Machine Discovery and Machine Construction

AI for Mathematics (General Elective) — Week 13

30 June 2026

Roadmap — Week 13

Two complementary strands of *conjecture-grade* AI:

Part A — Discovery Find a regularity hidden in mathematical data: a numerical identity, a closed-form formula, a structural rule, a new conjecture. The AI proposes; a human (or a prover) verifies.

Part B — Construction Find an explicit mathematical object or algorithm that beats a known record: extremal graphs, fast matrix-multiply schemes, large cap sets, better sorting code. Search guided by a language model, evaluated by an exact checker.

Common pattern across both: a *cheap exact evaluator* converts the problem into a fitness landscape an LLM-driven loop can climb.

Part A.1 — Relations among numbers

PSLQ: finding a relation among numbers

PSLQ detects **integer relations**. Given reals $x_1, \ldots, x_n$ to high precision, it returns integers $a_1, \ldots, a_n$ (not all zero) with

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0,$$

or certifies that none exists below a height bound.

- Ferguson–Forcade (1979); the stable PSLQ is Ferguson–Bailey–Arno (1999); related to LLL (Lenstra–Lenstra–Lovász) lattice reduction.
- Feed it a constant's value plus candidate terms (π, logs, ζ values, powers); it returns an exact-looking identity.
- Named one of the "top ten algorithms of the century" (2000).

The identity is a candidate: true to the computed precision, still to be proved.

An integer relation is a short vector in a lattice

Embed the numbers: take $\mathbf{b}_i = (\mathbf{e}_i,\ Nx_i)$, where $\mathbf{e}_i = (0, \ldots, 1, \ldots, 0)$ is the i-th unit vector and N is large. Each $\mathbf{b}_i$ is an $(n{+}1)$-vector, so

$$\sum_i a_i \mathbf{b}_i = \big(\underbrace{a_1, \ldots, a_n}_{\text{from the } \mathbf{e}_i},\ \underbrace{N \textstyle\sum_i a_i x_i}_{\text{last coordinate}} \big).$$

If $\sum_i a_i x_i \approx 0$ with small a_i, this lattice vector is **short**.

So **an integer relation** $=$ **a short vector in this lattice**, and the goal is to find the shortest one. That is what lattice reduction (LLL) does.

LLL: reduce the basis

Input a lattice basis $\mathbf{b}_1, \ldots, \mathbf{b}_n$.

Output a basis of the same lattice with a short first vector, $\|\mathbf{b}_1\| \le 2^{(n-1)/2}\lambda_1$ (λ_1 = true shortest length); $\mathbf{b}_1$ is the candidate relation.

Minimizes the potential $D = \prod_k \prod_{i \le k} \|\mathbf{b}_i^*\|^2$ (product of the partial Gram determinants).

($\mathbf{b}_i^*$: Gram–Schmidt vectors; $\mu_{ij} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle / \langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle$.)

A descent on D, alternating two moves until neither applies:

- **Size-reduce**: $\mathbf{b}_i \leftarrow \mathbf{b}_i - \lfloor \mu_{ij} \rceil \mathbf{b}_j$ until $|\mu_{ij}| \le \frac{1}{2}$ — near-orthogonal to earlier vectors; D fixed.
- **Lovász swap**: if $\|\mathbf{b}_i^*\|^2 < (\delta - \mu_{i,i-1}^2)\|\mathbf{b}_{i-1}^*\|^2$ ($\delta = \frac{3}{4}$), swap $\mathbf{b}_{i-1} \leftrightarrow \mathbf{b}_i$ — the shorter vector moves up; each swap shrinks D by a factor $< \delta$, so the loop stops.

PSLQ: input, output, objective

Input reals $x_1, \ldots, x_n$ to high precision.

Output a nonzero integer vector $\mathbf{a}$ with $\sum_i a_i x_i = 0$; or a bound M with every relation $\|\mathbf{a}\| \geq M$ (none below height M).

Objective the smallest integer relation, $\min \|\mathbf{a}\|$ over $0 \neq \mathbf{a} \in \mathbb{Z}^n$ with $\sum_i a_i x_i = 0$ — the same shortest vector LLL seeks.

The difference from LLL is in *how*: PSLQ reduces the real numbers x_i directly, kept to high precision, instead of building LLL's integer lattice $\mathbf{b}_i = (\mathbf{e}_i, N x_i)$. So there is no large multiplier N to pick or inflate, and the arithmetic stays numerically stable.

PSLQ: the matrix $H_{\mathbf{x}}$

Every integer relation lies in the **relation space** $\mathbf{x}^{\perp} = \{\mathbf{v} \in \mathbb{R}^n : \mathbf{v} \cdot \mathbf{x} = 0\}$. Normalize $|\mathbf{x}| = 1$ and set $s_k = \sqrt{x_k^2 + \cdots + x_n^2}$ (so $s_1 = 1$). These define the $n \times (n-1)$ matrix

$$(H_{\mathbf{x}})_{jj} = \frac{s_{j+1}}{s_j}, \quad (H_{\mathbf{x}})_{ij} = -\frac{x_i x_j}{s_j s_{j+1}} \ (i > j), \quad (H_{\mathbf{x}})_{ij} = 0 \ (i < j),$$

whose $n-1$ columns are orthonormal and orthogonal to $\mathbf{x}$ — a basis of $\mathbf{x}^{\perp}$.

Certificate (Ferguson–Bailey–Arno): for any invertible integer matrix A and orthogonal Q with $AH_{\mathbf{x}}Q$ lower-trapezoidal (zero above the diagonal), every relation $\mathbf{m}$ satisfies $|\mathbf{m}| \geq 1/\max_j |(AH_{\mathbf{x}}Q)_{jj}|$.

PSLQ: the loop

Keep an integer matrix A (with its inverse A^{-1}) and an orthogonal Q; write $H = AH_{\mathbf{x}}Q$. Fix a parameter $\gamma > 2/\sqrt{3}$, then repeat:

Reduce subtract rounded integer multiples of earlier rows ($A \leftarrow DA$, D integer) until $|H_{ij}| \le \frac{1}{2}|H_{jj}|$ for $i > j$.

Exchange pick r maximizing $\gamma^r |H_{rr}|$; swap rows $r, r{+}1$ of A.

Corner the swap breaks the lower-trapezoidal form; a plane (Givens) rotation $Q \leftarrow QP$ on columns $r, r{+}1$ restores it (the "LQ").

Stop when $\mathbf{x}A^{-1}$ has a zero entry: that column of A^{-1} is the relation $\mathbf{m}$. Until then $1/\max_j |H_{jj}|$ lower-bounds every relation — the "no relation below height M" certificate.

A worked discovery: the BBP formula for π

The Bailey–Borwein–Plouffe (BBP) formula:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

- Found numerically in 1995 (Plouffe) by an integer-relation search; **proved** after the discovery, and published with the proof in 1997.
- Gives the n-th hexadecimal digit of π — and the $4n$-th binary digit — without the earlier digits, in little memory.
- Base 16, not base 10: cheap decimal-digit extraction is still open.

Discovered from the numbers, proved afterward.

PSLQ: rediscovering the BBP formula

Feed PSLQ the eight building blocks $S_j = \sum_{k \geq 0} 1/(16^k(8k+j))$, with no prior knowledge of the coefficients:

```
import mpmath as mp;  mp.mp.dps = 100
S = [mp.nsum(lambda k, j=j: 1/(mp.mpf(16)**k*(8*k+j)), [0, mp.inf])
     for j in range(1, 9)]              # S1 .. S8
rel = mp.pslq([mp.pi] + S, maxcoeff=10**6)
```

Output:

```
rel = [1, -4, 0, 0, 2, 1, 1, 0, 0]   ==  published BBP
  ->  pi = 4*S1 - 2*S4 - S5 - S6        (rel. error vs pi: 9e-102)
spigot: pi hex digits at offset 999 = 349F1C   (matches mpmath)
PSLQ finds nothing below 16 digits, then locks on (height 4)
```

The exact formula comes back from the numbers alone; the spigot step then extracts the n-th hexadecimal digit of π directly.

From computed numbers to a closed form: OEIS, LIReC

- **OEIS**, the On-Line Encyclopedia of Integer Sequences (oeis.org; Sloane; over $390{,}000$ sequences) — type the first terms of a computed sequence; it returns matches with conjectured formulas, generating functions, recurrences, and references.
- **LIReC**, the Library of Integer RElations and Constants (github.com/RamanujanMachine/LIReC) — a database of constants and a hypergraph of PSLQ-found relations; a 2024 sweep reported 75 previously unknown connections, including new π–e relations.

LIReC: recovering an integer relation

Three constants to 190 digits — $\zeta(3)$, $\zeta(5)$, and $x = 2 + 3\zeta(3) - 7\zeta(5)$ built from them — handed to LIReC's PSLQ core:

```
from LIReC.lib.pslq_utils import PreciseConstant as C, check_consts
import mpmath as mp;  mp.mp.dps = 200
z3 = C(mp.zeta(3), 190, 'zeta3')
z5 = C(mp.zeta(5), 190, 'zeta5')
x  = C(2 + 3*mp.zeta(3) - 7*mp.zeta(5), 190, 'x')
check_consts([z3, z5, x], degree=1)
```

Output (run on the cluster, mu012):

```
[ x - 3*zeta3 + 7*zeta5 - 2 = 0   (188) ]
```

The integer relation comes back exactly, valid to 188 digits. Put an unknown constant in place of x and the same call is a discovery.

Part A.2 — Formulas from data

Symbolic regression: a formula from data

Input numeric pairs (x_i, y_i) and an operator set $\{+, -, \times, \div, \exp, \log, \sqrt{}\}$.

Output a *Pareto front* of closed-form formulas f — for each complexity, the best-fitting one.

Minimizes two quantities jointly — the data misfit $\sum_i \big(f(x_i) - y_i\big)^2$ and the expression complexity (number of operators/nodes in the formula). They trade off, so the answer is a front, not one formula.

Algorithm: genetic programming over expression trees — keep a population of candidate formulas, repeatedly mutate and recombine them, and select by a fitness that rewards low error and small size.

- **PySR** (Cranmer, 2023) is this implementation; AI Feynman and SINDy (sparse identification of nonlinear dynamics) are relatives.
- It returns the functional form — no proof, no error bound; a fitted constant is pinned afterward by an integer-relation step.

Partition asymptotics

Let $p(n)$ count the partitions of n — the ways to write n as a sum of positive integers, order ignored. Its generating function is

$$\sum_{n} p(n) x^n = \prod_{k \geq 1} (1 - x^k)^{-1},$$

which lets $p(n)$ be computed exactly in integer arithmetic, with no rounding, up to $n = 20{,}000$.

The Hardy–Ramanujan formula (1918) is

$$p(n) \sim \frac{1}{4n\sqrt{3}} \exp\left(\pi \sqrt{\tfrac{2n}{3}}\right).$$

Discovery task: compute $p(n)$, take logarithms, run symbolic regression on $(n, \log p(n))$.

Euler's pentagonal number theorem

The product $\prod_{k\geq 1}(1-x^k)$ — the reciprocal of the partition generating function — expands to an almost-empty signed sum:

$$\prod_{k\geq 1}(1-x^k) = \sum_{j=-\infty}^{\infty} (-1)^j\, x^{j(3j-1)/2} = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + \cdots$$

Non-zero terms sit only at the generalized pentagonal numbers $\frac{k(3k\pm 1)}{2} = 1, 2, 5, 7, 12, 15, \ldots$, with coefficient $(-1)^k$ on the k-th pair.

The coefficient of x^n counts the partitions of n into distinct parts with sign $(-1)^{\#\text{parts}}$; even and odd cancel in pairs (Franklin's bijection, 1881), leaving ± 1 only at a pentagonal n.

Matching x^n coefficients in $\left(\sum_n p(n)x^n\right)\prod_k(1-x^k) = 1$ gives the exact recurrence

$$p(n) = p(n{-}1) + p(n{-}2) - p(n{-}5) - p(n{-}7) + p(n{-}12) + \cdots,$$

an integer sum over the $O(\sqrt{n})$ pentagonal numbers below n.

Symbolic regression recovers the Hardy–Ramanujan form

Run PySR on $(n, \log p(n))$ with no form supplied — four random seeds in parallel on the CPU cluster, minutes each. Three of the four independently recover (complexity 11, loss $\sim 10^{-10}$):

$$\log p(n) \approx 2.565099\,\sqrt{n - 0.349} - \log n - 1.9355.$$

- $\sqrt{n}$ coefficient 2.565099 vs $\pi\sqrt{2/3} = 2.5650997$; constant -1.9355 vs $-\log(4\sqrt{3}) = -1.93560$ — 5–7 significant figures.
- The -0.349 inside the root is a sub-leading correction, beyond the crude 1918 leading term.
- The proof of the formula is separate from the fit: the circle method, the modular transformation of the generating function, Rademacher's exact series (1937).

The improved asymptotic: Rademacher's leading term

The 1918 formula is only the crude leading term. The first term of Rademacher's exact series (1937) is

$$p(n) \sim \frac{\sqrt{12}}{24n-1}\left(1-\frac{1}{\mu}\right)e^{\mu}, \qquad \mu=\frac{\pi}{6}\sqrt{24n-1},$$

with relative error $\sim 10^{-14}$ by $n=2000$, where the crude term is still 1% off.

The fitted -0.349 shift is built from two refinements inside this term:

- $24n-1=24\big(n-\frac{1}{24}\big)$, so $\mu=\pi\sqrt{\frac{2}{3}}\sqrt{n-\frac{1}{24}}$: the root is shifted by $\frac{1}{24}$.
- the factor $1-\frac{1}{\mu}$ contributes a further $O(1/\sqrt{n})$ term.

Folded into a single $\sqrt{}$-shift, the two give $\frac{1}{24}+\frac{3}{\pi^2}=0.346$, against the fitted 0.349.

The Pareto front for $\log p(n)$

Symbolic regression returns the best formula at each complexity, not a single answer. For this run:

complexity	best formula	loss
4	$2.460\,\sqrt{n}$	7.5
9	$2.5652\,\sqrt{n} - \log n - 1.952$	1.0×10^{-5}
11	$2.565099\,\sqrt{n - 0.349} - \log n - 1.9355$	2.7×10^{-10}

Complexity 11 is the knee: past it, extra terms cut the loss by only $\sim 10^{-10}$. The complexity-4 row, $2.460\,\sqrt{n}$, is the leading term by itself — the search reaches the full form only after it adds the $-\log n$ term.

Recovering the constant by least squares

With the form assumed, the coefficients follow from a linear least-squares fit of $a\sqrt{n} + b\log n + c$ to $(n, \log p(n))$, $50 \le n \le 20{,}000$:

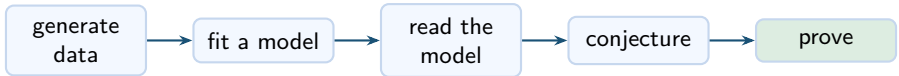
fit	a	ground truth	error
$a\sqrt{n} + b\log n + c$	2.56491	$\pi\sqrt{2/3} = 2.56510$	rel. 7×10^{-5}
$a\sqrt{n}$ only	2.460	same	4%

The full form pins $\pi\sqrt{2/3}$ to five significant figures; the leading term alone is 4% off — the same 2.460 the symbolic-regression front shows at complexity 4. The $-\log n$ term, from the $1/n$ prefactor in the Hardy–Ramanujan formula, is what fixes the constant.

Part A.3 — Rules from labelled data

The pipeline: fit a model, then read it

Supervised learning can surface a hidden rule. The pipeline is the one Davies and collaborators used for knot theory and Kazhdan–Lusztig polynomials (Nature 2021).



- Exact labels from a computer algebra system — no measurement noise.
- Read the model through feature importance or saliency — which inputs move the output.
- The machine stops at the conjecture; a human proves it.

Decision tree: a rule from labelled data

Input	labelled triples — each (a, b, c) tagged appears (1) or absent (0), presented as raw spins or as difference features.
Output	a tree of one-variable threshold tests ("is $a + b - c < 0$?"), a label at each leaf; each root-to-leaf path is an explicit rule.
Minimizes	at each node, the weighted Gini impurity of the two children — $\text{Gini} = 1 - p^2 - (1-p)^2 = 2p(1-p)$ for a node with appears-fraction p, and 0 at a pure leaf. Equivalently, it maximizes the impurity drop.

Algorithm: greedily pick the single feature and threshold that minimizes that weighted child impurity; recurse on each branch until the leaves are pure (or a depth cap). The model is white-box — the rule reads off the tree.

Two other readouts of a fitted model: inductive logic programming returns a logical clause; a black-box classifier is read by saliency, the gradient of the output with respect to each input.

Clebsch–Gordan selection rules

For $SU(2)$, the tensor product of spin-a and spin-b decomposes as

$$V_a \otimes V_b = \bigoplus_{c=|a-b|}^{a+b} V_c,$$

in integer steps; each multiplicity is 0 or 1. It is 1 exactly when

$$|a-b| \le c \le a+b \quad \text{and} \quad a+b+c \in \mathbb{Z}$$

— a triangle inequality and an integrality condition.

As a discovery task, label all triples (a, b, c) with $a, b \le 10$ by an independent computation of the multiplicity — weight counting, not the triangle rule. About $11{,}000$ triples, 30% positive, so "always absent" already scores 0.70.

Raw coordinates fail

A decision tree on raw (a, b, c) scores 0.58 — below the 0.70 baseline.

- The boundary $|a - b| \leq c \leq a + b$ is diagonal in these coordinates, so an axis-aligned tree only staircases it.
- The integrality condition alternates with c; no single coordinate is a threshold for it.

Handing the tree the three difference features $(a{+}b{-}c,\ c{-}|a{-}b|,\ [\,a{+}b{+}c \in \mathbb{Z}\,])$ makes a depth-4 tree exact. But those features *are* the rule — supplying them is doing the discovery by hand, leaving the tree only to place thresholds at 0.

Recovering the rule from a feature library

Instead of supplying the answer, give the tree a generic library and let it choose:

- Library: every signed combination $\varepsilon \cdot (a, b, c)$ with $\varepsilon \in \{-1, 0, 1\}$, each one's integrality flag, and $|a - b|, |a - c|, |b - c|$ — 23 candidates, none marked.
- Search: every feature subset up to size 4, scored by extrapolation (train $a, b \leq 6$, test larger spins).
- Result: no subset of size 1–3 extrapolates exactly; at size 4, one of 8855 does — $\{[a+b+c \in \mathbb{Z}],\ a+b-c,\ a-b+c,\ a-b-c\}$.

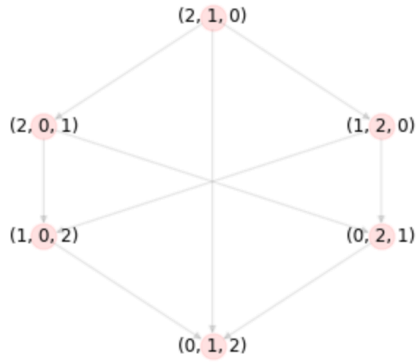
That tree reads back as the rule, with $c \geq a - b$ and $c \geq b - a$ together giving $c \geq |a - b|$:

$$\text{appears} \iff a + b + c \in \mathbb{Z} \ \wedge \ |a - b| \leq c \leq a + b.$$

Trained on $a, b \leq 6$, it matches the independent oracle on all 11,025 triples.

The symmetric group and the Bruhat graph

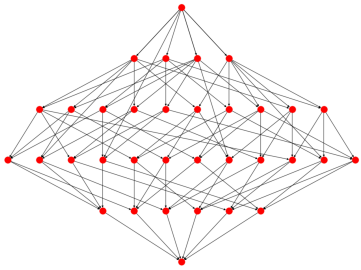
- The symmetric group S_N is all permutations of $\{0, 1, \ldots, N-1\}$, written in string notation: 2031 is the permutation $0 \mapsto 2,\ 1 \mapsto 0,\ 2 \mapsto 3,\ 3 \mapsto 1$.
- Bruhat graph: one node per permutation; an edge joins two permutations that differ by a transposition — a swap of two of the values.
- The length $\ell(x) = \#\{\, i < j : x(i) > x(j) \,\}$ counts inversions. It gives each node a height and orients every edge downward, from longer to shorter.



The ordered Bruhat graph of S_3: top is the reverse permutation 210, bottom is the identity 012.

Bruhat order and the interval $[x, y]$

- Bruhat order: $x \leq y$ when there is a downward path from y to x in the Bruhat graph. The identity is the least element, the reverse permutation w_0 the greatest.
- The interval $[x, y]$ is the set of permutations z with $x \leq z \leq y$, together with the edges among them — a directed graph in its own right.
- Erasing the vertex names leaves the *unlabelled* interval: the shape of that graph alone.



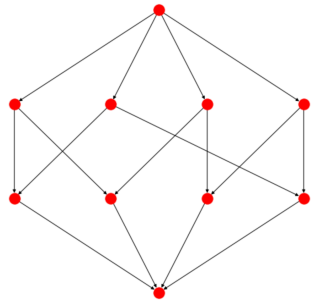
An interval $[03214,\ 34201]$ inside S_5.

Kazhdan–Lusztig polynomials

- Kazhdan and Lusztig (1979) attached to each pair $x \leq y$ a polynomial $P_{x,y}(q)$ with integer coefficients.
- The definition is inductive on length — computing $P_{x,y}$ may call on $P_{u,v}$ for many pairs below. It is awkward by hand, but a computer generates billions.
- Two facts already tie the polynomial to the graph alone:
  - $P_{x,y} \neq 0$ exactly when $x \leq y$;
  - $P_{x,y} = 1$ exactly when the unoriented interval $[x, y]$ is regular (every vertex has the same degree).
- Why they matter: $P_{x,y}$ records the local intersection cohomology of Schubert varieties (Kazhdan–Lusztig 1980) and composition multiplicities of Verma modules in representation theory.
- Small values: $1, \quad 1+q, \quad 1+q^2, \quad 1+3q+q^2$.

The combinatorial invariance conjecture

- Conjecture (Lusztig and Dyer, independently, 1980s): $P_{x,y}$ depends only on the isomorphism type of the unlabelled interval $[x, y]$ — not on which permutations x and y are.
- Example in S_4: the intervals $[0213, 2301]$ and $[1032, 3120]$ are different, yet both are the same directed graph, the "4-crown", and both have $P_{x,y} = 1 + q$.
- Known when x is the identity; open in general.



The "4-crown". It occurs as two different intervals in S_4, both with KL polynomial $1 + q$.

Graph neural networks

- A graph neural network computes on a graph directly. Each node holds a vector of numbers, started from its features — here its in-degree and out-degree.
- **Message passing:** in each round every node replaces its vector by combining its own with an aggregate (sum or mean) of its neighbours' vectors, through a small learned function. Four rounds were used, so each node ends up summarising its four-hop neighbourhood.
- **Readout:** the node vectors are pooled into one graph-level output — here the predicted coefficients of $P_{x,y}$.
- One set of weights is shared across all nodes and all graphs, so a single network handles intervals of any size and depends only on the connectivity, not on the node names.

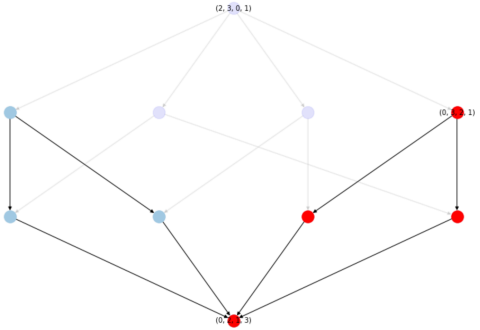
Learning the polynomial from the graph

- **Data:** every interval in the symmetric groups up to S_9 — about 2.9 million in S_9, reduced to $24{,}322$ distinct interval graphs; split $80/20$ into train and test.
- **Input:** the unlabelled interval graph, each node carrying its in-degree and out-degree; edge labels (which transposition) are withheld.
- **Model:** a message-passing graph neural network, four rounds, predicting each coefficient of q, q^2, q^3, q^4 as a separate classification.
- **Result:** the coefficients were predicted with about 98% accuracy (Williamson, 2023) — evidence that $P_{x,y}$ is a function of the graph alone.

Geordie Williamson — representation theorist, University of Sydney; found the hypercube decomposition.

Saliency and the hypercube decomposition

- Gradient saliency ranks each edge by its effect on the prediction. The extremal reflections — those moving the first or last position — stand out far beyond chance.
- Williamson turned that into a decomposition of every interval, along its extremal reflections, into
  - a hypercube, from one set of extremal edges;
  - a subgraph isomorphic to an interval in S_{N-1}.
- From it $P_{x,y}$ is computed directly from the graph — checked on all intervals up to S_9 (over a million); conjectured in general, which would imply combinatorial invariance for S_N (Blundell et al., 2022).



Interval $[(0,2,1,3),(2,3,0,1)]$ in S_4: hypercube piece (blue), interval-in-S_{N-1} piece (red).

Littlewood–Richardson and Kronecker coefficients

Two more multiplicities, both central in algebraic combinatorics:

- Littlewood–Richardson $c^{\lambda}_{\mu\nu}$ — V_λ inside $V_\mu \otimes V_\nu$ for GL_n; the Schur structure constants.
- Kronecker $g(\lambda, \mu, \nu)$ — S^λ inside $S^\mu \otimes S^\nu$ for the symmetric group; no combinatorial rule is known (Murnaghan's problem, open since 1938).

The value is hard to compute: #P-complete for Littlewood–Richardson, #P-hard for Kronecker — #P is the class of counting problems ("how many solutions", e.g. counting the satisfying assignments of a Boolean formula), at least as hard as NP — and for Kronecker even deciding which coefficients are nonzero is NP-hard. But the *support* — which are zero — is a classification problem, and a computer algebra system supplies exact labels in any quantity. Classifiers predict it well and sharpen as n grows (the ML4AlgComb — machine learning for algebraic combinatorics — datasets, Kvinge et al. 2025).

The same pipeline in the Langlands program

Lanard and Mínguez (2024) used this pipeline to find an algorithm for Aubert–Zelevinsky duality — an involution on the irreducible representations of a p-adic group — on SO_{2n+1} and Sp_{2n}, extending the Mœglin–Waldspurger algorithm, the known combinatorial rule for that duality on GL_n.

- Predicting the whole answer failed, so they predicted one piece of the combinatorial data at a time — the smallest of the "segments" indexing the representation — and read the formula off that.
- They calibrated the saliency on GL_n, where the algorithm is known, before trusting it on the classical groups, where it was not.

Part A.4 — Conjecture generators

Graffiti: conjecturing inequalities between invariants

Graffiti (Fajtlowicz, mid-1980s) proposes inequalities among graph invariants.

- Keep a database of graphs; compute ~ 60 invariants (independence number, domination number, average distance, matching number, eigenvalues).
- Propose an inequality whenever it holds for every graph in the database.
- The Dalmatian heuristic keeps a new conjecture only if it is sharper on some graph than all kept so far.

Circulated as "Written on the Wall"; about eighty papers followed, by Chung, Erdős, Lovász, Seymour and others.

The Ramanujan Machine: continued fractions for constants

Raayoni et al. (*Nature* 590, 2021) generate conjectured *polynomial continued fractions* (PCFs) for fundamental constants — π, e, Catalan's constant G, and zeta values $\zeta(2) = \pi^2/6$, $\zeta(3)$:

$$x = a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \ddots}}}, \qquad a_n, b_n \in \mathbb{Z}[n]$$

— the partial numerators b_n and denominators a_n are polynomials in n with integer coefficients.

- **Input:** a target constant, and bounded families of integer polynomials a_n, b_n.
- **Output:** an identity "constant = PCF", matched numerically — a conjecture, with no proof attached.

The Ramanujan Machine: how it searches

Both methods match numerical values only — no proof, no assumed structure.

MITM-RF (Meet-in-the-Middle):

- Input: the target constant; a library of candidate PCFs.
- Evaluate both sides to low precision; hash one side, look up the other.
- Re-check every hit at high precision to discard false matches.

Descent&Repel (gradient descent): minimise the gap between a PCF's value and the target over the integer coefficients; a "repel" step pushes away from solutions already found, to surface new ones.

Matches are tested to up to 2000 digits: a chance match in a 10^9 search at 50-digit accuracy has probability below 10^{-40}.

The Ramanujan Machine: $\zeta(3)$ and Catalan's constant

Extending the search produced previously-unknown continued fractions for $\zeta(3)$ (Apéry's constant) and Catalan's constant $G = 0.91596\ldots$:

$$\frac{12}{7\zeta(3)} = 2 - \cfrac{16 \cdot 1^6}{36 - \cfrac{16 \cdot 2^6}{160 - \cfrac{16 \cdot 3^6}{434 - \ddots}}}$$

$$\frac{2}{2G - 1} = 3 - \cfrac{6 \cdot 1^3}{13 - \cfrac{8 \cdot 2^3}{29 - \cfrac{10 \cdot 3^3}{51 - \ddots}}}$$

Both were matched from the numerical value alone, with no assumed structure, and were unproven when published; the Catalan-constant conjectures are still open.

The Ramanujan Machine: example formulas and status

Two formulas the machine found (both later proved):

$$\frac{e}{e-2} = 4 - \cfrac{1}{5 - \cfrac{2}{6 - \cfrac{3}{7 - \ddots}}}, \qquad \frac{4}{3\pi - 8} = 3 - \cfrac{1 \cdot 1}{6 - \cfrac{2 \cdot 3}{9 - \cfrac{3 \cdot 5}{12 - \ddots}}}.$$

- The program finds the identity; it cannot prove it.
- Yamamoto (2024) proved 38 of the conjectures: each PCF reduces to a second-order linear difference equation, solved via a differential equation or Petkovšek's algorithm; 31 were generalised to wider families.
- Many others, including Catalan's-constant conjectures, remain open.

Part A.5 — Murmurations of elliptic curves

Elliptic curves over $\mathbb{Q}$

- An **elliptic curve** over the rationals is the solution set of

$$E:\ y^2 = x^3 + ax + b, \qquad a, b \in \mathbb{Q},$$

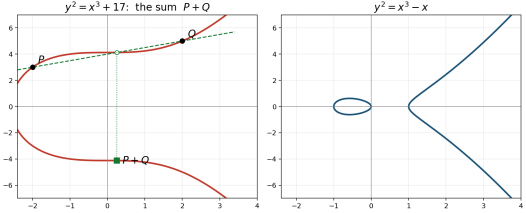
  together with one point O "at infinity".
- **Smoothness:** require the discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$, so the curve has no cusp or self-crossing.
- $E(\mathbb{Q})$ is the set of rational points (x, y) on E, plus O. The basic questions: how many are there, and how are they related?
- Over $\mathbb{R}$ the curve is one branch, or one branch plus a closed oval — $y^2 = x^3 - x$ has both.

The group law: chord and tangent

$E(\mathbb{Q})$ is an abelian group. A line meets the cubic in three points (with multiplicity); the operation $P + Q$ is built from that:

- **Sum.** The line through P, Q meets E in a third point R; reflect R across the x-axis to get $P + Q$.
- **Double.** For $2P$, use the *tangent* at P in place of the chord, then reflect.
- **Identity.** O, the point at infinity where all vertical lines meet, is neutral: $P + O = P$.
- **Inverse.** $-P$ is the reflection of P; the line through them is vertical, so its third point is O and $P + (-P) = O$.

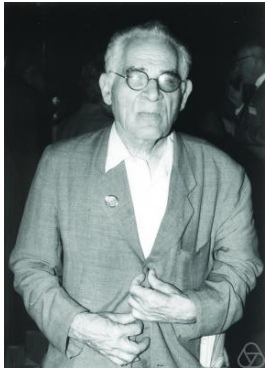
Commutative and associative; $nP = P + \cdots + P$ builds infinitely many points from a few.



Left: the sum $P + Q$ on $y^2 = x^3 + 17$ via the chord and the reflection. Right: the two-component locus $y^2 = x^3 - x$.

Mordell's theorem and the rank

- **Mordell (1922):** $E(\mathbb{Q})$ is a finitely generated abelian group,
$$E(\mathbb{Q}) \;\cong\; \mathbb{Z}^r \;\oplus\; E(\mathbb{Q})_{\mathrm{tors}},$$
with $E(\mathbb{Q})_{\mathrm{tors}}$ finite.
- The **rank** $r \geq 0$ is the number of independent infinite-order points that generate, by the group law, all but finitely many rational points.
- Rank 0: finitely many rational points; rank ≥ 1: infinitely many.
- Computing the rank exactly is hard — no algorithm is proven to terminate for every curve.
- But it can be *predicted* from the a_p with high accuracy — statistics, not proof.



Louis Mordell (1888–1972): proved $E(\mathbb{Q})$ finitely generated, 1922. Sadleirian Chair, Cambridge.

From $E(\mathbb{Q})$ to $E(\mathbb{F}_p)$: why count mod p

- **The rank is hard to reach directly.** It lives over $\mathbb{Q}$, the rational points can be infinite, and no algorithm is proven to compute it for every curve.
- **Reduce mod p.** For a good prime $p \nmid \Delta$, reduce the coefficients a, b mod p: the result is an elliptic curve over the finite field $\mathbb{F}_p = \{0, \ldots, p-1\}$. Now x, y each take only p values, so the points, plus O, form a *finite* group $E(\mathbb{F}_p)$ —a directly computable count.
- **The link.** Reducing coordinates is a group homomorphism $E(\mathbb{Q}) \to E(\mathbb{F}_p)$ (good p): each rational point maps to a mod-p point. So $E(\mathbb{Q})$ is reflected in the finite groups, and the rank shows up as a *bias* in the counts N_p —more rational points, more points mod p on average. The Birch–Swinnerton-Dyer conjecture makes this link exact.

a_p and the L-function

- For each good prime, the point count gives

$$N_p = \#E(\mathbb{F}_p), \qquad a_p = p + 1 - N_p,$$

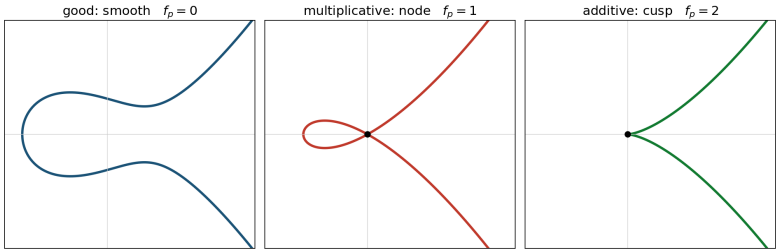
  the **Frobenius trace** — how far N_p strays from the expected $p+1$. Hasse: $|a_p| \le 2\sqrt{p}$.
- The (a_p) assemble into the L-**function**

$$L(E,s) = \prod_{p \text{ good}} \left(1 - a_p p^{-s} + p^{1-2s}\right)^{-1} = \sum_{n \ge 1} a_n n^{-s}, \quad \mathrm{Re}(s) > \tfrac{3}{2}.$$

- By the modularity theorem (Wiles; Taylor–Wiles; Breuil–Conrad–Diamond–Taylor) $L(E,s)$ extends to all of $\mathbb{C}$. Its center $s = 1$ is where the arithmetic concentrates.

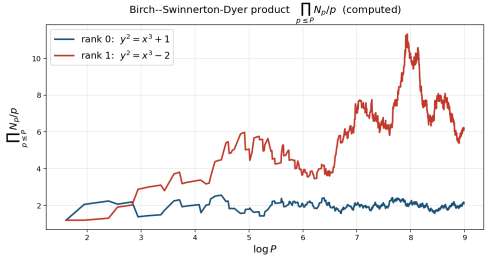
Reduction types and the conductor

- At a bad prime $p \mid \Delta$ the reduced curve is singular: a **node** (two tangent lines) is *multiplicative* reduction, a **cusp** (one) is *additive*; good reduction is smooth.
- The **conductor** is $N(E) = \prod_p p^{f_p}$ with $f_p = 0$ (good), 1 (mult.), 2 (additive, $p \geq 5$).
- At $p = 2, 3$ a wild term can raise it ($f_2 \leq 8$, $f_3 \leq 5$); in general $f_p = v_p(\Delta_{\min}) + 1 - m_p$ ($m_p = \#$components of the special fibre; Ogg, via Tate's algorithm).
- N square-free $\iff$ E semistable. The conductor orders curves by size and sets the murmuration bands.



The Birch–Swinnerton-Dyer conjecture, found by computer

- Birch and Swinnerton-Dyer, on the **EDSAC-2** at Cambridge (1958–62), computed N_p and studied $\displaystyle\prod_{p\le P}\frac{N_p}{p}$, finding it grows like $C(\log P)^{\,r}$ — higher rank, more points mod p.
- **Conjecture.** $\operatorname{rank} E(\mathbb{Q}) = \operatorname{ord}_{s=1} L(E,s)$: algebraic rank = analytic rank.
- A Clay Millennium Prize Problem; proved only for analytic rank ≤ 1 (Gross–Zagier, Kolyvagin), open in general.
- So BSD itself was **found by computer** — the conjecture read off the data, not derived from theory.



$\prod_{p\le P} N_p/p$ vs $\log P$ (computed): rank-1 grows, rank-0 stays bounded.

Birch–Swinnerton-Dyer: the precise statement

For $E/\mathbb{Q}$ with $r = \operatorname{ord}_{s=1} L(E,s)$, the conjecture has two parts:

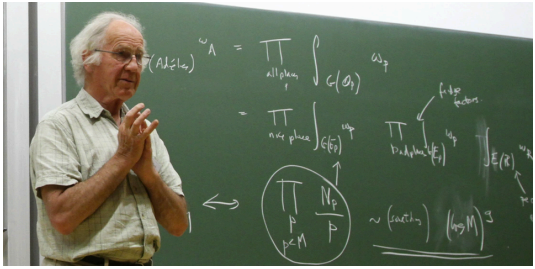
- **Rank:** $r = \operatorname{rank} E(\mathbb{Q})$.
- **Leading coefficient:** $\displaystyle\lim_{s\to 1} \frac{L(E,s)}{(s-1)^r} = \frac{\Omega_E \cdot \operatorname{Reg}(E) \cdot \#Ш(E) \cdot \prod_p c_p}{\left(\#E(\mathbb{Q})_{\mathrm{tors}}\right)^2}.$

Invariants (minimal model):

- Ω_E — real period $\int_{E(\mathbb{R})} |\omega|$ (ω the Néron differential).
- $\operatorname{Reg}(E)$ — regulator: $\det$ of the Néron–Tate height pairing on a basis of $E(\mathbb{Q})/\mathrm{tors}$ ($= 1$ if $r = 0$).
- $\#Ш(E)$ — order of the Tate–Shafarevich group (failure of the local–global principle; conjectured finite).
- c_p — Tamagawa number $[E(\mathbb{Q}_p) : E^0(\mathbb{Q}_p)]$ ($= 1$ at good p); $\prod_p c_p$ finite.
- $\#E(\mathbb{Q})_{\mathrm{tors}}$ — order of the torsion subgroup.

Birch and Swinnerton-Dyer

Working from their EDSAC-2 point counts at Cambridge in the early 1960s, Birch and Swinnerton-Dyer read off the rank–L-function correlation that became the conjecture.



Bryan Birch (b. 1931), British; University of Oxford.
Also known for Birch's theorem.

Sir Peter Swinnerton-Dyer (1927–2018), British; University of Cambridge; 16th Baronet.

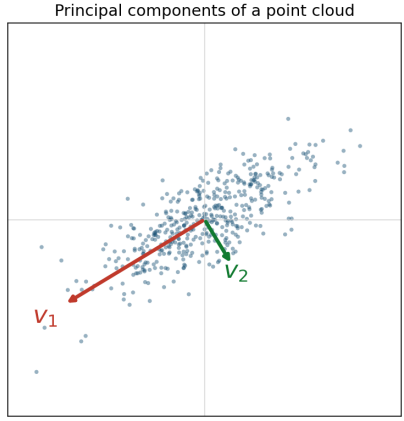
The machine: EDSAC 2

- A vacuum-tube (valve) computer at the Cambridge Mathematical Laboratory, running from **1958** — successor to EDSAC (1949).
- Built under Maurice Wilkes with David Wheeler: the **first computer with a microprogrammed control unit**, and the first bit-slice architecture.
- Birch and Swinnerton-Dyer ran their elliptic-curve point counts on it; the conjecture came out of those runs.
- Its small store and slow speed limited the runs to small-conductor curves.

EDSAC 2 in 1960, Cambridge Mathematical Laboratory. Wikimedia Commons.

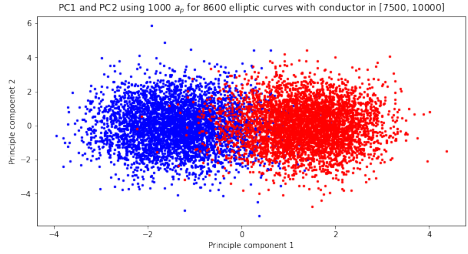
Principal component analysis (PCA)

- An **unsupervised** linear dimensionality-reduction method: orthogonal directions (principal components) ordered by the variance they capture.
- Centre the points; the eigenvectors $v_1, v_2, \dots$ of the covariance $C = \frac{1}{n} X^\top X$ (eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots$) are the components — v_1 the most-variance direction, $v_2 \perp v_1$ next, λ_i the variance along v_i (the SVD of X).
- Project onto the top two, $x \mapsto (v_1 \cdot x,\ v_2 \cdot x)$, to plot high-dimensional data in 2-D.
- Unsupervised: uses the data, not the labels, so any clusters are intrinsic.



A machine-learning rank classifier

- The **LMFDB** (L-functions and Modular Forms Database) lists elliptic curves with rank, conductor, and (a_p) — labelled data.
- He, Lee, and Oliver, with Lee's undergraduate Pozdnyakov (2020–22), trained classifiers to predict rank from $(a_{p_1}, \ldots, a_{p_{1000}})$.
- Ranks separate cleanly: logistic regression reaches precision 0.96 on $\{0, 1\}$ and 0.9998 on $\{1, 2\}$; principal component analysis (PCA), a top-variance projection, separates them visually.
- So the a_p carry the rank. *What* in them? Pozdnyakov averaged them and plotted.



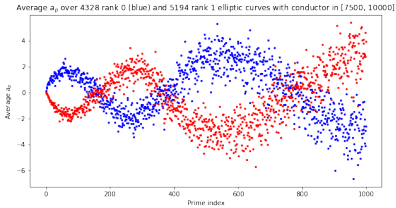
PCA of the a_p vectors, conductor $[7500, 10000]$; ranks separate. He–Lee–Oliver–Pozdnyakov, arXiv:2204.10140.

The murmuration: the averaged statement

- The **conductor** $N(E)$ is built from the bad primes (dividing Δ); it orders curves by size.
- In a band $[N_1, N_2]$, let $\mathcal{E}_r$ be the rank-r curves. Average the n-th Frobenius trace:

$$f_r(n) = \frac{1}{\#\mathcal{E}_r} \sum_{E \in \mathcal{E}_r} a_{p_n}(E).$$

- **Finding (2022):** f_0, f_1 trace two mirrored oscillating curves, amplitude and period growing with n; approximately scale-invariant across bands.
- Named *murmurations* for the look of starling flocks. Empirical: no formula, no proof.



f_0, f_1, conductor $[7500, 10000]$ (arXiv:2204.10140).

A starling murmuration.

The murmuration discoverers

- **Yang-Hui He** (London Institute for Mathematical Sciences) — brought machine learning to these number-theory datasets.
- **Kyu-Hwan Lee** (University of Connecticut) and **Thomas Oliver** (University of Westminster) — set up the rank-classification experiments on L-function data.
- **Alexey Pozdnyakov** (then Lee's undergraduate student) — averaged the a_p by rank and plotted them: the murmuration.

Published in *Experimental Mathematics* 34 (2025) (arXiv:2204.10140).

Yang-Hui He, London Institute for Mathematical Sciences. Wikimedia Commons.

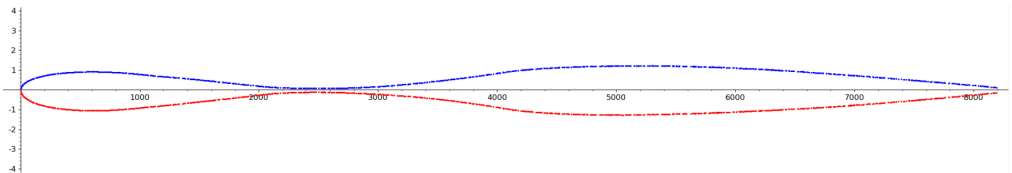
From elliptic curves to modular forms

- Sutherland (MIT) reproduced murmurations on $> 10^9$ curves and in far more general L-functions — the phenomenon is not special to elliptic curves.
- The clean setting is **modular forms**: a holomorphic function on the upper half-plane transforming under $\Gamma_0(N)$ (level N, weight k); a *newform* is a Hecke eigenform genuinely of level N, with Fourier coefficients $a_f(p)$.
- By modularity an elliptic curve *is* a weight-2 newform with the same a_p — so the elliptic-curve murmuration is the weight-2 case.
- Each f has a **root number** $\varepsilon(f) = \pm 1$ (sign of its functional equation), the analogue of rank parity — group forms by $\varepsilon(f)$. Normalized coefficient $\lambda_f(p) = a_f(p)/p^{(k-1)/2}$, $|\lambda_f(p)| \leq 2$ (the analogue of $a_p/\sqrt{p}$).

Sutherland's limit curve

The elliptic-curve murmuration averaged a_p over rank classes and plotted it against the prime index — and was scale-invariant. Sutherland makes that exact:

- Fix a band of levels $N \in [X, cX]$ and one prime P that scales with the level, $P \sim N$.
- **Root-number-weighted average:** multiply each $a_f(P)$ by its sign $\varepsilon(f)$, then average over all weight-k newforms in the band — the signs make the bias add up instead of cancel.
- As $N \to \infty$ this converges to one **continuous curve** depending only on $y = P/N$ (prime size ÷ level size).



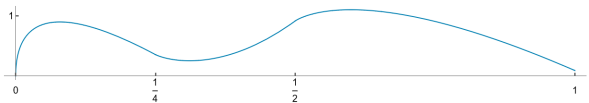
Average of $\sqrt{P}\,\lambda_f(P)\,\varepsilon(f)$ over levels $[2^{13}, 2^{14}]$ vs $y = P/N$. Courtesy of Sutherland, in arXiv:2310.07681.

Zubrilina's theorem (2023): the murmuration density

Average $\sqrt{P}\,\lambda_f(P)\,\varepsilon(f)$ over all weight-k newforms f of square-free level $N \in [X, X+Y]$, with $Y = (1+o(1))X^{1-\delta}$ for a suitable $\delta > 0$; $\lambda_f(P) = a_f(P)/P^{(k-1)/2}$ is the normalized coefficient, $\varepsilon(f) = \pm 1$ the root number, and $y = P/X$. As $X \to \infty$:

$$\frac{\sum_N \sum_f \sqrt{P}\,\lambda_f(P)\,\varepsilon(f)}{\sum_N \sum_f 1} \;=\; \mathcal{M}_k(y) \;+\; (\text{power-saving error}).$$

- $\mathcal{M}_k$ is explicit and continuous: a finite sum of Chebyshev polynomials U_{k-2} with Euler-product coefficients, plus a $\sqrt{y}$ term and (for $k = 2$) a y term.
- Proof: the Eichler–Selberg trace formula turns the average into an average of class numbers. The elliptic-curve murmuration is the case $k = 2$ (arXiv:2310.07681).



The murmuration is now a family of theorems

Zubrilina's weight-k density was the first proof. The phenomenon has since been established for further families:

- **Dirichlet characters** — Lee, Oliver, Pozdnyakov (arXiv:2307.00256).
- **Weight aspect** — Bober, Booker, Lee, Lowry-Duda; level fixed, weight $\to \infty$ (arXiv:2310.07746).
- **Maass forms** — the non-holomorphic case (arXiv:2409.00765).
- **Ordered by height** — the same curves ordered by naive height (arXiv:2504.12295).
- **Function fields** — the function-field analogue (arXiv:2603.13802).

Computing the rank vs predicting it

- **Certified** rank: descent (e.g. `mwrank`, Sage) — correct when it finishes, but it can stall when the Tate–Shafarevich group is large; no algorithm is proven to terminate for every curve.
- **Predicting** the rank from the a_p is easier — and the a_p determine it, through the explicit formula

$$-\frac{L_E'(s)}{L_E(s)} = \sum_{n\geq 1} \frac{c_n \Lambda(n)}{n^s} = -\frac{r}{s-1} + \cdots,$$

  where Λ is the von Mangoldt function, $c_p = a_p$, and $r = \operatorname{ord}_{s=1} L(E,s)$ is the analytic rank; by BSD it equals the algebraic rank. The minus sign is the bias: higher rank pushes the a_p sums more negative.
- So a weighted prime sum of the a_p estimates r — prediction, not proof.

Mestre–Nagao sums

Explicit prime sums that estimate the analytic rank (Mestre 1982, Nagao 1992; the list S_0–S_6 collected in arXiv:2207.06699). The basic one:

$$S_0(B) = \frac{1}{\log B} \sum_{\substack{p<B \\ \text{good}}} \frac{a_p(E)\,\log p}{p}.$$

- **Kim–Murty:** under the Riemann Hypothesis for L_E, if the limit exists, $\lim_{B\to\infty} S_0(B) = -r + \frac{1}{2}$.
- Variants: Nagao's S_3 (logarithmic derivative of the partial Euler product at $s = 1$, large for large rank); Bober's S_6 (from the explicit formula, $\to r$ under RH, but infeasible beyond a small range).
- A sum returns a *number*; reading a rank off it needs a decision rule and a conductor-dependent cutoff.

Learning the rank: trees and a CNN

- **Gradient-boosted trees** (Alessandretti–Baronchelli–He, 2019, arXiv:1911.02008): 2.5M Cremona curves; an ensemble of decision trees finds correlations among rank, Weierstrass coefficients, conductor, regulator, Tamagawa number, and the order of the Tate–Shafarevich group.
- **A 1-D CNN** (Kazalicki–Vlah, 2022, arXiv:2207.06699) learns the rank from a curve's a_p sequence.
- On the LMFDB it beats every Mestre–Nagao sum: Matthews correlation coefficient (a balanced score, max 1) 0.9958 — only 0.25% misclassified — vs 0.87 for the best single sum. A net fed all sums at once beats any one sum.
- On a harder set (conductor up to 10^{30}, rank to 10) accuracy drops; very large conductors need more a_p.

The CNN rank classifier: input, output, product

- **Input.** One curve becomes a $3 \times \pi(N)$ matrix; each column is a prime $p < N$ ($N = 10^3, 10^4, 10^5$):
  - row 1: $a_p/\sqrt{p} \in [-2, 2]$ (Hasse) — this row carries the rank;
  - row 2: the normalized conductor $\log N_E / \log N_{\max}$;
  - row 3: the prime's position in the sequence.
- **Output.** A probability for each rank class $0, 1, 2, \ldots$; the prediction is the largest. A classification, not a single number.
- **Training.** Supervised on LMFDB curves whose rank is already known (certified by descent); minimise the cross-entropy loss by back-propagation. The 1-D convolution slides one learned filter along the prime axis.
- **Product.** A fixed-weight function f_θ: (a_p matrix) $\mapsto$ rank — a predictor with no certificate. It estimates the analytic rank (= algebraic under BSD); certifying the rank still needs descent.

The 1-D convolution: a sliding weighted sum

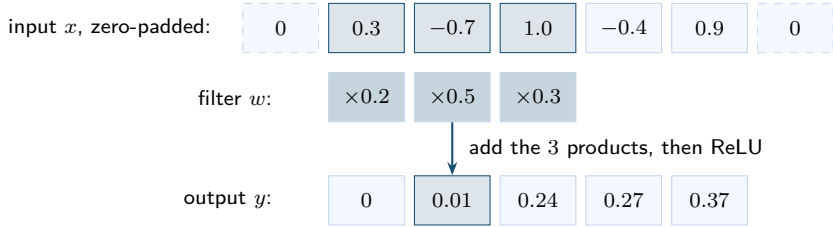
First layer, kernel size 17. Output channel $j \in \{1, \ldots, 64\}$ at prime position i:

$$y_j[i] \;=\; \mathrm{ReLU}\Big(b_j + \sum_{c=1}^{3} \sum_{k=-8}^{8} W_j[c,k]\; x_c[i+k] \Big), \qquad \mathrm{ReLU}(t) = \max(t, 0).$$

- **Window:** the 17 columns centered at position i, all 3 channels — a 3×17 block of the input.
- **Filter:** W_j, a 3×17 array of weights, plus a bias b_j — 52 learned numbers.
- Multiply window and filter entry by entry, add the 51 products, add b_j, clip negatives to 0: one number — entry i of output row j.
- The same (W_j, b_j) is used at every $i = 1, \ldots, 1229$ — the filter slides. 64 filters give the 64 output rows.
- At the two ends the window sticks out; missing entries are set to 0 (zero padding), so the length is kept.

One convolution by hand: kernel size 3, one channel

Filter $w = (0.2,\ 0.5,\ 0.3)$, bias $b = 0$: $\quad y[i] = \mathrm{ReLU}\big(0.2\,x[i{-}1] + 0.5\,x[i] + 0.3\,x[i{+}1]\big)$.



- $y[2] = \mathrm{ReLU}\big(0.2(0.3) + 0.5(-0.7) + 0.3(1.0)\big) = \mathrm{ReLU}(0.01) = 0.01.$
- Slide one step and repeat: $y[3] = 0.24$, $y[4] = 0.27$, $y[5] = 0.37$.
- Left edge: $y[1] = \mathrm{ReLU}\big(0.2(0) + 0.5(0.3) + 0.3(-0.7)\big) = \mathrm{ReLU}(-0.06) = 0.$
- The same three weights at every position; a fully-connected layer would have separate weights for each position.

Stride 2, the later layers, and the parameter count

- **Stride 2** (the reducing layers): the same formula, computed only at every other position $i = 1, 3, 5, \ldots$ — the length halves.
- **From the second layer on:** the input has 64 channels, so each filter is a 64×17 block — one output entry is a sum of $64 \cdot 17 = 1088$ products plus a bias. Still 64 filters per layer. (Once the sequence is shorter than 17, the kernel is capped at the sequence length.)
- **Parameter count:** first layer $64 \cdot 3 \cdot 17 + 64 = 3{,}328$; a full-length later layer $64 \cdot 64 \cdot 17 + 64 = 69{,}696$; with the shrinking kernels near the end, the whole model holds $596{,}293 \approx 6 \times 10^5$ learned numbers, all set by gradient descent on the cross-entropy loss.
- Without its ReLU, a convolution layer is a linear map that is translation-equivariant: shift the input along the prime axis and the output shifts the same way.
- After the 11 halvings, each of the final 64 numbers depends on all 1229 primes — 64 learned whole-sequence statistics; the dense layer is a linear classifier on them. A Mestre–Nagao sum is one hand-designed statistic of the same kind.

The CNN design (Kazalicki–Vlah): four layer types

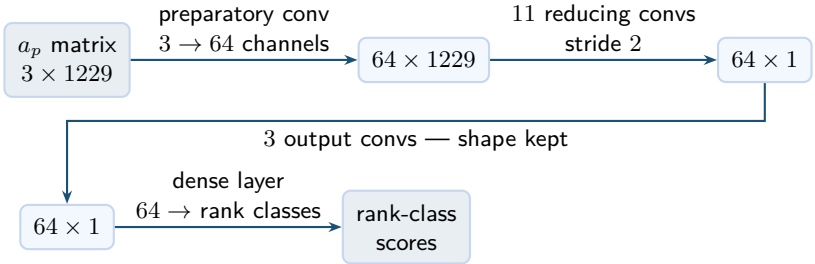
Design rationale: a Mestre–Nagao sum applies the same rule to every a_p — a shared weighting — and a 1-D convolution is exactly such a shared filter slid along the primes. Write $C \times L$ for C channels of length L. The network stacks 1-D convolutions in four groups:

- **Preparatory** (1 layer): $3 \times L \to 64 \times L$ — widens the 3 input rows to 64 channels; the length is unchanged.
- **Input** (L_1 layers): $64 \times L \to 64 \times L$ — same shape; extra processing steps before any shortening.
- **Reducing** (L_2 layers): stride 2 — each halves the length; L_2 is set so the length reaches exactly 1.
- **Output** (L_3 layers): $64 \times 1 \to 64 \times 1$ — same shape; extra processing steps after the shortening.
- **Final dense layer:** the last 64 numbers $\to$ one score per rank class; the largest score is the predicted rank.

Every convolution is followed by ReLU and batch normalization; one kernel size KS throughout, capped at the sequence length once the sequence is shorter than the kernel. The whole net is fixed by (L_1, L_2, L_3, KS), chosen by Bayesian optimization over 500 trained variants.

The LMFDB $p < 10^4$ model $(0, 11, 3, 17)$, layer by layer

$\pi(10^4) = 1229$ primes below 10^4, so one curve enters as a 3×1229 matrix; $L_1 = 0$, so this model has no input layers.



- $2^{10} < 1229 \le 2^{11}$: eleven halvings take the length $1229 \to \cdots \to 1$ (the code sets $L_2 = \lceil \log_2 \pi(N) \rceil$).
- Each convolution: `Conv1d` with zero padding $\lfloor \text{kernel}/2 \rfloor$, then ReLU, then batch norm.
- Training: class-weighted cross-entropy, Adam, one-cycle schedule, 40 epochs, batch size 2048.

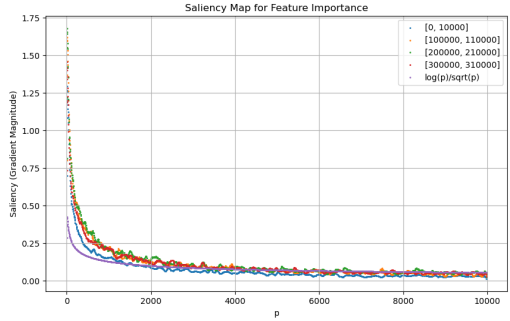
The $p < 10^4$ network, every layer (from the published code)

layer	operation	kernel	stride	output shape
input	$a_p/\sqrt{p}$, conductor, position	—	—	3×1229
preparatory	conv $3 \to 64$	17	1	64×1229
R_1	conv $64 \to 64$	17	2	64×615
R_2	conv $64 \to 64$	17	2	64×308
R_3	conv $64 \to 64$	17	2	64×154
R_4	conv $64 \to 64$	17	2	64×77
R_5	conv $64 \to 64$	17	2	64×39
R_6	conv $64 \to 64$	17	2	64×20
R_7	conv $64 \to 64$	17	2	64×10
R_8	conv $64 \to 64$	11	2	64×5
R_9	conv $64 \to 64$	5	2	64×3
R_{10}	conv $64 \to 64$	3	2	64×2
R_{11}	conv $64 \to 64$	3	2	64×1
O_1–O_3	conv $64 \to 64$ ($\times 3$)	1	1	64×1
head	flatten + linear $64 \to 5$	—	—	5 scores (ranks 0–4)

Every conv row = `Conv1d` $\to$ ReLU $\to$ batch norm. The kernel is capped at the sequence length, so it shrinks near the end; the O layers are 64×64 channel-mixing maps. Parameters: $594{,}048 + 1{,}920 + 325 = 596{,}293$.

The network's saliency: murmuration early, Mestre–Nagao late

- Saliency w_p — the gradient of the output with respect to each a_p — shows what the CNN relies on.
- **Early** in training the rank-0-vs-rank-1 saliency shows the *murmuration* oscillation.
- **Later** the rank-0 saliency flattens and weight moves to small primes, matching the Mestre–Nagao weighting $\log p/\sqrt{p}$.
- Murmurations matter more for small conductors. Test accuracy 99.85% on $[0, 10^4]$, 98.57% on $[3\times10^5, +10^4]$.



Trained CNN saliency across four conductor ranges vs the Mestre–Nagao weighting $\log p/\sqrt{p}$ (purple).
arXiv:2603.17681.

Part B.0 — A construction problem: 26 circles in a unit square

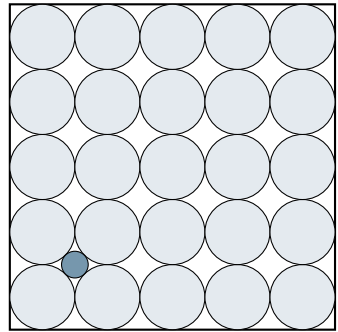
26 circles in a unit square

Problem. Place 26 circles inside the unit square:

- no two circles overlap (touching is allowed);
- no circle crosses the boundary;
- the radii need not be equal.

Maximize the sum of the 26 radii.

Best published placement (2025): radius sum 2.635, found by a computer search.



A placement made by hand — radius sum 2.541.

The by-hand placement: radius sum 2.541

- 25 circles of radius 0.1, centers at $(0.1 + 0.2i,\ 0.1 + 0.2j)$ for $i, j \in \{0, \ldots, 4\}$:
  - neighbouring centers are 0.2 apart $= 0.1 + 0.1$ — the circles touch but do not overlap;
  - the outer circles touch the sides of the square.

  Radius sum so far: $25 \times 0.1 = 2.5$.
- One more circle in a gap, centered at $(0.2,\ 0.2)$:
  - the four nearest centers are at distance $\sqrt{0.1^2 + 0.1^2} \approx 0.1414$;
  - $0.1 + 0.041 = 0.141 < 0.1414$ — a circle of radius 0.041 fits.
- Total radius sum: $2.5 + 0.041 = 2.541$.

Distance to the record: $2.635 - 2.541 = 0.094$.

Checking a placement is exact arithmetic

A candidate is a list of 26 triples (x_i, y_i, r_i) — 78 numbers. Checking it is a finite list of inequalities:

- inside the square: $r_i \le x_i \le 1 - r_i$ and $r_i \le y_i \le 1 - r_i$ for each i;
- no overlap: $(x_i - x_j)^2 + (y_i - y_j)^2 \ge (r_i + r_j)^2$ for each of the $\binom{26}{2} = 325$ pairs.

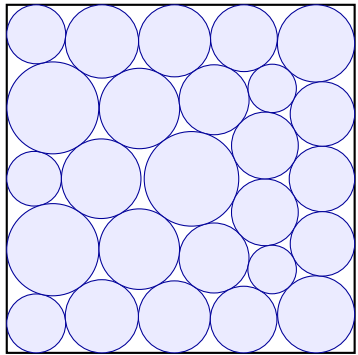
The check does not depend on who or what proposed the placement.

The two lower bounds: 2.541 and 2.635

- The by-hand placement proves: the maximum radius sum is at least 2.541.
- The 2025 placement (right) proves: it is at least 2.635.
- Whether a placement with a larger sum exists: open.

A record proves a bound; optimality stays open.

The placement drawn here is the published 2025 record itself (26 circles, radius sum 2.6358); checking it — every circle inside, every pair disjoint — is the finite arithmetic described on the previous frames, and was re-done independently for this lecture.



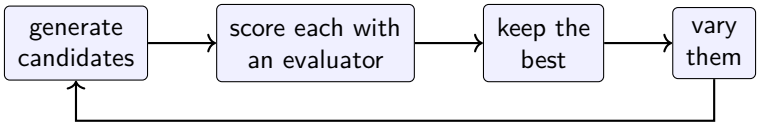
Construction problems

The circle problem, in general terms — a **construction problem** has three ingredients:

- a **search space**: a finite (or finitely described) set of candidate objects — here, placements of 26 circles;
- a **score function**: a computable number attached to each candidate — here, the sum of the radii;
- a **known record**: the best score any published construction achieves — here, 2.635.

The goal: an explicit candidate that scores past the record. Nothing needs to be proved about the optimum. In mathematical terms this is **extremal combinatorics**: maximize (or minimize) a score over a combinatorial class — here, sum of radii over packings.

Evolutionary search: generate → score → select → vary



- An **evolutionary search** keeps a pool of candidates and repeats the four steps.
- The **evaluator** is the program computing the score function — for the circles: check the inequalities, return the radius sum.
- The loop calls the evaluator thousands to millions of times per run, so scoring must be automatic and cheap — and a wrong evaluator makes the search optimize the wrong thing.

Part B.1 — First systems: Wagner, AlphaTensor, AlphaDev (2021–2023)

Conjectured inequalities in spectral graph theory

A. Z. Wagner, *Constructions in combinatorics via neural networks*, arXiv:2104.14516 (April 2021).

- Spectral graph theory studies a graph through eigenvalues of matrices attached to it — first of all the **adjacency matrix**: the $n \times n$ matrix with a 1 in entry (i, j) when vertices i and j are joined by an edge, 0 otherwise.
- Conjectures are produced here in bulk by computer search: **AutoGraphiX** (Caporossi–Hansen) hunts for inequalities among graph invariants that hold on every example it can generate; the Aouchiche–Hansen survey (*Linear Algebra Appl.* 432, 2010) collects conjectures produced this way.
- A conjectured inequality is a universal claim; one explicit violating graph settles it.

Adam Zsolt Wagner — research scientist, Google DeepMind; previously assistant professor of mathematics, Worcester Polytechnic Institute. Photo: personal homepage.

The target conjecture: $\lambda_1 + \mu \geq \sqrt{n-1} + 1$

Conjecture 2.1 of the survey (originally produced by AutoGraphiX): for every connected graph on $n \geq 3$ vertices,

$$\lambda_1 + \mu \ \geq \ \sqrt{n-1} + 1,$$

where

- λ_1 is the largest eigenvalue of the adjacency matrix;
- μ is the **matching number**: the maximum number of pairwise disjoint edges.

As a construction problem:

- search space: graphs on n vertices;
- score of a graph: $\sqrt{n-1} + 1 - \lambda_1 - \mu$ — the amount of violation;
- any graph with positive score is a counterexample.

Encoding and objective

Input a conjectured inequality, and an encoding of the candidate as a word — a graph on n vertices is a 0–1 string of length $\binom{n}{2}$, one bit per potential edge.

Output an explicit graph violating the inequality, if the search finds one.

Objective maximize the violation score, treated as a black box.

The graph is built bit by bit — edge slot by edge slot, each bit drawn from a probability that depends on the edges already placed.

What the network represents: a distribution over graphs

There are $2^{\binom{n}{2}}$ graphs on n vertices — too many to score one by one. The search needs to spend its trials where high-scoring graphs are likely, and to move that focus as it learns.

- The device for this is a **probability distribution over graphs we can sample from and adjust**. The network *is* that distribution.
- A graph is built one edge-bit at a time, so the distribution is fixed by one number per bit: the probability this edge is 1 *given the edges already decided*. That single conditional probability is the network's output.
- The network answers one question, once per edge slot: given the partial graph, how likely should this next edge be? Nothing about the conjecture is built in — it starts uniform (≈ 0.5) and only the score reshapes it.

What the output p is used for: it is the bias of a coin flip for this one edge; $\binom{n}{2}$ flips produce one graph; 1000 graphs make a batch to score. The *graphs* are the deliverable — the network is only the mechanism pulled toward where the best graphs sit, then discarded.

The proposal network: a multilayer perceptron

Fix an order on the $\binom{n}{2}$ edge slots and decide them one bit at a time. At step $t \in \{0, 1, \ldots, \binom{n}{2} - 1\}$ the network sees two vectors over the same $\binom{n}{2}$ slots.

- A — **the partial graph.** $A \in \{0,1\}^{\binom{n}{2}}$. For $i < t$, $A[i]$ is the bit already chosen for edge slot i. For $i \geq t$, $A[i] = 0$ (placeholder; slot not yet decided).
- B — **the position pointer.** $B \in \{0,1\}^{\binom{n}{2}}$ with $\sum_i B[i] = 1$ (one-hot). The unique 1 sits at position t: "decide edge slot t now". Linking constraint: $A[i] = 0$ whenever $B[j] = 1$ with $j \leq i$ — the network sees no edges past the pointer.
- **Concatenate** (A, B)**:** $2\binom{n}{2}$ inputs (342 for $n = 19$).
- **Hidden:** dense 128, dense 64, dense 4; ReLU $x \mapsto \max(0, x)$ on each layer.
- **Output:** one sigmoid neuron $\to p \in (0, 1)$. Sample $b \sim \mathrm{Bernoulli}(p)$, set $A[t] \leftarrow b$, move B's 1 from position t to position $t{+}1$, query again. After $\binom{n}{2}$ queries one graph is finished.
- **Training:** binary cross-entropy on the bit decisions taken in the best-scoring sampled graphs.

The cross-entropy method loop

A sample–select–imitate loop around the network:

1. The network reads the partial word and the current position, and outputs the probability that the next bit is 1.
2. Sample a batch of complete constructions (1000 per iteration in the released code) bit by bit from these probabilities.
3. Score every construction; keep the **elite** — the top few percent (93rd percentile in the released code).
4. Retrain the network to imitate the elite's decisions; carry the very best constructions into the next batch.
5. Repeat — the proposal distribution concentrates on high-scoring structures.

A 19-vertex counterexample

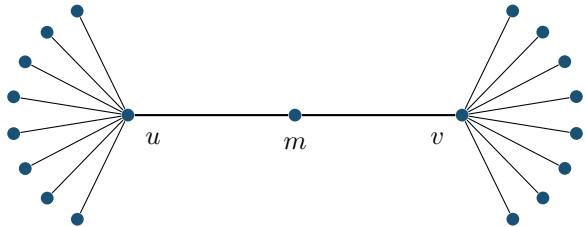
On Conjecture 2.1 ($\lambda_1 + \mu \geq \sqrt{n-1} + 1$):

- The agent found a 19-vertex counterexample — a tree Wagner calls a *balanced double star* — with $\lambda_1 = \sqrt{10}$, $\mu = 2$:

$$\lambda_1 + \mu \approx 5.16 \;<\; 5.24 \approx \sqrt{18} + 1.$$

- The conjecture holds for all $n \leq 18$, so no smaller counterexample exists.
- History, stated in Wagner's paper: Stevanović had already disproved the conjecture in 2010, with a 600-vertex counterexample.

The counterexample: two stars joined through a middle vertex



- A middle vertex m joined to the centers u, v of two stars with 8 leaves each: 19 vertices.
- $\mu = 2$: every edge contains u or v, so a matching has at most two edges; one leaf edge at u plus one at v are disjoint.
- $\lambda_1 = \sqrt{10}$: in the top eigenvector the centers carry equal value x; each leaf carries x/λ, the middle vertex $2x/\lambda$; the center equation $\lambda x = 8(x/\lambda) + 2x/\lambda$ forces $\lambda^2 = 10$.

Seven problems; the 203-vertex counterexample

- The paper refutes or answers seven problems: five by this loop, two by linear programming.
- Among them: a second conjecture from the same survey, about the eigenvalues of the graph's distance matrix, refuted at 203 vertices; a 1989 conjecture of Collins on characteristic-polynomial coefficients of trees.
- Wagner: "the only thing we need to change in the code is the function that calculates the score of a given construction."

The 203-vertex case:

- The run at $n = 30$ found *no* counterexample.
- Its near-optimal structures showed a pattern — a path with many pendant vertices (vertices of degree 1).
- Wagner scaled the pattern up by hand to 203 vertices, where it violates the conjecture.

Next problem: fast matrix multiplication

Wagner's loop searched for a *graph*. AlphaTensor (DeepMind, 2022) searches for an *algebraic algorithm*.

- **Concrete instance: 2×2.** Compute $C = AB$ for 2×2 matrices over a field $\mathbb{F}$. The textbook formula uses 8 multiplications $a_{ij} b_{k\ell}$. Can fewer than 8 scalar multiplications do it?
- **Search space.** A *bilinear scheme with r multiplications*: three coefficient tables $\alpha_{ij}^{(k)}, \beta_{ij}^{(k)}, \gamma_{ij}^{(k)} \in \mathbb{F}$ ($k = 1, \ldots, r$; $i, j \in \{1, 2\}$) defining

$$m_k \;=\; \Big(\textstyle\sum_{i,j} \alpha_{ij}^{(k)} a_{ij}\Big)\Big(\sum_{i,j} \beta_{ij}^{(k)} b_{ij}\Big), \qquad c_{ij} \;=\; \textstyle\sum_{k=1}^{r} \gamma_{ij}^{(k)}\, m_k.$$

  Each m_k is one multiplication in $\mathbb{F}$. The α, β, γ are the unknowns to search for; the scheme is valid iff $c_{ij} = (AB)_{ij}$ as polynomials in a_{ij}, b_{ij}.
- **Score: r, the multiplication count.** Applied recursively, the 2×2 scheme multiplies $N \times N$ matrices in $\Theta(N^{\log_2 r})$ time — r becomes the exponent on N; block-additions add only a constant factor. Lower r = lower exponent.
- **Known record.** Strassen 1969, $r = 7$ for 2×2 (next frame): $\Theta(N^{2.807})$ vs textbook $\Theta(N^3)$.

Strassen 1969: 2×2 matrices in 7 multiplications

Instead of the 8 products of the textbook formula, compute

$$
\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) & m_5 &= (a_{11} + a_{12})\, b_{22} \\
m_2 &= (a_{21} + a_{22})\, b_{11} & m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_3 &= a_{11}(b_{12} - b_{22}) & m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11})
\end{aligned}
$$

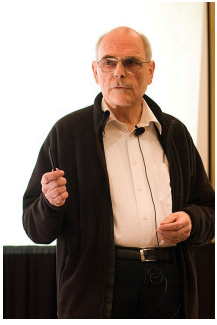
and assemble the product from them:

$$
\begin{aligned}
c_{11} &= m_1 + m_4 - m_5 + m_7, & c_{12} &= m_3 + m_5, \\
c_{21} &= m_2 + m_4, & c_{22} &= m_1 - m_2 + m_3 + m_6.
\end{aligned}
$$

7 multiplications, 18 additions/subtractions, valid over any field.

Recursion on blocks: $O(n^{\log_2 7})$

- The entries a_{ij}, b_{ij} may themselves be matrix blocks — Strassen's identities use no commutativity.
- Applied recursively to 2×2 block matrices: 7 multiplications per level, so running time $O(n^{\log_2 7}) = O(n^{2.807})$ instead of $O(n^3)$.
- Additions are cheap ($O(n^2)$ per level); recursion amplifies the multiplication count, so the number of multiplications is the quantity to minimize.
- A 4×4 matrix, treated as a 2×2 matrix of 2×2 blocks: $7 \times 7 = 49$ multiplications.



Volker Strassen (b. 1936) — German mathematician; the 1969 seven-multiplication algorithm started fast matrix multiplication. Photo: Wikimedia Commons.

The matrix-multiplication tensor $\mathcal{T}_n$

$C = AB$ is bilinear in (A, B). Any bilinear map is described by a 3D table of structure constants; $\mathcal{T}_n$ is that table.

- **Flatten.** Read $A, B, C \in \mathbb{F}^{n\times n}$ row-by-row as $a, b, c \in \mathbb{F}^{n^2}$. For $n = 2$: $(a_1, a_2, a_3, a_4) := (a_{11}, a_{12}, a_{21}, a_{22})$.
- **Defining equations ($n = 2$).** From the textbook expansion of $C = AB$:

$$c_1 = a_1b_1 + a_2b_3, \quad c_2 = a_1b_2 + a_2b_4, \quad c_3 = a_3b_1 + a_4b_3, \quad c_4 = a_3b_2 + a_4b_4.$$

- **Tensor.** $\mathcal{T}_2 \in \{0, 1\}^{4\times4\times4}$ with $(\mathcal{T}_2)_{ijk}$ = coefficient of a_ib_j in c_k:

$$c_k \;=\; \sum_{i,j=1}^{4} (\mathcal{T}_2)_{ijk}\, a_ib_j.$$

  Eight ones: $(1,1,1), (2,3,1), (1,2,2), (2,4,2), (3,1,3), (4,3,3), (3,2,4), (4,4,4)$; other 56 entries are 0.
- **General n.** $\mathcal{T}_n \in \{0, 1\}^{n^2\times n^2\times n^2}$, n^3 ones.

Rank-R decomposition = algorithm with R multiplications

$u, v, w \in \mathbb{F}^{n^2}$ are vectors of coefficients. Their **outer product** is the rank-one tensor with entries $(u \otimes v \otimes w)_{ijk} = u_i v_j w_k$. A **rank-$R$ decomposition** of $\mathcal{T}_n$ is

$$\mathcal{T}_n \;=\; \sum_{r=1}^{R} u^{(r)} \otimes v^{(r)} \otimes w^{(r)} \quad \Longleftrightarrow \quad (\mathcal{T}_n)_{ijk} = \sum_{r=1}^{R} u_i^{(r)}\, v_j^{(r)}\, w_k^{(r)}.$$

Read columnwise it is an R-multiplication algorithm for $C = AB$:

$$m_r = \Big(\textstyle\sum_i u_i^{(r)} a_i\Big)\Big(\textstyle\sum_j v_j^{(r)} b_j\Big), \qquad c_k = \textstyle\sum_r w_k^{(r)}\, m_r.$$

- $u^{(r)}, v^{(r)}, w^{(r)}$ are exactly $\alpha^{(r)}, \beta^{(r)}, \gamma^{(r)}$ from the bilinear-scheme frame, with the matrix index (i, j) flattened to a single index in $\{1, \ldots, n^2\}$.
- Strassen = rank-7 decomposition of $\mathcal{T}_2$: each r gives one m_r, the seven columns together reproduce all 64 entries of $\mathcal{T}_2$.
- Rank of $\mathcal{T}_n$:= the minimal such R. Finding low-rank decompositions of a given tensor is NP-hard in general (Håstad 1990).

Known ranks before 2022; the exponent ω

- The rank of $\mathcal{T}_3$ is unknown: between 19 (Bläser's lower bound) and 23 (Laderman's 1976 scheme).
- Before 2022, nothing below 49 was known for 4×4 over any field.
- Separate research line: the asymptotic exponent ω — the smallest τ such that two $n \times n$ matrices can be multiplied in time $O(n^{\tau+\varepsilon})$ for every $\varepsilon > 0$ — currently $\omega < 2.371339$ (Alman–Duan–Vassilevska Williams–Xu–Xu–Zhou, SODA 2025). Those algorithms hide constants so large they never beat practical algorithms at real sizes.

TensorGame: decomposition as a single-player game

Fawzi et al., *Discovering faster matrix multiplication algorithms with reinforcement learning*, Nature 610 (5 Oct 2022) 47–53 — DeepMind's system **AlphaTensor**.
Target: exact, usable decompositions at small fixed sizes — not the exponent ω.

Input	the tensor $\mathcal{T}_{n,m,p}$ and a finite coefficient set F — e.g. $F = \{-2, -1, 0, 1, 2\}$ for standard arithmetic, or $F = \{0, 1\}$ with reduction mod 2 for arithmetic in $\mathbb{Z}_2$.
Output	triples $(u^{(r)}, v^{(r)}, w^{(r)})$ forming a verified rank-R decomposition — a correct algorithm with R multiplications.
Objective	minimize R.

- **State**: the residual tensor, starting at $S_0 = \mathcal{T}_n$.
- **Move**: pick one triple (u, v, w) over F; update $S_t = S_{t-1} - u \otimes v \otimes w$.
- **Reward**: -1 per move; if the move limit is hit, an extra penalty bounding the rank of the remainder.
- **End**: $S_t = 0$ — the moves played are a valid decomposition.

The agent: AlphaZero adapted to tensors

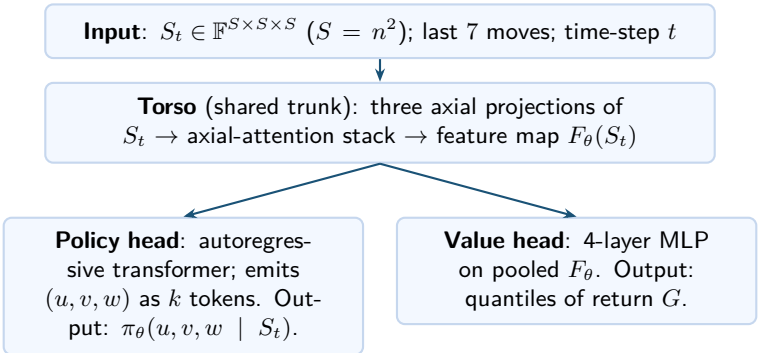
AlphaZero: Monte Carlo tree search (MCTS) guided by a network that proposes moves (policy) and values states (value), trained by self-play (reinforcement learning: trial and error driven by a numeric reward).

- Per-move action space above 10^{12} (the number of triples (u, v, w) over the coefficient set F; board games: hundreds of legal moves) — actions are *sampled* from the policy network, never enumerated.
- **Synthetic demonstrations**: sum R random triples — a tensor with a known decomposition; pre-train on 5 million such pairs.
- **Change of basis**: re-express A, B, C in new coordinates; $\mathcal{T}_n$ changes but its rank does not, so any basis's decomposition converts back. Each game uses one of ∼100,000 random bases.

Alhussein Fawzi — first author of AlphaTensor; led the work as a research scientist at Google DeepMind; PhD EPFL 2016. Photo: fawzi.ai.

AlphaTensor's network: shared trunk + two heads

One network $f_\theta(S_t) = (\pi_\theta, V_\theta)$, produced by a shared *trunk* (Torso) and two task *heads* (Policy, Value).



Why this shape: at each MCTS visit the agent needs a move suggestion *and* a value of the state; sharing the trunk computes features once, two heads read them. **Axial attention**: on an $S\times S$ grid attend along one axis at a time — $O(S^3)$ per layer instead of $O(S^4)$.

AlphaTensor's network: what's optimized, what's trained

- **End goal (optimization target).** For a fixed $\mathcal{T}_n$, minimize R — the number of TensorGame moves to drive S_t to $\mathbf{0}$. Reward -1 per move, total return $G = -R$. The decomposition itself is produced *at inference*: MCTS guided by the trained network plays one game on $\mathcal{T}_n$; the recorded triples $(u^{(r)}, v^{(r)}, w^{(r)})$ are the decomposition.
- **Training targets (what the heads chase).**
  - Policy head $\pi_\theta(\cdot|S)$ chases the **MCTS improved policy** $\pi^*(\cdot|S)$ — the visit-count distribution over moves after MCTS searches from S. Loss: cross-entropy $L_\pi = -\sum_a \pi^*(a|S) \log \pi_\theta(a|S)$.
  - Value head $V_\theta(S)$ chases the **observed return** G from S. AlphaTensor models the full distribution of G; $L_V =$ quantile-regression loss.

  Joint loss $L_\theta = L_\pi + L_V$; minimized by SGD on θ.
- **Training data (two streams, mixed in each batch).**
  - *Self-play*: the current network plays games against itself via MCTS; stored as (S, π^*, G) triples.
  - *Synthetic demos*: sum R random rank-one triples $\rightarrow$ a tensor with a *known* R-rank decomposition; the R summands are supervised (S, action) examples.

AlphaTensor: results

format	arithmetic	previous best	AlphaTensor
4×4	$\mathbb{Z}_2$ (mod 2)	49	**47**
4×4	standard	49	49 (not improved)
4×5 by 5×5	standard	80	**76**
5×5	$\mathbb{Z}_2$	98	**96**

- The 47 is the first count below the 49 of Strassen applied twice, since 1969; the naive count for 4×5 by 5×5 is 100.
- Matched or improved the best known rank for over 70 tensors ($n, m, p \leq 12$).
- The rank-47 scheme uses mod-2 arithmetic, so it works only in characteristic 2; the 4×4 record over $\mathbb{R}, \mathbb{C}$ stayed at 49. Recursively it gives $O(N^{\log_4 47}) = O(N^{2.778})$ — a runnable exponent, unlike the smaller asymptotic $\omega < 2.371$ (only for matrices too large to compute).
- At rank 49 (standard arithmetic) it found more than 14,000 inequivalent schemes.

Why a mod-2 scheme is not an algorithm for real matrices

A multiplication scheme is a set of *algebraic identities*: the products m_r and the assembly $c_k = \sum_r w_k^{(r)} m_r$ must expand to the right entries of AB. Whether an identity is *true* depends on the arithmetic the entries obey.

- Strassen uses only the coefficients $+1, -1$; the identities hold by ordinary algebra valid in **every** field — real, complex, rational, mod-p alike.
- A scheme found "mod 2" may use $1 + 1 = 0$ (equivalently $x + x = 0$): an identity leaning on that is true mod 2 but **false over the reals**. So AlphaTensor's 47 is a correct algorithm only where $1 + 1 = 0$ holds.

The **characteristic** of a field is how many times you add 1 to reach 0:

- characteristic 2: $1 + 1 = 0$ — e.g. $\mathbb{F}_2 = \{0, 1\}$, the arithmetic of bits (coding theory, cryptography);
- characteristic 0: never returns to 0 — $\mathbb{Q}, \mathbb{R}, \mathbb{C}$, the matrices people actually compute with.

Over characteristic 0 the 4×4 record stayed at Strassen's 49 — the barrier AlphaEvolve broke.

Kauers–Moosbauer: $96 \to 95$ in three days

- Kauers and Moosbauer (arXiv:2210.04045, 8 October 2022 — three days after the Nature paper): starting from AlphaTensor's 5×5 mod-2 scheme, hand-guided transformations remove one more multiplication: 95.
- They systematized the idea as random walks on *flip graphs* — vertices are correct multiplication schemes, edges are local modifications (ISSAC 2023) — improving further formats over arbitrary fields.
- Jakob Moosbauer: doctoral work at the same institute (Johannes Kepler University Linz).

Manuel Kauers — professor of algebra, Johannes Kepler University Linz; head of its Institute for Algebra. Photo: JKU.

Hardware-tailored decompositions

- Change the terminal reward to the measured runtime of the found algorithm on a target device — multiplying 8192×8192 matrices as 4×4 blocks.
- Targets: an Nvidia V100 GPU and a Google TPU v2 (tensor processing unit, Google's machine-learning accelerator chip).
- Found schemes run 10–20% faster (DeepMind's reported range) than the standard library multiplication on large matrices.
- Schemes tuned for one device lose their advantage on the other.

What AlphaTensor released

`github.com/google-deepmind/alphatensor` — code Apache 2.0, other materials CC-BY 4.0. Four directories:

- `algorithms/` — the discovered factorizations as compressed array files (the `.npz` format of NumPy, Python's array library; one file for standard arithmetic, one for mod 2), plus a notebook that loads and verifies them.
- `benchmarking/` — the speed measurement of the discovered 4×4-block algorithm on an Nvidia V100 GPU.
- `nonequivalence/` — the 14,236 rank-49 factorizations of $\mathcal{T}_4$, with invariants certifying pairwise nonequivalence.
- `recombination/` — code building larger formats from factorizations of smaller ones.

Not included: any training or search code. The paper's Methods give the withheld system's scale: training on 64 TPU v3 cores for 600,000 iterations, self-play on 1,600 TPU v4 actors (workers that play games to generate training data), about one week to converge.

AlphaDev: sorting routines as a game over assembly

Mankowitz et al., *Faster sorting algorithms discovered using deep reinforcement learning*, Nature 618 (June 2023) 257–263.

The object constructed is a program for sorting a small fixed number of values, built one x86 assembly instruction at a time (assembly: the processor's lowest-level instruction language).

- **State**: the program so far, plus the contents of registers (the processor's working storage) and memory after running it on test inputs.
- **Move**: append one instruction (`mov` = copy, `cmp` = compare, `cmov` = copy only if a condition holds — the branchless conditional that makes the discovered sorts fast).
- **Reward**: correctness on test sequences, plus a latency term (program length or measured latency).
- Agent: an AlphaZero extension, as in TensorGame.

AlphaDev: results and deployment

Program length in instructions, against the human benchmark:

routine	human	AlphaDev
sort3	18	**17**
sort4	28	28 (matched)
sort5	46	**42**
VarSort5 (variable length)	115	**63**

- The routines were translated back to C++ and merged into libc++ — the C++ standard library of the LLVM compiler infrastructure — in April 2022.
- First change to these small-sort subroutines in over a decade; the paper estimates the code runs trillions of times per day.

Daniel J. Mankowitz — first author of AlphaDev; staff research scientist, Google DeepMind. Photo: X profile.

Two limits shared by Wagner, AlphaTensor, AlphaDev

- **Move-level generation.** The generator emits one bit / one rank-one tensor / one instruction at a time, starting from zero knowledge.
  - Everything the search knows, it learned from the score.
  - No mathematical literature and no domain idiom enters the search.
- **One problem, one system.** AlphaTensor's network, game, and action space are purpose-built for tensors; AlphaDev's for x86 assembly.
  - A new problem means a new encoding, a new action space, often a new network.

Part B.2 — FunSearch: a language model as the generator

Cap sets in $\mathbb{F}_3^n$

$\mathbb{F}_3^n$: the length-n vectors with entries $0, 1, 2$, added coordinate-wise mod 3. A **cap set** in $\mathbb{F}_3^n$ is a set of vectors no three of which (distinct) sum to zero. Equivalent formulations:

- Since $2 \equiv -1 \pmod 3$: $x + y + z = 0 \iff x + z = 2y$ — no 3-term arithmetic progression.
- Lines of the affine space $\mathrm{AG}(n, 3)$ are exactly the triples $\{a,\ a + d,\ a + 2d\}$ — a cap set contains no full line.

The card game SET is the case $n = 4$:

- the 81 cards are the points of $\mathbb{F}_3^4$ (four attributes, three values each);
- a "SET" is a line;
- the largest collection of cards containing no SET has exactly 20 cards (Pellegrino, 1971).

Known maximum sizes c_n

c_n = maximum size of a cap set in dimension n; exact values known only up to $n = 6$:

n	1	2	3	4	5	6	7	8
c_n	2	4	9	20	45	112	?	?
known range							236–288	≥ 496 (pre-2023)

- $c_6 = 112$: construction from Hill's 1973 cap; maximality proved by Potechin (2008).
- $n = 7$: best known construction 236 points.
- Brute force in dimension 8 is out of reach: $3^8 = 6561$ vectors, $\binom{6561}{512} \approx 10^{779}$ candidate sets of size 512.

The cap set capacity C

Define $C = \sup_n c_n^{1/n}$, the growth rate of maximum cap sets.
From above: $C \leq 2.756$ — Croot–Lev–Pach, then Ellenberg–Gijswijt (*Annals* 185, 2017), polynomial method.

From below — one explicit cap set $S \subset \mathbb{F}_3^n$ per row, giving $C \geq |S|^{1/n}$. A **construction problem on cap sets** (score $= |S|$):

bound	construction	year
$C \geq 2.2101$	Calderbank–Fishburn	1994
$C \geq 2.2173$	Edel	2004
$C \geq 2.2180$	Tyrrell (SAT solvers)	2022

Terence Tao — UCLA, Fields Medal 2006. On the cap set problem: "perhaps my favourite open question." Photo: Wikimedia Commons.

FunSearch: input, output, objective

Romera-Paredes et al., *Mathematical discoveries from program search with large language models*, Nature 625 (online 14 Dec 2023) 468–475. "FunSearch" = searching in *fun*ction space.

FunSearch trains no model. The LLM is used *frozen* as a code sampler; all improvement accumulates in a population of Python programs evolved by selection on the evaluator's score. This is the break from Wagner/AlphaTensor/AlphaDev, where the network is trained from scratch on the score.

Input	(a) an **evaluator**: a trusted scoring program — run the candidate, check the cap property exactly, return the size; (b) a **skeleton**: a short greedy program with one inner heuristic function left trivial; (c) a pretrained code **LLM**, Codey (Google's PaLM 2, code-specialized) — used frozen as a black-box sampler.
Output	a short Python function; plugged into the skeleton, it produces the record object.
Objective	maximize the evaluator's score of the skeleton + function combination.

The cap-set skeleton: only `priority` evolves

```
def solve(n):
    cap = []
    for v in sorted(F3n(n), key=lambda v: -priority(v, n)):
        if not creates_line(cap, v):
            cap.append(v)
    return cap                # the candidate cap set in F_3^n

def priority(v, n):           # <- the ONLY part the LLM rewrites
    return 0.0
```

- `solve(n)` returns an explicit cap set $S \subset \mathbb{F}_3^n$. The FunSearch **score** is its size: $|S| = \mathrm{len}(\mathtt{solve(n)})$.
- Enumeration, the line-check `creates_line`, and the greedy loop are human-written — correct by construction.
- The LLM only proposes new `priority(v, n)` bodies. Search space = functions $\mathbb{F}_3^n \to \mathbb{R}$; a better ordering yields a larger greedy cap set.

The evolutionary loop

1. **Sample.** Show the LLM $k = 2$ stored programs sorted by score (`priority_v0`, `priority_v1`); ask it to write `priority_v2` ("best-shot prompting").
2. **Evaluate.** Run each generated program in a sandbox with time and memory limits; discard crashes, timeouts, and low scores.
3. **Store.** Keep scored programs in a database of 10 **islands** — independent subpopulations.
   - Every 4 hours, wipe the worst half of the islands; reseed from the best survivors.
   - Within an island, cluster programs by behavior signature (the tuple of scores on the test inputs); prefer shorter programs among equals.
4. Repeat, asynchronously.

Scale and cost

- About 15 LLM samplers and $\sim$150 CPU evaluators in parallel.
- On the order of 10^6 generated programs per experiment, over a few days.
- The LLM is never trained — all improvement accumulates in the population of programs.
- Reproduction estimate from the paper, with the open code model StarCoder-15B (15 billion parameters): about two days on 15 GPUs plus 5 CPU servers, roughly \$800–1,400 of cloud compute.

Dimension 8: a 512-point cap set

- FunSearch found a cap set of size **512** in $\mathbb{F}_3^8$.
- Previous record: 496, from a 2001–2004 line of constructions combining lower-dimensional caps (Edel–Bierbrauer, Edel).
- FunSearch matched the known records for all $n \leq 7$.
- The 512 was rare even for FunSearch: 4 out of 140 runs reached it.

Bernardino Romera-Paredes — first author of FunSearch; research scientist, Google DeepMind; PhD UCL, postdoc Oxford. Photo: X profile.

Admissible sets: cap-set templates

Shift of target. A cap set in dim n gives one row of the lower-bound table. An *admissible set* $\mathcal{A} \subset \{0,1,2\}^n$ of weight-w vectors (each with exactly w nonzero entries) gives, via a fixed substitution, a cap-set family in $\mathbb{F}_3^{nk}$ for every $k \geq 1$, hence one bound $C \geq |\mathcal{A}|^{1/n}$.

The substitution. Fix a small cap set $C_0 \subset \mathbb{F}_3^k$ with two disjoint "slot" subsets. Lift each template $v \in \mathcal{A}$ block by block: at block i place a zero block if $v_i = 0$, a slot-1 vector if $v_i = 1$, a slot-2 vector if $v_i = 2$. Varying the slot choices gives the cap-set family above (Calderbank–Fishburn 1994, Edel 2004).

Admissibility (what makes the lift a cap set). For any three distinct templates a, b, c, at some coordinate i the multiset $\{a_i, b_i, c_i\}$ is one of $\{0,1,2\}, \{0,0,1\}, \{0,0,2\}$ — i.e. zero + slot-1 + slot-2, or two zeros + one slot-j. At block i the three lifted blocks cannot form a 3-AP, so a, b, c are not a 3-AP globally.

Full-size: $|\mathcal{A}| = \binom{n}{w}$, written $\mathcal{I}(n, w)$. FunSearch evolves `priority` on $\{0,1,2\}^n$, same skeleton.

Three capacity improvements: $2.2184 \to 2.219486 \to 2.2202$

1. FunSearch found a full-size admissible set $\mathcal{I}(12, 7)$: $C \geq 2.2184$ — past Tyrrell's SAT-solver record.
2. The authors read the discovered program; it treated coordinates symmetrically in blocks. They defined symmetry-restricted admissible sets and re-ran the search in the smaller space: $\mathcal{I}(15, 10)$, giving $C \geq 2.219486$.
3. The restricted search produced a partial admissible set in $\mathcal{A}(24, 17)$ with 237,984 vectors:

$$C \;\geq\; 2.2202.$$

The paper: "the largest improvement in 20 years to the asymptotic lower bound" (i.e. since Edel 2004).

Program output vs. object output

- **Compression.** The 237,984-vector admissible set is generated by a program of a few lines. The paper: FunSearch "attempts to find solutions that have low Kolmogorov complexity" (the length of the shortest program producing the object); "because most problems we care about are structured (highly non-random), we believe that solutions are described more concisely with a computer program."
- **Scalability.** The previous record ($\mathcal{I}(11, 7)$, Tyrrell 2022) came from SAT solvers, which enumerate the object and stop scaling; a program instantiates at sizes no explicit search reaches.
- **Readability.** The $\mathcal{I}(12, 7)$ program's block symmetry was read by the authors and turned into a restricted search space — which then broke the record again.
- The paper on the LLM's role: "a source of diverse (syntactically correct) programs with occasionally interesting ideas."

Jordan Ellenberg on the discovered programs

> *"The solutions generated by FunSearch are far conceptually richer than a mere list of numbers. When I study them, I learn something."*

Jordan Ellenberg — University of Wisconsin–Madison; co-author of the Ellenberg–Gijswijt upper bound $C \leq 2.756$, and a co-author of the FunSearch paper.

Jordan Ellenberg. Photo: Wikimedia Commons.

Online bin packing with the same loop

Online bin packing: items of sizes in $(0, 1]$ arrive one at a time; each must be placed immediately into a bin of capacity 1; minimize the number of bins used.

- Classical baselines: *first fit* (leftmost bin that fits), *best fit* (fullest bin that fits).
- Skeleton: assign each arriving item to the bin maximizing an evolved `heuristic(item, bins)`.
- Benchmarks: OR-Library (Beasley's standard operations-research test instances) and instances with Weibull-distributed item sizes (a model of realistic scheduling workloads).

Bin packing: excess bins over the lower bound

Fraction of bins beyond the standard lower bound on the optimal offline packing — the best packing computable with all items known in advance (smaller is better):

	OR1	OR2	OR3	OR4	Weib. 5k	10k	100k
first fit	6.42%	6.45%	5.74%	5.23%	4.23%	4.20%	4.00%
best fit	5.81%	6.06%	5.37%	4.94%	3.98%	3.90%	3.79%
FunSearch	**5.30%**	**4.19%**	**3.11%**	**2.47%**	**0.68%**	**0.32%**	**0.03%**

- The discovered rule: place the item best-fit-style only when the fit is tight; otherwise prefer bins with more room left — avoiding small unfillable gaps.
- Trained on small instances; the same function generalizes to instances $100\times$ larger.

FunSearch's scope conditions

Stated by the authors as scope conditions:

- an efficient evaluator;
- a graded score — partial credit, not yes/no;
- a skeleton isolating a small evolvable heuristic.

Problems lacking these are out of scope — the paper names theorem proving explicitly: the signal is binary, and there is no cheap evaluator short of a proof checker run on a complete proof.

What FunSearch released

`github.com/google-deepmind/funsearch` — six directories. Five hold results:

- `cap_set/` (the priority functions and the 512-point cap set as a text file), `admissible_set/`, `bin_packing/` (with an evaluation suite reproducing the paper's table), `cyclic_graphs/`, `corner_free_sets/`.

The sixth, `implementation/`, is the method itself: about 900 lines of single-threaded Python.

- `funsearch.py` — the pipeline; the problem is one Python file in which the decorator `@funsearch.evolve` marks the function to evolve and `@funsearch.run` marks the scoring entry point (a decorator is a label attached to a function definition).
- `programs_database.py` — islands, score-signature clusters, softmax sampling, periodic reset of the worst half.
- `config.py` — the paper's defaults: 2 programs per prompt, 10 islands, reset every 4 hours, 15 samplers, 140 evaluators.

Two deliberate stubs: `LLM` and `Sandbox`

Two classes in the released code are stubs — placeholder methods that raise an error until the user supplies an implementation:

- `sampler.LLM`: `raise NotImplementedError('Must provide a language model.')`
- `evaluator.Sandbox`: `raise NotImplementedError('Must provide a sandbox for executing untrusted code.')` — a sandbox is an isolated execution environment, so machine-written code cannot damage the host.

The README on scope: the directory "does not contain language models for generating new programs, the sandbox for executing untrusted code, nor the infrastructure for running FunSearch on our distributed system," and is "intended … for adapting it for use with any available language models, sandboxes, and distributed systems."

Reproducible in design, not runnable out of the box: the user supplies an LLM binding, a sandbox, and parallelism.

Part B.3 — AlphaEvolve: from one function to whole code files

Evolve blocks, two Gemini models, diffs

Novikov et al., DeepMind, 14 May 2025 (arXiv:2506.13131).

- The user marks evolvable regions of a real code file:

```
# EVOLVE-BLOCK-START
  ... code the system may rewrite ...
# EVOLVE-BLOCK-END
```

  Everything outside is fixed scaffolding. Several regions can evolve together — one discovered training-pipeline improvement required 15 coordinated mutations across optimizer, loss, and hyperparameters.

- Two Gemini models generate: Flash (fast, cheap — breadth) and Pro (slower — occasional higher-quality suggestions).
- Mutations are emitted as SEARCH/REPLACE diffs (not full rewrites):

```
<<<<<<< SEARCH
... old code chunk ...
=======
... new code chunk ...
>>>>>>> REPLACE
```

  The host applies the patch by exact-string substitution on the file.

How the search learns: the prompt

The model's weights are frozen — the only thing that improves from step to step is *what the model is shown*. So each prompt is assembled to carry the current state of the search:

- the best programs found so far, with their scores — the model sees what already works;
- the problem description, relevant equations, literature, human hints;
- the actual output of running the previous candidate — its error messages, printed numbers, timings — so the model fixes a concrete failure instead of guessing.

The system can even evolve the wording of its own prompt instructions (*meta-prompting*). All "learning" lives here, in the text handed to a fixed model.

Filtering and storing candidates: cascade and database

Evaluation cascade — *problem:* fully scoring a candidate can take minutes to hours, and most candidates are bad.

- Score each candidate on a cheap test first; only survivors advance to the expensive tests. Bad candidates die after the cheap stage.
- Several metrics can be scored at once — optimizing several together often improves even the single target metric.

Program database — *problem:* keeping only the single best program collapses the search onto one approach and sticks in a local optimum. Keep a *diverse* set instead, two ways combined:

- **MAP-Elites:** bin programs by a few cheap features (length, a diversity measure); keep the best in each bin — many distinct styles survive.
- **Islands:** several separate populations evolve in parallel, exchanging members only occasionally, so independent lines of attack do not homogenize.

AlphaEvolve vs. FunSearch (the paper's comparison)

The same loop as FunSearch, with every component upgraded two years later — the paper's own comparison:

FunSearch (2023)	AlphaEvolve (2025)
evolves a single function	evolves an entire code file
up to 10–20 lines, Python only	up to hundreds of lines, any language
fast evaluation needed ($\leq$ 20 min, 1 CPU)	evaluation may take hours, on accelerators
millions of LLM samples	thousands of samples suffice
small code model; no gain from bigger	benefits from frontier LLMs
minimal prompt context	rich context: problem, literature, feedback
one score	several scores optimized simultaneously

More compute per candidate, far fewer candidates.

4×4 over $\mathbb{C}$ in 48 multiplications

- Strassen recursion: rank 49 for $\mathcal{T}_4$, over any field. AlphaTensor's 47 needs $1 + 1 = 0$ (characteristic 2). Over characteristic 0, rank below 49 was open for 56 years.
- AlphaEvolve found rank **48** over $\mathbb{C}$.
  - What evolves: a gradient-based search program (loss, optimizer, initialization). Run it $\rightarrow$ numerical decomposition $\rightarrow$ round entries to integers or half-integers $\rightarrow$ verify the rounded identity exactly.
  - Applies recursively. Coefficients are complex, but the verified identity is an algebraic equality, so it multiplies real matrices too.
- Same agent — no tensor-specific game. Only the evaluator and the prompt change between problems.

What the rank-48 scheme concretely is

Three 16×48 complex matrices U, V, W: the 16 rows index the entries of A, of B, of the output C (each 4×4, flattened); the 48 columns are the 48 products.

- For $r = 1, \ldots, 48$: $\quad m_r = (U_{:,r} \cdot a)\,(V_{:,r} \cdot b)$ — a complex combination of A's entries times one of B's. Then $c_{ik} = (W\text{-row}) \cdot m$.
- Every coefficient is one of **nine values**: $\{0,\ \pm\frac{1}{2},\ \pm\frac{i}{2},\ \pm\frac{1}{2} \pm \frac{i}{2}\}$ — this is what "half-integer complex" means.

One of the 48 products (common $\frac{1}{2}$ factored out):

$$m_1 = \tfrac{1}{2}\big[(1{+}i)(a_{11}{+}a_{12}) + (1{-}i)(a_{21}{+}a_{22}{+}a_{31}{+}a_{32}{+}a_{41}{+}a_{42})\big]$$

$$\cdot\ \tfrac{1}{2}\big[-b_{11} - b_{21} + b_{31} - i\,b_{41}\big].$$

Full scheme: $3 \times 16 \times 48 = 2304$ numbers; one output sums 28 of the 48 products. The certificate is the point — expand the 48 products and the 4×4 identity checks out (DeepMind's notebook, re-run for this lecture).

14 formats improved; the 4×4 record over time

- Improved the known rank for 14 matrix-multiplication formats — e.g. $\langle 3, 4, 7 \rangle$: 66 $\to$ 63; $\langle 4, 5, 6 \rangle$: 93 $\to$ 90.
- Matched or surpassed the best known rank for all formats with $m, n, p \leq 5$.

The 4×4 record over time:

1969	Strassen, human	49	any field
2022	AlphaTensor, reinforcement learning	47	$\mathbb{Z}_2$ only
2025	AlphaEvolve, coding agent	48	over $\mathbb{C}$ (char. 0)
2025	humans, from AlphaEvolve's scheme	48	rational (arXiv:2506.13242)

The last row: Dumas–Pernet–Sedoglavic applied an *isotropy* of the tensor (a symmetry — a simultaneous change of basis on the A, B, C slots taking one valid scheme to another) chosen to land on rational coefficients, so 48 now works over $\mathbb{Q}, \mathbb{R}$, not only $\mathbb{C}$.

Over 50 open problems: 75% matched, 20% improved

- Over 50 open problems in analysis, combinatorics, number theory, and geometry, formulated with Javier Gómez-Serrano and Terence Tao.
- For each problem, the agent evolves a *search heuristic program*: given a time budget and the best known construction, output a better one.
- Outcome: $\sim$75% rediscovered the best known construction; $\sim$20% improved on it.
- Follow-up at 67 problems: Georgiev, Gómez-Serrano, Tao, Wagner, arXiv:2511.02864 (November 2025).

Three of the improvements

- **Kissing number, dimension 11.** The kissing number is how many non-overlapping unit spheres can simultaneously touch one central unit sphere. AlphaEvolve found a configuration of **593** spheres; the previous record was 592.
- **Erdős's minimum overlap.** Split $\{1, \ldots, 2n\} = A \sqcup B$ in half. Overlap of the split: $\max_k |\{(a, b) \in A \times B : a - b = k\}|$. Let $M(n) = \min$ overlap over splits; the asymptotic constant $c = \lim_n M(n)/n$ is open. Upper bound: $0.380926 \to 0.380924$.
- **Packing.** Circles in the unit square ($n = 26$), max sum of radii: $2.6340 \to \mathbf{2.6358}$ (this is the source of the 2.635 record). Unit regular hexagons in a smallest enclosing regular hexagon, outer side: $4.000 \to \mathbf{3.942}$. Further records on circle-packing variants, Heilbronn-type problems, and autocorrelation inequalities.

Each problem was configured only through its evaluator and prompt — no problem-specific system was built for any of them.

Datacenter scheduling: 0.7% of fleet compute

- Target: a scoring heuristic inside Borg (Google's cluster scheduler) that re-ranks candidate machines for each arriving job.
- The evolved function is one line — a sum of four terms built from each machine's leftover processor and memory capacity.
- DeepMind chose it over a deep-RL alternative for its "interpretability, debuggability, predictability and ease of deployment" (the announcement).
- In fleet-wide production: recovers on average 0.7% of Google's compute resources that would otherwise be stranded.

Gemini's training kernels; FlashAttention; a TPU circuit

- **Gemini training kernels.** A tiling heuristic — how each multiplication is cut into hardware-sized blocks — for the matmul kernels used to train Gemini. 23% average kernel speedup; $\sim$1% off total Gemini training time; tuning turnaround cut from months to days.
- **FlashAttention** (the standard fast attention implementation in transformers): 32% kernel speedup, obtained by editing the compiler's intermediate representation.
- **TPU circuit.** A simplification of Verilog code (the hardware-description language) removed unnecessary bits in an arithmetic unit inside the TPU's matmul circuit. Validated by the designers, slated for an upcoming TPU. Caveat (from the paper): downstream synthesis tools caught it independently.

Each target had an exact automated evaluator: correctness checks plus measured speed or resource metrics.

AlphaEvolve's stated limitation

From the paper:

> *"The main limitation of AlphaEvolve is that it handles problems for which it is possible to devise an automated evaluator."*

- Tasks requiring manual experimentation — much of the natural sciences — are out of scope.
- Proposed next step (not done in the paper): use the programs and results AlphaEvolve discovers as training data for the next base model, so the model itself absorbs the capability.

Alexander Novikov — first author of the AlphaEvolve white paper; research scientist, Google DeepMind. Photo: X profile.

AlphaEvolve: results only

`github.com/google-deepmind/alphaevolve_results` contains a notebook with the mathematical discoveries of the white paper's Section 3 and code verifying their correctness — and states:

> *"This repository does not contain the code to run AlphaEvolve."*

- No agent code, no prompts, no program database, no pipeline.
- Access to the system goes through Google. The announcement blog: "We're planning an Early Access Program for selected academic users" and "exploring possibilities to make AlphaEvolve more broadly available," with an interest-registration form.

Per system: AlphaTensor ships its outputs; FunSearch ships outputs and the method as readable reference code; AlphaEvolve ships outputs only.

Part B.4 — OpenEvolve: an open re-implementation of AlphaEvolve

OpenEvolve: an open re-implementation of the AlphaEvolve design

- The `openevolve` repository under the `algorithmicsuperintelligence` GitHub org; created and maintained by Asankhaya Sharma. Apache 2.0.
- `pip install openevolve`; version 0.2.27; ∼6,500 GitHub stars (June 2026).
- Model-agnostic: talks to any OpenAI-compatible API endpoint — OpenAI, Gemini, or self-hosted open models served by vLLM or Ollama.
- Command-line tool or Python library; about 9,600 lines of Python.

Asankhaya Sharma — creator and maintainer of OpenEvolve. Photo: GitHub profile.

OpenEvolve from the source: evolve blocks, diffs, ensemble

- **Evolve blocks.** The user marks evolvable regions with the same `# EVOLVE-BLOCK-START` / `# EVOLVE-BLOCK-END` comments AlphaEvolve uses.
- **Diffs.** Default mutation mode is diff-based; the parsing pattern in `config.py` is AlphaEvolve's `<<<<<<< SEARCH` / `=======` / `>>>>>>> REPLACE` format verbatim. Default is diff mode (`diff_based_evolution: true`); switching to full-rewrite mode is a manual config choice.
- **Model ensemble.** A weighted list of models; each generation call picks one by weighted random choice — the Flash/Pro split generalized to arbitrary ensembles. Selection is seeded, so runs are repeatable.
- **Prompts** carry the current program, 3 top programs and 2 diverse programs from the database, optional evolved meta-instructions, and up to 20 KB of *artifacts* — execution output (errors, timings, printouts) from the previous evaluation fed back to the model.

From the source: an evolve block, a diff

Initial program with one evolvable region:

```
# EVOLVE-BLOCK-START
def priority(v, n):
    return 0.0
# EVOLVE-BLOCK-END

def solve(n): ...                  # fixed scaffolding
```

One LLM response (AlphaEvolve SEARCH/REPLACE format, verbatim):

```
<<<<<<< SEARCH
def priority(v, n):
    return 0.0
=======
def priority(v, n):
    return sum(c * (i + 1) for i, c in enumerate(v)) % n
>>>>>>> REPLACE
```

- `openevolve/utils/code_utils.py` (`extract_diffs`, `apply_diff`) extracts SEARCH/REPLACE blocks and applies them by exact-string substitution. If SEARCH is not found verbatim the candidate is discarded as a parse failure.

Where mutation happens: one iteration in code

One iteration = `run_iteration_with_shared_db` in `openevolve/iteration.py`:

```
parent, inspirations = database.sample(...)
prompt   = prompt_sampler.build_prompt(...)   # code+scores -> text
response = await llm_ensemble.generate_with_context(...)
diffs    = extract_diffs(response)            # SEARCH/REPLACE
child    = apply_diff(parent.code, ...)       # patched parent
metrics  = await evaluator.evaluate_program(child, ...)
database.add(child)                           # MAP-Elites insert
```

- The mutation operator is the one LLM call. There is no hand-coded genetic operator in the package: no `mutate()`, no `crossover()` — no bit flips, no syntax-tree edits.
- The child = the parent's code with the diff applied; `parent_id` is recorded — a run leaves an evolution tree.
- Recombination happens inside the model, if at all: the prompt shows top and inspiration programs, and the model may merge their ideas into its diff. No function combines two programs.

What the model sees each iteration

System message, complete (`BASE_SYSTEM_TEMPLATE`): "You are an expert software developer tasked with iteratively improving a codebase. Your job is to analyze the current program and suggest improvements based on feedback from previous attempts. Focus on making targeted changes that will increase the program's performance metrics."

User message: a fixed skeleton of slots (`DIFF_USER_TEMPLATE`):

```
# Current Program Information
- Current performance metrics: {metrics}
- Areas identified for improvement: {improvement_areas}
{artifacts}
# Program Evolution History
{evolution_history}
# Current Program
{current_program}
# Task
...You MUST use the exact SEARCH/REPLACE diff format...
IMPORTANT: Do not rewrite the entire program - focus on
targeted improvements.
```

How the prompt slots are filled

- `{improvement_areas}` is computed, not static (`_identify_improvement_areas`): the parent's fitness — computed excluding the feature dimensions — is compared with the previous attempt's, and a sentence is emitted: fitness improved / declined / stable; plus the program's current region of the feature grid.
- `{artifacts}`: the previous evaluation's execution output — error messages, timings, printouts — capped at 20 KB. The model reads its candidate's own runtime feedback.
- `{evolution_history}`: previous attempts with their scores, the top programs, and the inspiration programs (3 top + 2 diverse by default, all from the parent's island).
- `{current_program}`: the parent's full code.

Selection pressure enters the prompt as text: scores, score changes, and runtime errors are written into the next generation's instructions.

From the source: an evaluator

Contract. `evaluate(path)` returns a dict of metric → value; `combined_score` is what selection optimizes.

```
def evaluate(path):
    spec = importlib.util.spec_from_file_location("cand", path)
    cand = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(cand)
    try:
        signal.alarm(300)
        S = cand.solve(n=8)
        signal.alarm(0)
    except Exception as e:
        return {"combined_score": 0.0, "valid": 0.0, "error": str(e)}
    if not is_cap_set(S):
        return {"combined_score": 0.0, "valid": 0.0}
    return {"combined_score": float(len(S)), "valid": 1.0,
            "complexity": len(open(path).read())}
```

- Cascade: define `evaluate_stage1/2/3` for early culling; every stage must return every metric key (placeholder `0.0`) — a missing key silently drops the candidate at database insert.

MAP-Elites: one champion per cell of a feature grid

Mouret–Clune 2015, *Illuminating search spaces by mapping elites*. Each program carries two kinds of numbers:

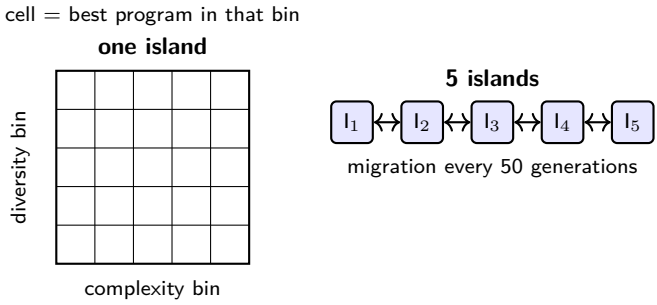
- a **fitness** — the evaluator's score, the thing being maximized;
- **feature coordinates** — descriptive axes, here: code length ("complexity") and edit-distance diversity.

Cut the feature space into a grid. Each cell keeps **one** program: the best-scoring one that landed there.

- Insertion: empty cell $\rightarrow$ occupy; occupied $\rightarrow$ replace only if strictly better.
- The feature axes are coordinates, not objectives: each cell still maximizes the single fitness.
- Best-of-population selection collapses onto one kind of program; the grid keeps the best program of each kind alive as a future parent.

The database: MAP-Elites grid × 5 islands

Each island carries a MAP-Elites grid; the database keeps 5 islands.



- Defaults: 10 bins per axis, complexity × edit-distance diversity; population 1000 per island, archive 100. Any evaluator metric can replace a feature axis.
- **Insert.** Compute the candidate's feature bin; replace the cell's occupant only if the candidate scores higher.

The grid in code: one store, id-only views

- The grid is not an array. Each island holds a dictionary {cell key $\rightarrow$ program id}; the key joins the bin indices: coordinates $[3, 7] \rightarrow$ `"3–7"`. Unoccupied cells are simply absent.
- Placing a program: each feature value (an evaluator metric, or `len(program.code)` for complexity) is min–max scaled by the running min/max observed so far, multiplied by the bin count, and clamped to a bin index.
- One central store `programs: {id` $\rightarrow$ `Program}` holds the objects; grids, island sets, and the archive hold only id strings.
- Fitness never mixes with features: the comparison score explicitly excludes the metrics used as feature axes (source comment: "excludes MAP-Elites feature dimensions").
- Three kinds of "best", kept separately: the per-cell champion; the **archive**, a global top-100 elite pool; and the absolute best program, tracked on its own — docstring: "to ensure it's never lost."

Choosing the parent: three tiers

Each iteration draws one random number $r \in [0, 1)$ and picks the parent accordingly (defaults from `config.py`):

- $r < 0.2$: a **random member** of the current island — exploration;
- $0.2 \leq r < 0.9$: an **elite from the archive** — exploitation;
- $r \geq 0.9$: a **fitness-weighted** island member.

The inspiration programs — the additional scored programs shown to the model in the prompt — are drawn only from the parent's own island.

Islands: independent grids, coupled only by migration

- Each island evolves independently and carries its **own** MAP-Elites grid (`island_feature_maps`: one dictionary per island).
- Prompts never show programs from other islands; migration is the only coupling between islands.
- Migration: at each migration event, the island's top fraction is **copied** to its two ring neighbours, islands $(i+1) \bmod N$ and $(i-1) \bmod N$.
- A copied program is marked `migrant` and is never migrated again. The guard records a real failure — source comment: "Program cb5d07f2 had 183 descendant copies by iteration 850" — identical copies all map to the same cell, so only one survives and the rest are wasted evaluations.

Evaluation cascade, operations, novelty judge

Evaluation cascade — *problem:* fully scoring a candidate may take minutes, and most candidates are bad.

- The user's evaluator may define stages `evaluate_stage1/2/3`.
- OpenEvolve detects them at startup and runs them in order; candidates failing an early stage skip the rest.
- Per-candidate timeout: 300 s.

Operations — single-machine pipeline; defaults sized for a workstation, not a datacenter.

- Async controller; process-based parallel workers.
- Checkpoint every 100 iterations; resume from any checkpoint.
- Deterministic seeding (default 42) — runs are repeatable.
- Browser-based visualizer of the program tree.

Beyond the AlphaEvolve paper. An optional LLM novelty judge (adapted from Sakana AI's ShinkaEvolve) rejects candidates whose behavior is not meaningfully different from existing programs.

OpenEvolve on circle packing, $n = 26$

Documented run: sum of radii **2.634292** (99.97% of AlphaEvolve's 2.635, matching but not beating). The discovered program:

```
def construct_packing():
    n, centers, count = 26, np.zeros((n, 2)), 0
    # (1) Layout: graded radii on a 5x5 grid, alternating column offset
    radii = np.linspace(0.12, 0.05, n)
    xs = ys = np.linspace(0.15, 0.85, 5)
    for i in range(5):
        for j in range(5):
            centers[count] = [xs[i] + 0.05*(j%2), ys[j]]; count += 1

    # (2) Refine with SLSQP: maximize sum of radii subject to
    #     non-overlap + boundary constraints
    x0 = np.concatenate([centers.flatten(), radii])
    result = minimize(objective, x0, method="SLSQP",
                      bounds=[(0,1)]*(2*n) + [(0.03, 0.2)]*n,
                      constraints={"type":"ineq", "fun": constraint})
    return result.x[:2*n].reshape(n,2), result.x[2*n:]
```

- The LLM contributes the *architecture* (graded-radii grid + SLSQP refinement) and good initial guesses; the optimal coordinates come from `scipy.optimize`.

Running OpenEvolve

A minimal run needs three files — an initial program with one evolve block, an evaluator returning a score dictionary, a config naming the model — and a served model:

```
pip install openevolve
python openevolve-run.py initial_program.py evaluator.py \
       --config config.yaml --iterations 50
```

- With a self-hosted open-weight model behind vLLM, the loop runs with no external API.
- Process isolation and timeouts are built in; a hardened security sandbox is not.
- The requirement of a cheap, exact evaluator is inherited, not removed.

Run: deciding Littlewood–Richardson positivity — the data

For partitions λ, μ, ν (weakly decreasing lists of positive integers), $c_{\mu\nu}^{\lambda}$ is a nonnegative integer, computable exactly by a counting rule — the Littlewood–Richardson (LR) rule. (It is the coefficient of s_λ in $s_\mu s_\nu$ for Schur polynomials: background.)

One data point = one triple with its exactly computed label:

λ	μ	ν	$c_{\mu\nu}^{\lambda}$	label
$(3,2,1)$	$(2,1)$	$(2,1)$	2	1
$(3,3)$	(2)	$(2,2)$	0	0

The second triple passes every cheap sanity check (sizes add up, μ and ν fit inside λ, …) and is still zero — a **hard zero**. Hard zeros are what the run is after.

Dataset: 700 training triples ($|\lambda|$ 6–12); 300 test triples, deliberately larger ($|\lambda|$ 13–16). Score of a predicate: accuracy on positives and on zeros, averaged, on the test triples.

The seed program; what one iteration does

The evolve block is one Python predicate. The seed: four cheap necessary conditions (abridged from the run's actual file):

```
def is_positive(lam, mu, nu):              # EVOLVE-BLOCK
    # contains(a, b): b[i] <= a[i] for every row i
    if sum(lam) != sum(mu) + sum(nu):            return False
    if not (contains(lam,mu) and contains(lam,nu)): return False
    if lam[0] > mu[0] + nu[0]:                   return False
    if len(lam) > len(mu) + len(nu):             return False
    return True
```

- Seed score: every positive right, every hard zero wrong — test score 0.667.
- One iteration: OpenEvolve puts the best programs so far, with their scores, into a prompt; Qwen-7B rewrites the block; the evaluator runs the new predicate on the 700 training triples and stores its score in the program database.
- 5,000 iterations.

The champion program after 5,000 iterations

Keeps the seed checks; adds two loops (abridged, as evolved):

```
for i in range(len(lam)):                # Weyl inequalities
    for j in range(i + 1):
        if j < len(mu) and i-j < len(nu):
            if lam[i] > mu[j] + nu[i-j]: return False
for k in range(len(lam)):                # dominance partial sums
    if sum(lam[:k]) > sum(mu[:k]) + sum(nu[:k]): return False
```

- These are the Weyl inequalities $\lambda_{i+j+1} \leq \mu_{i+1} + \nu_{j+1}$ and the dominance condition — the two simplest layers of the **Horn system** of inequalities characterizing positivity (Klyachko, Knutson–Tao).
- Test score $0.667 \to 0.814$; no true positive is ever rejected — every added condition is genuinely necessary. The deeper Horn layers were not found.
- Also evolved: blocks that check nothing — e.g. a "conjugate" step that re-sorts an already sorted tuple.
- An earlier run over-weighted hard zeros: a reject-most-positives predicate won. Rebalanced before this run.

Run: Naruse's skew hook-length formula — the target

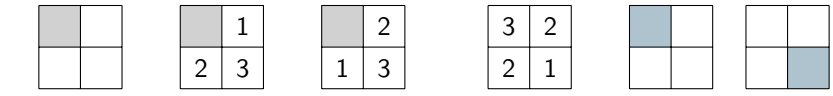
The **Young diagram** of λ has λ_i cells in row i; the skew shape λ/μ removes μ's cells from the top-left. $f^{\lambda/\mu}$ counts the **standard Young tableaux** of the shape: fillings with $1, \ldots, |\lambda/\mu|$, increasing along rows and down columns.

- Straight shapes: $f^{\lambda} = |\lambda|! / \prod_c h(c)$ (Frame–Robinson–Thrall 1954); the **hook length** $h(c)$ counts the cells to the right of c, plus the cells below c, plus 1.
- Skew shapes have no such product formula. Naruse (2014):

$$f^{\lambda/\mu} \;=\; |\lambda/\mu|! \sum_{D} \prod_{c \in [\lambda] \setminus D} \frac{1}{h_{\lambda}(c)},$$

summed over **excited diagrams** D: copies of μ's cell set pushed by repeated diagonal moves $(i, j) \to (i+1, j+1)$ inside λ.

The formula on one example: $\lambda = (2,2)$, $\mu = (1)$



λ/μ (μ shaded) | filling 1 | filling 2 | hooks $h_\lambda(c)$ | $D_0 = \mu$ | D_1

- The skew shape has 3 cells; exactly two standard fillings, so $f^{\lambda/\mu} = 2$.
- Excited diagrams of μ: the unmoved $D_0 = \{(1,1)\}$ and $D_1 = \{(2,2)\}$, its one diagonal move. The product skips D's cells:

$$f^{\lambda/\mu} = 3! \left(\frac{1}{2 \cdot 2 \cdot 1} + \frac{1}{3 \cdot 2 \cdot 2} \right) = 6 \left(\frac{1}{4} + \frac{1}{12} \right) = 2.$$

What evolves: the family generator

The shell is frozen in the evaluator: for each shape it computes, in exact arithmetic,

$$F \;=\; |\lambda/\mu|! \sum_{D \in \texttt{family}(\lambda,\mu)} \;\prod_{c \in [\lambda]\setminus D} \frac{1}{h_\lambda(c)},$$

and checks $F = f^{\lambda/\mu}$ (true value computed independently). Score: fraction of shapes matched exactly — 700 training, 300 larger test shapes. Only the family generator evolves; the seed returns μ unmoved:

```
def candidate_family(lam, mu):              # EVOLVE-BLOCK
    return [frozenset((i, j) for i in range(len(mu))
                             for j in range(mu[i]))]
```

- Seed: exact only when no diagonal move exists — test exact rate 0.15.
- The prompt never names excited diagrams. Probe before evolving: asked point-blank for a skew hook-length formula, the model gives a wrong one (a product formula attributed to "RSK", wrong hooks).

Output: three runs, the family is not found

- Diff mode, 600 iterations: 600/600 "No valid diffs found" (7B model); all later runs use full-rewrite mode.
- Full rewrite, high temperature, 5,000 iterations: best = seed; the children collapsed onto one brute-force subset-search program scoring below the seed.
- Full rewrite, low temperature, 5,000 iterations: all ∼5,000 children scored exactly the seed's 0.3128 — verbatim copies of the seed.

The excited-diagram family was neither recalled (the probe) nor discovered (the runs).

Run: circle packing $n = 26$ — task, inputs, output

Task: place 26 disjoint circles in the unit square, maximizing the sum of radii (the problem that opened this lecture).

- **Evaluator:** the exact check — every circle inside the square, every pair disjoint — returning the sum of radii; made deterministic by seeding the RNG.
- **Seed program:** the 5×5 grid of radius-0.1 circles plus one in a gap, sum 2.541.
- **What evolves:** the whole constructor `construct_packing() -> 26×(x,y,r)` (full rewrite each step).

Output: best deterministic sum **2.541** — equal to the grid seed. The 7B model produced no valid packing beating the grid, and about **97%** of its candidates were invalid (overlaps or out-of-bounds). Reaching AlphaEvolve's 2.6358 needs the graded-radius layout, which a 7B on one GPU did not find: the loop runs and scores correctly on open weights, but a small model does not reach the frontier record on this geometric problem.

The evaluator must be exact *and* deterministic

Caught while running OpenEvolve on the 26-circle problem (XMU HPC, June 2026). The evolved program placed the circles, then ran a local optimizer using an *unseeded* random number generator.

- The same program scored a radius sum of 2.56 on one evaluation and 2.47 on the next.
- A noisy score breaks selection: the search cannot tell a real improvement from luck, and it stalls.
- Fix: seed the random generator inside the evaluator, so every candidate is scored reproducibly.

The "cheap exact evaluator" precondition silently includes *deterministic*: a score that wobbles is not an exact score — the most practical lesson from running the method by hand.

Run: a representation-theory selector — task, inputs, output

Task: resolve the last unknown of a correspondence from our own research (painted bipartition $\to$ local system, type D) — at each column a 2-bit choice $(\text{sign}, \text{swap})$ picking which of four candidate values to emit.

- **Evaluator:** assemble the full local system from the candidate's per-column choices, exact-match against 28,384 oracle-verified pairs; headline metric the exact-match rate on held-out *larger* instances.
- **Seed program:** the trivial selector (always the canonical choice).
- **What evolves:** one function `select(rec) -> {column: (sign, swap)}`.

Output: best held-out exact match about **23%** (orbit ceiling 90%). The search **independently recovered the predicted program shape** — a state carried left to right across the columns (parities of cumulative symbol counts, an alternating counter) — but found no clean rule. A second independent method (after an earlier search over a hand-built grammar) reaching the same wall corroborates that the correspondence is irreducibly the recursion.

Part B.5 — PatternBoost: local search plus a small transformer

PatternBoost: local search plus a small transformer

Charton (Meta), Ellenberg (Wisconsin), Wagner (Worcester Polytechnic), Williamson (Sydney), *PatternBoost: constructions in mathematics with a little help from AI*, arXiv:2411.00566 (November 2024).

Input	a score function; an encoding of constructions as token sequences; a classical **local search** for the problem (start from a construction, repeatedly make small changes that raise the score, from many random starts).
Output	record or near-record constructions.
Objective	maximize the score, alternating two phases.

Geordie Williamson — representation theorist, University of Sydney; PatternBoost co-author. Photo: course asset.

The worked example: triangle-free graphs on 20 vertices

Maximize the number of edges of a 20-vertex graph containing no triangle. The answer is classical (Mantel 1907): $\lfloor 20^2/4 \rfloor = 100$, achieved by the complete bipartite graph $K_{10,10}$ — the method is demonstrated on known ground.

- **Encoding:** the upper-triangular part of the adjacency matrix, flattened row by row, rows separated by commas — a 3×3 matrix with rows $010,\ 01,\ 0$ becomes the token string `010,01,0`. For $n = 20$: 190 bits.
- **Local phase:** start from the empty graph; delete an edge from some triangle until none remains; then greedily add random edges that create no triangle.
- 40,000 local-search runs: best score 99 — "achieved quite rarely, by two lucky runs out of the 40,000 tries"; 100 is never reached.

One transformer round on the worked example

- Train a small transformer on the top 25% of the 40,000 scored graphs, encoded as token strings.
- Sample 37,000 new graphs from the trained model; run the same local search from each of them.
- Result: score 99 appears 47 times (twice in the previous round); the optimum 100 appears **46** times.
- After 5 more loops, "the model quickly learns to produce essentially only complete bipartite graphs" — the Mantel extremal structure.

The two phases

Stated aim of the paper: "a useful and simple tool for working mathematicians, one which does not require extensive expertise in machine learning or access to industry-level computing power."

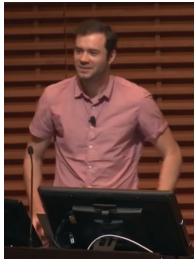
1. **Local phase.** Run classical local search (greedy plus repair moves) from many random seeds; collect a large pool of scored constructions.
2. **Global phase.** Train a small transformer on the best fraction of the pool (top 25% in the worked example), encoded as token sequences — e.g. a graph's upper-triangular adjacency matrix flattened row by row. Sample new sequences; use them as *seeds* for the next local phase. Repeat 10–50 times.

- The transformer never sees the score function; it learns the distribution of good constructions and proposes more from it.
- No reinforcement learning, no frontier model, no million-program budget.

Makemore: a minimal GPT-2-style transformer

Makemore — Andrej Karpathy's minimal decoder-only transformer, the block structure of GPT-2 (OpenAI's 2019 language model) at small size:

- each token gets a learned embedding vector plus a learned position embedding;
- 2–6 identical blocks; each = **masked self-attention** (every token attends only to *earlier* tokens; 4 heads, dimension 16–128) then a two-layer fully connected network, each with a residual connection (the input added back to the output);
- a final linear layer: at each position, a probability distribution over the vocabulary;
- training = next-token prediction on the best constructions; generation = sample token by token, decode into a candidate.



Andrej Karpathy — AI researcher; author of Makemore. Photo: Wikimedia Commons.

A 1992 hypercube conjecture refuted: 81 edges, not 82

Question (Erdős–Hamburger–Pippert–Weakley; Graham–Harary 1992): how many edges can be deleted from the d-dimensional hypercube without increasing its diameter (the largest graph distance between two vertices), which equals d?

- The classical construction keeps $2^d + \binom{d}{\lfloor d/2 \rfloor} - 2$ edges; Niall Graham conjectured in 1992 (Graham–Harary, *Discrete Applied Mathematics* 37) that this is optimal.
- For $d = 6$ that count is 82. PatternBoost found a spanning subgraph of the 6-cube (a subgraph keeping all 64 vertices) with diameter 6 and **81** edges.
- The paper: "the first progress on this problem in 30 years."

François Charton — PatternBoost co-author; research engineer at Meta AI at the time; since 2025 at Axiom Math. Photo: personal homepage.

No-isosceles sets; Brualdi–Cao permanents

The paper tests nine problems. Two more outcomes:

- **No-isosceles point sets** — largest subset of an $n \times n$ grid with no three points forming an isosceles triangle (degenerate ones included):
  - $n = 64$: three months of problem-specific classical search had reached 108 points; one transformer loop on top reached **110** within days.
  - $n = 100$: 154 $\rightarrow$ **160**.
- **Brualdi–Cao permanents** — the permanent of a matrix sums, over all ways to pick one entry per row and per column, the product of those entries (the determinant without the $\pm$ signs). Maximize it over a structured class of $n \times n$ 0–1 matrices (those avoiding a fixed forbidden sub-pattern): here a handcrafted specialized algorithm still wins, 5,200,384 vs 5,101,230 at $n = 25$. The paper: "handcrafted specialized algorithms can still outperform generic ML methods."

References

Primary sources: Wagner, AlphaTensor, AlphaDev

- A. Z. Wagner, *Constructions in combinatorics via neural networks*, arXiv:2104.14516 (2021).
- A. Fawzi et al., *Discovering faster matrix multiplication algorithms with reinforcement learning*, Nature 610 (2022) 47–53.
- M. Kauers, J. Moosbauer, arXiv:2210.04045 (2022); *Flip graphs for matrix multiplication*, ISSAC 2023.
- D. J. Mankowitz et al., *Faster sorting algorithms discovered using deep reinforcement learning*, Nature 618 (2023) 257–263.
- J. S. Ellenberg, D. Gijswijt, *On large subsets of* $\mathbb{F}_q^n$ *with no three-term arithmetic progression*, Annals of Mathematics 185 (2017) 339–343.

Primary sources: FunSearch, AlphaEvolve, PatternBoost

- B. Romera-Paredes et al., *Mathematical discoveries from program search with large language models*, Nature 625 (2024) 468–475.
- A. Novikov et al., *AlphaEvolve: a coding agent for scientific and algorithmic discovery*, Google DeepMind white paper, May 2025; arXiv:2506.13131.
- B. Georgiev, J. Gómez-Serrano, T. Tao, A. Z. Wagner, *Mathematical exploration and discovery at scale*, arXiv:2511.02864 (2025).
- F. Charton, J. Ellenberg, A. Z. Wagner, G. Williamson, *PatternBoost: constructions in mathematics with a little help from AI*, arXiv:2411.00566 (2024).
- Repositories (read 2026-06-11): `github.com/google-deepmind/alphatensor`, `google-deepmind/funsearch`, `google-deepmind/alphaevolve_results`, `algorithmicsuperintelligence/openevolve`.