Google DeepMind

AlphaEvolve: A coding agent for scientific and algorithmic discovery

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*
Google DeepMind[1]

In this white paper, we present *AlphaEvolve*, an evolutionary coding agent that substantially enhances capabilities of state-of-the-art LLMs on highly challenging tasks such as tackling open scientific problems or optimizing critical pieces of computational infrastructure. *AlphaEvolve* orchestrates an autonomous pipeline of LLMs, whose task is to improve an algorithm by making direct changes to the code. Using an evolutionary approach, continuously receiving feedback from one or more evaluators, *AlphaEvolve* iteratively improves the algorithm, potentially leading to new scientific and practical discoveries. We demonstrate the broad applicability of this approach by applying it to a number of important computational problems. When applied to optimizing critical components of large-scale computational stacks at Google, *AlphaEvolve* developed a more efficient scheduling algorithm for data centers, found a functionally equivalent simplification in the circuit design of hardware accelerators, and accelerated the training of the LLM underpinning *AlphaEvolve* itself. Furthermore, *AlphaEvolve* discovered novel, provably correct algorithms that surpass state-of-the-art solutions on a spectrum of problems in mathematics and computer science, significantly expanding the scope of prior automated discovery methods (Romera-Paredes et al., 2023). Notably, *AlphaEvolve* developed a search algorithm that found a procedure to multiply two 4×4 complex-valued matrices using 48 scalar multiplications; offering the first improvement, after 56 years, over Strassen's algorithm in this setting. We believe *AlphaEvolve* and coding agents like it can have a significant impact in improving solutions of problems across many areas of science and computation.

1. Introduction

Discovering new high-value knowledge, such as making a novel scientific discovery or developing a commercially valuable algorithm, generally requires a prolonged process of ideation, exploration, backtracking on unpromising hypotheses, experimentation, and validation. There has been much recent interest in using large language models (LLMs) to automate significant parts of this process. Hopes of success here are driven by the breathtaking power of recent LLMs [32, 76], which can enhance their capabilities using test-time compute, and the rise of *agents* that combine language generation and action [88, 114]. These advances have improved performance across a range of established benchmarks and accelerated discovery-oriented tasks like hypothesis generation [34] and experiment design [7, 43]. However, getting LLM pipelines all the way to making entirely new scientific or practical discoveries remains challenging.

In this white paper, we present an LLM code superoptimization agent, called *AlphaEvolve*, that takes on this challenge using a combination of evolutionary computation and LLM-based code generation. *AlphaEvolve* focuses on the broad spectrum of scientific and engineering

[1]See Acknowledgments and Author information section. *Equal contributions.

discovery problems in which the candidates of discovery can be automatically evaluated. It represents the candidates (for example, new mathematical objects or practical heuristics) as algorithms and uses a set of LLMs to generate, critique, and evolve a pool of such algorithms. The LLM-directed evolution process is grounded using code execution and automatic evaluation. This evaluation mechanism allows *AlphaEvolve* to avoid any incorrect suggestions from the base LLM [44].

The evolutionary process in *AlphaEvolve* leverages modern LLMs' ability to respond to feedback, enabling the discovery of candidates that are substantially different from the initial candidate pool in syntax and function. It is applicable both to problems where discovering new algorithms is the intrinsic goal, as well as to the broad range of problems where the solution of interest is not an algorithm itself but an algorithm can *describe* how that solution is to be constructed or found. In the latter case, discovering the algorithm is only an instrumental goal, but it turns out to be a surprisingly effective strategy compared to searching for the solution directly [83].

The idea of combining evolutionary methods with coding LLMs has been previously explored in various specialized settings. In particular, *AlphaEvolve* is a substantial enhancement of *FunSearch* [83] (see Table 1), which used LLM-guided evolution to discover heuristics in order to construct novel mathematical objects or to drive the operation of online algorithms. Also, related approaches have been used in tasks such as discovering policies for simulated robots [57], symbolic regression [35, 89], and the synthesis of heuristic functions for combinatorial optimization [63]. In contrast to these systems, *AlphaEvolve* leverages state-of-the-art (SOTA) LLMs to evolve large pieces of code that implement complex algorithms spanning multiple functions and components. As a result, it is able to go significantly beyond its predecessors in scale and generality.

FunSearch [83]	*AlphaEvolve*
evolves single function	evolves entire code file
evolves up to 10-20 lines of code	evolves up to hundreds of lines of code
evolves code in Python	evolves any language
needs fast evaluation ($\leq$ 20min on 1 CPU)	can evaluate for hours, in parallel, on accelerators
millions of LLM samples used	thousands of LLM samples suffice
small LLMs used; no benefit from larger	benefits from SOTA LLMs
minimal context (only previous solutions)	rich context and feedback in prompts
optimizes single metric	can simultaneously optimize multiple metrics

Table 1 | Capabilities and typical behaviours of *AlphaEvolve* and our previous agent.

While the use of an automated evaluation metric offers *AlphaEvolve* a key advantage, it is also a limitation—in particular, it puts tasks that require manual experimentation out of our scope. Because problems in mathematics, computer science, and system optimization typically permit automated evaluation metrics, our efforts on *AlphaEvolve* focus on these domains. Specifically, we use *AlphaEvolve* to make progress on several well-known open problems in algorithm design and constructive mathematics, as well as the optimization of critical layers in the large-scale computation stacks at Google.

Within algorithm design, we consider the fundamental problem of discovering fast algorithms for multiplying matrices, a problem to which a more specialized AI approach had been applied previously [26]. Despite being general-purpose, *AlphaEvolve* goes beyond [26], improving the SOTA for 14 matrix multiplication algorithms; notably, for 4×4 matrices, *AlphaEvolve* improves Strassen (1969)'s algorithm by discovering an algorithm using 48 multiplications to multiply 4×4 complex-valued matrices.[2]

In mathematics, we consider a broad range of open problems on which one can make progress by discovering constructions (objects) with better properties than all previously known constructions, according to given mathematical definitions. We apply *AlphaEvolve* to a large number (over 50) of such problems and match the best known constructions on ~75% of them (in many cases these constructions are likely to already be optimal). On ~20% of the problems, *AlphaEvolve* surpasses the SOTA and discovers new, provably better constructions. This includes an improvement on the Minimum Overlap Problem set by Erdős [25] and an improved construction on the Kissing Numbers problem in 11 dimensions [8, 31].

Finally, we use *AlphaEvolve* in four engineering problems spanning different layers of Google's compute stack: discovering scheduling heuristics for Google's cluster management system, optimizing matrix-multiplication kernels used to train LLMs, optimizing arithmetic circuits used within TPUs, and optimizing the runtime of attention in Transformers. Because these components are run repeatedly over a long period of time, any improvements are highly valuable.

2. *AlphaEvolve*

AlphaEvolve is a coding agent that orchestrates an autonomous pipeline of computations including queries to LLMs, and produces algorithms that address a user-specified task. At a high level, the orchestrating procedure is an evolutionary algorithm that gradually develops programs that improve the score on the automated evaluation metrics associated with the task. A high-level overview of *AlphaEvolve* is shown in Figure 1, and Figure 2 gives an expanded view.

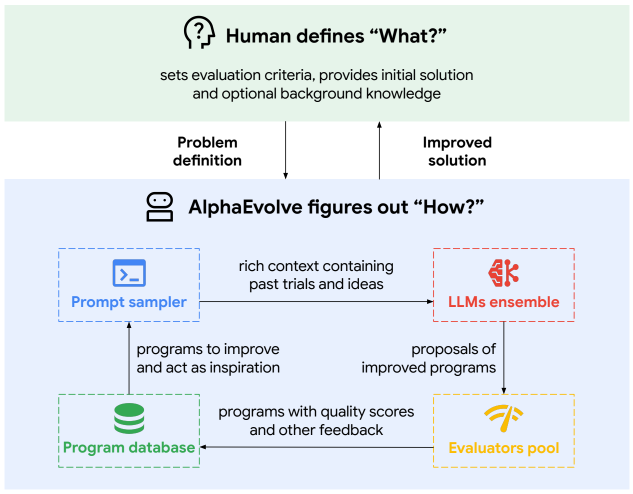
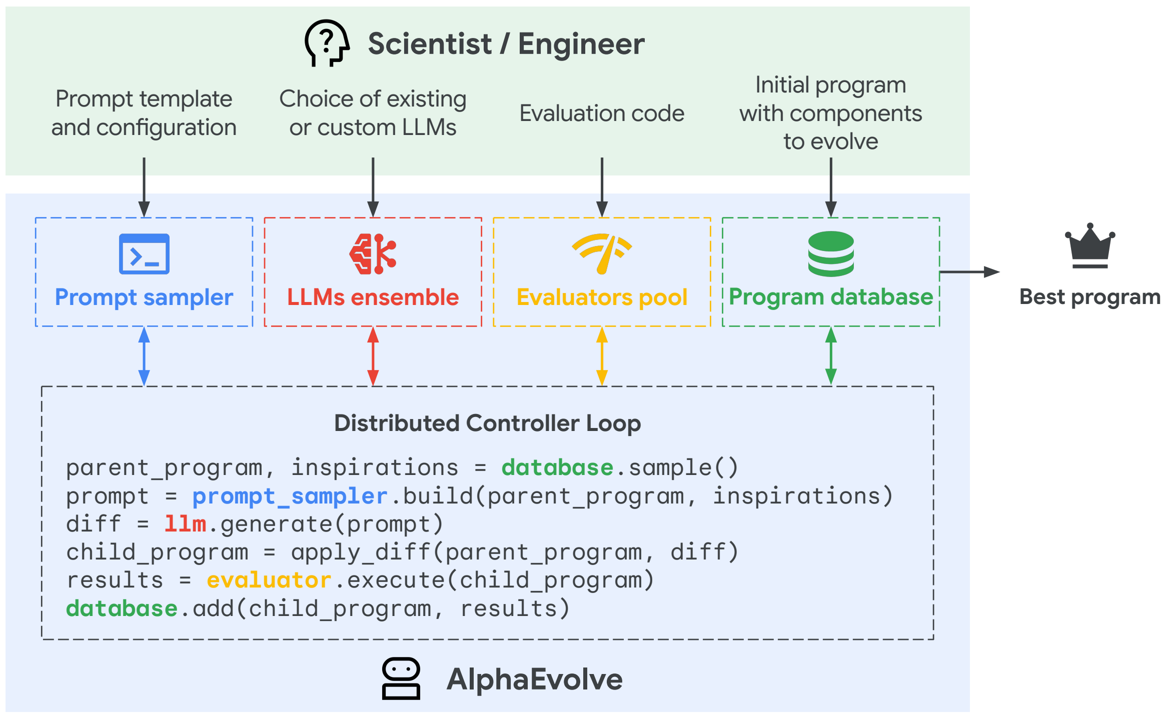

Human defines "What?"
sets evaluation criteria, provides initial solution and optional background knowledge
Problem definition
Improved solution
AlphaEvolve figures out "How?"
Prompt sampler
rich context containing past trials and ideas
LLMs ensemble
programs to improve and act as inspiration
proposals of improved programs
Program database
programs with quality scores and other feedback
Evaluators pool

Figure 1 | *AlphaEvolve* high-level overview.

2.1. Task specification

Evaluation. Since *AlphaEvolve* tackles problems with machine-gradeable solutions, the user must provide a mechanism for automatically assessing generated solutions. This mechanism takes the form of a function h mapping a solution to a set of scalar evaluation metrics. By convention, these metrics are maximized. In our current setup, h is typically implemented

[2]These discovered algorithms as well as our other new mathematical results can be found at https://colab.research.google.com/github/google-deepmind/alphaevolve_results/blob/master/mathematical_results.ipynb.


Scientist / Engineer
Prompt template and configuration
Choice of existing or custom LLMs
Evaluation code
Initial program with components to evolve
Prompt sampler
LLMs ensemble
Evaluators pool
Program database
Best program
Distributed Controller Loop
AlphaEvolve

```
parent_program, inspirations = database.sample()
prompt = prompt_sampler.build(parent_program, inspirations)
diff = llm.generate(prompt)
child_program = apply_diff(parent_program, diff)
results = evaluator.execute(child_program)
database.add(child_program, results)
```

Figure 2 | Expanded view of the *AlphaEvolve* discovery process. The user provides an initial program (with components to evolve marked), evaluation code, and optional configurations (Section 2.1). *AlphaEvolve* then initiates an evolutionary loop. The *Prompt sampler* uses programs from the *Program database* to construct rich prompts (Section 2.2). Given these prompts, the *LLMs* generate code modifications (diffs), which are applied to create new programs (Section 2.3). These are then scored by *Evaluators* (Section 2.4), and promising solutions are registered back into the *Program database* (Section 2.5), driving the iterative discovery of better and better programs.

as a Python function, called `evaluate`, with a fixed input/output signature, returning a dictionary of scalars.

Depending on the application, executing this function may take only seconds on a single device or spawn extensive computations. For mathematical problems, the function h is typically very simple. For example, when wishing to find largest possible graphs satisfying a given property, h invokes the evolved code to generate a graph, checks whether the property holds, and then simply returns the size of the graph as the score. In more complicated cases, the function h might involve performing an evolved search algorithm, or training and evaluating a machine learning model.

API. To support evolving multiple components across a codebase, *AlphaEvolve* exposes an input API where blocks of code can be annotated as to-be-evolved-by-the-system; see Figure 3a for an illustration. This design facilitates integrating it with existing codebases while requiring only minimal changes, simply by adding special markers (`# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END`) as comments into the code.

Any user-provided code inside such evolution blocks serves as the initial solution to be improved by *AlphaEvolve*, and the rest of the code forms a skeleton that ties the evolved pieces together, so that they can be invoked from `evaluate`. While this initial implementation must be complete, it can be rudimentary—for instance, consisting of single-line functions that return constants of the appropriate types.

Flexibility in choosing the abstraction. *AlphaEvolve* can be applied to the same problem in very different ways—especially when the evolved programs are not the final output but a means to discover solutions. For example, *AlphaEvolve* can evolve the solution in raw string representation (as in classical evolutionary algorithms); evolve a function of a definite form that specifies how to construct the solution from scratch (the approach taken in [83]); evolve a bespoke search algorithm to find the solution within some fixed compute budget; or even co-evolve intermediate solutions and search algorithms together, such that each search algorithm is specifically tailored to further improve upon a particular intermediate solution.

We find that different levels of abstraction work better for different problems. For example, we hypothesize that for problems with highly symmetric solutions it is advantageous to evolve constructor functions as these tend to be more concise [83], whereas for problems with non-symmetric solutions it works better to evolve customized search algorithms.

2.2. Prompt sampling

As *AlphaEvolve* leverages SOTA LLMs, it supports various types of customization and providing long contexts as part of the primary evolution prompt. This prompt comprises multiple previously discovered solutions sampled from the program database, as well as system instructions on how to propose changes to a particular solution. Beyond these key ingredients, users can further tailor prompts to their specific needs in different ways, such as the following.

- *Explicit context*: details about the problem being solved, such as fixed human-written instructions, equations, code snippets, or relevant literature (e.g., pdf files).
- *Stochastic formatting*: template placeholders with human-provided alternatives for increased diversity, instantiated using probability distributions provided in a separate config file.
- *Rendered evaluation results*: usually this will include a program, the result of executing that program, and the scores assigned by the `evaluate` function.
- *Meta prompt evolution*: instructions and context suggested by the LLM itself in an additional prompt-generation step, co-evolved in a separate database analogous to the solution programs.

2.3. Creative generation

To drive the evolutionary procedure, *AlphaEvolve* leverages the capabilities of SOTA LLMs, whose principal role is to digest information about previously developed solutions and propose new, diverse ways to improve the solutions. Although *AlphaEvolve* is model-agnostic, in ablations we observe that *AlphaEvolve* performs increasingly better as the underlying LLM improves (see Section 4).

```python
# EVOLVE-BLOCK START
"""Image classification experiment in jaxline."""

import jax
...
# EVOLVE-BLOCK-END

...

# EVOLVE-BLOCK-START
class ConvNet(hk.Module):
  def __init__(self, num_classes): ...
  def __call__(self, inputs, is_training): ...


def sweep():
  return hyper.zipit([...])
# EVOLVE-BLOCK-END

...

def evaluate(eval_inputs) -> dict[str, float]:
  ...
  return metrics
```

(a)

```
The current model uses a simple ResNet architecture with only
three ResNet blocks. We can improve its performance by
increasing the model capacity and adding regularization. This
will allow the model to learn more complex features and
generalize better to unseen data. We also add weight decay to
the optimizer to further regularize the model and prevent
overfitting. AdamW is generally a better choice than Adam,
especially with weight decay.


<<<<<<< SEARCH
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels * 2, stride=2)
    self._block3 = ResNetBlock(num_channels * 4, stride=2)
=======
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels, stride=1)
    self._block3 = ResNetBlock(num_channels * 2, stride=2)
    self._block4 = ResNetBlock(num_channels * 2, stride=1)
    self._block5 = ResNetBlock(num_channels * 4, stride=2)
    self._block6 = ResNetBlock(num_channels * 4, stride=1)
>>>>>>> REPLACE
<<<<<<< SEARCH
  def optimizer(self, learning_rate):
    return optax.adam(learning_rate)
=======
  def optimizer(self, learning_rate):
    return optax.adamw(learning_rate, weight_decay=1e-4)
>>>>>>> REPLACE
```

(c)

```
Act as an expert software developer. Your task is to iteratively
improve the provided codebase. [...]

- Prior programs

Previously we found that the following programs performed well
on the task at hand:

top_1_acc: 0.796; neg_eval_log_loss: 0.230; average_score: 0.513

"""Image classification experiment in jaxline."""
[...]
class ConvNet(hk.Module):
  """Network."""

  def __init__(self, num_channels=32, num_output_classess=10):
    super().__init__()
    self._conv1 = hk.Conv2D(num_channels, kernel_shape=3)
    self._conv2 = hk.Conv2D(num_channels * 2, kernel_shape=3)
    self._conv3 = hk.Conv2D(num_channels * 4, kernel_shape=3)
    self._logits_module = hk.Linear(num_output_classes)
[...]


- Current program

Here is the current program we are trying to improve (you will
need to propose a modification to it below).

top_1_acc: 0.862; neg_eval_log_loss: 0.387; average_score: 0.624

"""Image classification experiment in jaxline."""
[...]
class ConvNet(hk.Module):
  """Network."""

  def __init__(self, num_channels=32, num_output_classes=10):
    super().__init__()
    self._conv1 = hk.Conv2D(num_channels, kernel_shape=3)
    self._block1 = ResNetBlock(num_channels)
    self._block2 = ResNetBlock(num_channels * 2, stride=2)
    self._block3 = ResNetBlock(num_channels * 4, stride=2)
    self._logits_module = hk.Linear(num_output_classes)
[...]

SEARCH/REPLACE block rules:
[...]

Make sure that the changes you propose are consistent with each
other. For example, if you refer to a new config variable
somewhere, you should also propose a change to add that
variable.

Example:
[...]

Task
Suggest a new idea to improve the code that is inspired by your
expert knowledge of optimization and machine learning.

Describe each change with a SEARCH/REPLACE block.
```

(b)

Figure 3 | Illustrative example of applying *AlphaEvolve* to evolving a supervised learning pipeline. All snippets are abbreviated, with ellipsis (...) indicating skipped lines. (a) The user-provided file with blocks marked for evolution, and the special `evaluate` function that can be invoked to score the current version of the code. (b) Example of an assembled prompt to be provided to the LLMs. (c) Example output generated by the LLM. The proposed diffs in (c) will be applied to the "current program" shown in the prompt (b), and the resulting modified program will then be sent to the evaluators. The evaluators will invoke the `evaluate` function from (a) in order to obtain the scores of the newly proposed program.

Output format. When *AlphaEvolve* asks an LLM to modify existing code, especially within larger codebases, it requests the changes to be provided as a sequence of diff blocks in a specific format:

```
<<<<<<< SEARCH
  # Original code block to be found and replaced
=======
  # New code block to replace the original
>>>>>>> REPLACE
```

Here, the code between `<<<<<<< SEARCH` and `=======` is the exact segment to match in the current program version. The code between `=======` and `>>>>>>> REPLACE` is the new segment that will replace the original one. This allows for targeted updates to specific parts of the code.

In cases where the code being evolved is very short, or when a complete rewrite is more appropriate than a small modification, *AlphaEvolve* can be configured to instruct the LLM to output the entire code block directly, rather than using the diff format.

Models used. *AlphaEvolve* employs an ensemble of large language models. Specifically, we utilize a combination of Gemini 2.0 Flash and Gemini 2.0 Pro. This ensemble approach allows us to balance computational throughput with the quality of generated solutions. Gemini 2.0 Flash, with its lower latency, enables a higher rate of candidate generation, increasing the number of ideas explored per unit of time. Concurrently, Gemini 2.0 Pro, possessing greater capabilities, provides occasional, higher-quality suggestions that can significantly advance the evolutionary search and potentially lead to breakthroughs. This strategic mix optimizes the overall discovery process by maximizing the volume of evaluated ideas while retaining the potential for substantial improvements driven by the more powerful model.

2.4. Evaluation

To track *AlphaEvolve*'s progress and to select which ideas to propagate in future generations, each new solution proposed by the LLMs is automatically evaluated. In principle, this process amounts to simply executing the user-provided evaluation function h on the generated solution. In practice, *AlphaEvolve* supports optional mechanisms to make this evaluation more flexible and more efficient:

- *Evaluation cascade (hypothesis testing)*: the user can specify ensembles of test cases of increasing difficulty, such that new solutions are evaluated on the next stage only if they achieve sufficiently promising results in all earlier stages. This helps to prune out less promising solutions more quickly. Moreover, new solutions are initially evaluated on a small scale before being subjected to the main test cases, to filter out faulty programs early.
- *LLM-generated feedback*: in some applications, desirable solutions have certain characteristics that are difficult to capture precisely in the user-provided evaluation function

h; for example, simplicity of the discovered program. These properties can be graded using separate LLM calls and added to the dictionary of scores to steer evolution, or they can be used to discard solutions when a criterion is not fulfilled.
- *Parallelized evaluation*: the sample efficiency of *AlphaEvolve* makes it feasible to spend on the order of 100 compute-hours to evaluate any new solution. However, unless individual evaluations are parallelized to reduce their wall-clock duration, this can slow down the rate at which new generations appear, limiting the ability of the evolutionary algorithm to apply several consecutive mutations. In many applications, evaluation is embarrassingly parallel (for example, running a search algorithm from multiple randomized initializations), allowing *AlphaEvolve* to distribute this work through asynchronous calls to an evaluation cluster.

Multiple scores. *AlphaEvolve* allows for optimizing multiple user-provided scores, i.e., evolving objects that achieve a high score under one or multiple evaluation metrics. This has both an intrinsic and instrumental value. While in multiple applications we genuinely care about developing solutions for multiple evaluation metrics (or one solution that is strong on all of them simultaneously), we find that even if one metric is of particular interest, optimizing for multiple metrics often improves results for the single target metric. Perhaps this occurs because programs excelling under different evaluation criteria often possess distinct structures or logic and, by incorporating examples of these diverse, high-performing programs—each representing a different definition of "good"—into the prompts provided to the language model, we can stimulate the generation of more varied candidate solutions, increasing the chances of discovering novel approaches that are highly effective for the target metric.

2.5. Evolution

During its evolutionary procedure, *AlphaEvolve* continually generates a growing number of solutions with evaluation results (scores and program outputs) attached to them. These solutions are stored in an evolutionary database, the primary goal of which is to optimally resurface previously explored ideas in future generations. A key challenge in designing such databases is balancing exploration and exploitation, to continuously improve the best programs while maintaining diversity to encourage exploration of the entire search space. In *AlphaEvolve*, the evolutionary database implements an algorithm that is inspired by a combination of the MAP elites algorithm [74] and island-based population models [83, 97].

2.6. Distributed pipeline

AlphaEvolve is implemented as an asynchronous computational pipeline (using the `asyncio` Python library) in which many computations are run concurrently, with each computation blocking (waiting) whenever its next step relies on the result of another, yet unfinished computation. More specifically, the asynchronous pipeline comprises a controller, LLM samplers, and evaluation nodes. The entire pipeline is optimized for throughput (rather than the speed of any one particular computation), in order to maximize the number of ideas that can be proposed and evaluated within a specific overall computation budget.

$\langle m, n, p\rangle$	best known [reference]	*AlphaEvolve*
$\langle 2,4,5\rangle$	33 [42]	**32**
$\langle 2,4,7\rangle$	46 [93]	**45**
$\langle 2,4,8\rangle$	52 [93]	**51**
$\langle 2,5,6\rangle$	48 [93]	**47**
$\langle 3,3,3\rangle$	23 [52]	23
$\langle 3,4,6\rangle$	56 [48]	**54**
$\langle 3,4,7\rangle$	66 [91]	**63**
$\langle 3,4,8\rangle$	75 [91]	**74**
$\langle 3,5,6\rangle$	70 [48]	**68**
$\langle 3,5,7\rangle$	82 [91]	**80**
$\langle 4,4,4\rangle$	49 [95]	**48**
$\langle 4,4,5\rangle$	62 [47]	**61**
$\langle 4,4,7\rangle$	87 [93]	**85**
$\langle 4,4,8\rangle$	98 [95]	**96**
$\langle 4,5,6\rangle$	93 [48]	**90**
$\langle 5,5,5\rangle$	93 [72]	93

Table 2 | Upper bounds on the rank of the tensor $\langle m, n, p\rangle$ representing the product of an $m \times n$ matrix and an $n \times p$ matrix, i.e. the number of scalar multiplications required to compute this matrix product. Beyond the examples shown here, for all parameters $m, n, p \leq 5$, *AlphaEvolve* either matched or surpassed the best known solutions, and provided exact algorithms (see Table 3 in appendix for full results). For $\langle 3,4,7\rangle$, $\langle 4,4,4\rangle$, and $\langle 4,4,8\rangle$, the algorithms discovered by *AlphaEvolve* use complex-valued multiplications which can be used for exact multiplication of complex or real-valued matrices. The decompositions shown in this table can be found in the accompanying Google Colab.

3. Results

3.1. Faster matrix multiplication via finding novel algorithms for tensor decomposition

From accelerating machine learning computations to enabling realistic computer graphics, matrix multiplication serves as a fundamental operation underpinning numerous critical algorithms and applications within computer science. Since the pioneering work of Strassen [95], it has been known that a rich space of algorithms for multiplying two matrices can be represented as decompositions of a given 3D tensor into rank-one tensors. The rank (number of terms) of the decomposition exactly specifies the number of scalar multiplications needed to compute the matrix product. Hence, to develop faster matrix multiplication algorithms one needs to find low-rank decompositions of particular tensors. This problem has been tackled with many approaches, from specialized alternating least squares solvers [93] to deep reinforcement learning [26] and custom search algorithms [47]; yet, despite decades of effort, even for the simple case of multiplying two 3×3 matrices, the minimum achievable rank is not known, showcasing the difficulty of the problem.

Starting from the problem description and a standard gradient-based algorithm (including an initializer, a reconstruction loss function, and an Adam optimizer [50]), *AlphaEvolve* is able to develop sophisticated tensor decomposition algorithms that outperform existing approaches. To evaluate each evolved program, we choose a set of matrix multiplication targets and run the algorithm, initialized with multiple random seeds using the evaluation cascade described in Section 2.4. The performance is then measured as the best (lowest) rank achieved on each target as well as the fraction of seeds that achieved this rank, providing a signal for *AlphaEvolve* to hill-climb. To ensure the exactness of the decomposition and avoid any potential numerical error, when evaluating, we round each element to the nearest integer or the nearest half-integer; and, to encourage the algorithm to generate near-integral solutions, we include this request in natural language in the LLM's prompt.

In Table 2, one can see that the various algorithms developed by *AlphaEvolve* improve the state of the art for 14 different matrix multiplication targets. Notably, for multiplying two 4×4 matrices, applying the algorithm of Strassen [95] recursively results in an algorithm with rank (number of scalar multiplications) equal to 49, which works over any field. For the very specific case of multiplying in the field with 2 elements, Fawzi et al. [26] found an algorithm with rank 47. For 56 years, designing an algorithm with rank less than 49 over any field with characteristic 0 was an open problem.[3] *AlphaEvolve* is the first method to find a rank-48 algorithm to multiply two 4×4 complex-valued matrices.

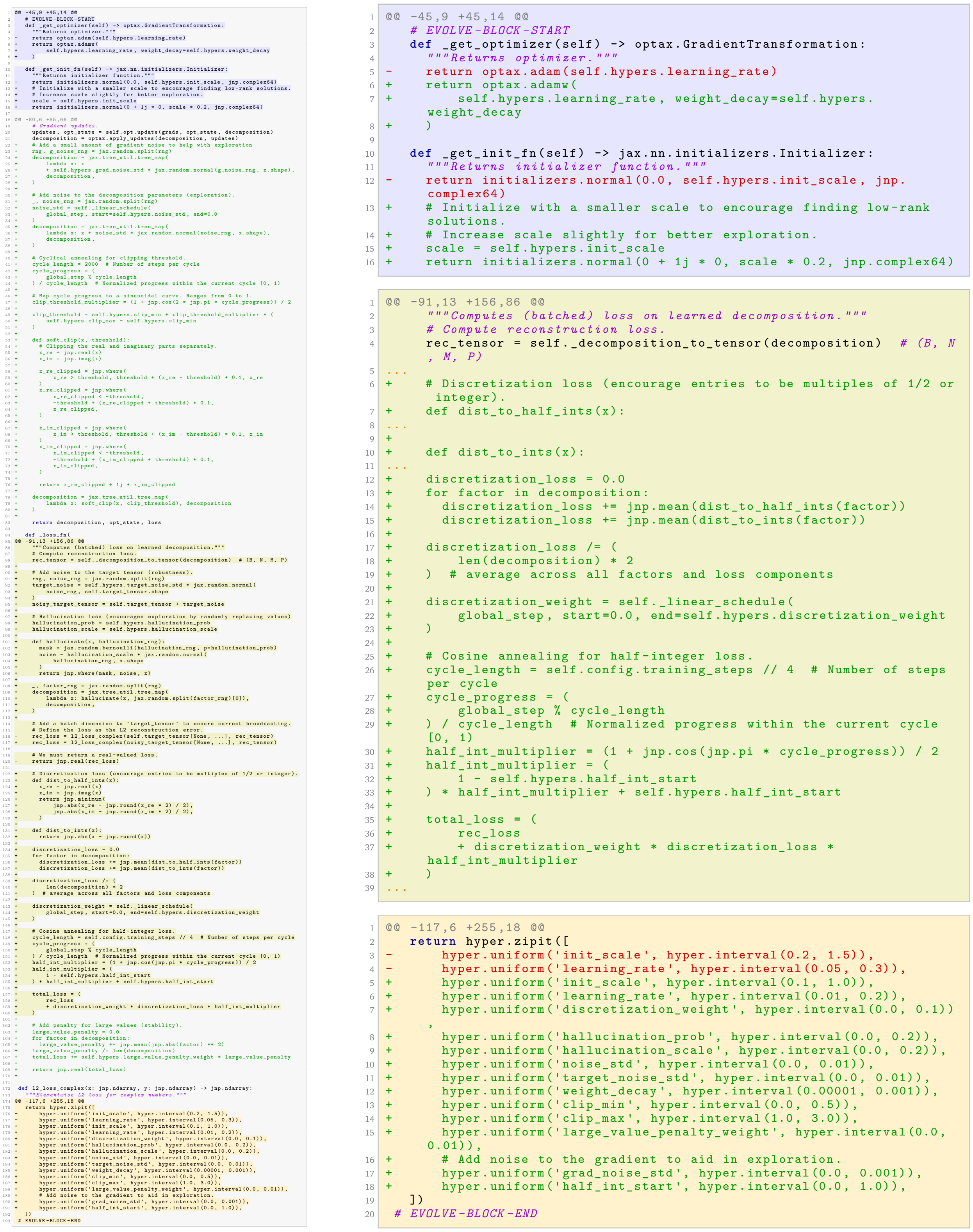
As shown in Figure 4, *AlphaEvolve* makes significant changes to the initial program, introducing several original ideas to design increasingly better algorithms. While most results in Table 2 (including $\langle 4, 4, 4 \rangle$) were obtained from a simple initial program, we found that for some parameters, seeding the initial program with our own ideas (such as adding stochasticity to the evaluation function or using evolutionary approaches) could further boost performance, highlighting the possibility of scientific collaboration between researchers and *AlphaEvolve*.

3.2. Finding tailored search algorithms for a wide range of open mathematical problems

A significant frontier in mathematical research involves discovering objects or *constructions* that possess optimal, or near-optimal, properties according to some measure. Examples range from finding dense packings of geometric shapes [29] to identifying functions or sets satisfying specific combinatorial or analytic constraints (e.g., [39, 40, 70, 104]). Progress often relies on finding a single construction that surpasses all previously known examples, thereby establishing new lower or upper bounds for the optimal value. We demonstrate that *AlphaEvolve* serves as a powerful tool for exploring the vast search space inherent in these problems, successfully tackling a diverse array of open mathematical challenges.

To assess its capabilities, we apply *AlphaEvolve* to a curated set of over 50 mathematical problems, spanning more than five different branches of mathematics, including analysis, combinatorics, number theory, and geometry, evaluated across numerous specific parameter settings (e.g., different dimensions or sizes). In 75% of the cases *AlphaEvolve* rediscovered the best known constructions, and in 20% of the cases it discovered a new object that is better than a previously known best construction, thereby improving the SOTA. In all these cases, the initial starting point was a simple or a random construction. These results underscore *AlphaEvolve*'s broad potential as a versatile tool for mathematical research.

[3]There exist algorithms using fewer than 49 multiplications, but they do not correspond to decompositions of the matrix multiplication tensor, and they cannot be applied recursively to multiplying larger matrices.



```diff
@@ -45,9 +45,14 @@
   # EVOLVE-BLOCK-START
   def _get_optimizer(self) -> optax.GradientTransformation:
     """Returns optimizer."""
-    return optax.adam(self.hypers.learning_rate)
+    return optax.adamw(
+        self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
+    )

   def _get_init_fn(self) -> jax.nn.initializers.Initializer:
     """Returns initializer function."""
-    return initializers.normal(0.0, self.hypers.init_scale, jnp.complex64)
+    # Initialize with a smaller scale to encourage finding low-rank solutions.
+    # Increase scale slightly for better exploration.
+    scale = self.hypers.init_scale
+    return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)

@@ -80,6 +85,66 @@
     # Gradient updates.
     updates, opt_state = self.opt.update(grads, opt_state, decomposition)
     decomposition = optax.apply_updates(decomposition, updates)
+    # Add a small amount of gradient noise to help with exploration
+    rng, g_noise_rng = jax.random.split(rng)
+    decomposition = jax.tree_util.tree_map(
+        lambda x: x
+        + self.hypers.grad_noise_std * jax.random.normal(g_noise_rng, x.shape),
+        decomposition,
+    )
+
+    # Add noise to the decomposition parameters (exploration).
+    _, noise_rng = jax.random.split(rng)
+    noise_std = self._linear_schedule(
+        global_step, start=self.hypers.noise_std, end=0.0
+    )
+    decomposition = jax.tree_util.tree_map(
+        lambda x: x + noise_std * jax.random.normal(noise_rng, x.shape),
+        decomposition,
+    )
+
+    # Cyclical annealing for clipping threshold.
+    cycle_length = 2000  # Number of steps per cycle
+    cycle_progress = (
+        global_step % cycle_length
+    ) / cycle_length  # Normalized progress within the current cycle [0, 1)
+
+    # Map cycle progress to a sinusoidal curve. Ranges from 0 to 1.
+    clip_threshold_multiplier = (1 + jnp.cos(2 * jnp.pi * cycle_progress)) / 2
+
+    clip_threshold = self.hypers.clip_min + clip_threshold_multiplier * (
+        self.hypers.clip_max - self.hypers.clip_min
+    )
+
+    def soft_clip(x, threshold):
+      # Clipping the real and imaginary parts separately.
+      x_re = jnp.real(x)
+      x_im = jnp.imag(x)
+
+      x_re_clipped = jnp.where(
+          x_re > threshold, threshold + (x_re - threshold) * 0.1, x_re
+      )
+      x_re_clipped = jnp.where(
+          x_re_clipped < -threshold,
+          -threshold + (x_re_clipped + threshold) * 0.1,
+          x_re_clipped,
+      )
+
+      x_im_clipped = jnp.where(
+          x_im > threshold, threshold + (x_im - threshold) * 0.1, x_im
+      )
+      x_im_clipped = jnp.where(
+          x_im_clipped < -threshold,
+          -threshold + (x_im_clipped + threshold) * 0.1,
+          x_im_clipped,
+      )
+
+      return x_re_clipped + 1j * x_im_clipped
+
+    decomposition = jax.tree_util.tree_map(
+        lambda x: soft_clip(x, clip_threshold), decomposition
+    )
+
     return decomposition, opt_state, loss

   def _loss_fn(
@@ -91,13 +156,86 @@
     """Computes (batched) loss on learned decomposition."""
     # Compute reconstruction loss.
     rec_tensor = self._decomposition_to_tensor(decomposition)  # (B, N, M, P)
+
+    # Add noise to the target tensor (robustness).
+    rng, noise_rng = jax.random.split(rng)
+    target_noise = self.hypers.target_noise_std * jax.random.normal(
+        noise_rng, self.target_tensor.shape
+    )
+    noisy_target_tensor = self.target_tensor + target_noise
+
+    # Hallucination loss (encourages exploration by randomly replacing values)
+    hallucination_prob = self.hypers.hallucination_prob
+    hallucination_scale = self.hypers.hallucination_scale
+
+    def hallucinate(x, hallucination_rng):
+      mask = jax.random.bernoulli(hallucination_rng, p=hallucination_prob)
+      noise = hallucination_scale * jax.random.normal(
+          hallucination_rng, x.shape
+      )
+      return jnp.where(mask, noise, x)
+
+    _, factor_rng = jax.random.split(rng)
+    decomposition = jax.tree_util.tree_map(
+        lambda x: hallucinate(x, jax.random.split(factor_rng)[0]),
+        decomposition,
+    )
+
     # Add a batch dimension to `target_tensor` to ensure correct broadcasting.
     # Define the loss as the L2 reconstruction error.
-    rec_loss = l2_loss_complex(self.target_tensor[None, ...], rec_tensor)
+    rec_loss = l2_loss_complex(noisy_target_tensor[None, ...], rec_tensor)

     # We must return a real-valued loss.
-    return jnp.real(rec_loss)

+    # Discretization loss (encourage entries to be multiples of 1/2 or integer).
+    def dist_to_half_ints(x):
+      x_re = jnp.real(x)
+      x_im = jnp.imag(x)
+      return jnp.minimum(
+          jnp.abs(x_re - jnp.round(x_re * 2) / 2),
+          jnp.abs(x_im - jnp.round(x_im * 2) / 2),
+      )
+
+    def dist_to_ints(x):
+      return jnp.abs(x - jnp.round(x))
+
+    discretization_loss = 0.0
+    for factor in decomposition:
+      discretization_loss += jnp.mean(dist_to_half_ints(factor))
+      discretization_loss += jnp.mean(dist_to_ints(factor))
+
+    discretization_loss /= (
+        len(decomposition) * 2
+    )  # average across all factors and loss components
+
+    discretization_weight = self._linear_schedule(
+        global_step, start=0.0, end=self.hypers.discretization_weight
+    )
+
+    # Cosine annealing for half-integer loss.
+    cycle_length = self.config.training_steps // 4  # Number of steps per cycle
+    cycle_progress = (
+        global_step % cycle_length
+    ) / cycle_length  # Normalized progress within the current cycle [0, 1)
+    half_int_multiplier = (1 + jnp.cos(jnp.pi * cycle_progress)) / 2
+    half_int_multiplier = (
+        1 - self.hypers.half_int_start
+    ) * half_int_multiplier + self.hypers.half_int_start
+
+    total_loss = (
+        rec_loss
+        + discretization_weight * discretization_loss * half_int_multiplier
+    )
+
+    # Add penalty for large values (stability).
+    large_value_penalty = 0.0
+    for factor in decomposition:
+      large_value_penalty += jnp.mean(jnp.abs(factor) ** 2)
+    large_value_penalty /= len(decomposition)
+    total_loss += self.hypers.large_value_penalty_weight * large_value_penalty
+
+    return jnp.real(total_loss)
+

 def l2_loss_complex(x: jnp.ndarray, y: jnp.ndarray) -> jnp.ndarray:
   """Elementwise L2 loss for complex numbers."""
@@ -117,6 +255,18 @@
   return hyper.zipit([
-      hyper.uniform('init_scale', hyper.interval(0.2, 1.5)),
-      hyper.uniform('learning_rate', hyper.interval(0.05, 0.3)),
+      hyper.uniform('init_scale', hyper.interval(0.1, 1.0)),
+      hyper.uniform('learning_rate', hyper.interval(0.01, 0.2)),
+      hyper.uniform('discretization_weight', hyper.interval(0.0, 0.1)),
+      hyper.uniform('hallucination_prob', hyper.interval(0.0, 0.2)),
+      hyper.uniform('hallucination_scale', hyper.interval(0.0, 0.2)),
+      hyper.uniform('noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('target_noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('weight_decay', hyper.interval(0.00001, 0.001)),
+      hyper.uniform('clip_min', hyper.interval(0.0, 0.5)),
+      hyper.uniform('clip_max', hyper.interval(1.0, 3.0)),
+      hyper.uniform('large_value_penalty_weight', hyper.interval(0.0, 0.01)),
+      # Add noise to the gradient to aid in exploration.
+      hyper.uniform('grad_noise_std', hyper.interval(0.0, 0.001)),
+      hyper.uniform('half_int_start', hyper.interval(0.0, 1.0)),
   ])
 # EVOLVE-BLOCK-END
```

```diff
@@ -45,9 +45,14 @@
   # EVOLVE-BLOCK-START
   def _get_optimizer(self) -> optax.GradientTransformation:
     """Returns optimizer."""
-    return optax.adam(self.hypers.learning_rate)
+    return optax.adamw(
+        self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
+    )

   def _get_init_fn(self) -> jax.nn.initializers.Initializer:
     """Returns initializer function."""
-    return initializers.normal(0.0, self.hypers.init_scale, jnp.complex64)
+    # Initialize with a smaller scale to encourage finding low-rank solutions.
+    # Increase scale slightly for better exploration.
+    scale = self.hypers.init_scale
+    return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)
```

```diff
@@ -91,13 +156,86 @@
     """Computes (batched) loss on learned decomposition."""
     # Compute reconstruction loss.
     rec_tensor = self._decomposition_to_tensor(decomposition)  # (B, N, M, P)

...
+    # Discretization loss (encourage entries to be multiples of 1/2 or integer).
+    def dist_to_half_ints(x):
...
+
+    def dist_to_ints(x):
...
+    discretization_loss = 0.0
+    for factor in decomposition:
+      discretization_loss += jnp.mean(dist_to_half_ints(factor))
+      discretization_loss += jnp.mean(dist_to_ints(factor))
+
+    discretization_loss /= (
+        len(decomposition) * 2
+    )  # average across all factors and loss components
+
+    discretization_weight = self._linear_schedule(
+        global_step, start=0.0, end=self.hypers.discretization_weight
+    )
+
+    # Cosine annealing for half-integer loss.
+    cycle_length = self.config.training_steps // 4  # Number of steps per cycle
+    cycle_progress = (
+        global_step % cycle_length
+    ) / cycle_length  # Normalized progress within the current cycle [0, 1)
+    half_int_multiplier = (1 + jnp.cos(jnp.pi * cycle_progress)) / 2
+    half_int_multiplier = (
+        1 - self.hypers.half_int_start
+    ) * half_int_multiplier + self.hypers.half_int_start
+
+    total_loss = (
+        rec_loss
+        + discretization_weight * discretization_loss * half_int_multiplier
+    )
...
```

```diff
@@ -117,6 +255,18 @@
  return hyper.zipit([
-      hyper.uniform('init_scale', hyper.interval(0.2, 1.5)),
-      hyper.uniform('learning_rate', hyper.interval(0.05, 0.3)),
+      hyper.uniform('init_scale', hyper.interval(0.1, 1.0)),
+      hyper.uniform('learning_rate', hyper.interval(0.01, 0.2)),
+      hyper.uniform('discretization_weight', hyper.interval(0.0, 0.1)),
+      hyper.uniform('hallucination_prob', hyper.interval(0.0, 0.2)),
+      hyper.uniform('hallucination_scale', hyper.interval(0.0, 0.2)),
+      hyper.uniform('noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('target_noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('weight_decay', hyper.interval(0.00001, 0.001)),
+      hyper.uniform('clip_min', hyper.interval(0.0, 0.5)),
+      hyper.uniform('clip_max', hyper.interval(1.0, 3.0)),
+      hyper.uniform('large_value_penalty_weight', hyper.interval(0.0, 0.01)),
+      # Add noise to the gradient to aid in exploration.
+      hyper.uniform('grad_noise_std', hyper.interval(0.0, 0.001)),
+      hyper.uniform('half_int_start', hyper.interval(0.0, 1.0)),
  ])
 # EVOLVE-BLOCK-END
```

Figure 4 | Changes proposed by *AlphaEvolve* to discover faster matrix multiplication algorithms. The full diff is outlined on the left (see magnified version in Figures 9a to 9c) and some excerpts are highlighted on the right. In this example, *AlphaEvolve* proposes extensive changes across several components, including the optimizer and weight initialization (top right), the loss function (middle right), and hyperparameter sweep (bottom right). These changes are highly non-trivial, requiring 15 mutations during the evolutionary process.

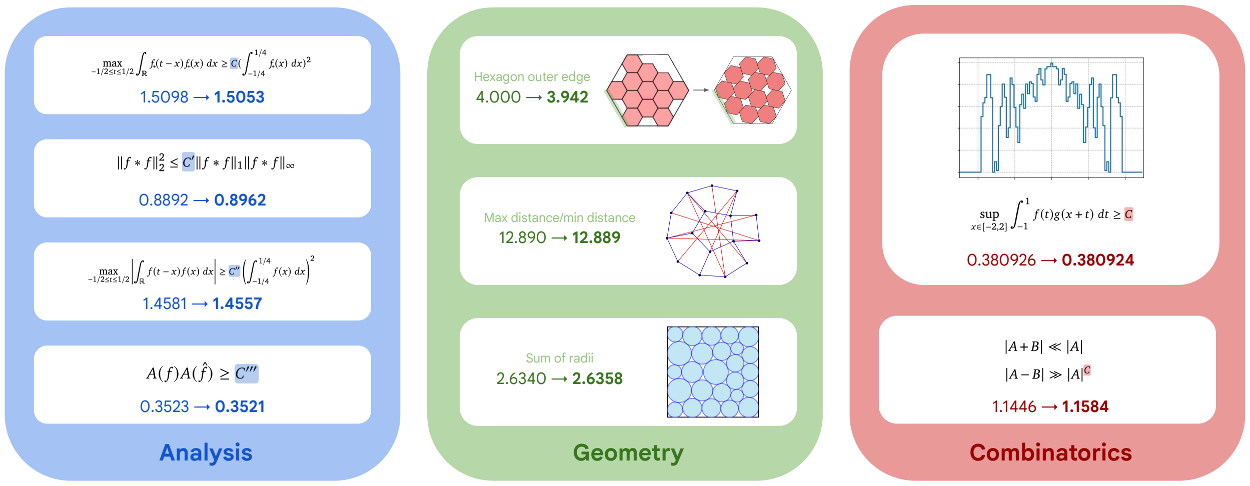

$\max_{-1/2\le t\le 1/2}\int_{\mathbb{R}} f_+(t-x)f_+(x)\,dx \ge C(\int_{-1/4}^{1/4} f_+(x)\,dx)^2$
1.5098 → **1.5053**
$\|f*f\|_2^2 \le C'\|f*f\|_1\|f*f\|_\infty$
0.8892 → **0.8962**
$\max_{-1/2\le t\le 1/2}\left|\int_{\mathbb{R}} f(t-x)f(x)\,dx\right| \ge C''\left(\int_{-1/4}^{1/4} f(x)\,dx\right)^2$
1.4581 → **1.4557**
$A(f)A(\hat{f}) \ge C'''$
0.3523 → **0.3521**
Analysis
Hexagon outer edge
4.000 → **3.942**
Max distance/min distance
12.890 → **12.889**
Sum of radii
2.6340 → **2.6358**
Geometry
$\sup_{x\in[-2,2]}\int_{-1}^{1} f(t)g(x+t)\,dt \ge C$
0.380926 → **0.380924**
$|A+B| \ll |A|$
$|A-B| \gg |A|^C$
1.1446 → **1.1584**
Combinatorics

Figure 5 | Examples of SOTA-breaking mathematical constructions discovered with *AlphaEvolve*. The versatility of *AlphaEvolve* allows us to tackle problems in analysis (autocorrelation and uncertainty inequalities), geometry (packing and minimum/maximum distance problems) and combinatorics (Erdős's minimum overlap problem and sums and differences of finite sets).

A significant advantage of the *AlphaEvolve* configuration used here is its versatility and speed of application. The core methodology, focused on evolving heuristic search programs (detailed below), can be rapidly deployed across a diverse range of mathematical construction problems and conjectures, often requiring less initial problem-specific expert tailoring compared to traditional bespoke approaches. While deep mathematical insight naturally aids in problem formulation and search space definition, *AlphaEvolve* often demonstrates a capacity to autonomously discover effective search patterns and attack strategies by identifying subtle structures within the problem landscape. This allows for efficient, large-scale exploration across many different problems.

The key methodological innovation enabling these discoveries is *AlphaEvolve*'s ability to evolve *heuristic search algorithms* rather than directly evolving the constructions themselves. For many problems, particularly those with fast objective function evaluations—which are common in mathematics—we employed an iterative refinement strategy. Each generation of *AlphaEvolve* was tasked with evolving a program representing a search heuristic. This program was given a fixed time budget (e.g., 1000 seconds) and was shown the best construction found by the previous best heuristic. Its goal was to leverage this starting point and the allotted time to find an even better construction. The evolutionary process thus selects for heuristics that are effective at improving already high-quality solutions. The final constructions were often the result of a sequence of different, specialized heuristics discovered by *AlphaEvolve*—early heuristics proficient at making large gains from random or simple initial states, and later heuristics adept at fine-tuning near-optimal configurations. This automated discovery of multi-stage, adaptive search strategies is challenging to replicate manually and proved crucial for surpassing the SOTA.

Below are high-level descriptions of some of the problems where *AlphaEvolve* yielded new results. Full list of problems and details are provided in Appendix B.

- **Analysis**
  - **Autocorrelation inequalities.** *AlphaEvolve* was able to improve the best known bounds on several autocorrelation inequalities.
  - **Uncertainty principles.** *AlphaEvolve* was able to produce a refined configuration for a problem arising in Fourier analysis, by polishing an uncertainty principle construction [33] leading to a slightly better upper bound.
- **Combinatorics and number theory**
  - **Erdős's minimum overlap problem.** *AlphaEvolve* established a new upper bound for the minimum overlap problem [25], slightly improving upon the previous record [40].
- **Geometry and packing**
  - **Kissing number problem.** In 11 dimensions, *AlphaEvolve* improved the lower bound on the kissing number, finding a configuration of 593 non-overlapping unit spheres that can simultaneously touch a central unit sphere, surpassing the previous record of 592 [31].
  - **Packing problems.** *AlphaEvolve* achieved several new results in packing problems, such as packing N points in a shape to minimize the ratio of the maximum and minimum distance, packing various polygons in other polygons in the most efficient way, and variants of the Heilbronn problem concerning point sets avoiding small-area triangles [29].

The full list of problems appears in Appendix B and the new constructions found by *AlphaEvolve* can be found in the accompanying Google Colab. More examples and details on these problems and the methods used will be provided in an upcoming paper. Most of these discoveries are on open problems suggested to us by external mathematicians Javier Gomez Serrano and Terence Tao, who also advised on how to best formulate them as inputs to *AlphaEvolve*. This highlights the potential for synergistic partnerships between AI-driven discovery engines like *AlphaEvolve* and human mathematical expertise.

3.3. Optimizing Google's computing ecosystem

In addition to the scientific applications presented in preceding sections, here we demonstrate how *AlphaEvolve* has been used to improve performance of mission-critical infrastructure and deliver real-world impact.

3.3.1. Improving data center scheduling

Efficiently scheduling compute jobs onto a cluster of machines is a critical optimization problem, particularly at the scale of Google's data centers, orchestrated by Borg [102]. This task involves assigning jobs to available machines based on job resource requirements and machine capacity. Inefficient assignments can result in stranded resources: when a machine can no longer accept jobs because it has run out of one kind of resource (e.g., memory) but still has other resources free (e.g., CPU). Improvements in scheduling efficiency can recover these stranded resources, allowing more jobs to be completed on the same amount

```python
def alpha_evolve_score(required, free):
  cpu_residual = required.cpu / free.cpu
  mem_residual = required.mem / free.mem

  return -1.0 * (cpu_residual + mem_residual +
                 mem_residual / cpu_residual +
                 cpu_residual / mem_residual)
```

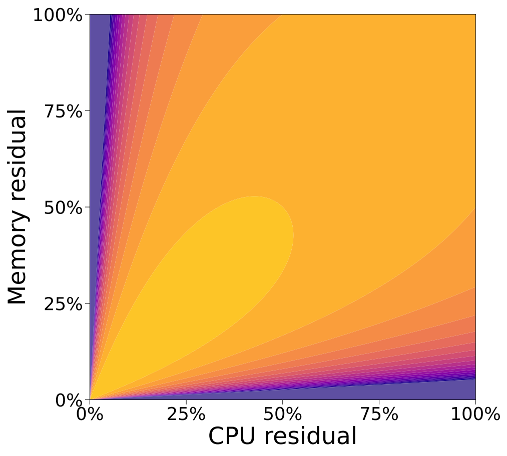


Figure 6 | Left: The heuristic function discovered by *AlphaEvolve*, tailored to Google's workloads and capacity. Right: Visualization of the heuristic scoring function. Yellow regions represent high scores, while purple regions represent low scores.

of computational footprint. This recovery is essential to accommodate growing compute needs without a proportional increase in resource consumption. Furthermore, this problem is challenging since it combines typical engineering difficulties, such as debuggability and scale, on top of the classically difficult bin-packing problem.

We address this challenge by framing the online job scheduling problem as a vector bin-packing problem with two variables. In this context, machines represent bins with defined capacities for CPU and memory, and incoming jobs are items with specific resource demands. A heuristic function takes as input a pending job's CPU and memory requirements and a potential machine's CPU and memory availability. This function then outputs a priority score for the machine. The Borg scheduler subsequently assigns the pending job to the machine with the highest priority score as determined by the heuristic function, among other objectives. Because this heuristic only influences the ranking of machines already determined by Borg to be available and capable of running each pending job, the resulting scheduling decisions are effectively correct by construction.

An early version of *AlphaEvolve* was used to discover a remarkably simple yet effective heuristic function (shown in Figure 6), evolving from the existing one in production. We use a simulator of our data centers to provide feedback to *AlphaEvolve* based on historical snapshots of workloads and capacity across Google's fleet. We measure the performance of *AlphaEvolve*'s heuristic function on an unseen test dataset of recent workloads and capacity to ensure generalization. Observing that *AlphaEvolve*'s heuristic function outperforms the one in production, we rolled out *AlphaEvolve*'s heuristic function to the entire fleet. Post-deployment measurements across Google's fleet confirmed the simulator results, revealing that this heuristic function continuously recovers on average 0.7% of Google's fleet-wide compute resources, which would otherwise be stranded. *AlphaEvolve* was chosen over a deep reinforcement learning approach because its code solution not only leads to better performance, but also offers clear advantages in interpretability, debuggability, predictability, and ease of deployment—essential qualities for a mission-critical system.

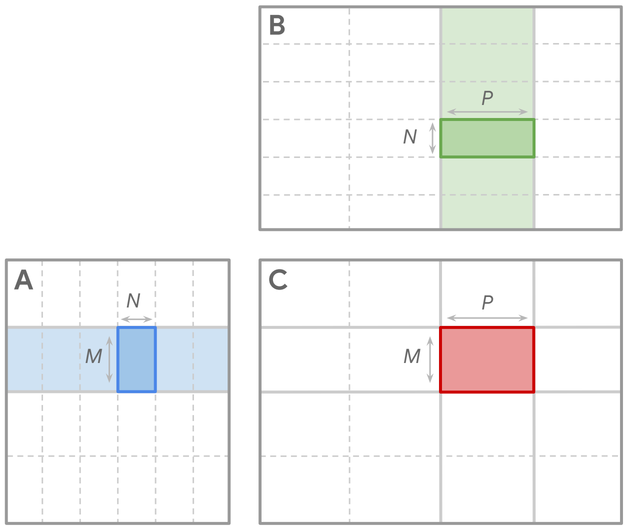


Figure 7 | Visualization of the tiling heuristic problem for a matrix product $AB = C$. Creating a heuristic that automatically chooses the right tile size (M, N, P) for all input shapes is difficult because one has to know the matrix multiplication unit's optimal shapes and memory capacity, the memory requirements of surrounding operations, extra operations that are fused into the kernel, and low-level compiler intricacies, among other details.

3.3.2. Enhancing Gemini kernel engineering

Training large models like Gemini requires substantial computational resources. Gemini is built on JAX [9], and Pallas is an extension to JAX that enables writing custom, highly specialized programs (kernels) tailored for optimal execution on hardware accelerators. Therefore, efficient Pallas kernels are crucial for optimizing Gemini's training performance. A critical aspect of kernel optimization is tuning the tiling strategy for matrix multiplication operations (see Figure 7). This technique involves dividing a large matrix multiplication computation into smaller subproblems to better balance computation with data movement, which is key to accelerating the overall computation. Traditionally, kernel engineers rely on either search-based autotuning or manually crafted heuristics to determine near-optimal tiling configurations for various input shapes. Search-based tuning interrupts the research workflow, necessitating retuning for every input shape change. Conversely, manually crafting effective tiling heuristics is a major engineering bottleneck due to its complexity, demanding a deep understanding of both kernel functionality and hardware intricacies. The key advantage of a performant heuristic is its ability to deliver high performance across arbitrary input shapes. Consequently, to expedite the design of performant kernels for emerging hardware and to simplify their utilization by model developers, we aim to facilitate the heuristic generation process.

We address this challenge by employing *AlphaEvolve* to optimize tiling heuristics for an important matrix multiplication kernel used to train Gemini. The objective is to minimize the kernel's actual runtime. *AlphaEvolve* iteratively explores and refines tiling heuristics for this kernel by proposing candidate code, aiming to minimize this runtime on various input shapes on real TPU accelerators. The kernel's correctness is maintained by construction because *AlphaEvolve* is optimizing the tiling strategy for this kernel rather than altering its underlying mathematical operation. To build the training and evaluation datasets for *AlphaEvolve*, we automatically collect realistic kernel input shapes from kernel users. Half of these input shapes form the training set, providing the optimization targets during the evolutionary process. The remaining input shapes form the evaluation set, used to test the general applicability of the resulting heuristic.

This automated approach enables *AlphaEvolve* to discover a heuristic that yields an average 23% kernel speedup across all kernels over the existing expert-designed heuristic, and a corresponding 1% reduction in Gemini's overall training time. In addition, the use of *AlphaEvolve* significantly reduced the kernel optimization time, from several months of dedicated engineering effort to just days of automated experimentation. This acceleration speeds up the deployment of optimized kernels, allowing kernel engineers to dedicate their expertise to more strategic, higher-level optimization problems. Furthermore, *AlphaEvolve* offers a path towards automating the manual tuning process and improving the ergonomics of Gemini kernel usage. The tiling heuristic discovered by *AlphaEvolve* has been deployed in production, directly enhancing Gemini's training efficiency and the Gemini team's research and engineering velocity. This deployment also marks a novel instance where Gemini, through the capabilities of *AlphaEvolve*, optimizes its own training process.

3.3.3. Assisting in hardware circuit design

Specialized hardware, such as Google's Tensor Processing Units (TPUs), is crucial for achieving the resource efficiency required to run modern AI systems at scale. However, designing new computer chips is a complex and time-consuming process, often spanning years. Register-Transfer Level (RTL) optimization, a critical step in this process, involves manually rewriting hardware descriptions to improve metrics like power, performance, and area, demanding months of iteration by highly skilled engineers.

In this work, *AlphaEvolve* was challenged to optimize an already highly optimized Verilog implementation of a key TPU arithmetic circuit within the matrix multiplication unit. The optimization objectives were to reduce both area and power consumption while preserving the component's core functionality. Crucially, the final proposal must pass robust verification methods to confirm that the modified circuit maintains functional correctness. *AlphaEvolve* was able to find a simple code rewrite that removed unnecessary bits, a change validated by TPU designers for correctness. While this specific improvement was also independently caught by downstream synthesis tools, *AlphaEvolve*'s contribution at the RTL stage demonstrates its capability to refine source RTL and provide optimizations early in the design flow.

Integrated into an upcoming TPU, this improvement represents Gemini's first direct contribution to TPU arithmetic circuits, achieved via *AlphaEvolve*, paving the way for future contributions. A key advantage of *AlphaEvolve* is that it communicates the suggested changes directly in Verilog, the standard language used by hardware engineers, fostering trust and simplifying adoption. This early exploration demonstrates a novel approach where LLM-powered code evolution assists in hardware design, potentially reducing time to market.

3.3.4. Directly optimizing compiler-generated code

The transformer architecture [100] is used in the majority of modern neural networks, ranging from LLMs to AlphaFold [1]. The core computation of transformers is the attention mechanism [4], which is most commonly implemented using FlashAttention [22]. In our stack, FlashAttention is implemented as an accelerator kernel in Pallas, wrapped by higher-level code in JAX that handles input preparation and output postprocessing. The machine learning compiler (XLA [77]) then translates this implementation into a sequence of intermediate representations (IRs), each adding more detail for execution on particular hardware. At these stages, improved decisions on memory access orchestration or computation scheduling can significantly reduce runtime on specific hardware.

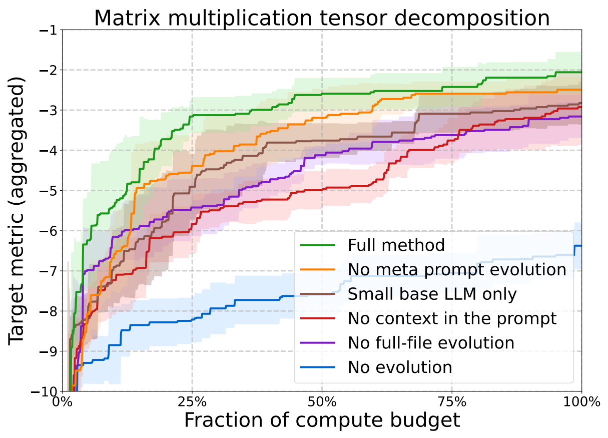
We challenged *AlphaEvolve* to directly optimize the XLA-generated IRs encapsulating the FlashAttention kernel along with pre- and postprocessing code. We optimized a configuration corresponding to a highly impactful transformer model used for inference at scale on GPUs, with the goal of minimizing the module's overall execution time. This was a particularly challenging task, because (1) the IR is designed for debugging purposes rather than for direct editing by developers, and (2) it is compiler-generated and already highly optimized. Each modification proposed by *AlphaEvolve* was checked against the reference (unmodified) code on randomized inputs in order to ensure numerical correctness throughout optimization. The final version of the code was rigorously confirmed by human experts to be correct for all possible inputs.

AlphaEvolve was able to provide meaningful optimizations for both levels of abstraction exposed by the IR. Firstly, the FlashAttention kernel for the configuration of interest was sped up by 32%. Secondly, *AlphaEvolve* found improvements in pre- and postprocessing of kernel inputs and outputs, resulting in a 15% speed up in this part. These results demonstrate the ability of *AlphaEvolve* to optimize compiler-generated code, offering the potential of incorporating discovered optimizations into existing compilers for specific use cases, or, in the longer term, incorporating *AlphaEvolve* into the compiler workflow itself.

4. Ablations

We carried out ablations on two tasks: finding tensor decompositions for faster matrix multiplication (Section 3.1) and computing lower bounds on kissing numbers (Section 3.2), aiming to understand the efficacy of the following components of *AlphaEvolve*.

- **Evolutionary approach.** *AlphaEvolve* utilizes an evolutionary approach, where previously generated programs are stored in a database and used to obtain better programs in subsequent iterations. To analyze the importance of evolution, we consider an alternative approach, which repeatedly feeds the same initial program to the language model. We refer to this approach as "No evolution".
- **Context in prompts.** *AlphaEvolve* uses powerful language models with large context windows, whose output can be improved significantly by providing problem-specific context in the prompt. To test the importance of context, we consider an alternative approach where no explicit context is added to the prompt. We refer to this approach as "No context in the prompt".



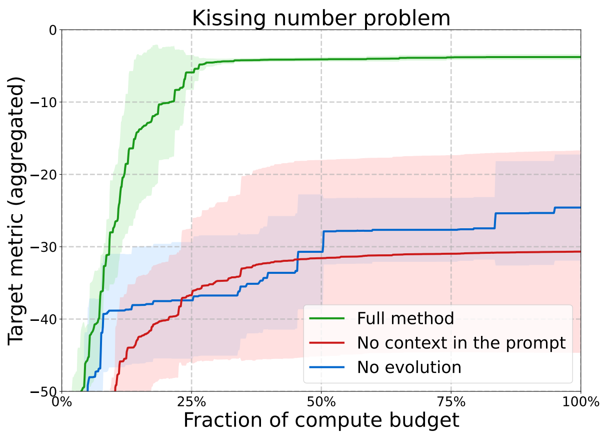


Figure 8 | Left: Ablations of *AlphaEvolve* on the problem of finding low-rank tensor decomposition for faster matrix multiplication. Right: Ablations of *AlphaEvolve* on the problem of finding sphere packings for improving kissing numbers. Each curve shows the performance of an individual setting with increasing compute budget, averaged over all considered targets (higher values on the target metric are better). The shades indicate intra-target standard deviation, averaged over three independent runs of *AlphaEvolve*, initialized with different random seeds.

- **Meta prompts.** *AlphaEvolve* also uses meta prompts in order to improve the prompts that are provided to the language model. This allows it to potentially surpass the performance one can obtain using a human prompter. To test the efficacy of meta prompting, we disable it for the task of tensor decomposition. We refer to this approach as "No meta prompt evolution".
- **Full-file evolution.** Unlike previous approaches such as FunSearch, *AlphaEvolve* can evolve an entire codebase instead of focusing on a single function. To test the importance of full-file evolution, we consider an alternative in the context of tensor decomposition where only the loss function is evolved. We refer to this approach as "No full-file evolution".
- **Powerful language models.** *AlphaEvolve* relies on a mixture of small and large language models in order to obtain highly diverse samples. To understand the importance of this component, we consider an alternative where only a single small base model is used. We refer to this approach as "Small base LLM only".

Figure 8 shows the results of the all-inclusive *AlphaEvolve* approach as well as the various alternatives listed above. As can be seen, each of the components is responsible for a significant improvement in the results.

5. Related work

Evolutionary methods. *AlphaEvolve* extends a long tradition of research on *evolutionary* or *genetic programming* [54], where one repeatedly uses a set of mutation and crossover operators to evolve a pool of programs [5, 51]. In particular, classical evolutionary techniques have succeeded in symbolic regression applications [66, 87], automated scientific [21] or algorithmic [16] discovery, and scheduling [118] problems. However, a challenge with these methods is the use of handwritten evolution operators, which can be hard to design and may fail to capture important properties of the domain. In contrast, *AlphaEvolve* uses LLMs to automate the construction of these operators—it leverages the LLM's world knowledge to mutate programs without the need to pre-define a set of allowed mutation operations.

AlphaEvolve was preceded by a body of recent efforts that combine LLMs and evolution; specifically, it extends the FunSearch system, introduced by Romera-Paredes et al. [83] as an approach to mathematical discovery. FunSearch was subsequently used in downstream tasks such as learning acquisition functions for Bayesian optimization [2], discovering cognitive models [13], computing distances between graphs [103], or combinatorial competitive programming [101]. *AlphaEvolve* goes beyond FunSearch and its recent reimplementation [24] in three key ways. First, while FunSearch only allowed the evolution of a single Python function, *AlphaEvolve* allows evolution over entire codebases written in a wide range of programming languages. Second, FunSearch optimized a single objective function, while *AlphaEvolve* provides the ability to perform multiobjective optimization. Third, the LLMs in FunSearch were relatively small and solely trained on code. By contrast, *AlphaEvolve* uses frontier LLMs and rich forms of natural-language context and feedback. As has been demonstrated in this paper, these extensions allow *AlphaEvolve* to address important challenging problems that were not amenable to FunSearch.

Other efforts in this category include the approach by Lehman et al. [57], which uses an LLM-guided evolution process to discover programmatic policies for a set of simulated robots; or the approach by Hemberg et al. [41] for code synthesis. Similar approaches have found use in several scientific and mathematical tasks, including symbolic regression [35, 89], discovering heuristics for combinatorial optimization [63, 115, 117], and synthesizing molecular structures [105]. LLM-guided evolution has also been used to improve AI systems by enhancing LLM prompts [27] and searching over neural architectures [14, 73]. *AlphaEvolve* differs from these approaches in its scale, flexibility, and general applicability to a broad range of domains.

Some recent efforts have augmented the basic paradigm of LLM-guided evolution with complementary ideas. For example, Surina et al. [96] complement the evolution process by continuously finetuning the LLM through reinforcement learning. Grayeli et al. [35] enhance the evolution process with an LLM-directed concept learning step that summarizes high-performing programs in the pool into natural language. More investigation is required to understand the benefits of these ideas at the scale at which *AlphaEvolve* operates.

Evolutionary methods have also found use in the recent AI Co-Scientist work [34], which seeks to automate scientific discovery using distinct agents for tasks like hypothesis discovery, ranking of hypotheses, and literature review. While AI Co-Scientist represents scientific hypotheses and their evaluation criteria in *natural language*, *AlphaEvolve* focuses on evolving *code*, and directs evolution using programmatic evaluation functions. This choice enables us to substantially sidestep LLM hallucinations, which allows *AlphaEvolve* to carry on the evolution process for a large number of time steps. Nevertheless, it is possible in principle to combine the two approaches, leading to a method that allows a flexible combination of natural-language and programmatic idioms.

Superoptimization and algorithm discovery. *AlphaEvolve* can be viewed as a method for *code superoptimization* in that it iteratively improves an initial program using execution feedback. The idea of code superoptimization goes back to the 1980s [69]; pre-LLM approaches to the problem included systematic enumeration [69], genetic search [20], Monte Carlo sampling [86], and deep reinforcement learning [68]. Additionally, in limited settings that focus on a single problem such as matrix multiplication, there have been systems such as AlphaTensor that were also able to discover provably correct algorithms [26].

More recently, a body of LLM-based approaches to superoptimization and algorithm discovery have emerged. This literature builds on the success of LLMs in coding tasks, perhaps best illustrated by their success in (simulated) programming competitions as in the case of AlphaCode [60]. For instance, LLM agents have been used to optimize certain operations in GPU kernels, such as the attention operation [15] or more general user-specified operations [56]. There is also work on using LLMs to discover novel evolutionary algorithms [55], train language models [58], and optimize warehouse-scale computers [61]. Other recent work [108] has also proposed the use of multiple LLM agents that converse with each other to accomplish mathematical and coding tasks.

While previous work on using LLMs for algorithm discovery provided promising results, *AlphaEvolve*'s approach to leverage it for evolutionary algorithms allows us to address significantly more challenging problems, as demonstrated in Section 3.

AI for scientific and mathematical discovery. Over the last decade, AI systems have been applied to a wide range of scientific disciplines and tasks, from protein structure prediction [46] to quantum physics [6, 84] to climate sciences [53]. In particular, there are numerous recent LLM-based methods that target scientific problems in multiple disciplines, such as materials science [45, 71, 94, 119], chemistry [12, 64], bioinformatics [67, 85], geoscience [79], and quantum physics [30, 78] (for surveys on the topic, see [36, 65, 81]).

Many of these methods use LLMs to automate several distinct stages of the scientific discovery process [37, 59, 106, 109, 112], e.g., for generating and ranking hypotheses and ideas [38, 90]. Of these methods, especially related to *AlphaEvolve* are the methods that use LLM-guided tree search-based algorithms [11] or LLM-guided evolutionary algorithms [34, 113, 120]. Other works use LLMs to optimize experimental planning and design [7, 10, 43, 75] or experiment execution and workflow [28, 62, 82, 105, 116]. Finally, there are also works focusing on the data analysis stage [80]. *AlphaEvolve* differs from most of these methods in its use of programmatic hypothesis representations and evaluation metrics.

AI systems have also contributed to advances in pure mathematics [23]. In this context, the FunSearch approach [24, 83] established LLM-guided evolution as a powerful tool for discovering witnesses for, and counterexamples to, mathematical statements—a problem that is complementary to that of finding formal and informal proofs of mathematical statements [3, 19, 98, 99, 110, 111].

6. Discussion

AlphaEvolve demonstrates the surprising power of combining state-of-the-art LLMs with automated evaluation metrics within an evolutionary framework, which can lead to new discoveries on decades-old mathematical problems as well as practical improvements to highly optimized compute stacks.

Interestingly, *AlphaEvolve* often allows approaching the same problem in different ways: searching for the solution directly, finding a function that constructs it from scratch, or evolving a search algorithm to find it. Applying *AlphaEvolve* in different ways comes with different biases (for example, finding constructive functions may favor discovering highly symmetric objects [83]) and thus can suit different problems.

AlphaEvolve can also be seen as a test-time compute agent that, through its evolutionary procedure, significantly enhances the capability of the base LLM (compared to, e.g., repeated sampling). On one hand, this can be seen as a compelling demonstration of how machine feedback is able to sustain test-time compute scaling up to regimes where new scientific discoveries and highly valuable practical optimizations are made. On the other hand, a natural next step will be to consider distilling the *AlphaEvolve*-augmented performance of the base LLMs into the next generation of the base models. This can have intrinsic value and also, likely, uplift the next version of *AlphaEvolve*.

Beyond distillation, it is also intriguing that *AlphaEvolve* can make practical discoveries that increase the efficiency of its own infrastructure and of (future versions of) its base LLMs. Currently, the gains are moderate and the feedback loops for improving the next version of *AlphaEvolve* are on the order of months. However, with these improvements we envision that the value of setting up more environments (problems) with robust evaluation functions will become more widely recognized, which in turn will result in more high-value practical discoveries going forward.

The main limitation of *AlphaEvolve* is that it handles problems for which it is possible to devise an automated evaluator. While this is true of many problems in the mathematical and computational sciences, there are domains such as the natural sciences where only some experiments can be simulated or automated. While *AlphaEvolve* does allow for LLM-provided evaluation of ideas, this is not a setting we have optimized for. However, concurrent work shows this is possible [34], and a natural step would be to link the two settings, with LLMs providing feedback on high-level ideas before transitioning to an implementation stage, for which machine feedback is available through code execution.

Acknowledgements

We thank Michael Figurnov for reviewing this white paper; Alhussein Fawzi, Bernardino Romera-Paredes, and Ankit Anand for early explorations and insightful discussions; Stig Petersen and Demis Hassabis for support and advice; JD Velasquez for helpful advice on managing the practical applications; and all early users and collaborators of *AlphaEvolve* for their diverse use cases and insightful feedback, which shaped it into a more robust and versatile tool for a wide range of applications. We gratefully acknowledge the invaluable contributions of these individuals towards the applications highlighted in this white paper:

Terence Tao, Javier Gomez Serrano, and Jordan Ellenberg for suggesting specific open mathematical problems and advising on how to best formulate them for *AlphaEvolve*; Bogdan Georgiev, Ray Jiang, and Johannes Bausch for their contributions to applying *AlphaEvolve* to such problems.

Mohammadamin Barekatain, Patrick Heisel, Chase Hensel, Robert O'Callahan, and Pengming Wang for co-leading the application to data center scheduling; Federico Piccinini, Sultan Kenjeyev, and Andrea Michi for making significant contributions; Kieran Milan, Daniel Mankowitz, Cosmin Paduraru, Calin Cascaval, Tammo Spalink, and Natasha Antropova for providing helpful advice; Aaron Gentleman, Gaurav Dhiman, Parthasarathy Ranganatha, and Amin Vahdat for reviewing this work.

Yanislav Donchev for leading the application to Gemini kernel engineering; Richard Tanburn for making significant contributions; Justin Chiu and Julian Walker for providing helpful advice; Jean-Baptiste Alayrac, Dmitry Lepikhin, Sebastian Borgeaud, Koray Kavukcuoglu and Jeff Dean for reviewing this work.

Timur Sitdikov for leading the application to TPU circuit design; Georges Rotival for providing the circuit evaluation infrastructure; Kirk Sanders, Srikanth Dwarakanath, Indranil Chakraborty, Christopher Clark for verifying and validating the results in the TPU design; Vinod Nair, Sergio Guadarrama, Dimitrios Vytiniotis, and Daniel Belov for their helpful advice; Kerry Takenaka, Jeff Dean, Sridhar Lakshmanamurthy, Parthasarathy Ranganathan, and Amin Vahdat for reviewing this work.

Benjamin Chetioui, Sergei Lebedev, Alexander Belyaev, Henning Becker, Oleg Shyshkov, and Aliia Khasanova for their help with XLA modifications as well as their helpful advice; Giorgio Arena, Marco Cornero, and Sebastian Bodenstein for reviewing this work.

Author information

These authors contributed equally: Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, and Matej Balog.

Contributions. A.N. and M.B. designed and implemented the initial version of *AlphaEvolve*. M.B., A.N., N.V. and P.K. developed project vision and scoped problems. N.V. and P.-S.H. oversaw the practical applications. E.D. and M.E. implemented the first benchmark problem used for iterating on *AlphaEvolve*, with input from F.J.R.R. and M.B. A.N. and M.E. developed the final version of *AlphaEvolve*, with contributions from S.S., P.-S.H., and input from M.B., E.D., A.Z.W. and N.V. A.N., S.S., P.-S.H. and M.E. maintained the infrastructure underlying *AlphaEvolve*. M.E. and E.D. used *AlphaEvolve* to discover new algorithms for matrix multiplication, with input from F.J.R.R. A.Z.W. worked on the applications to open mathematical problems, with help from A.M., M.E., and A.N. A.N. contributed to the Borg scheduling application. P.-S.H. and N.V. worked on the application to Gemini kernel engineering. P.-S.H. and A.N. contributed to the TPU circuit design application. B.K. and S.S. worked on applying *AlphaEvolve* to directly optimize compiler-generated code. M.E. performed the ablation experiments. M.B., A.N., M.E., S.S. and P.-S. H. performed the

majority of code reviews. M.B., E.D., S.C., N.V., A.Z.W., F.J.R.R., M.E., A.N., B.K., S.S., A.M., and M.P.K. wrote the paper, with input from A.S., P.-S.H and P.K. N.V., E.D., M.E., S.C., A.N., and A.Z.W. created the figures. F.J.R.R., A.M., and A.Z.W. assembled the accompanying Google Colab. S.N., A.D. and P.K. advised and enabled multiple strands of this work. M.B., A.N., N.V. and G.H. coordinated the team. P.K. supervised and coordinated the research program.

Corresponding authors. Matej Balog, Alexander Novikov and Pushmeet Kohli.

References

[1] J. Abramson, J. Adler, J. Dunger, R. Evans, T. Green, A. Pritzel, O. Ronneberger, L. Willmore, A. J. Ballard, J. Bambrick, et al. Accurate structure prediction of biomolecular interactions with alphafold 3. *Nature*, 630(8016):493–500, 2024.

[2] V. Aglietti, I. Ktena, J. Schrouff, E. Sgouritsa, F. J. R. Ruiz, A. Malek, A. Bellot, and S. Chiappa. FunBO: Discovering acquisition functions for Bayesian optimization with FunSearch. In *International Conference on Machine Learning*, 2025.

[3] AlphaProof and AlphaGeometry teams. AI achieves silver-medal standard solving International Mathematical Olympiad problems, 2024. URL https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications*. The Morgan Kaufmann Series in Artificial Intelligence, 1998.

[6] J. Bausch, A. W. Senior, F. J. H. Heras, T. Edlich, A. Davies, M. Newman, C. Jones, K. Satzinger, M. Y. Niu, S. Blackwell, G. Holland, D. Kafri, J. Atalaya, C. Gidney, D. Hassabis, S. Boixo, H. Neven, and P. Kohli. Learning high-accuracy error decoding for quantum processors. *Nature*, 635(8040):834–840, 2024. doi: 10.1038/s41586-024-08148-8.

[7] D. A. Boiko, R. MacKnight, B. Kline, and G. Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, 2023. doi: 10.1038/s41586-023-06792-0.

[8] P. Boyvalenkov, S. Dodunekov, and O. Musin. A survey on the kissing numbers. *Serdica Math. J.*, 38(4):507–522, 2012. ISSN 1310-6600.

[9] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

[10] A. M. Bran, S. Cox, O. Schilter, C. Baldassari, A. D. White, and P. Schwaller. Augmenting large language models with chemistry tools. *Nature Machine Intelligence*, 6(5):525–535, 2024. doi: 10.1038/s42256-024-00832-8.

[11] A. M. Bran, T. A. Neukomm, D. P. Armstrong, Z. Jončev, and P. Schwaller. Chemical reasoning in LLMs unlocks steerable synthesis planning and reaction mechanism elucidation. In *arXiv preprint arXiv:2503.08537*, 2025.

[12] M. Caldas Ramos, C. J. Collison, and A. D. White. A review of large language models and autonomous agents in chemistry. *Chemical Science*, 16:2514–2572, 2025. doi: 10.1039/D4SC03921A.

[13] P. S. Castro, N. Tomasev, A. Anand, N. Sharma, R. Mohanta, A. Dev, K. Perlin, S. Jain, K. Levin, N. Éltető, W. Dabney, A. Novikov, G. C. Turner, M. K. Eckstein, N. D. Daw, K. J. Miller, and K. L. Stachenfeld. Discovering symbolic cognitive models from human and animal behavior. In *International Conference on Machine Learning*, 2025.

[14] A. Chen, D. M. Dohan, and D. R. So. EvoPrompting: Language models for code-level neural architecture search. In *Advances in Neural Information Processing Systems*, 2023.

[15] T. Chen, B. Xu, and K. Devleker. Automating GPU kernel generation with DeepSeek-R1 and inference time scaling, 2025. URL https://developer.nvidia.com/blog/automating-gpu-kernel-generation-with-deepseek-r1-and-inference-time-scaling.

[16] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, Y. Lu, and Q. V. Le. Symbolic discovery of optimization algorithms. *Advances in Neural Information Processing Systems*, 2023.

[17] A. Cloninger and S. Steinerberger. On suprema of autoconvolutions with an application to Sidon sets. *Proceedings of the American Mathematical Society*, 145(8):3191–3200, 2017.

[18] H. Cohn and F. Gonçalves. An optimal uncertainty principle in twelve dimensions via modular forms. *Inventiones mathematicae*, 217:799–831, 2019.

[19] K. M. Collins, A. Q. Jiang, S. Frieder, L. Wong, M. Zilka, U. Bhatt, T. Lukasiewicz, Y. Wu, J. B. Tenenbaum, W. Hart, et al. Evaluating language models for mathematics through interactions. *Proceedings of the National Academy of Sciences*, 121(24):e2318124121, 2024.

[20] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23:7–22, 2002.

[21] M. Cranmer. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*, 2023.

[22] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

[23] A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, M. Lackenby, G. Williamson, D. Hassabis, and P. Kohli. Advancing mathematics by guiding human intuition with AI. *Nature*, 600 (7887):70–74, 2021. doi: 10.1038/s41586-021-04086-x.

[24] J. S. Ellenberg, C. S. Fraser-Taliente, T. R. Harvey, K. Srivastava, and A. V. Sutherland. Generative modelling for mathematical discovery. *arXiv preprint arXiv:2503.11061*, 2025.

[25] P. Erdős. Some remarks on number theory. *Riveon Lematematika*, 9:45–48, 1955.

[26] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022. doi: 10.1038/s41586-022-05172-4.

[27] C. Fernando, D. Banarse, H. Michalewski, S. Osindero, and T. Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

[28] N. Ferruz and B. Höcker. Controllable protein design with language models. *Nature Machine Intelligence*, 4(6):521–532, 2022.

[29] E. Friedman. Erich's Packing Center. https://erich-friedman.github.io/packing/, 2025. Accessed: 2025-04-22.

[30] F. Frohnert, X. Gu, M. Krenn, and E. van Nieuwenburg. Discovering emergent connections in quantum physics research via dynamic word embeddings. *Machine Learning: Science and Technology*, 6(1):015029, 2025. doi: 10.1088/2632-2153/adb00a.

[31] M. Ganzhinov. Highly symmetric lines. In *arXiv preprint arXiv:2207.08266v1*, 2022.

[32] Gemini team. Gemini 2.5: Our most intelligent AI model, 2025. URL https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025.

[33] F. Gonçalves, D. O. e Silva, and S. Steinerberger. Hermite polynomials, linear flows on the torus, and an uncertainty principle for roots. *Journal of Mathematical Analysis and Applications*, 451(2):678–711, 2017.

[34] J. Gottweis, W.-H. Weng, A. Daryin, T. Tu, A. Palepu, P. Sirkovic, A. Myaskovsky, F. Weissenberger, K. Rong, R. Tanno, K. Saab, D. Popovici, J. Blum, F. Zhang, K. Chou, A. Hassidim, B. Gokturk, A. Vahdat, P. Kohli, Y. Matias, A. Carroll, K. Kulkarni, N. Tomasev, Y. Guan, V. Dhillon, E. D. Vaishnav, B. Lee, T. R. D. Costa, J. R. Penadés, G. Peltz, Y. Xu, A. Pawlosky, A. Karthikesalingam, and V. Natarajan. Towards an AI co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

[35] A. Grayeli, A. Sehgal, O. Costilla Reyes, M. Cranmer, and S. Chaudhuri. Symbolic regression with a learned concept library. *Advances in Neural Information Processing Systems*, 37:44678–44709, 2024.

[36] M. Gridach, J. Nanavati, C. Mack, K. Z. E. Abidine, and L. Mendes. Agentic AI for scientific discovery: A survey of progress, challenges, and future directions. In *ICLR Workshop: Towards Agentic AI for Science: Hypothesis Generation, Comprehension, Quantification, and Validation*, 2025.

[37] X. Gu and M. Krenn. Interesting scientific idea generation using knowledge graphs and LLMs: Evaluations with 100 research group leaders. In *arXiv preprint arXiv:2405.17044*, 2024.

[38] S. Guo, A. H. Shariatmadari, G. Xiong, and A. Zhang. Embracing foundation models for advancing scientific discovery. In *Proceedings of the IEEE International Conference on Big Data*, pages 1746–1755, 2024. doi: 10.1109/bigdata62323.2024.10825618.

[39] K. Gyarmati, F. Hennecart, and I. Z. Ruzsa. Sums and differences of finite sets. *Functiones et Approximatio Commentarii Mathematici*, 37(1):175–186, 2007.

[40] J. K. Haugland. The minimum overlap problem revisited. *arXiv preprint arXiv:1609.08000*, 2016.

[41] E. Hemberg, S. Moskal, and U.-M. O'Reilly. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21, 2024. doi: 10.1007/s10710-024-09494-2.

[42] J. E. Hopcroft and L. R. Kerr. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM J. Appl. Math.*, 20(1):30–36, Jan. 1971. ISSN 0036-1399. doi: 10.1137/0120004.

[43] K. Huang, Y. Qu, H. Cousins, W. A. Johnson, D. Yin, M. Shah, D. Zhou, R. Altman, M. Wang, and L. Cong. CRISPR-GPT: An LLM agent for automated design of gene-editing experiments. In *arXiv preprint arXiv:2404.18021*, 2024.

[44] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, et al. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2): 1–55, 2025.

[45] S. Jia, C. Zhang, and V. Fung. LLMatDesign: Autonomous materials discovery with large language models. In *arXiv preprint arXiv:2406.13163*, 2024.

[46] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873): 583–589, 2021. doi: 10.1038/s41586-021-03819-2.

[47] M. Kauers and J. Moosbauer. Flip graphs for matrix multiplication. In *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation*, pages 381–388, 2023.

[48] M. Kauers and J. Moosbauer. Some new non-commutative matrix multiplication algorithms of size $(n, m, 6)$. *ACM Commun. Comput. Algebra*, 58(1):1–11, Jan. 2025. ISSN 1932-2232. doi: 10.1145/3712020.3712021.

[49] M. Kauers and I. Wood. Consequences of the Moosbauer-Poole algorithms. *arXiv preprint arXiv:2505.05896*, 2025.

[50] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[51] J. R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994. doi: 10.1007/BF00175355.

[52] J. D. Laderman. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. *Bulletin of the American Mathematical Society*, 82(1):126 – 128, 1976.

[53] R. Lam, A. Sanchez-Gonzalez, M. Willson, P. Wirnsberger, M. Fortunato, F. Alet, S. Ravuri, T. Ewalds, Z. Eaton-Rosen, W. Hu, A. Merose, S. Hoyer, G. Holland, O. Vinyals, J. Stott, A. Pritzel, S. Mohamed, and P. Battaglia. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023. doi: 10.1126/science.adi2336.

[54] W. B. Langdon and R. Poli. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

[55] R. Lange, Y. Tian, and Y. Tang. Large language models as evolution strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO ’24 Companion, pages 579–582. Association for Computing Machinery, 2024. doi: 10.1145/3638530.3654238.

[56] R. T. Lange, A. Prasad, Q. Sun, M. Faldor, Y. Tang, and D. Ha. The AI CUDA engineer: Agentic CUDA kernel discovery, optimization and composition. Technical report, Sakana AI, 02 2025.

[57] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pages 331–366. Springer, 2023.

[58] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley. *Evolution Through Large Models*, pages 331–366. Springer Nature Singapore, 2024. doi: 10.1007/978-981-99-3814-8_11.

[59] P.-H. Li, Y.-Y. Sun, H.-F. Juan, C.-Y. Chen, H.-K. Tsai, and J.-H. Huang. A large language model framework for literature-based disease–gene association prediction. *Briefings in Bioinformatics*, 26(1):bbaf070, 02 2025. ISSN 1477-4054. doi: 10.1093/bib/bbaf070.

[60] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. de Masson d’Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz,

E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

[61] H. Lin, M. Maas, M. Roquemore, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa, et al. ECO: An LLM-driven efficient code optimizer for warehouse scale computers. *arXiv preprint arXiv:2503.15669*, 2025.

[62] Z. Lin, H. Akin, R. Rao, B. Hie, Z. Zhu, W. Lu, N. Smetanin, R. Verkuil, O. Kabeli, Y. Shmueli, A. dos Santos Costa, M. Fazel-Zarandi, T. Sercu, S. Candido, and A. Rives. Evolutionary-scale prediction of atomic-level protein structure with a language model. *Science*, 379(6637):1123–1130, 2023. doi: 10.1126/science.ade2574.

[63] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

[64] F. Luo, J. Zhang, Q. Wang, and C. Yang. Leveraging prompt engineering in large language models for accelerating chemical research. *ACS Central Science*, 2025. doi: 10.1021/acscentsci.4c01935.

[65] Z. Luo, Z. Yang, Z. Xu, W. Yang, and X. Du. LLM4SR: A survey on large language models for scientific research. In *arXiv preprint arXiv:2501.04306*, 2025.

[66] H. Ma, A. Narayanaswamy, P. Riley, and L. Li. Evolving symbolic density functionals. *Science Advances*, 8(36):eabq0279, 2022. doi: 10.1126/sciadv.abq0279.

[67] A. Madani, B. Krause, E. R. Greene, S. Subramanian, B. P. Mohr, J. M. Holton, J. L. Olmos, C. Xiong, Z. Z. Sun, R. Socher, J. S. Fraser, and N. Naik. Large language models generate functional protein sequences across diverse families. *Nature Biotechnology*, 41 (8):1099–1106, August 2023. ISSN 1087-0156. doi: 10.1038/s41587-022-01618-2.

[68] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, T. Köppe, K. Millikin, S. Gaffney, S. Elster, J. Broshear, C. Gamble, K. Milan, R. Tung, M. Hwang, T. Cemgil, M. Barekatain, Y. Li, A. Mandhane, T. Hubert, J. Schrittwieser, D. Hassabis, P. Kohli, M. Riedmiller, O. Vinyals, and D. Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618 (7964):257–263, 2023. doi: 10.1038/s41586-023-06004-9.

[69] H. Massalin. Superoptimizer - A look at the smallest program. In R. H. Katz and M. Freeman, editors, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987*, pages 122–126. ACM Press, 1987. doi: 10.1145/36206.36194.

[70] M. Matolcsi and C. Vinuesa. Improved bounds on the supremum of autoconvolutions. *Journal of mathematical analysis and applications*, 372(2):439–447, 2010.

[71] S. Miret and N. M. A. Krishnan. Are LLMs ready for real-world materials discovery? In *arXiv preprint arXiv:2402.05200*, 2024.

[72] J. Moosbauer and M. Poole. Flip graphs with symmetry and new matrix multiplication schemes. *arXiv preprint arXiv:2502.04514*, 2025.

[73] C. Morris, M. Jurado, and J. Zutty. Llm guided evolution-the automation of models advancing models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 377–384, 2024.

[74] J.-B. Mouret and J. Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[75] V. Naumov, D. Zagirova, S. Lin, Y. Xie, W. Gou, A. Urban, N. Tikhonova, K. Alawi, M. Durymanov, F. Galkin, S. Chen, D. Sidorenko, M. Korzinkin, M. Scheibye-Knudsen, A. Aspuru-Guzik, E. Izumchenko, D. Gennert, F. W. Pun, M. Zhang, P. Kamya, A. Aliper, F. Ren, and A. Zhavoronkov. DORA AI scientist: Multi-agent virtual research team for scientific exploration discovery and automated report generation. In *bioRxiv preprint:10.1101/2025.03.06.641840*. Cold Spring Harbor Laboratory, 2025. doi: 10.1101/2025.03.06.641840.

[76] OpenAI. Introducing OpenAI o3 and o4-mini, 2025. URL https://openai.com/index/introducing-o3-and-o4-mini/.

[77] OpenXLA. XLA: composable transformations of Python+NumPy programs. URL https://github.com/openxla/xla.

[78] H. Pan, N. Mudur, W. Taranto, M. Tikhanovskaya, S. Venugopalan, Y. Bahri, M. P. Brenner, and E.-A. Kim. Quantum many-body physics calculations with large language models. *Communications Physics*, 8(1):49, 2025. doi: 10.1038/s42005-025-01956-y.

[79] D. Pantiukhin, B. Shapkin, I. Kuznetsov, A. A. Jost, and N. Koldunov. Accelerating Earth science discovery via multi-agent LLM systems. In *arXiv preprint arXiv:2503.05854*, 2025.

[80] Z. Rasheed, M. Waseem, A. Ahmad, K.-K. Kemell, W. Xiaofeng, A. N. Duc, and P. Abrahamsson. Can large language models serve as data analysts? a multi-agent assisted approach for qualitative data analysis. *arXiv preprint arXiv:2402.01386*, 2024.

[81] S. Ren, P. Jian, Z. Ren, C. Leng, C. Xie, and J. Zhang. Towards scientific intelligence: A survey of LLM-based scientific agents. In *arXiv preprint arXiv:2503.24047*, 2025.

[82] A. Rives, J. Meier, T. Sercu, S. Goyal, Z. Lin, J. Liu, D. Guo, M. Ott, C. L. Zitnick, J. Ma, and R. Fergus. Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences. *Proceedings of the National Academy of Sciences*, 118(15):e2016239118, 2021. doi: 10.1073/pnas.2016239118.

[83] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995): 468–475, 2023. doi: 10.1038/s41586-023-06924-6.

[84] F. J. R. Ruiz, T. Laakkonen, J. Bausch, M. Balog, M. Barekatain, F. J. H. Heras, A. Novikov, N. Fitzpatrick, B. Romera-Paredes, J. van de Wetering, A. Fawzi, K. Meichanetzidis, and P. Kohli. Quantum circuit optimization with AlphaTensor. *Nature Machine Intelligence*, 7(3):374–385, 2025. doi: 10.1038/s42256-025-01001-1.

[85] O. A. Sarumi and D. Heider. Large language models and their applications in bioinformatics. *Computational and Structural Biotechnology Journal*, 23:3498–3505, 2024. ISSN 2001-0370. doi: https://doi.org/10.1016/j.csbj.2024.09.031.

[86] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 305–316. ACM, 2013. doi: 10.1145/2451116.2451150.

[87] M. Schmidt and H. Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009. doi: 10.1126/science.1165893.

[88] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[89] P. Shojaee, K. Meidani, S. Gupta, A. B. Farimani, and C. K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *International Conference on Learning Representations*, 2025.

[90] C. Si, D. Yang, and T. Hashimoto. Can LLMs generate novel research ideas? a large-scale human study with 100+ NLP researchers. In *International Conference on Learning Representations*, 2025.

[91] A. Smirnov. Several bilinear algorithms for matrix multiplication problems $\langle 3, P, Q \rangle$. `https://www.researchgate.net/publication/350897049_Several_Bilinear_Algorithms_for_Matrix_Multiplication_Problems_3_P_Q`, 04 2021.

[92] A. Smirnov. Bilinear algorithm for matrix multiplication $\langle 4\times4\times9; 104 \rangle$. an irreducibly irrational solution of the brent system? `https://www.researchgate.net/publication/364167198_Bilinear_Algorithm_for_Matrix_Multiplication_4x4x9_104_An_irreducibly_irrational_solution_of_the_Brent_system`, 10 2022.

[93] A. V. Smirnov. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics*, 53(12):1781–1795, 2013.

[94] Z. Song, M. Ju, C. Ren, Q. Li, C. Li, Q. Zhou, and J. Wang. LLM-Feynman: Leveraging large language models for universal scientific formula and theory discovery. In *arXiv preprint arXiv:2503.06512*, 2025.

[95] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4): 354–356, 1969.

[96] A. Surina, A. Mansouri, L. Quaedvlieg, A. Seddas, M. Viazovska, E. Abbe, and C. Gulcehre. Algorithm discovery with LLMs: Evolutionary search meets reinforcement learning. In *arXiv preprint arXiv:2504.05108*, 2025.

[97] R. Tanese. *Distributed genetic algorithms for function optimization*. University of Michigan, 1989.

[98] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri. An in-context learning agent for formal theorem-proving. In *Conference on Language Models*, 2024.

[99] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

[100] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[101] P. Veličković, A. Vitvitskyi, L. Markeeva, B. Ibarz, L. Buesing, M. Balog, and A. Novikov. Amplifying human performance in combinatorial competitive programming. 2024.

[102] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741964.

[103] S. Verma, A. Goyal, A. Mathur, A. Anand, and S. Ranu. GRAIL: Graph edit distance and node alignment using llm-generated code. In *International Conference on Machine Learning*, 2025.

[104] C. Vinuesa del Rio. *Generalized Sidon sets*. PhD thesis, Universidad Autónoma de Madrid, 2010.

[105] H. Wang, M. Skreta, C. T. Ser, W. Gao, L. Kong, F. Strieth-Kalthoff, C. Duan, Y. Zhuang, Y. Yu, Y. Zhu, Y. Du, A. Aspuru-Guzik, K. Neklyudov, and C. Zhang. Efficient evolutionary search over chemical space with large language models. In *International Conference on Learning Representations*, 2025.

[106] Q. Wang, D. Downey, H. Ji, and T. Hope. SciMON: Scientific inspiration machines optimized for novelty. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 279–299, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.18.

[107] E. P. White. A new bound for Erdős' minimum overlap problem. *Acta Arithmetica*, 208:235–255, 2023.

[108] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *arXiv preprint arXiv:2308.08155*, 2023.

[109] Y. Xia, P. Jin, S. Xie, L. He, C. Cao, R. Luo, G. Liu, Y. Wang, Z. Liu, Y.-J. Chen, Z. Guo, Y. Bai, P. Deng, Y. Min, Z. Lu, H. Hao, H. Yang, J. Li, C. Liu, J. Zhang, J. Zhu, R. Bi, K. Wu, W. Zhang, K. Gao, Q. Pei, Q. Wang, X. Liu, Y. Li, H. Zhu, Y. Lu, M. Ma, Z. Wang, T. Xie, K. Maziarz, M. Segler, Z. Yang, Z. Chen, Y. Shi, S. Zheng, L. Wu, C. Hu, P. Dai, T.-Y. Liu, H. Liu, and T. Qin. Nature language model: Deciphering the language of nature for scientific discovery. In *arXiv preprint arXiv:2502.07527v2*, 2025.

[110] K. Yang, A. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. J. Prenger, and A. Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

[111] K. Yang, G. Poesia, J. He, W. Li, K. Lauter, S. Chaudhuri, and D. Song. Formal mathematical reasoning: A new frontier in AI. *arXiv preprint arXiv:2412.16075*, 2024.

[112] Z. Yang, X. Du, J. Li, J. Zheng, S. Poria, and E. Cambria. Large language models for automated open-domain scientific hypotheses discovery. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13545–13565. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.804.

[113] Z. Yang, W. Liu, B. Gao, T. Xie, Y. Li, W. Ouyang, S. Poria, E. Cambria, and D. Zhou. MOOSE-Chem: Large language models for rediscovering unseen chemistry scientific hypotheses. In *International Conference on Learning Representations*, 2025.

[114] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[115] S. Yao, F. Liu, X. Lin, Z. Lu, Z. Wang, and Q. Zhang. Multi-objective evolution of heuristic using large language model. In *AAAI Conference on Artificial Intelligence*, volume 39, pages 27144–27152, 2025. doi: 10.1609/aaai.v39i25.34922.

[116] G. Ye, X. Cai, H. Lai, X. Wang, J. Huang, L. Wang, W. Liu, and X. Zeng. DrugAssist: A large language model for molecule optimization. In *arXiv preprint arXiv:2401.10334*, 2023.

[117] H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

[118] F. Zhang, S. Nguyen, Y. Mei, and M. Zhang. *Genetic Programming for Production Scheduling*. Springer, 2021.

[119] H. Zhang, Y. Song, Z. Hou, S. Miret, and B. Liu. HoneyComb: A flexible LLM-based agent system for materials science. In Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 3369–3382. Association for Computational Linguistics, Nov. 2024. doi: 10.18653/v1/2024.findings-emnlp.192.

[120] Y. Zhou, H. Liu, T. Srivastava, H. Mei, and C. Tan. Hypothesis generation with large language models. In L. Peled-Cohen, N. Calderon, S. Lissak, and R. Reichart, editors,

Proceedings of the 1st Workshop on NLP for Science (NLP4Science), pages 117–139. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.nlp4science-1.10.

A. Faster matrix multiplication: Full results

Full table of results. We provide the best ranks obtained by *AlphaEvolve* in Table 3. Overall, we considered 54 matrix multiplication sizes in our experiments. These were chosen roughly representing sizes $\langle m, n, p\rangle$ where $2 \le m, n \le 5$, with some reasonable cutoff for p. Due to symmetries of the underlying matrix multiplication tensor, there exist equivalent algorithms for any permutations of the three axes, hence we focus on sorted sizes $m \le n \le p$.

In all but two considered sizes, *AlphaEvolve* discovered programs which either match or surpass the best known rank. Anecdotally, we encountered some difficulty when increasing the problem size: when we run the discovered programs on sizes beyond $\langle 5, 5, 5\rangle$ on 1000 random seeds on evaluators with a single GPU accelerator, we often run out of memory. Hence, extending our setup to larger matrix sizes requires further optimization.

$\langle m, n, p\rangle$	best known [reference]	*AlphaEvolve*
$\langle 2,2,2\rangle$	7 [95]	7
$\langle 2,2,3\rangle$	11 [93]	11
$\langle 2,2,4\rangle$	14 [93]	14
$\langle 2,2,5\rangle$	18 [93]	18
$\langle 2,2,6\rangle$	21 [93]	21
$\langle 2,2,7\rangle$	25 [93]	25
$\langle 2,2,8\rangle$	28 [93]	28
$\langle 2,2,9\rangle$	32 [93]	32
$\langle 2,2,10\rangle$	35 [93]	35
$\langle 2,2,11\rangle$	39 [93]	39
$\langle 2,2,12\rangle$	42 [93]	42
$\langle 2,2,13\rangle$	46 [93]	46
$\langle 2,2,14\rangle$	49 [93]	49
$\langle 2,2,15\rangle$	53 [93]	53
$\langle 2,2,16\rangle$	56 [93]	56
$\langle 2,3,3\rangle$	15 [93]	15
$\langle 2,3,4\rangle$	20 [93]	20
$\langle 2,3,5\rangle$	25 [93]	25
$\langle 2,3,6\rangle$	30 [93]	30
$\langle 2,3,7\rangle$	35 [93]	35
$\langle 2,3,8\rangle$	40 [93]	40
$\langle 2,3,9\rangle$	45 [93]	45
$\langle 2,3,10\rangle$	50 [93]	50
$\langle 2,4,4\rangle$	26 [93]	26
$\langle 2,4,5\rangle$	33 [42]	**32**
$\langle 2,4,6\rangle$	39 [93]	39
$\langle 2,4,7\rangle$	46 [93]	**45**
$\langle 2,4,8\rangle$	52 [93]	**51**
$\langle 2,5,5\rangle$	40 [93]	40
$\langle 2,5,6\rangle$	48 [93]	**47**
$\langle 3,3,3\rangle$	23 [52]	23
$\langle 3,3,4\rangle$	29 [93]	29
$\langle 3,3,5\rangle$	36 [93]	36
$\langle 3,3,6\rangle$	40 [93]	40
$\langle 3,3,7\rangle$	49 [93]	49
$\langle 3,3,8\rangle$	55 [93]	55
$\langle 3,4,4\rangle$	38 [93]	38
$\langle 3,4,5\rangle$	47 [26]	47
$\langle 3,4,6\rangle$	56 [48]	**54**
$\langle 3,4,7\rangle$	66 [91]	**63**
$\langle 3,4,8\rangle$	75 [91]	**74**
$\langle 3,5,5\rangle$	58 [91]	58
$\langle 3,5,6\rangle$	70 [48]	**68**
$\langle 3,5,7\rangle$	82 [91]	**80**
$\langle 4,4,4\rangle$	49 [95]	**48**
$\langle 4,4,5\rangle$	62 [47]	**61**
$\langle 4,4,6\rangle$	73 [48]	73
$\langle 4,4,7\rangle$	87 [93, 95]	**85**
$\langle 4,4,8\rangle$	98 [95]	**96**
$\langle 4,4,9\rangle$	104 [92]	108
$\langle 4,5,5\rangle$	76 [26]	76
$\langle 4,5,6\rangle$	93 [48]	**90**
$\langle 5,5,5\rangle$	93 [72]	93
$\langle 6,6,6\rangle$	153 [72]	156

Table 3 | Full version of Table 2, showing the best ranks obtained by *AlphaEvolve* for tensor decomposition for all considered parameters. Of the 54 targets, *AlphaEvolve* matches the state of the art in 38 cases, surpasses it in 14 cases (green), and falls behind in 2 cases (red). In all cases, *AlphaEvolve* provides exact algorithms, using integer or half-integer entries in the decomposition. For $\langle 3, 4, 7\rangle$, $\langle 4, 4, 4\rangle$, and $\langle 4, 4, 8\rangle$, the algorithms discovered by *AlphaEvolve* use complex-valued multiplications which can be used for exact multiplication of complex or real-valued matrices. The decompositions shown in this table can be found in the accompanying Google Colab.

Note: Concurrent work [49] has also found a rank-90 algorithm for $\langle 4, 5, 6\rangle$.

Magnified version of Figure 4 (left). In Figures 9a to 9c, we show a magnified version of Figure 4 (left), which corresponds to the program that discovers a decomposition of rank 48 for the 3D tensor representing the operation of multiplying two 4×4 matrices.

```
@@ -45,9 +45,14 @@
    # EVOLVE-BLOCK-START
    def _get_optimizer(self) -> optax.GradientTransformation:
      """Returns optimizer."""
-     return optax.adam(self.hypers.learning_rate)
+     return optax.adamw(
+         self.hypers.learning_rate, weight_decay=self.hypers.weight_decay
+     )

    def _get_init_fn(self) -> jax.nn.initializers.Initializer:
      """Returns initializer function."""
-     return initializers.normal(0.0, self.hypers.init_scale, jnp.complex64)
+     # Initialize with a smaller scale to encourage finding low-rank
     solutions.
+     # Increase scale slightly for better exploration.
+     scale = self.hypers.init_scale
+     return initializers.normal(0 + 1j * 0, scale * 0.2, jnp.complex64)

@@ -80,6 +85,66 @@
      # Gradient updates.
      updates, opt_state = self.opt.update(grads, opt_state, decomposition)
      decomposition = optax.apply_updates(decomposition, updates)
+     # Add a small amount of gradient noise to help with exploration
+     rng, g_noise_rng = jax.random.split(rng)
+     decomposition = jax.tree_util.tree_map(
+         lambda x: x
+         + self.hypers.grad_noise_std * jax.random.normal(g_noise_rng, x.
     shape),
+         decomposition,
+     )
+
+     # Add noise to the decomposition parameters (exploration).
+     _, noise_rng = jax.random.split(rng)
+     noise_std = self._linear_schedule(
+         global_step, start=self.hypers.noise_std, end=0.0
+     )
+     decomposition = jax.tree_util.tree_map(
+         lambda x: x + noise_std * jax.random.normal(noise_rng, x.shape),
+         decomposition,
+     )
+
+     # Cyclical annealing for clipping threshold.
+     cycle_length = 2000  # Number of steps per cycle
+     cycle_progress = (
+         global_step % cycle_length
+     ) / cycle_length  # Normalized progress within the current cycle [0,
     1)
+
+     # Map cycle progress to a sinusoidal curve. Ranges from 0 to 1.
+     clip_threshold_multiplier = (1 + jnp.cos(2 * jnp.pi * cycle_progress))
      / 2
+
+     clip_threshold = self.hypers.clip_min + clip_threshold_multiplier * (
+         self.hypers.clip_max - self.hypers.clip_min
+     )
+
+     def soft_clip(x, threshold):
+       # Clipping the real and imaginary parts separately.
+       x_re = jnp.real(x)
+       x_im = jnp.imag(x)
+
+       x_re_clipped = jnp.where(
+           x_re > threshold, threshold + (x_re - threshold) * 0.1, x_re
+       )
+       x_re_clipped = jnp.where(
+           x_re_clipped < -threshold,
+           -threshold + (x_re_clipped + threshold) * 0.1,
+           x_re_clipped,
+       )
```

Figure 9a | Magnified version of Figure 4(left), giving the program that discovers a faster algorithm to multiply 4×4 matrices (*1/3*).

```
+
+      x_im_clipped = jnp.where(
+          x_im > threshold, threshold + (x_im - threshold) * 0.1, x_im
+      )
+      x_im_clipped = jnp.where(
+          x_im_clipped < -threshold,
+          -threshold + (x_im_clipped + threshold) * 0.1,
+          x_im_clipped,
+      )
+
+      return x_re_clipped + 1j * x_im_clipped
+
+    decomposition = jax.tree_util.tree_map(
+        lambda x: soft_clip(x, clip_threshold), decomposition
+    )
+
     return decomposition, opt_state, loss

   def _loss_fn(
@@ -91,13 +156,86 @@
     """Computes (batched) loss on learned decomposition."""
     # Compute reconstruction loss.
     rec_tensor = self._decomposition_to_tensor(decomposition)  # (B, N, M, P)
+
+    # Add noise to the target tensor (robustness).
+    rng, noise_rng = jax.random.split(rng)
+    target_noise = self.hypers.target_noise_std * jax.random.normal(
+        noise_rng, self.target_tensor.shape
+    )
+    noisy_target_tensor = self.target_tensor + target_noise
+
+    # Hallucination loss (encourages exploration by randomly replacing values)
+    hallucination_prob = self.hypers.hallucination_prob
+    hallucination_scale = self.hypers.hallucination_scale
+
+    def hallucinate(x, hallucination_rng):
+      mask = jax.random.bernoulli(hallucination_rng, p=hallucination_prob)
+      noise = hallucination_scale * jax.random.normal(
+          hallucination_rng, x.shape
+      )
+      return jnp.where(mask, noise, x)
+
+    _, factor_rng = jax.random.split(rng)
+    decomposition = jax.tree_util.tree_map(
+        lambda x: hallucinate(x, jax.random.split(factor_rng)[0]),
+        decomposition,
+    )
+
     # Add a batch dimension to `target_tensor` to ensure correct broadcasting.
     # Define the loss as the L2 reconstruction error.
-    rec_loss = l2_loss_complex(self.target_tensor[None, ...], rec_tensor)
+    rec_loss = l2_loss_complex(noisy_target_tensor[None, ...], rec_tensor)

     # We must return a real-valued loss.
-    return jnp.real(rec_loss)

+    # Discretization loss (encourage entries to be multiples of 1/2 or integer).
+    def dist_to_half_ints(x):
+      x_re = jnp.real(x)
+      x_im = jnp.imag(x)
+      return jnp.minimum(
+          jnp.abs(x_re - jnp.round(x_re * 2) / 2),
+          jnp.abs(x_im - jnp.round(x_im * 2) / 2),
+      )
+
```

Figure 9b | Magnified version of Figure 4(left), giving the program that discovers a faster algorithm to multiply 4×4 matrices (*2/3*).

```
+    def dist_to_ints(x):
+      return jnp.abs(x - jnp.round(x))
+
+    discretization_loss = 0.0
+    for factor in decomposition:
+      discretization_loss += jnp.mean(dist_to_half_ints(factor))
+      discretization_loss += jnp.mean(dist_to_ints(factor))
+
+    discretization_loss /= (
+        len(decomposition) * 2
+    )  # average across all factors and loss components
+
+    discretization_weight = self._linear_schedule(
+        global_step, start=0.0, end=self.hypers.discretization_weight
+    )
+
+    # Cosine annealing for half-integer loss.
+    cycle_length = self.config.training_steps // 4  # Number of steps per cycle
+    cycle_progress = (
+        global_step % cycle_length
+    ) / cycle_length  # Normalized progress within the current cycle [0, 1)
+    half_int_multiplier = (1 + jnp.cos(jnp.pi * cycle_progress)) / 2
+    half_int_multiplier = (
+        1 - self.hypers.half_int_start
+    ) * half_int_multiplier + self.hypers.half_int_start
+
+    total_loss = (
+        rec_loss
+        + discretization_weight * discretization_loss * half_int_multiplier
+    )
+
+    # Add penalty for large values (stability).
+    large_value_penalty = 0.0
+    for factor in decomposition:
+      large_value_penalty += jnp.mean(jnp.abs(factor) ** 2)
+    large_value_penalty /= len(decomposition)
+    total_loss += self.hypers.large_value_penalty_weight * large_value_penalty
+
+    return jnp.real(total_loss)
+

 def l2_loss_complex(x: jnp.ndarray, y: jnp.ndarray) -> jnp.ndarray:
   """Elementwise L2 loss for complex numbers."""
@@ -117,6 +255,18 @@
   return hyper.zipit([
-      hyper.uniform('init_scale', hyper.interval(0.2, 1.5)),
-      hyper.uniform('learning_rate', hyper.interval(0.05, 0.3)),
+      hyper.uniform('init_scale', hyper.interval(0.1, 1.0)),
+      hyper.uniform('learning_rate', hyper.interval(0.01, 0.2)),
+      hyper.uniform('discretization_weight', hyper.interval(0.0, 0.1)),
+      hyper.uniform('hallucination_prob', hyper.interval(0.0, 0.2)),
+      hyper.uniform('hallucination_scale', hyper.interval(0.0, 0.2)),
+      hyper.uniform('noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('target_noise_std', hyper.interval(0.0, 0.01)),
+      hyper.uniform('weight_decay', hyper.interval(0.00001, 0.001)),
+      hyper.uniform('clip_min', hyper.interval(0.0, 0.5)),
+      hyper.uniform('clip_max', hyper.interval(1.0, 3.0)),
+      hyper.uniform('large_value_penalty_weight', hyper.interval(0.0, 0.01)),
+      # Add noise to the gradient to aid in exploration.
+      hyper.uniform('grad_noise_std', hyper.interval(0.0, 0.001)),
+      hyper.uniform('half_int_start', hyper.interval(0.0, 1.0)),
   ])
 # EVOLVE-BLOCK-END
```

Figure 9c | Magnified version of Figure 4(left), giving the program that discovers a faster algorithm to multiply 4×4 matrices (*3/3*). Here `hyper` is a user-provided library for generating hyperparameter sweeps.

B. Details of mathematical discoveries of *AlphaEvolve*

The data and verification code for all constructions reported in this section appear in the accompanying Google Colab.

B.1. First autocorrelation inequality

For any function $f : \mathbb{R} \to \mathbb{R}$, define the *autoconvolution* of f as

$$f * f(t) := \int_{\mathbb{R}} f(t-x)f(x)\,dx.$$

Let C_1 denote the largest constant satisfying

$$\max_{-1/2\le t\le 1/2} f * f(t) \ge C_1 \left(\int_{-1/4}^{1/4} f(x)\,dx\right)^2 \tag{1}$$

for all non-negative $f : \mathbb{R} \to \mathbb{R}$. This problem arises in additive combinatorics, relating to the size of Sidon sets. It is currently known that

$$1.28 \le C_1 \le 1.5098,$$

with the lower bound achieved in [17] and the upper bound achieved in [70] via a step function construction. *AlphaEvolve* found a step function with 600 equally-spaced intervals on $[-1/4, 1/4]$ that gives a slightly better upper bound $C_1 \le 1.5053$.

B.2. Second autocorrelation inequality

Let C_2 be the smallest constant for which one has

$$\|f * f\|_2^2 \le C_2 \|f * f\|_1 \|f * f\|_\infty$$

for all non-negative $f : \mathbb{R} \to \mathbb{R}$. It is known that

$$0.88922 \le C_2 \le 1$$

with the lower bound coming from a step function construction [70]. *AlphaEvolve* found a step function with 50 equally-spaced intervals on $[-1/4, 1/4]$ that gives a slightly better lower bound $0.8962 \le C_2$.

B.3. Third autocorrelation inequality

Let C_3 be the largest constant satisfying

$$\max_{-1/2\le t\le 1/2} |f * f(t)| \ge C_3 \left(\int_{-1/4}^{1/4} f(x)\,dx\right)^2$$

for any function $f : \mathbb{R} \to \mathbb{R}$. Clearly $C_3 \le C_1$, since we now allow f to take positive and negative values. There is a step function that gives the upper bound $C_3 \le 1.45810$ [104, page 75]. *AlphaEvolve* found a step function with 400 equally-spaced intervals on $[-1/4, 1/4]$ that gives a slightly better upper bound $C_3 \le 1.4557$.

B.4. An uncertainty inequality

Given a function $f : \mathbb{R} \to \mathbb{R}$, define the Fourier transform $\hat{f}(\xi) := \int_{\mathbb{R}} f(x)e^{-2\pi i x\xi}\, dx$ and

$$A(f) := \inf\{r > 0 : f(x) \geq 0 \text{ for all } |x| \geq r\}.$$

Let C_4 be the largest constant for which one has

$$A(f)A(\hat{f}) \geq C_4$$

for all even f with $\max(f(0), \hat{f}(0)) < 0$. It is known [33] that

$$0.2025 \leq C_4 \leq 0.3523.$$

(The upper bound is stated as 0.353 in the paper, but rounding their solution to the fourth digit gives 0.3523). We improved the upper bound to $C_4 \leq 0.3521$ with a similar linear combination as in [33], but with refined constants that were found by *AlphaEvolve*.

To obtain upper bounds for C_4, one constructs a specific "test function" f satisfying the conditions and calculates the value $A(f)A(\hat{f})$ for this function, which provides an upper bound $C_4 \leq A(f)A(\hat{f})$. Following the approach in [33], the test function is sought in the form $f(x) = P(x)e^{-\pi x^2}$, where $P(x)$ is an even polynomial constructed as a linear combination of Hermite polynomials $H_{4k}(x)$. This form is particularly useful because the Fourier transform of $H_n(x)e^{-\pi x^2}$ is $i^n H_n(\xi)e^{-\pi\xi^2}$. For an even polynomial $P(x) = \sum c_{4k}H_{4k}(x)$, the Fourier transform of $f(x)$ is $\hat{f}(\xi) = \sum c_{4k}i^{4k}H_{4k}(\xi)e^{-\pi\xi^2} = (\sum c_{4k}H_{4k}(\xi))e^{-\pi\xi^2} = P(\xi)e^{-\pi\xi^2}$. Thus, $A(f)$ is related to the largest positive root of $P(x)$, and $A(\hat{f})$ is related to the largest positive root of $P(\xi)$. Specifically, if $P(x) \geq 0$ for large $|x|$, $A(f)$ is the largest positive root of $P(x)$, and $A(\hat{f})$ is the largest positive root of $P(\xi)$, implying $A(f) = A(\hat{f})$. The inequality becomes $C_4 \leq (A(f))^2$.

The method involves finding coefficients $c_0, c_1, c_2, \ldots$ for the polynomial $P(x) = c_0H_0(x) + c_1H_4(x) + c_2H_8(x) + \ldots$ such that $P(x)$ satisfies certain constraints (related to $f(0) < 0$, $\hat{f}(0) < 0$ and being positive for large $|x|$) and minimizes the largest positive root of $P(x)$. In our approach, the polynomial $P(x)$ is constructed such that $P(0) = 0$ (a condition used in the optimization process to simplify constraints), meaning $P(x)$ has a factor of x^2. The largest positive root $r_{\max}$ of $P(x)$ is then the largest positive root of $P(x)/x^2$. The upper bound on C_4 derived from this construction is $r_{\max}^2/(2\pi)$.

The refined constants found by *AlphaEvolve* for $P(x) = c_0H_0(x) + c_1H_4(x) + c_2H_8(x)$ are $[c_0, c_1, c_2] \approx [0.32925, -0.01159, -8.9216 \times 10^{-5}]$. Using these coefficients to construct $P(x)$, finding its largest positive root $r_{\max}$ (by finding the largest positive root of $P(x)/x^2$), and calculating $r_{\max}^2/(2\pi)$ yields the improved upper bound $C_4 \leq 0.3521$. Qualitatively our linear combination is very similar to the one found in [33], thus empirically confirming their hypothesis the construction is nearly optimal.

Note: After publishing the first version of this manuscript, Henry Cohn pointed out that in a recent paper [18] they used a similar, but more refined approach to get the better constant 0.3284. By incorporating their refined approach into *AlphaEvolve*, we improved our reported constant further to 0.3216. For details, we refer to the accompanying Google Colab.

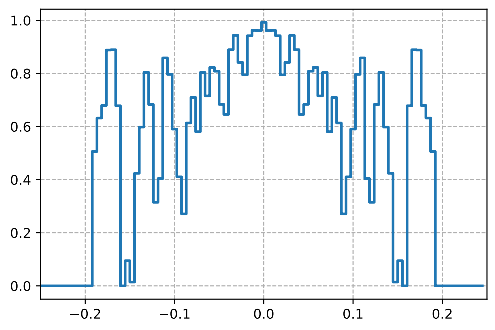


Figure 10 | Construction found by *AlphaEvolve* for the minimum overlap problem of Erdős.

B.5. Erdős' minimum overlap problem

Let C_5 be the largest constant for which

$$\sup_{x\in[-2,2]} \int_{-1}^{1} f(t)g(x+t)\,dt \geq C_5$$

for all non-negative $f, g : [-1,1] \to [0,1]$ with $f + g = 1$ on $[-1,1]$ and $\int f = 1$, where we extend f, g by zero outside of $[-1,1]$. This constant controls the asymptotics of the Minimum Overlap Problem of [25]. The bounds

$$0.379005 \leq C_5 \leq 0.380927$$

are known, where the lower bound was obtained in [107] via convex programming methods.

It is known (see [40]) that this constant is equal to the infimum, over all step functions h on $[0,2]$ with values in $[0,1]$ and satisfying $\int_0^2 h(x)dx = 1$ of

$$\max_k \int h(x)(1 - h(x+k))dx.$$

The upper bound to the Erdős minimum overlap problem was then obtained by using this result, in [40] by a step function construction. The step function depicted in Figure 10 does ever so slightly better than the previous bound, giving the upper bound of $C_5 \leq 0.380924$.

B.6. Sums and differences of finite sets

Let C_6 be the largest constant for which the following statement holds: there exist arbitrarily large finite sets of integers A, B with $|A+B| \ll |A|$ and $|A-B| \gg |A+B|^{C_6}$. (Here $A + B = \{a + b : a \in A, b \in B\}$ and $A - B = \{a - b : a \in A, b \in B\}$ denote the sumset and difference set, respectively. The notation $X \ll Y$ means that $X \leq CY$ for some constant C independent of the sets A, B (for sufficiently large sets A, B). The notation $X \gg Y$ means that $X \geq C'Y$ for some positive constant C' independent of the sets A, B (for sufficiently large sets A, B).)

$$1.14465 \leq C_6 \leq \frac{4}{3}; \tag{2}$$

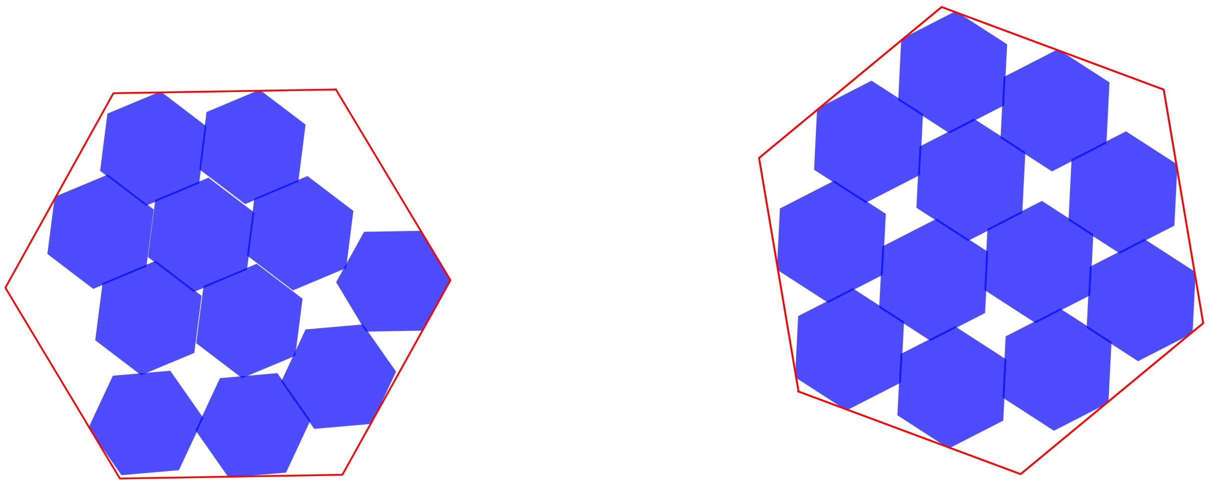


Figure 11 | Constructions of the packing problems found by *AlphaEvolve*. Left: Packing 11 unit hexagons into a regular hexagon of side length 3.931. Right: Packing 12 unit hexagons into a regular hexagon of side length 3.942.

see [39, Corollary 3] for the upper bound and [39, Theorem 1] for the lower bound. The main tool for the lower bound is the following result of Gyarmati et al. [39]:

$$C_6 \geq 1 + \frac{\log \frac{|U-U|}{|U+U|}}{\log(2\max(U)+1)} \tag{3}$$

for any finite set U of non-negative integers containing zero satisfying $|U-U| \leq 2\max(U)+1$. *AlphaEvolve* found a set U_1 of size 2003 improving the lower bound to $1.1479 \leq C_6$, and another set U_2 of size 54265 further improving the lower bound to $1.1584 \leq C_6$.

B.7. Packing unit regular hexagons inside a regular hexagon

Consider the problem of packing n disjoint regular hexagons with unit side length into a larger regular hexagon, minimizing the side length of the outer hexagon. For $n = 11$ and $n = 12$, the best known constructions use outer hexagons of side lengths 3.943 and 4.0, respectively [29]. *AlphaEvolve* found packing arrangements that improve these bounds to 3.931 and 3.942, respectively. These arrangements are shown in Figure 11.

B.8. Minimizing the ratio of maximum to minimum distance

For any n and d, the goal of this problem is to find n points in the d-dimensional space so as to minimize the ratio between the maximum and minimum pairwise distances. *AlphaEvolve* found two new constructions improving the best known bounds. The found constructions are shown in Figure 12.

In 2 dimensions, *AlphaEvolve* found 16 points with ratio $\approx \sqrt{12.889266112}$, improving the best known bound of $\sqrt{12.890}$ [29]. (In this reference, instead of the ratio itself, the square of the ratio is reported, and we use the same convention.)

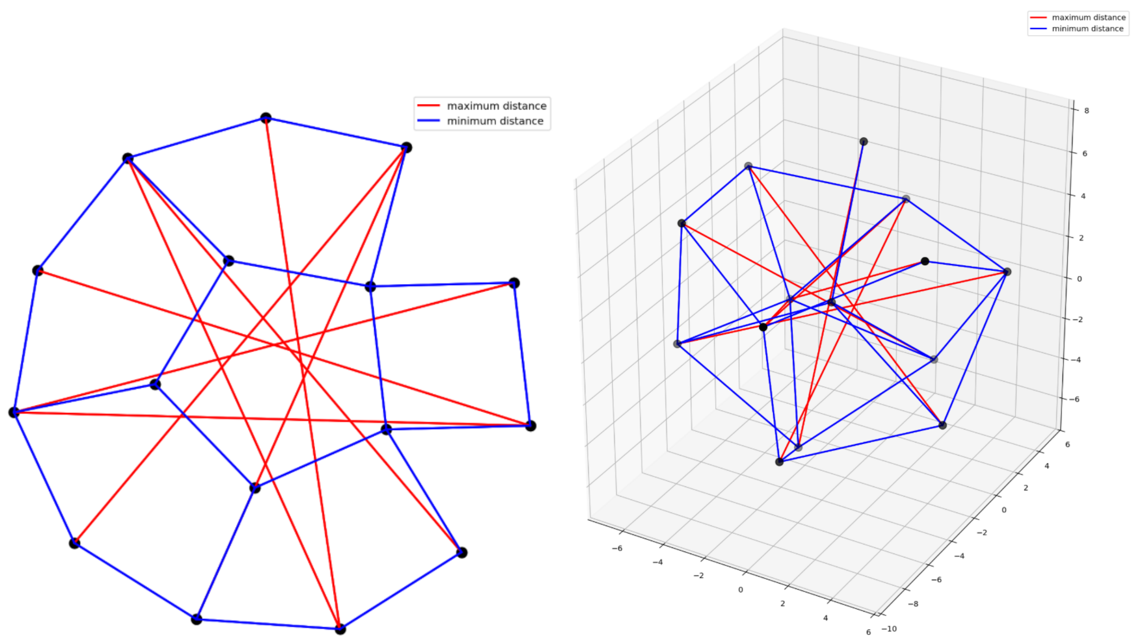


Figure 12 | Left: 16 points in 2 dimensions achieving a ratio of maximum distance to minimum distance of $\approx \sqrt{12.889266112}$. Right: 14 points in 3 dimensions achieving a ratio of $\approx \sqrt{4.165849767}$. Both constructions improve the best known bounds.

In 3 dimensions, *AlphaEvolve* found 14 points with ratio $\approx \sqrt{4.165849767}$, improving the best known bound of $\sqrt{4.168}$ [29].

B.9. The Heilbronn problem for triangles

The goal of this problem is to find n points on or inside a triangle with unit area so that the area of the smallest triangle formed by these points is maximized. For $n = 11$, the SOTA was 0.036 [29], and AlphaEvolve found a construction with minimum area larger than 0.0365, which is shown in Figure 13 (left).

B.10. The Heilbronn problem for convex regions

The goal of this problem is to find n points on or inside a convex region with unit area so that the area of the smallest triangle formed by these points is maximized. *AlphaEvolve* improved two of the best known bounds.

For $n = 13$, the SOTA was 0.0306 [29], and *AlphaEvolve* improved it to 0.0309 (see Figure 13 (middle)). For $n = 14$, the SOTA was 0.0277 [29] and *AlphaEvolve* improved it to 0.0278 (see Figure 13 (right)).

B.11. Kissing number in dimension 11

The kissing problem asks how many disjoint unit spheres can be packed tangent to a given unit sphere. The maximum such number in d dimensions is called the *d-dimensional kissing number [8]*. For $d = 11$, the best known lower bound was 592 [31] and *AlphaEvolve* improved

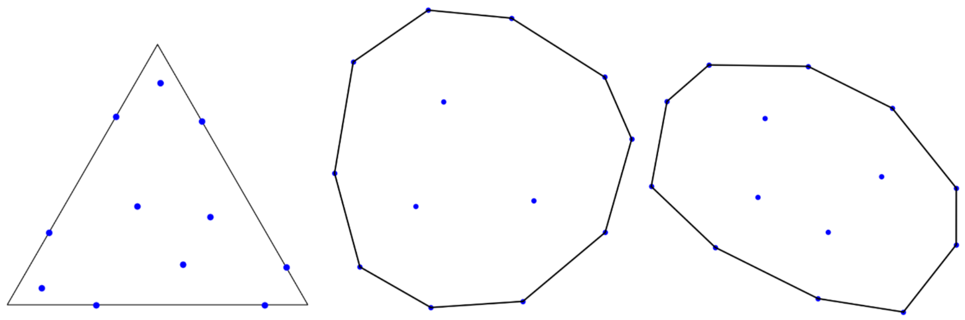


Figure 13 | New constructions found by *AlphaEvolve* improving the best known bounds on two variants of the Heilbronn problem. Left: 11 points in a unit-area triangle with all formed triangles having area ≥ 0.0365. Middle: 13 points inside a convex region with unit area with all formed triangles having area ≥ 0.0309. Right: 14 points inside a unit convex region with minimum area ≥ 0.0278.

this to 593. To prove the lower bound of 593 for the kissing number in dimension 11, *AlphaEvolve* found 593 many 11-dimensional non-zero points with integral coordinates such that the maximum norm of these points is smaller than their minimum pairwise distance. By the following lemma, this implies the kissing number in dimension 11 is at least 593.

Lemma 1. *Let $C \subset \mathbb{R}^d$ be a set of points satisfying $0 \notin C$ and*

$$\min\{\|x-y\| : x \neq y \in C\} \geq \max\{\|x\| : x \in C\}.$$

Then unit spheres centred at $\left\{\frac{2x}{\|x\|} : x \in C\right\}$ form a valid kissing configuration in dimension d. In particular, the kissing number in dimension d is at least $|C|$.

Proof. For any $x \neq y \in C$, the inequality $\|x-y\|^2 \geq \max\{\|x\|^2, \|y\|^2\}$ implies

$$2\langle x, y\rangle \leq \|x\|^2 + \|y\|^2 - \max\{\|x\|^2, \|y\|^2\} = \min\{\|x\|^2, \|y\|^2\} \leq \|x\| \cdot \|y\|, \tag{4}$$

where the last inequality holds because the minimum of two positive numbers is less than or equal to their geometric mean. The points $\left\{\frac{2x}{\|x\|} : x \in C\right\}$ have norm 2, so unit spheres centred at them are tangent to a unit sphere centred at the origin. The last step is to show that these spheres do not overlap. This is equivalent to showing, for all $x \neq y \in C$, that

$$\left\|\frac{2x}{\|x\|} - \frac{2y}{\|y\|}\right\| \geq 2.$$

After simplifying, this is equivalent to $2\langle x, y\rangle \leq \|x\| \cdot \|y\|$, which we have proved in (4). Thus unit spheres centred at $\left\{\frac{2x}{\|x\|} : x \in C\right\}$ form a valid kissing configuration in dimension d, as required. □

B.12. Packing circles inside a unit square to maximize sum of radii

Given a positive integer n, the problem is to pack n disjoint circles inside a unit square so as to maximize the sum of their radii. *AlphaEvolve* found two new constructions improving the state of the art [29].

For $n = 26$, the SOTA was 2.634, and AlphaEvolve improved it to 2.635; see Figure 14 (left). For $n = 32$, the SOTA was 2.936, and AlphaEvolve improved it to 2.937; see Figure 14 (middle).

B.13. Packing circles inside a rectangle of perimeter 4 to maximize sum of radii

Given a positive integer n, the problem is to pack n disjoint circles inside a rectangle of perimeter 4 so as to maximize the sum of their radii. *AlphaEvolve* found a new construction for $n = 21$, improving the state of the art from 2.364 [29] to 2.3658; see Figure 14 (right).

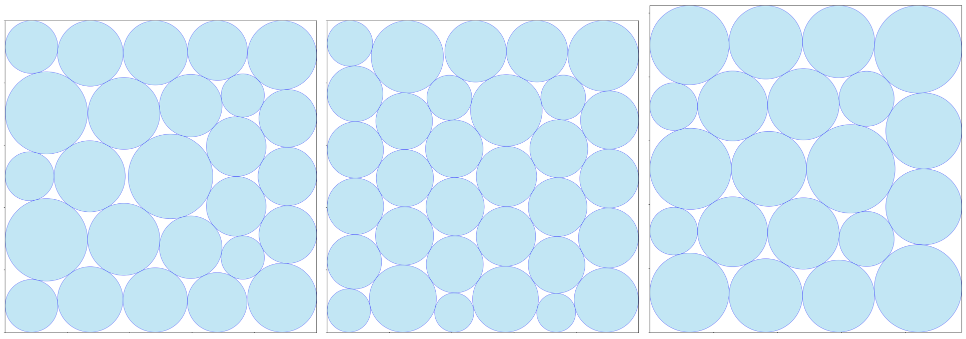


Figure 14 | New constructions found by *AlphaEvolve* improving the best known bounds on packing circles to maximize their sum of radii. Left: 26 circles in a unit square with sum of radii ≥ 2.635. Middle: 32 circles in a unit square with sum of radii ≥ 2.937. Right: 21 circles in a rectangle with perimeter 4, with sum of radii ≥ 2.365.