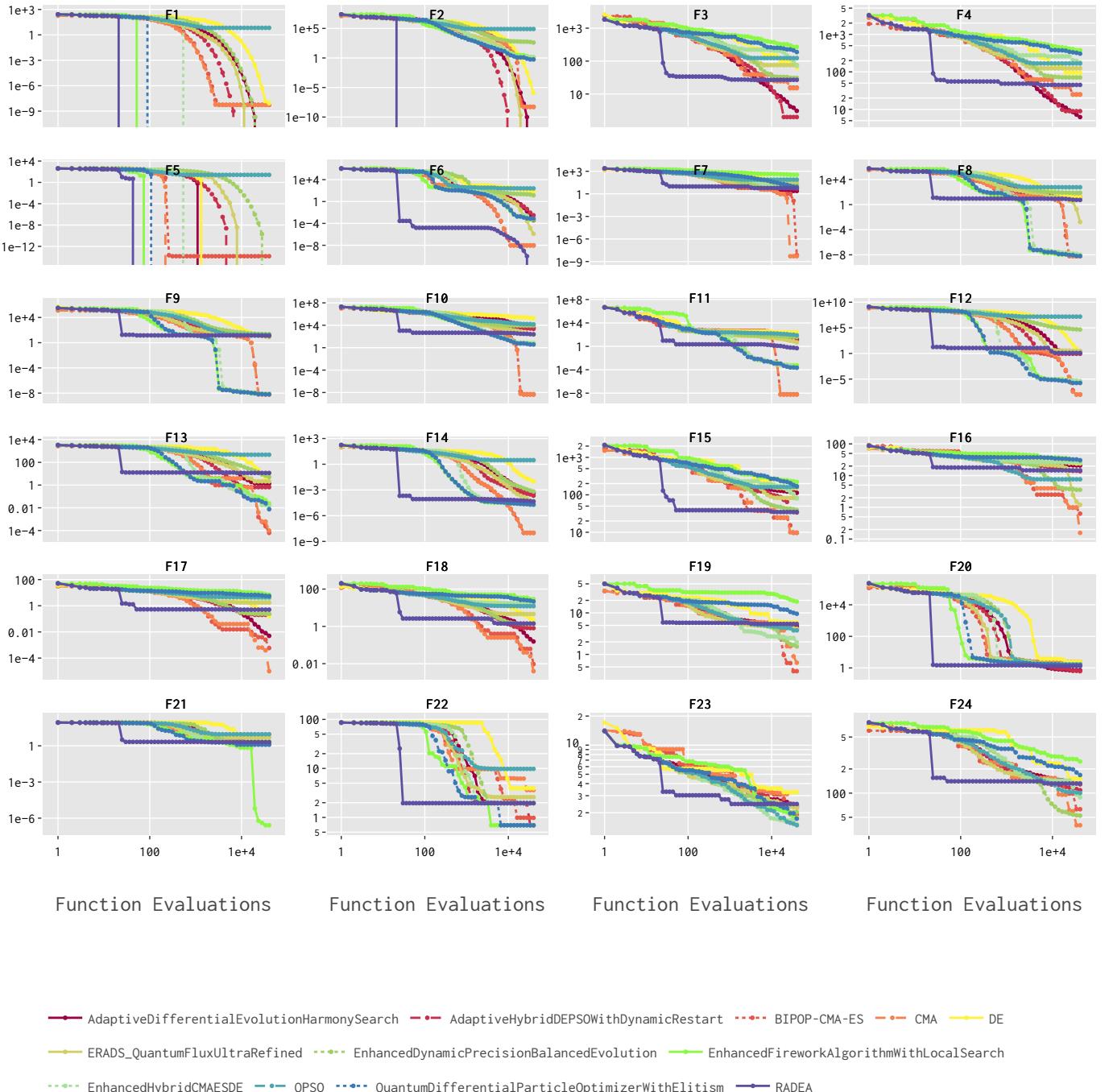


Fig. 12. Median best-so-far function value (y -axis) over the $2000 \times d$ function evaluations (x -axis) per BBOB function for the best algorithm per configuration and the baselines; CMA-ES, DE and BIPOP-CMA-ES (for $d = 20$).