# SYNTHESIZING PHYSICS FORMULAE WITH TRANSFORMERS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Finding a compact formula that fits a set of input-output pairs and predicts outputs on unseen inputs is a fundamental problem in science. Symbolic regression automates the search for such formulae: search-based methods explore the space of possible formulae directly, while transformers pre-trained on synthetic data produce formulae of comparable quality substantially faster. Existing transformers, however, are prone to overfitting — they find formulae that fit the training data well but do not extrapolate to input ranges unseen during training. We address this by shaping the set of formulae used to train a transformer, and show that the resulting formulae extrapolate substantially better. Fine-tuning the transformer on data with noise-corrupted target values further makes the synthesized formulae robust to noise in the observations. On SRBench and LLM-SRBench our transformer synthesizes a formula in about ten seconds and extrapolates better than all evaluated methods at a comparable budget. Search-based methods surpass our accuracy only when given one to three orders of magnitude more time.

## 1 INTRODUCTION

Symbolic regression seeks to produce a formula that fits a set of observations (Schmidt & Lipson, 2009; Lample & Charton, 2020). Genetic algorithms (GAs) search the space of formulae seeking one that fits the data the most. While GAs can find formulae with a high quality of fit (Udrescu & Tegmark, 2020) the process can be slow (La Cava et al., 2021). A recent line of work pre-trains transformers to produce symbolic formulae based on input observations. As real-life observations are often limited in volume, the transformers are trained on synthetic observations produced by symbolic formulae randomly drawn from a given grammar. This allows one to readily produce large and diverse sets of training data, each labeled with the generated formula. Once trained the transformers can produce symbolic formulae fitting test-time observations with the quality of fit comparable with state-of-the-art GA methods while doing so orders of magnitude faster (Biggio et al., 2021; Kamienny et al., 2022; Shojaee et al., 2023). A fast formula synthesis can be useful for practical applications with time constraints such as programmatic reinforcement learning (Kubalík et al., 2021; Derner et al., 2020). The downside of using existing pre-trained transformers is that they often produce complex symbolic formulae that fit the training data well but fail to extrapolate, limiting the use of the synthesized symbolic formulae in practice (Voigt et al., 2025).

We address this problem by simplifying each randomly drawn formula into an equivalent but more compact and consistently ordered form and using that form as the training target. We show that a transformer trained on such targets produces more compact symbolic formulae that extrapolate substantially better to unseen input ranges than existing formula-trained transformers. This allows the formulae produced by our transformers to serve as predictive models rather than mere data compressors (Schmidhuber, 2009; Seguí & Armour, 2026; Schulte, 2009).

Our empirical work compares our method with genetic-algorithm, formula-trained transformer and language-model methods on SRBench (La Cava et al., 2021) and LLM-SRBench (Shojaee et al., 2025b). Our transformer synthesizes a formula in about ten seconds, and at that cost it extrapolates better than every other method we evaluate that takes under a minute per formula. Search-based methods extrapolate better when given one to three orders of magnitude more time: PySR with a 30-minute budget on SRBench, and AI Feynman on noise-free SRBench, and PySR with a 30-minute budget and LLM-SR with Gemini-3.5-flash on LSR-Transform. On the LSR-Synth tasks, whose



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

inputs lie at physical scales far from those of our training data, our transformer trails search-based methods. Our code[1] and model weights[2] are available.

## 2 Problem Formulation

Consider a function $f : \mathbb{R}^m \to \mathbb{R}$. The problem is to synthesize a formula $\tilde{f} \in F$ which approximates $f$ over a finite set of inputs $D \subset \mathbb{R}^m$. The quality of approximation is measured via the coefficient of determination:

$$R^2(f, \tilde{f}, D) \quad = 1 - \frac{\sum_{\mathbf{x}_i \in D} \big(f(\mathbf{x}_i) - \tilde{f}(\mathbf{x}_i)\big)^2}{\sum_{\mathbf{x}_i \in D} \big(f(\mathbf{x}_i) - \bar{f}\big)^2}$$

where $\bar{f}$ is the mean value of $f$ over the inputs $\mathbf{x}_i \in D$. The input to the synthesizer is a training set $D_{\text{train}} = \{(\mathbf{x}_i, f(\mathbf{x}_i))\}_{i=1}^n$ with $f$ the target function. The performance of the synthesized approximation is measured on $D_{\text{test}}$ and the task is to find $f^*$ to maximize the coefficient of determination:

$$f^* \quad = \arg\max_{\tilde{f} \in F} R^2(f, \tilde{f}, D_{\text{test}}).$$

$D_{\text{test}}$ is assumed to be different from $D_{\text{train}}$ but the ground-truth $f$ should fit it exactly: $R^2(f, f, D_{\text{test}}) = 1$. We prefer synthesis to have low computational cost and to produce compact formulae.

To measure extrapolation we shift the input range of $D_{\text{train}}$ by $k$ times its width:

$$D^k_{\text{test}} \quad = \{(\mathbf{x}_i, f(\mathbf{x}_i)) \mid \mathbf{x}_i \sim \mathcal{U}(\boldsymbol{\ell} + k \cdot \boldsymbol{\delta}, \mathbf{r} + k \cdot \boldsymbol{\delta})\}_{i=1}^n$$

where $\boldsymbol{\ell} = \min_{\mathbf{x} \in D_{\text{train}}}(\mathbf{x})$, $\mathbf{r} = \max_{\mathbf{x} \in D_{\text{train}}}(\mathbf{x})$ element-wise and $\boldsymbol{\delta} = \mathbf{r} - \boldsymbol{\ell}$. Here $\mathcal{U}(\boldsymbol{\ell} + k \cdot \boldsymbol{\delta}, \mathbf{r} + k \cdot \boldsymbol{\delta})$ is a multivariate uniform distribution over the hyper-rectangle $[\boldsymbol{\ell} + k \cdot \boldsymbol{\delta}, \mathbf{r} + k \cdot \boldsymbol{\delta}]$. We define the degree-$k$ $R^2$ denoted by $R^2_k(f, \tilde{f})$ as the coefficient of determination evaluated on $D^k_{\text{test}}$.

We report, for each $k$, the fraction of tasks on which the synthesized formula attains a near-exact fit on $D^k_{\text{test}}$. Let $\mathcal{T}$ be a benchmark of synthesis tasks, with $f_t$ the target formula of task $t \in \mathcal{T}$ and $\tilde{f}_t$ synthesized for it, we define the degree-$k$ solution rate as

$$\rho_k(\mathcal{T}) \quad = \tfrac{1}{|\mathcal{T}|} \left| \left\{ t \in \mathcal{T} \mid R^2_k(f_t, \tilde{f}_t) \geq \tau \right\} \right|$$

where $\tau$ is for a near-exact match. We also report the mean synthesis time per task and the mean *size* of a synthesized formula, defined as the number of nodes in it.

## 3 Related Work

Symbolic regression methods applicable to the problem defined in the previous section fall into three categories. *Search-based symbolic regression* (Schmidt & Lipson, 2009; Cranmer et al., 2020; Petersen et al., 2021; Bulitko et al., 2022; Wang et al., 2023; Sun et al., 2023; Cranmer, 2023) uses genetic algorithms or reinforcement learning to search the space of formulae. Such methods do not require pre-training but are typically computationally expensive.

*Symbolic regression based on large language models* (Grayeli et al., 2024; Ma et al., 2024; Hua et al., 2025; Shojaee et al., 2025a) involves training on a large corpus of text in natural languages (and other data) which then allows it to use knowledge of physics to synthesize formulae. During synthesis the large language models can be run for inference repeatedly within a search-based algorithm, trading synthesis time for fit accuracy. However, the performance of these methods depends on the quality of the underlying language models.

Instead of training a general large language model, one can also train a smaller *transformer on the input-output pairs and the formulae only*, without any natural language data (Biggio et al., 2021; Valipour et al., 2022; Kamienny et al., 2022; Shojaee et al., 2023; d'Ascoli et al., 2024; Vastl et al., 2024; Ying et al., 2025). Doing so speeds up and simplifies the training but during

[1] uploaded as supplementary material

[2] https://osf.io/ca7h6/overview?view_only=1718f6af44a8442ca914f7dbf75682bf

synthesis such transformers tend to produce formulae that overfit the input data, exhibiting poor extrapolation (Voigt et al., 2025). To improve accuracy of synthesized formulae on inputs outside of the training domain, researcher have attempted to shape the distribution of the data on which the transformer is trained (Biggio et al., 2021; Bendinelli et al., 2023; Saegert & Köthe, 2026). Shaping usually rewrites each generated formula with SymPy (Meurer et al., 2017), a throughput bottleneck when formulae are generated online that Saegert & Köthe (2026) replace with offline-mined rewrite rules applied by pattern matching. We instead keep one formula per equivalence class, with no redundant constants for overfitting to the training range.

# 4 Proposed Approach

We train an encoder-decoder transformer such that given $N$ data points $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ where for each $i$, $f(\mathbf{x}_i) = y_i$ with $f : \mathbb{R}^m \to \mathbb{R}$ the target formula, the transformer receives $D$ and synthesizes $\tilde{f}$ token by token. Afterwards the constants are fitted to $D$.

## 4.1 Training Data Generation

We define a space of formulae $\mathcal{G}$ by the following context-free grammar (Udrescu et al., 2020):

$$
\begin{aligned}
S &\to T \mid U \mid B \\
T &\to v_1 \mid v_2 \mid v_3 \mid v_4 \mid v_5 \mid v_6 \mid v_7 \mid v_8 \mid v_9 \mid v_{10} \mid C \\
U &\to S{+}{+} \mid S{-}{-} \mid -S \mid \sqrt{S} \mid e^S \mid \ln(S) \mid \sin(S) \mid \cos(S) \mid \tan(S) \mid |S| \mid \arcsin(S) \mid \arccos(S) \\
&\quad \arctan(S) \mid \frac{1}{S} \mid \tanh(S) \mid S^2 \mid S^3 \\
B &\to S + S \mid S - S \mid S \times S \mid \frac{S}{S} \mid S^S
\end{aligned}
$$

where $C$ is a set of constants.

During training, each data instance is of the form $(f, D)$ where $f$ is a target formula from the space $\mathcal{G}$ in standard Polish notation (Biggio et al., 2021, a prefix notation where operators come before their operands) and $D$ is a set of data points from the space $\mathbb{R}^{N\times(v+1)}$ where $v$ is the number of variables in $f$ with the last column of $D$ the outputs of $f$ given the $v$ inputs. A formula can be seen as a tree with the first operator the root node and the variables and constants as leaves.

**Formulae generation.** To create a formula for training we sample the number of variables $v$ from a uniform distribution $\mathcal{U}(1, V_{\text{max}})$. We then sample the total number of nodes $n$ of the formula from $\mathcal{D}(l, r, \gamma)$, a distribution over $[l, r]$ with decay $\gamma \in (0, 1]$ that biases the training data towards shorter formulae. We then sample the number of unary operators $u$ from $\mathcal{D}(0, \min(8, \max(0, \lfloor (n - 1)/2 \rfloor)), \gamma)$ with the number of binary operators following $n$ and $u$.

We then generate a formula skeleton with $v$ variables, $n$ nodes and $u$ unary operators following the method of Lample & Charton (2020) as adapted by Kamienny et al. (2022). The distribution over the set of binary operators and unary operators are set by hand (Appendix A.1). Once a formula skeleton is generated we sample from a log-uniform distribution (Matsubara et al., 2024; d'Ascoli et al., 2024) for each constant $c = s \cdot 10^q$ with $s \sim \mathcal{U}\{-1, 1\}$ and $q \sim \mathcal{U}(-3, 3)$.

Existing formula-trained transformers place fittable constants at the additive positions of a skeleton (Kamienny et al., 2022): each term of a sum is multiplied by a constant, the argument $S$ of a unary operator is replaced by $a+b\cdot S$ (unless that argument is itself a sum) and the formula becomes $c_0 + f$. Products and powers receive no constants of their own, so $v_1 \times v_2$ is drawn as $c_0 + c_1 v_1 v_2$, while $\sin(v_1)$ is drawn as $c_0 + \sin(c_1 + c_2 v_1)$. Our generator does not apply it.

**Formula simplification.** Many different formulae are functionally equivalent. A full standardization is undecidable in general (Richardson, 1968), so we do not attempt it. In the spirit of Cohen (2003) we flatten each chain of connected additive operators: $S + S$, $S - S$, and the following unary operators $S{+}{+}$, $S{-}{-}$, $-S$, into a sum $c_0 + \sum_i c_i t_i$ with like terms collected. Multiplicative chains $S \times S$, $S/S$, $\frac{1}{S}$ and the powers $S^n$ with integer $n$ are flattened the same way into the product $c \prod_i b_i^{e_i}$ with like factors collected. We merge exponents only when they are integers, which is value-safe as $S^m S^n = S^{m+n}$ for integers $m, n$ whenever $S \neq 0$. We call a subformula an *atom* when the flattening treats it as a single indivisible term or factor and rewrites only its argument. We keep $\sqrt{S}$, $|S|$,

$S^k$ with non-integer $k$, and the transcendental unary operators $e^S$, $\ln(S)$, $\sin(S)$, $\cos(S)$, $\tan(S)$, $\arcsin(S)$, $\arccos(S)$, $\arctan(S)$ and $\tanh(S)$ as atoms, and flatten their arguments recursively. We also apply the cancellations $\ln e^S = S$, $-(-S) = S$, $||S|| = |S|$, $e^{\ln S} = S$ and $1/(1/S) = S$. We then sort the resulting sums and products into a fixed order and re-index the variables in ascending order, so that $v_3 + v_2 \cdot v_4 - v_1$ becomes $v_1 + v_2 \cdot v_3 - v_4$. Appendix A.2 gives the full ordering rules, the effect of the rewrites on the domain, and the pseudocode for the flattening.

**Data points generation.** Once a formula is simplified by the method above we sample its inputs as Kamienny et al. (2022) do, from a mixture of 1 to 10 Gaussian or uniform clusters with centers drawn from $\mathcal{N}(\mathbf{0}, I)$, a scale per variable drawn from $\mathcal{U}(0, 1)$ and a random rotation per cluster, and calculate $y_i = f(\mathbf{x}_i)$. We discard inputs at which $f$ is undefined or overflows ($|f(\mathbf{x}_i)| \geq 10^9$) and draw a fresh mixture for the missing points until $N$ valid data points remain. We whiten the inputs, $\tilde{\mathbf{x}}_i = (\mathbf{x}_i - \boldsymbol{\mu})/\boldsymbol{\sigma}$ with $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ the per-variable mean and standard deviation over the $N$ data points, so every input the transformer receives has zero mean and unit variance.

We whiten the inputs during both training and synthesis, but not the outputs. Unlike Kamienny et al. (2022), we compute each $y_i$ from the raw inputs $\mathbf{x}_i$, so the transformer learns $f$ on the original inputs.

### 4.2 Transformer Architecture and Training

We introduce our embedding method for the data points as well as our transformer's architecture. We then detail the training process.

**Tokenization and embedding.** We round each floating point number to four significant digits (Kamienny et al., 2022) and encode it as four tokens (Charton, 2022): the sign ($-$ or $+$), the higher mantissa (00 to 99), the lower mantissa (00 to 99), and the exponent ($-100$ to 100), a vocabulary of size $2 + 100 + 201 = 303$. Variables absent from the formula are padded with a dedicated token `<INPUT_PAD>`. We embed and flatten the tokens, then a two-layer ReLU multi-layer perceptron maps them to a $d_{\text{emb}}$-dimensional input of the transformer encoder.

**Transformer architecture.** We use an encoder-decoder transformer architecture (Vaswani et al., 2017; Phuong & Hutter, 2022). Since the order of data points does not matter to the synthesized formulae we remove the positional embedding from the encoder (Biggio et al., 2021; Kamienny et al., 2022). Starting from the 89 million-parameter model proposed by Kamienny et al. (2022), we enlarge our transformer to have 12 encoder layers and 20 decoder layers. We set an embedding dimension of 512 and 20 attention heads with 32-dimensional queries, keys and values, for a total of about 145 million parameters. To stabilize training we use the Pre-LayerNorm (Xiong et al., 2020) architecture and the GELU activation function (Hendrycks & Gimpel, 2016) in the transformer.

**Training.** We use the Adam (Ba & Kingma, 2015) optimizer to minimize the cross-entropy loss, increasing the learning rate linearly from $1.0 \times 10^{-7}$ to $4.0 \times 10^{-4}$ over the first 10,000 training steps and then lowering the learning rate by inverse-square-root down to $1.0 \times 10^{-4}$. We use 18.2k generated data instances (equivalent to 200 batches) as validation data and train the transformer with a patience of 10 epochs for early stopping. To avoid wasteful padding we batch together data instances with formulae of similar lengths. There are approximately 1.1M tokens passing through the encoder for each full batch. We train our transformer on a single NVIDIA H100 GPU with 80 GB memory. A single epoch takes about 30 minutes and the full training about two weeks. The values of all the control parameters of our data generation, architecture, training and synthesis are collected in Appendix A.10.

**Fine tuning for noisy targets.** We further train the transformer on noisy targets: for each formula we draw a noise level $\epsilon \sim \mathcal{U}(0, 0.1)$ in the convention of Section 5.2, and set $y \leftarrow y + \epsilon\sqrt{\overline{y^2}}\xi$ with $\xi \sim \mathcal{N}(0, 1)$, holding half the formulae noise-free so the noise-free objective is not forgotten. We resume from the trained transformer with a peak learning rate of $5 \times 10^{-5}$.

### 4.3 Inference Methods

We compare beam search (Wiseman & Rush, 2016) with two hybrid inference methods for our transformer: transformer-based planning for symbolic regression (TPSR) (Shojaee et al., 2023) and the divide-and-conquer strategy (D&C) (Ying et al., 2025).

**Whitened and raw variables.** Our transformer reads whitened inputs, so the formula it synthesizes is a function of $\tilde{\mathbf{x}}$. Following Kamienny et al. (2022) one can substitute $v_i$ with $(v_i - \mu_i)/\sigma_i$ to express it in the original variables, which makes the synthesis insensitive to the scale of the inputs. We do not do so by default. The substitution writes the location and scale of $D_{\text{train}}$ into the formula as constants, and those constants are wrong once the inputs move away from the range they were computed on. It also replaces every occurrence of a variable with a subexpression, inflating the size of the synthesized formula. We instead keep the formula as the transformer emits it and fit its constants against the raw data points with BFGS, and we report the substituted variant separately as *ours (unscaled)*.

**Bagging and refinement.** Since our transformer is trained on at most $N$ data points, whenever there are more than $N$ data points available for a task we draw $B$ bags of $N$ data points each, sampling uniformly with replacement so the bags are independent but not a partition of the available points. We use the synthesized constants as initial values and refine them *a posteriori* with the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm (Fletcher, 2013). We keep the top $C$ candidate formulae with beam search giving us in total $BC$ candidate formulae.

**Transformer-based planning.** Following Shojaee et al. (2023), the next-token distribution of our transformer serves as the prior of a Monte-Carlo tree search. Beam search completes each expansion into a formula $\hat{f}$, and the reward $1/(1 + \text{NMSE}) + \lambda e^{-|\hat{f}|/200}$ on $D_{\text{train}}$ is propagated back up the tree.

**Divide and conquer.** We run beam search on the original task first. If its formula reaches $R^2 \geq \theta$ on a held-out split of $D_{\text{train}}$ we keep it, otherwise we decompose. We fit a feed-forward network to $D_{\text{train}}$, one per task and from scratch, as a differentiable proxy for the unknown function (Appendix A.3). Following Ying et al. (2025) we read the structure of $f$ off its second derivatives, testing additive and multiplicative separability only. Two variables are additively separable when $\partial^2 f/\partial x_i \partial x_j$ vanishes and multiplicatively separable when $\partial^2 \ln|f|/\partial x_i \partial x_j$ vanishes. Since $\partial^2 \ln|f|/\partial x_i \partial x_j = H_{ij}/f - (\partial_i f)(\partial_j f)/f^2$, one gradient and Hessian of the network give both tests. We take the median of each over the sample points to form a coupling matrix, threshold it at several cutoffs, and group the variables by the connected components of the pairs that stay coupled. For each grouping we query the network for data points on every group, holding the variables outside it at their median. Our transformer synthesizes a subformula per group, and we combine them into one formula whose constants we refine with BFGS against $D_{\text{train}}$. We keep the beam-search formula unless a combined one scores higher on the held-out split of $D_{\text{train}}$.

## 5 Empirical Evaluation

We introduce the two benchmarks we use for evaluation in Section 5.1 and go through the control parameters in Section 5.2 and then show our evaluation results in Section 5.3.

### 5.1 Benchmarks

We evaluate the methods using two benchmarks: the SRBench (La Cava et al., 2021), from which we take the 99 Feynman formulae with ground-truth formulae available to compute the extrapolation set $D^k_{\text{test}}$ at every $k$, and the LSR-Transform category of the LLM-SRBench (Shojaee et al., 2025b) which provides 111 ground-truth formulae. Both benchmarks have no more than ten variables. We also evaluate our method on the LSR-Synth category of the LLM-SRBench (Shojaee et al., 2025b) which contains 129 formulae.

### 5.2 Control Parameters

For each task we construct $D^k_{\text{test}}$ as in Section 2, with $n = 10{,}000$ points, for $k \in \{0, 1, 2, 4, 8, 16, 32, 64, 128\}$. Shifting every variable by $k\boldsymbol{\delta}$ can create a domain of the target for-

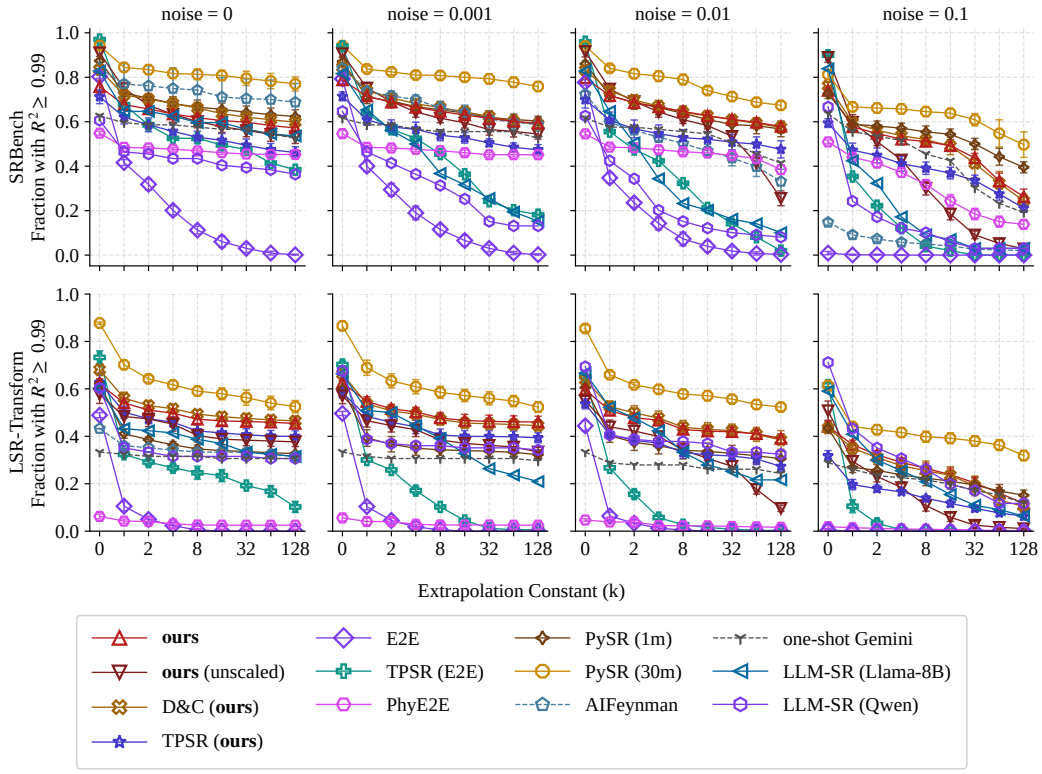



Figure 1: Solution rate $\rho_k$ versus extrapolation constant $k$ on the 99 SRBench formulae and on the 111 LSR-Transform formulae.

mula where it is undefined so we advance each variable as far as it can go and freeze the rest, keeping the formulae defined for every $k$ (Appendix A.7). At $k = 0$, $D^0_{\text{test}}$ is a sample over the range of $D_{\text{train}}$. We set the near-exact match threshold to $\tau = 0.99$ following existing research (Kamienny et al., 2022; Shojaee et al., 2023). To test robustness of the methods under noise we corrupt the target values of $D_{\text{train}}$ with additive Gaussian noise, $y \leftarrow y + \xi$ with $\xi \sim \mathcal{N}(0, \sigma^2)$ and $\sigma = \epsilon\sqrt{\overline{y^2}}$, where $\overline{y^2}$ is the mean squared target value over $D_{\text{train}}$, with the set of target noise levels $\epsilon \in \{0.0, 0.001, 0.01, 0.1\}$. The target values of every $D^k_{\text{test}}$ are left noise-free. We run each method with 10 random number-generator seeds and plot the means and the standard deviations over runs.

**Data budget.** For each task and run we draw a pool of data points uniformly at random from those the benchmark provides and split it $75/25$ into $D_{\text{train}}$ and a validation set, drawing the $B$ bags from $D_{\text{train}}$ as described in Section 4.3. On SRBench the pool holds $\lceil BN/0.75 \rceil = 53{,}334$ points, so $D_{\text{train}}$ holds $BN = 40{,}000$ points. On LLM-SRBench we use the training and test splits the benchmark provides and cap the pool at 20,000 points. We run beam search and TPSR fit and select on all of them, while the D&C strategy takes the same $75/25$ split. The validation set is used only to select among candidate formulae during synthesis.

### 5.3 Results

We compare our method with search-based methods, formula-trained transformers and LLMs, and read the comparison as a trade-off between the solution rate $\rho_k$ and the synthesis time.

**SRBench methods.** La Cava et al. (2021) ran 14 symbolic regression methods on the SRBench tasks and released the formula each method synthesized. We evaluate these released formulae on our $D^k_{\text{test}}$ and show the method with the highest $\rho_k$, AI Feynman (Udrescu et al., 2020).

**PySR.** We also evaluate PySR (Cranmer, 2023), a recent search-based method involving a genetic algorithm. We keep the operator set, the constraints on operator nesting and the population settings

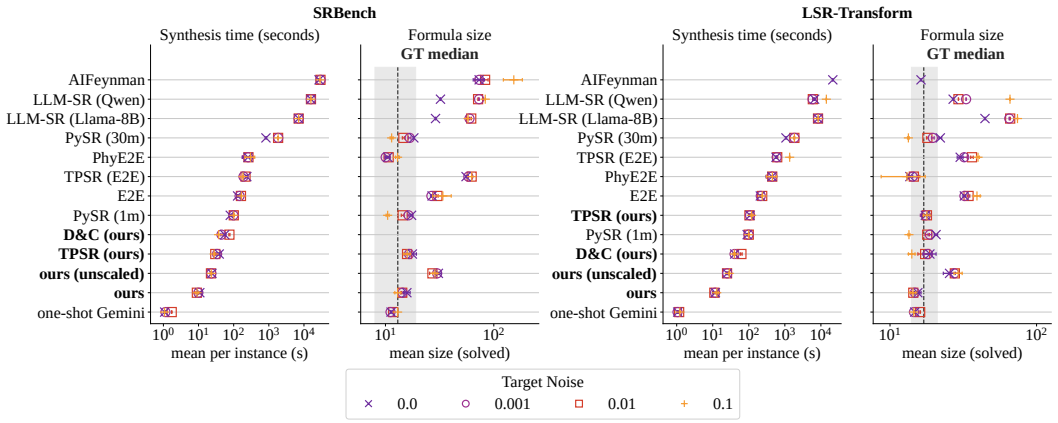



Figure 2: Average per formula synthesis time and average synthesized formula size on the 99 SRBench formulae and on the 111 LSR-Transform formulae. The median formula size and standard deviation of the ground-truth formulae are shown by the dashed vertical line and the gray region.

the SRBench specifies, with a largest formula size of 30 and a largest depth of 20. We run it at four time budgets per formula, 10 seconds, 1 minute, 30 minutes and 1 hour, and report them as *PySR (10s)*, *PySR (1m)*, *PySR (30m)* and *PySR (1h)*. Each fit runs on a single CPU core, and the measured time includes the start-up of PySR's Julia back end (about 40 seconds). The main figures show PySR (30m) only, since PySR (1h) differs from it by at most 0.02 in $\rho_k$.

**Formula-trained transformers.** We evaluate six formula-trained transformer methods: E2E (Kamienny et al., 2022), TPSR (Shojaee et al., 2023), PhyE2E (Ying et al., 2025), NeSymReS (Biggio et al., 2021), SymFormer (Vastl et al., 2024), and TF4SR (Lalande et al., 2023), and show the three best, E2E, TPSR and PhyE2E, here (all six in Appendix A.8). We use E2E with beam search. TPSR decodes with the pre-trained E2E model using Monte Carlo tree search guided by a reward that combines the quality of fit and the complexity of the formula. PhyE2E trains a similar architecture to E2E on LLM-generated data and applies D&C but we evaluated it using beam search.

**Large language models.** We evaluate three open-source models, Llama-3.1-8B-Instruct (Llama-8B), Llama-3.3-70B-Instruct (Llama-70b) Grattafiori et al. (2024) and Qwen-2.5-Coder-32B (Qwen) Hui et al. (2024), and one commercial model, Gemini-3.5-flash (Gemini), on the two benchmarks (Appendix A.9). We evaluate the LLMs in two ways: a one-shot method that prompts the LLM once for a formula skeleton and then applies BFGS to fit the constants to the training data (see Appendix A.4 for the prompt) and LLM-SR (Shojaee et al., 2025a) which uses the LLM to generate an initial population of candidate formulae and iteratively refine them based on loss. We set a budget limit of 1,000 LLM calls per LLM-SR run following previous work (Shojaee et al., 2025b). They are not restricted to the operators of $\mathcal{G}$ (Appendix A.5).

On SRBench the highest $\rho_0$ belongs to TPSR with the E2E model (0.97), but its $\rho_k$ falls to 0.38 at $k = 128$, whereas our transformer keeps $\rho_{128} = 0.57$, and 0.60 with D&C (Figure 1). Without noise three methods reach a higher $\rho_{128}$ than our transformer with or without D&C but each requires more synthesis time (Figure 2): PySR (1m) reaches $\rho_{128} = 0.62$ in 1.3 minutes per formula against our 11 seconds, AI Feynman 0.69 in 7.5 hours, and PySR (30m) 0.78 in 14 minutes. Under target noise AI Feynman's performance degrades ($\rho_{128} = 0.33$ at $\epsilon = 0.01$) while our transformer's does not (0.58, against 0.57 for PySR (1m) and 0.67 for PySR (30m)). LLM-SR with Llama-8B is comparable to our transformer without noise (0.54) at about 2 hours per task, but degrades under noise. Applying TPSR to our transformer makes $\rho_k$ worse than plain beam search even at $k = 0$, which we suspect is because its reward favors a close fit to $D_{\text{train}}$ over a compact formula, while applying D&C improves it only marginally, since it replaces the beam-search formula only on the tasks where that formula fails the held-out threshold $\theta$.

At target noise level of 0.1 the highest $\rho_0$ on SRBench is again TPSR with the E2E model (0.90), closely followed by *ours (unscaled)* (Section 4.3) (0.89) and LLM-SR with Llama-8B (0.84), but all three fall below 0.05 by $k = 128$ (Figure 1). For every $k \geq 2$ PySR (30m) has the highest $\rho_k$ (0.51 at $k = 128$), followed by PySR (1m) (0.39), against 0.26 for our transformer. One-shot Gemini

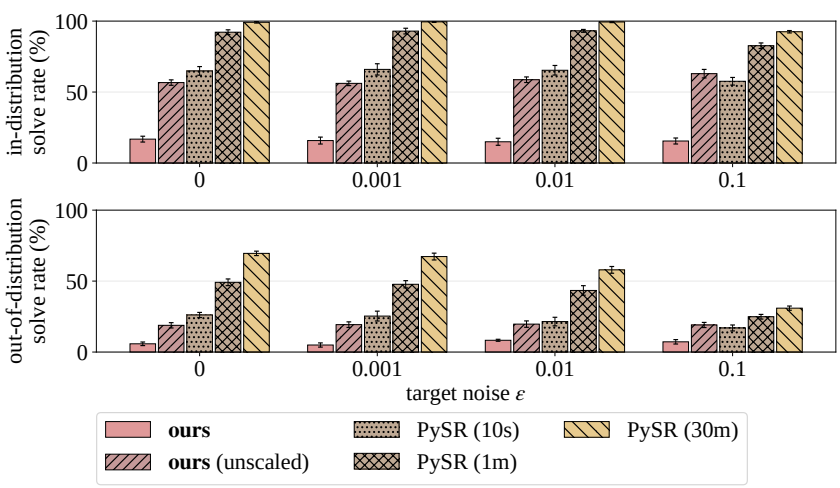



Figure 3: In-distribution and out-of-distribution solution rate on the 129 LSR-Synth formulae.

has a substantially higher solution rate than the two one-shot open-source models (Figure 8) and is relatively robust to target noise.

On LSR-Transform (Figure 1) PySR (30m) has the highest $\rho_k$ at every $k$ up to target noise $0.01$. Among the other methods TPSR with the E2E model has the highest $\rho_0$ ($0.73$) but falls to $0.10$ at $k = 128$. Up to target noise $0.01$ our transformer, with or without D&C, has the highest $\rho_k$ for every $k \geq 1$ among the methods in Figure 1 other than PySR (30m): $\rho_{128} = 0.45$ without noise and $0.39$ at $\epsilon = 0.01$, against $0.33$ and $0.31$ for PySR (1m). PySR (30m) extrapolates better ($0.54$ and $0.53$) at about $18$ minutes per task against our $12$ seconds (Figure 2). At target noise $0.1$ LLM-SR with Qwen has the highest $\rho_0$ ($0.71$), and at $k = 128$ PySR (30m) reaches $0.35$, PySR (1m) $0.15$ and our transformer $0.12$.

Due to API cost we ran LLM-SR with Gemini on LSR-Transform for a single run in the noise-free condition and compare it against the seed of each other method with the highest $\rho_0$ (Figure 5 in Appendix A.6). There PySR (30m) has the highest $\rho_0$ ($0.88$) and $\rho_1$, while LLM-SR with Gemini has the highest $\rho_k$ for every $k \geq 2$ ($0.61$ at $k = 128$, against $0.50$ for PySR (30m) and $0.47$ for our transformer), at about $2.2$ hours per formula against about $18$ minutes for PySR (30m) and $12$ seconds for ours.

For LSR-Synth we report the solution rate on the in-distribution and out-of-distribution test splits the benchmark provides (Figure 3). Without noise our transformer solves $17\%$ of the tasks in distribution and $6\%$ out of distribution, far below PySR (10s) ($65\%$ and $26\%$), PySR (1m) ($92\%$ and $49\%$) and PySR (30m) ($99\%$ and $70\%$). We attribute the gap to input range. Our raw training inputs lie in a narrow range around zero (Section 4.1), and those of SRBench and LSR-Transform lie mostly in $[1, 5]$. LSR-Synth inputs instead have physical ranges, such as temperatures in $[273, 543]$ in the material science domain. Reading the synthesized formula as a function of the whitened inputs, *ours (unscaled)*, brings them to our training range and raises our solution rate to $57\%$ in distribution and $19\%$ out of distribution. The largest gain is in the material science domain ($26\%$ to $84\%$), whose inputs are furthest from our training range, and the smallest in the physics domain ($9\%$ to $33\%$), whose inputs in $[-1.3, 1.3]$ are already close to it. Search-based methods nonetheless remain ahead on LSR-Synth. We suspect two reasons: they fit the raw data directly and so need no assumption about range, and they refine candidates against the data over many iterations, whereas our transformer must name the whole formula in one pass.

### 5.4 Ablation Study

Our transformer synthesizes a formula in about $11$ seconds and reaches $\rho_{128} = 0.57$ on SRBench, a higher solution rate out of distribution than any other method we evaluate that takes under a minute per task. This raises the question of which of our data-generation choices produce this gain, and whether either of them costs accuracy in distribution. We examine the two choices: dropping the

affine transformation that places constants at every additive position (Section 4.1), and simplifying each formula before training.

We study the contribution of the affine transformation and simplification within our approach by training four transformers, one for each combination of the two components being present or absent from our approach. Each of them uses the 89 million-parameter architecture of Kamienny et al. (2022) and is trained from scratch on 120 million generated training examples. We evaluate the checkpoint with the lowest validation loss on SRBench and LSR-Transform with 10 random seeds. On both benchmarks the transformer trained with simplification but without the affine transformation has the highest $\rho_k$ (Figure 4).

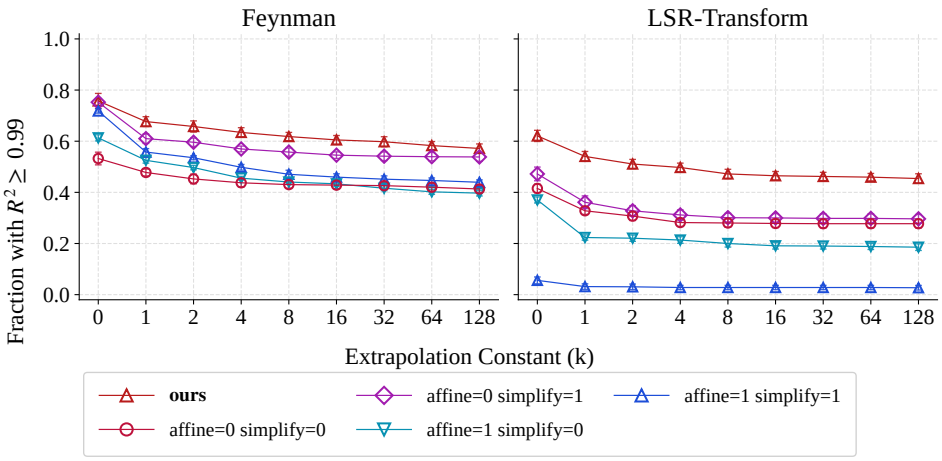



Figure 4: The ablation study results on affine transformation and simplification.

## 6 Future Work

Our transformer supports up to ten variables, while a symbolic regression task can involve hundreds, not all of them relevant. Future work will explore transformers accepting more variables and methods to identify the relevant ones. Our raw training inputs lie in a narrow range around zero, unlike the physical ranges of LSR-Synth inputs, such as temperatures in $[273, 543]$. Future work will draw the training input ranges at random, over several orders of magnitude, so that the transformer handles such inputs without the substitution of *ours (unscaled)*.

## 7 Conclusion

We proposed a way of shaping the training data of a transformer for symbolic regression on physics formulae: each training formula is simplified into a compact, consistently ordered form. On SRBench and LSR-Transform the resulting transformer synthesizes a formula in about ten seconds and extrapolates better than every other method evaluated that takes under a minute per formula. We attribute this performance to the fact that our training formulae possibly have a structure more similar to the actual physics formulae than used in previous studies. Search-based methods can extrapolate better but only when given one to three orders of magnitude more synthesis time. Additional training on noise-corrupted targets keeps our solution rate $\rho_k$ largely unchanged up to target noise $0.01$, where only PySR with a 30-minute or longer budget extrapolates better on both benchmarks.

### AI use statement

We used AI for code implementation, for drafting and condensing parts of the text, for checking typos and grammatical errors, for creating tables in LaTeX in the appendix, and for identifying related research. We have reviewed all AI contributions before inclusion in this paper and take responsibility for the full content of this paper.

### Reproducibility statement

Our source code is submitted as supplementary material. It contains the formula generator and simplification method, the training script, one evaluation entry point per method, and the scripts that construct the extrapolation sets. It also includes the script that downloads the LLM-SRBench data and the configuration files that define each LLM method. The weights of our six transformers (the transformer we evaluate, the same transformer before fine-tuning on noisy targets, and the four transformers of Section 5.4) are too large for the supplementary archive and are hosted at an anonymized link (`https://osf.io/ca7h6/overview?view_only=1718f6af44a8442ca914f7dbf75682bf`). The other methods are run from their authors' released implementations and, where available, their released weights, with the settings given in Section 5 and the Appendix.

## References


Jimmy Ba and Diederik P. Kingma. Adam: A method for stochastic optimization. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

Tommaso Bendinelli, Luca Biggio, and Pierre-Alexandre Kamienny. Controllable neural symbolic regression. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 2063–2077, 2023.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurélien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 936–945, 2021.

Vadim Bulitko, Shuwei Wang, Justin Stevens, and Levi HS Lelis. Portability and explainability of synthesized formula-based heuristics. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pp. 29–37, 2022.

François Charton. Linear algebra with transformers. *Transactions on Machine Learning Research (TMLR)*, 2022.

Joel S Cohen. *Computer algebra and symbolic computation: Mathematical methods*. CRC Press, 2003.

Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression. jl. *arXiv preprint arXiv:2305.01582*, 2023.

Miles Cranmer, Alvaro Sanchez Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 17429–17442, 2020.

Stéphane d'Ascoli, Sören Becker, Philippe Schwaller, Alexander Mathis, and Niki Kilbertus. ODEFormer: Symbolic regression of dynamical systems with transformers. In *Proceedings of International Conference on Learning Representations (ICLR)*, volume 2024, pp. 21943–21976, 2024.

Erik Derner, Jiří Kubalík, Nicola Ancona, and Robert Babuška. Constructing parsimonious analytic models for dynamic systems via symbolic regression. *Applied Soft Computing*, 94:106432, 2020.

Roger Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 44678–44709, 2024.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (GELUs). *arXiv preprint arXiv:1606.08415*, 2016.

Yingfan Hua, Ruikun Li, Jun Yao, Guohang Zhuang, Shixiang Tang, Bin Liu, Wanli Ouyang, and Yan Lu. Finetuning large language model as an effective symbolic regressor. *arXiv preprint arXiv:2508.09897*, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pp. 10269–10281, 2022.

Jiří Kubalík, Erik Derner, Jan Zegklitz, and Robert Babuška. Symbolic regression methods for reinforcement learning. *Institute of Electrical and Electronics Engineers (IEEE) Access*, 9:139697–139711, 2021.

William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. Contemporary symbolic regression methods and their relative performance. In *Proceedings of the Neural Information Processing Systems (NIPS) Track on Datasets and Benchmarks*, volume 1, 2021.

Florian Lalande, Yoshitomo Matsubara, Naoya Chiba, Tatsunori Taniai, Ryo Igarashi, and Yoshitaka Ushiku. A transformer model for symbolic regression towards scientific discovery. *arXiv preprint arXiv:2312.04070*, 2023.

Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of International Conference on Machine Learning (ICML)*, 2019.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B. Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. LLM and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. In *Proceedings of International Conference on Machine Learning (ICML)*, 2024.

Yoshitomo Matsubara, Naoya Chiba, Ryo Igarashi, and Yoshitaka Ushiku. Rethinking symbolic regression datasets and benchmarks for scientific discovery. *Journal of Data-centric Machine Learning Research (DMLR)*, 2024.

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

Brenden K. Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2021.

Mary Phuong and Marcus Hutter. Formal algorithms for transformers. *arXiv preprint arXiv:2207.09238*, 2022.

Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, 1968.

Paul Saegert and Ullrich Köthe. Breaking the simplification bottleneck in amortized neural symbolic regression. In *Proceedings of International Conference on Machine Learning (ICML)*, volume 306, 2026.

Jürgen Schmidhuber. Driven by compression progress: A simple principle explains essential aspects of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. In *Proceedings of the Workshop on Anticipatory Behavior in Adaptive Learning Systems (ABiALS)*, volume 5499, pp. 48–76, 2009.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324:81–85, 2009.

Oliver Schulte. Simultaneous discovery of conservation laws and hidden particles with smith matrix decomposition. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1481–1487, 2009.

Aleix Seguí and Wesley Armour. A geometric lens on physics-aligned data compression. In *Proceedings of International Conference on Machine Learning (ICML)*, volume 306, 2026.

Parshin Shojaee, Kazem Meidani, Amir Barati Farimani, and Chandan K. Reddy. Transformer-based planning for symbolic regression. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2025a.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D. Doan, and Chandan K. Reddy. LLM-SRBench: A new benchmark for scientific equation discovery with large language models. In *Proceedings of International Conference on Machine Learning (ICML)*, 2025b.

Fangzheng Sun, Yang Liu, Jian-Xun Wang, and Hao Sun. Symbolic physics learner: Discovering governing equations via monte carlo tree search. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2023.

Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 4860–4871, 2020.

Mojtaba Valipour, Bowen You, Maysum H. Panju, and Ali Ghodsi. SymbolicGPT: A generative transformer model for symbolic regression. In *Proceedings of the Workshop on Efficient Natural Language and Speech Processing (ENLSP)*, 2022.

Martin Vastl, Jonáš Kulhánek, Jiří Kubalík, Erik Derner, and Robert Babuška. Symformer: End-to-end symbolic regression using transformer-based architecture. *IEEE Access*, 12:37840–37849, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

Henrik Voigt, Paul Kahlmeyer, Kai Lawonn, Michael Habeck, and Joachim Giesen. Analyzing generalization in pre-trained symbolic regression. *arXiv preprint arXiv:2509.19849*, 2025.

Shuwei Wang, Vadim Bulitko, Taoan Huang, Sven Koenig, and Roni Stern. Synthesizing priority planning formulae for multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pp. 360–369, 2023.

Sam Wiseman and Alexander M Rush. Sequence-to-sequence learning as beam-search optimization. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1296–1306, 2016.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *Proceedings of International Conference on Machine Learning (ICML)*, pp. 10524–10533, 2020.

Jie Ying, Haowei Lin, Chao Yue, Yajie Chen, Chao Xiao, Quanqi Shi, Yitao Liang, Shing-Tung Yau, Yuan Zhou, and Jianzhu Ma. A neural symbolic model for space physics. *Nature Machine Intelligence*, 7:1726–1741, 2025.

# A Appendix

## A.1 Operator sampling weights

Table 1 lists the unnormalized sampling weight of each operator. We set these weights by hand, before any evaluation, and did not tune them on the benchmarks. They encode a domain-general ordering: arithmetic first, then squares, roots and reciprocals, then the core transcendental operators, and finally the rarer inverse trigonometric ones. When labeling a binary node of the skeleton we draw a binary operator with probability equal to its weight divided by the sum of the binary weights (3.5), and when labeling a unary node we draw a unary operator with probability equal to its weight divided by the sum of the unary weights (6.35). For example, + is drawn with probability $1/3.5 \approx 0.286$ at a binary node and `sqr` with probability $1/6.35 \approx 0.157$ at a unary node.

Table 1: Unnormalized sampling weight of each operator. The probability of drawing an operator is its weight divided by the sum of the weights in its group (binary or unary). `pow2` is identical to `sqr` and is kept only for compatibility with the token vocabulary.

| *Binary* | | | *Unary (continued)* | | |
|---|---|---|---|---|---|
| $S + S$ | + | 1 | $S --$ | `--` | 0.35 |
| $S \times S$ | * | 1 | $\lvert S \rvert$ | `abs` | 0.2 |
| $S - S$ | - | 0.8 | $S^3$ | `pow3` | 0.2 |
| $S/S$ | / | 0.6 | $\tan(S)$ | `tan` | 0.12 |
| $S^S$ | `pow` | 0.1 | $\arcsin(S)$ | `arcsin` | 0.1 |
| *Unary* | | | $\arctan(S)$ | `arctan` | 0.1 |
| $S^2$ | `sqr` | 1 | $\tanh(S)$ | `tanh` | 0.1 |
| $\sqrt{S}$ | `sqrt` | 0.7 | $\arccos(S)$ | `arccos` | 0.08 |
| $-S$ | `neg` | 0.6 | $S^2$ | `pow2` | 0.05 |
| $1/S$ | `invert` | 0.6 | | | |
| $\sin(S)$ | `sin` | 0.5 | | | |
| $\cos(S)$ | `cos` | 0.5 | | | |
| $e^S$ | `exp` | 0.4 | | | |
| $\ln(S)$ | `ln` | 0.4 | | | |
| $S ++$ | `++` | 0.35 | | | |

## A.2 Simplification method

The pseudocode for the simplification method is shown in Algorithm 1, with its subroutines in Algorithms 2, 3 and 4. The cancellations $\ln e^S = S$, $-(-S) = S$, $||S|| = |S|$, $e^{\ln S} = S$ and $1/(1/S) = S$ do not change the value of a formula at any input at which the original is defined. Of these, $\ln e^S = S$, $e^{\ln S} = S$ and $||S|| = |S|$ are applied as rewrites when a unary operator is normalized (NormFunc of Algorithm 1), while $-(-S) = S$ and $1/(1/S) = S$ need no rule of their own: the flattening negates every coefficient of a sum for each $-S$ and every exponent of a product for each $1/S$, and doing either twice is the identity. Some of the cancellations make a formula defined at additional inputs: merging exponents can cancel a factor against its reciprocal, as in $S \cdot \frac{1}{S} \to 1$, and $e^{\ln S} = S$ removes the requirement that $S$ be positive. The inputs affected are those at which a cancelled subformula is zero or leaves the domain of a cancelled operator, and they do not affect the training data: we generate the data points from the simplified formula rather than from the original draw, and discard any input at which it does not evaluate to a finite value.

Sums and products are sorted to place variables first in order of their index and all other subformulae in lexicographical order, so $v_{10} + v_2$ becomes $v_2 + v_{10}$, and every variable precedes every non-variable subformula, so $\cos(v_1) + v_3$ becomes $v_3 + \cos(v_1)$. Sorting applies within groups: when a sum is written back out its positive terms come before its negative ones, and a product its coefficient and numerator before its denominator, so $v_2 - v_1$ and $v_3/v_1$ are already in simplified form.

**Algorithm 1** SIMPLIFY

**function** SIMPLIFY($s$) ▷ $s$ is a given formula in standard Polish notation
  Parse $s$ into a formula tree $T = (\mathrm{op}, \mathrm{kids})$
  $I \leftarrow \mathrm{NORM}(T)$ ▷ Apply sum-of-products normalization to form an internal representation $I$
  **return** standard Polish notation converted from the internal representation $I$
**end function**

**function** NORM($T = (\mathrm{op}, \mathrm{kids})$)
  **if** op is a leaf **then** ▷ zero-arity: nothing to collect
    **return** $v_i$ for a variable, or $c$ for a numeric constant
  **else if** op is additive **then** ▷ $+, -, ++, --$, neg
    $(c_0, \{c_i, t_i\}) \leftarrow \mathrm{COLLECTSUM}(T)$
    **return** $\mathrm{MAKESUM}(c_0, \{t_i \mapsto c_i\})$ ▷ drops zero coefficients and sorts the terms
  **else if** op is multiplicative **then** ▷ $\times$, /, invert, sqr, pow2, pow3
    $(c, \{b_i \mapsto e_i\}) \leftarrow \mathrm{COLLECTPROD}(T)$
    **return** $\mathrm{MAKEPROD}(c, \{b_i \mapsto e_i\})$ ▷ drops zero exponents and sorts the factors
  **else if** op is pow **then**
    **return** $\mathrm{NORMPOW}(\mathrm{kids}[0], \mathrm{kids}[1])$
  **else** ▷ $\sqrt{\cdot}$, $|\cdot|$, and the transcendental unary operators
    $A \leftarrow \mathrm{NORM}(\mathrm{kids}[0])$
    **return** $\mathrm{NORMFUNC}(\mathrm{op}, A)$
  **end if**
**end function**

**function** NORMPOW(base, expNode) ▷ $\mathrm{base}^{\mathrm{expNode}}$; only integer exponents are merged
  $E \leftarrow \mathrm{NORM}(\mathrm{expNode})$
  **if** $E$ is a constant $n$ **then**
    **if** $n = 0$ **then**
      **return** $1$
    **else if** $n \in \mathbb{Z}$ **then**
      $(\mathrm{c}, \mathrm{f}) \leftarrow \mathrm{COLLECTPROD}(\mathrm{base})$
      **if** $n < 0$ and $\mathrm{c} = 0$ **then**
        **abort** ▷ $0^n$ with $n < 0$ is undefined; the formula is left unchanged
      **end if**
      **return** $\mathrm{MAKEPROD}(\mathrm{c}^n, \{b \mapsto e \times n \mid (b, e) \in \mathrm{f}\})$
    **else if** $n = 1/2$ **then**
      $A \leftarrow \mathrm{NORM}(\mathrm{base})$
      **return** $\mathrm{NORMFUNC}(\sqrt{\cdot}, A)$
    **end if**
  **end if**
  **return** the opaque node $(\mathrm{NORM}(\mathrm{base}))^E$ ▷ symbolic or non-integer exponent
**end function**

**function** NORMFUNC(op, $A$) ▷ $A$ is the already normalized argument
  **if** op is $e^{\cdot}$ and $A$ is $\ln(A')$ **then**
    **return** $A'$ ▷ $e^{\ln S} = S$
  **else if** op is ln and $A$ is $e^{A'}$ **then**
    **return** $A'$ ▷ $\ln e^{S} = S$
  **else if** op is $|\cdot|$ **then**
    **if** $A$ is a constant $c$ **then**
      **return** $|c|$
    **else if** $A$ is $|A'|$ **then**
      **return** $A$ ▷ $||S|| = |S|$
    **end if**
  **end if**
  **return** the opaque node $\mathrm{op}(A)$
**end function**

**Algorithm 2** COLLECTSUM

**function** COLLECTSUM($T = (\mathrm{op}, \mathrm{kids})$) ▷ returns $(c_0, \{t_i \mapsto c_i\})$ with value $c_0 + \sum_i c_i t_i$
  **if** op is $+$ **then**
    $(\mathrm{cL}, \mathrm{tL}) \leftarrow$ COLLECTSUM(kids[0])
    $(\mathrm{cR}, \mathrm{tR}) \leftarrow$ COLLECTSUM(kids[1])
    **return** $(\mathrm{cL} + \mathrm{cR}, \mathrm{MERGE}(\mathrm{tL}, \mathrm{tR}))$ ▷ a shared base has its coefficients added
  **else if** op is $-$ **then**
    $(\mathrm{cL}, \mathrm{tL}) \leftarrow$ COLLECTSUM(kids[0])
    $(\mathrm{cR}, \mathrm{tR}) \leftarrow$ COLLECTSUM(kids[1])
    $\mathrm{tR}' \leftarrow \mathrm{NEG}(\mathrm{tR})$ ▷ subtracting negates every coefficient
    **return** $(\mathrm{cL} - \mathrm{cR}, \mathrm{MERGE}(\mathrm{tL}, \mathrm{tR}'))$
  **else if** op is $++$ **then**
    $(\mathrm{c}, \mathrm{t}) \leftarrow$ COLLECTSUM(kids[0])
    **return** $(\mathrm{c} + 1, \mathrm{t})$
  **else if** op is $--$ **then**
    $(\mathrm{c}, \mathrm{t}) \leftarrow$ COLLECTSUM(kids[0])
    **return** $(\mathrm{c} - 1, \mathrm{t})$
  **else if** op is neg **then**
    $(\mathrm{c}, \mathrm{t}) \leftarrow$ COLLECTSUM(kids[0])
    **return** $(-\mathrm{c}, \mathrm{NEG}(\mathrm{t}))$
  **end if**
  $I \leftarrow \mathrm{NORM}(T)$ ▷ $T$ is not additive: normalize it and contribute it as a single term
  **if** $I$ is a sum $(c_0, \{(c_i, t_i)\}_i)$ **then**
    **return** $(c_0, \{c_i, t_i\}_i)$
  **end if**
  $(s, b) \leftarrow$ SPLITSCALAR($I$)
  **if** $b = \emptyset$ **then**
    **return** $(s, \emptyset)$ ▷ $I$ is a pure constant
  **end if**
  **return** $(0, \{b \mapsto s\})$
**end function**

### A.3 ORACLE NETWORK CONFIGURATION

The divide-and-conquer strategy of Section 4.3 uses a feed-forward neural network as a differentiable proxy for the unknown function. It is queried for its second-order derivatives, which determine how the variables are grouped, and for the data points of each group once that grouping is fixed. Table 2 gives its configuration.

**Algorithm 3** COLLECTPROD

**function** COLLECTPROD($T = (\text{op}, \text{kids})$) ▷ returns $(c, \{b_i \mapsto e_i\})$ with value $c \prod_i b_i^{e_i}$
  **if** op is $\times$ **then**
    $(\text{cL}, \text{fL}) \leftarrow$ COLLECTPROD(kids[0])
    $(\text{cR}, \text{fR}) \leftarrow$ COLLECTPROD(kids[1])
    **return** $(\text{cL} \times \text{cR}, \text{MERGE}(\text{fL}, \text{fR}))$ ▷ a shared base has its exponents added
  **else if** op is $/$ **then**
    $(\text{cL}, \text{fL}) \leftarrow$ COLLECTPROD(kids[0])
    $(\text{cR}, \text{fR}) \leftarrow$ COLLECTPROD(kids[1])
    **if** $\text{cR} = 0$ **then**
      **abort** ▷ division by zero is undefined; the formula is left unchanged
    **end if**
    $\text{fR}' \leftarrow \text{NEG}(\text{fR})$ ▷ dividing subtracts the divisor's exponents
    **return** $(\text{cL}/\text{cR}, \text{MERGE}(\text{fL}, \text{fR}'))$
  **else if** op is invert **then**
    $(\text{c}, \text{f}) \leftarrow$ COLLECTPROD(kids[0])
    **if** $\text{c} = 0$ **then**
      **abort** ▷ $1/0$ is undefined; the formula is left unchanged
    **end if**
    **return** $(1/\text{c}, \text{NEG}(\text{f}))$ ▷ the coefficient is reciprocated, every exponent negated
  **else if** op is in $\{\text{sqr}, \text{pow2}\}$ **then**
    $(\text{c}, \text{f}) \leftarrow$ COLLECTPROD(kids[0])
    **return** $(\text{c}^2, \{b \mapsto e \times 2 \mid (b, e) \in \text{f}\})$
  **else if** op is pow3 **then**
    $(\text{c}, \text{f}) \leftarrow$ COLLECTPROD(kids[0])
    **return** $(\text{c}^3, \{b \mapsto e \times 3 \mid (b, e) \in \text{f}\})$
  **end if**
  $I \leftarrow \text{NORM}(T)$ ▷ $T$ is not multiplicative: normalize it and contribute it as a single factor
  $(s, b) \leftarrow$ SPLITSCALAR($I$)
  **if** $b = \emptyset$ **then**
    **return** $(s, \emptyset)$ ▷ $I$ is a pure constant
  **else if** $b$ is a product $(c', \text{f}')$ **then**
    **return** $(s \times c', \text{f}')$ ▷ absorb the inner product
  **end if**
  **return** $(s, \{b \mapsto 1\})$
**end function**

**Algorithm 4** HELPER ROUTINES

**function** MERGE($m_1, m_2$) ▷ union of the keys; the values are added on a shared key
  **return** $\{k \mapsto m_1[k] + m_2[k] \mid k \in \text{keys}(m_1) \cup \text{keys}(m_2)\}$ ▷ an absent key contributes 0
**end function**

**function** NEG($m$) ▷ negate every value: subtracts terms in a sum, inverts factors in a product
  **return** $\{k \mapsto -m[k] \mid k \in \text{keys}(m)\}$
**end function**

**function** SPLITSCALAR($I$) ▷ writes $I = s \cdot b$ with a rational $s$ and a monic base $b$
  **if** $I$ is a rational constant $c$ **then**
    **return** $(c, \emptyset)$ ▷ $I$ is a pure constant; there is no base
  **else if** $I$ is a product $(c, \text{f})$ **then**
    **return** $(c, \text{MAKEPROD}(1, \text{f}))$ ▷ strip the coefficient
  **else if** $I$ is a sum $(0, \{(c, b)\})$ **then**
    **return** $(c, b)$ ▷ a lone $c \cdot b$ term
  **end if**
  **return** $(1, I)$
**end function**

Table 2: Configuration of the oracle network used by the divide-and-conquer strategy.

| Parameter | Value |
|---|---|
| *Architecture* | |
| hidden layers | $128, 128, 64, 64$ |
| hidden activation | $\tanh$ |
| output layer | linear, one unit |
| weights | $128m + 29{,}121$ for $m$ variables |
| *Input and target scaling* | |
| inputs | whitened per variable |
| target | standardized |
| *Training* | |
| loss | root mean squared error |
| optimizer | Adam, $\beta = (0.9, 0.999)$ |
| initial learning rate | $1.0 \times 10^{-2}$ |
| learning rate decay | $\div 10$ after 20 epochs without improvement |
| stopping criterion | learning rate below $1.0 \times 10^{-5}$ |
| epoch cap | 3,000 |
| batch size | 256 |
| *Data* | |
| data points | $\leq$ 20,000, drawn from $D_{\text{train}}$ only |
| validation split | 20% |
| derivative sample | 1,000 data points |

Both separability tests of Section 4.3 are evaluated at 1,000 points drawn from $D_{\text{train}}$ rather than from a box over the input range, since the network is only trustworthy where it saw data. Each pair of variables is scored by a dimensionless quantity so that a single threshold serves every task. For the additive test the score is the median of $|H_{ij}|\,|f|\,/\,(|\partial_i f|\,|\partial_j f|)$ over those points, which is 0 for a separable pair and of order 1 otherwise. For the multiplicative test, writing $t_1 = f H_{ij}$ and $t_2 = (\partial_i f)(\partial_j f)$, the numerator $t_1 - t_2$ of $\partial^2 \ln|f| / \partial x_i \partial x_j$ is a difference of two nearly equal terms exactly when the pair separates, so we score the relative cancellation, the median of $|t_1 - t_2|\,/\,(|t_1| + |t_2|)$, which is again 0 when the pair separates. We score the relative cancellation rather than the magnitude of the second derivative itself because that magnitude carries the error of the network's derivatives at the same size as a genuinely coupled pair, leaving no signal to threshold. Points at which $|f|$ is near zero are excluded from the multiplicative test alone, as $\ln|f|$ is undefined there, and we attempt no multiplicative decomposition when fewer than 8 points remain. A grouping of the variables is the set of connected components of the graph of the pairs scoring above a threshold. We enumerate the groupings that a ladder of thresholds produces, discard those from which the network cannot reconstruct its own outputs before any beam search is spent on them, and keep the one whose combined formula attains the highest $R^2$ on the held-out split of Section 4.3.

### A.4 One-shot LLM prompt

The one-shot LLM method issues one chat completion per task, with the fixed system prompt below and the user prompt after that. Angle brackets mark the per-task substitutions, and the fitting referred to in the prompt is done with BFGS.

```
You are an exceptional symbolic regression assistant. You analyse numerical
relationships between variables and infer the closed-form mathematical formula
that generated them. You answer with a formula and nothing else.
```

```
Find the mathematical formula for <output description> (<output>) as a function
of the following input variables:
  <variable 1>: <variable 1 description>
  ...
  <variable m>: <variable m description>

Below are 30 observed samples (columns: <variable 1>, ..., <variable m>,
<output>):
```

```
  <30 rows of numbers, one per sample, drawn from the training split>

Infer the formula that maps the inputs to the output. It must hold for all the
data above, and it must be physically/mathematically plausible, not a fit of
convenience.

Rules for your answer:
  - Use only these variables: <variable 1>, ..., <variable m>.
  - Use only these operations and functions: +, -, *, /, ** (power), exp, log,
    sqrt, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, abs, and the
    constant pi.
  - Write every unknown numeric coefficient as a placeholder symbol c0, c1, c2,
    ... (at most 10). Their values are fitted to the data afterwards by a
    numerical optimizer, so do NOT guess numbers: give the STRUCTURE, e.g.
    c0*x1*exp(-c1*x2) + c2.
    Structural exponents may stay literal (x1**2, sqrt(x1)).
  - Every placeholder must be multiplied/added into the formula somewhere it can
    matter; do not write a placeholder you do not need.
  - Use Python/SymPy infix syntax on a single line, e.g. c0*x1*exp(-c1*x2) + c2.
  - Output ONLY the right-hand side of the expression for <output>. No
    explanation, no units, no code fences, no '=' sign.
```

Listing 1: The one-shot user prompt template.

### A.5 Operator sets available to each method

The methods we evaluate are not given the same set of operators, so we state here what each one may use.

Our transformer emits only the tokens of the grammar $\mathcal{G}$ of Section 4.1. No operator outside $\mathcal{G}$ can appear at any point of the decoding.

The one-shot method instructs the model to use $+$, $-$, $\times$, $/$, exponentiation, `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `abs` and the constant $\pi$ (Listing 1). That list covers $\mathcal{G}$ — the unary operators $S{+}{+}$, $S{-}{-}$, $1/S$, $S^2$ and $S^3$ are expressible through $+$, $/$ and exponentiation — and adds $\sinh$ and $\cosh$, which $\mathcal{G}$ does not contain. The parser that reads a reply admits more than the prompt offers: it also accepts `asinh`, `acosh`, `atanh`, `cbrt`, `log10`, `atan2`, `sign` and the constant $e$. We admit those because the models write the same function in several spellings and we would otherwise discard a reply over its notation. A reply is rejected when it applies a function outside that set, introduces a variable the task does not have, carries more than the 10 constants we fit, or evaluates to a non-finite or complex value.

The LLM-SR algorithm states no set of operators. It prompts for the body of a Python function with the numerical library NumPy in scope and evolves that program (Shojaee et al., 2025a), so any function of that library and any control flow of the language may appear in a candidate, and a candidate need not be a closed-form formula. A candidate is kept when it returns a finite loss with at most the 10 constants we fit. We run the algorithm as its authors released it rather than restricting it to $\mathcal{G}$, so that our measurements of it remain comparable with the published ones.

Each of the two LLM methods therefore has at least the operators of $\mathcal{G}$ available to it, and LLM-SR is not confined to closed-form formulae. Any effect this has on the comparison favors them over our transformer.

### A.6 Best-seed comparison on LSR-Transform

Because of API cost we ran LLM-SR with Gemini on LSR-Transform for a single noise-free seed. Figure 5 compares that run against the seed of each other method with the highest $\rho_0$, showing $\rho_k$ on the left and synthesis time and formula size on the right. Section 5.3 discusses the results.

### A.7 Construction of the extrapolation input ranges

Shifting the whole input box by $k\boldsymbol{\delta}$ can leave the domain of the target formula, which would drop the task from the comparison at that $k$. We therefore choose a per-variable shift. At each $k$ we first try shifting all variables together; if the target formula does not evaluate to finite values there we advance the variables one at a time, in the order the benchmark lists them, keeping each advance

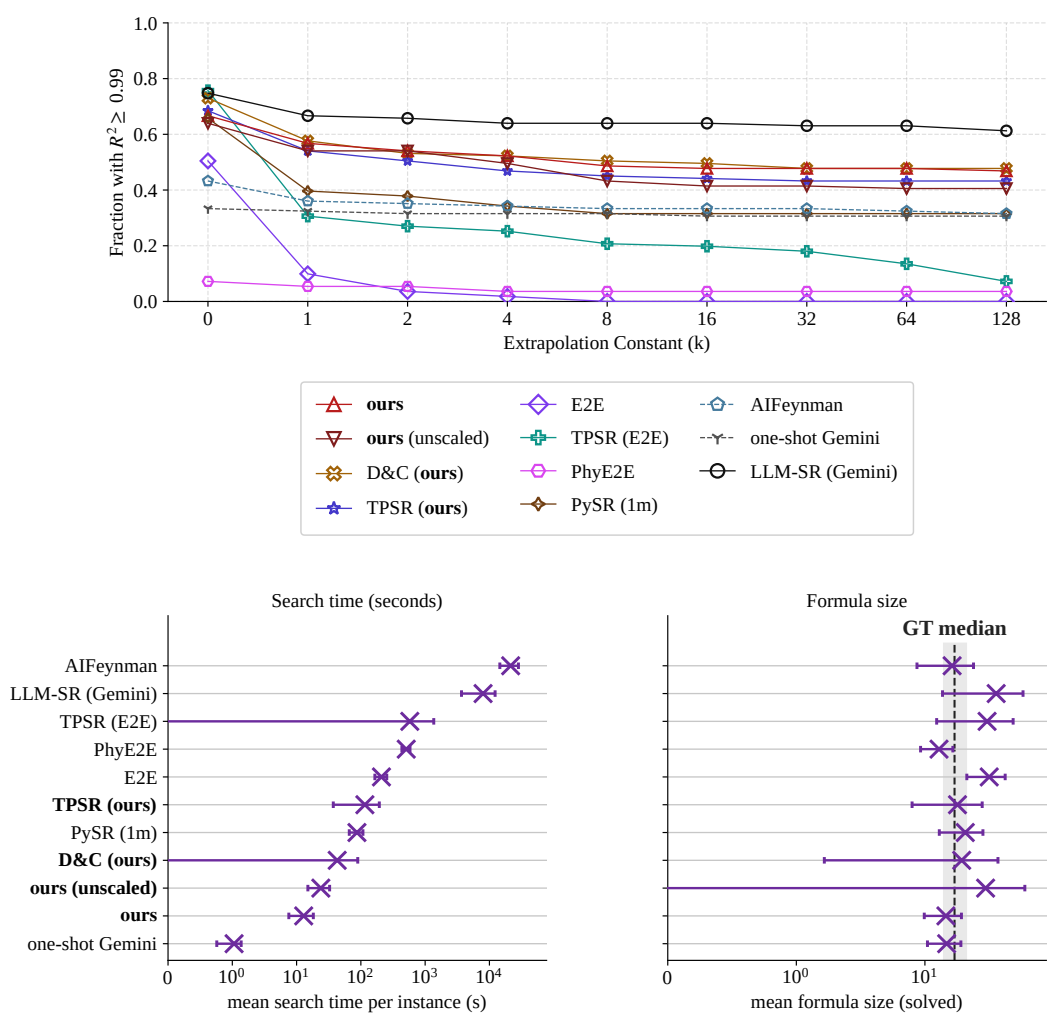



Figure 5: Left: solution rate $\rho_k$ versus extrapolation constant $k$ on the 111 LSR-Transform formulae. Right: mean per-formula synthesis time and mean synthesized formula size on the same formulae, with the median size and $\pm 1$ standard deviation of the ground-truth formulae shown by the dashed vertical line and the gray region. The LLM-SR with Gemini curve is compared against the single best seed of each other method in the noise-free setting.

whose outputs stay finite and leaving the remaining variables at their previous shift. A variable that cannot advance in one direction is offered the other, and the direction is settled in a first pass at the largest $k$ so that a variable does not reverse partway along the $k$ axis. The per-variable shifts are non-decreasing in $k$ by construction, so a task never moves back toward its training range as $k$ grows. This retains all 99 SRBench and all 111 LSR-Transform tasks at every $k$.

### A.8 Formula-trained transformers

NeSymReS encodes the data points with a set transformer (Lee et al., 2019) and decodes formula skeletons whose constants are placeholders, which we fit with BFGS for each skeleton in its beam. Its released model accepts at most three input variables. SymFormer decodes each symbol together with the value of its constant and then refines the constants by gradient descent. Its released models accept at most two input variables. TF4SR decodes a formula skeleton whose constants we fit with BFGS. Its released model accepts at most six input variables, all of them positive. Figures 6 and 7 show all six methods.

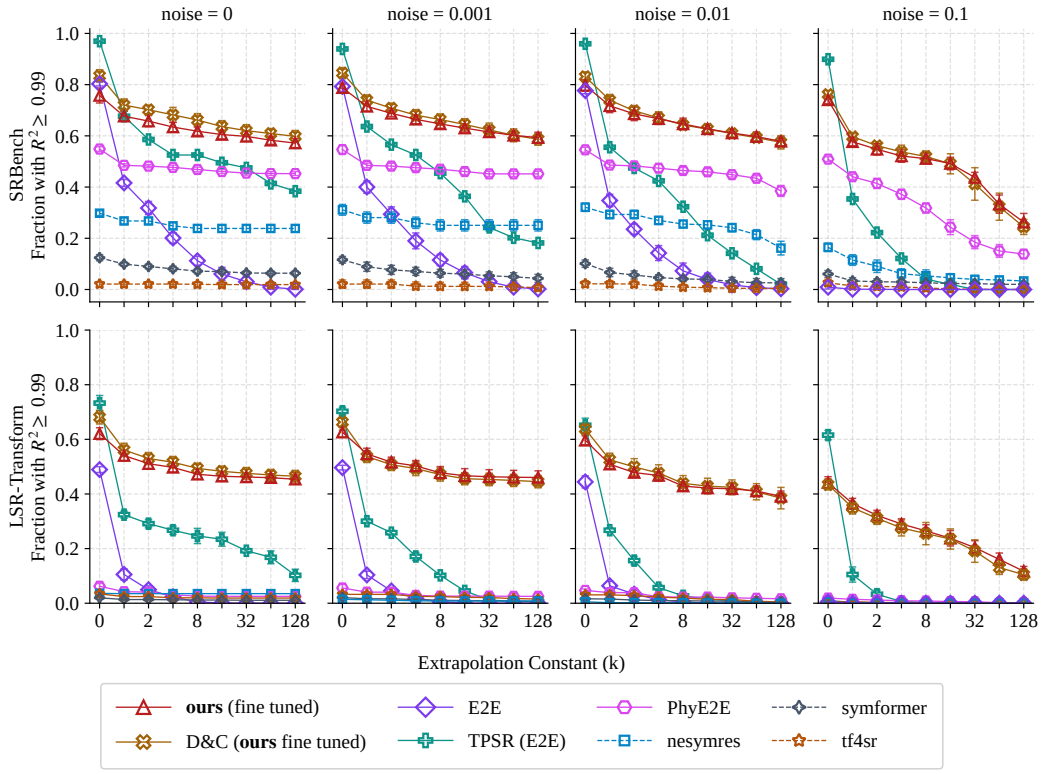



Figure 6: Solution rate $\rho_k$ versus extrapolation constant $k$ on the 99 SRBench formulae and the 111 LSR-Transform formulae, with every formula-trained arm drawn.

### A.9 Language-trained transformers

The body figures draw only one-shot Gemini among the LLMs, to keep the panels legible. Figures 8 and 9 redraw the same two benchmarks with every LLM arm we evaluated: one-shot Gemini, one-shot Llama-8B, one-shot Llama-70B, and LLM-SR with Llama-8B and with Qwen. All other arms and all control parameters are as in Section 5.2.

### A.10 Control parameter values

Table 3 gives the value of every control parameter that the body of the paper names, in the order in which the body introduces them.

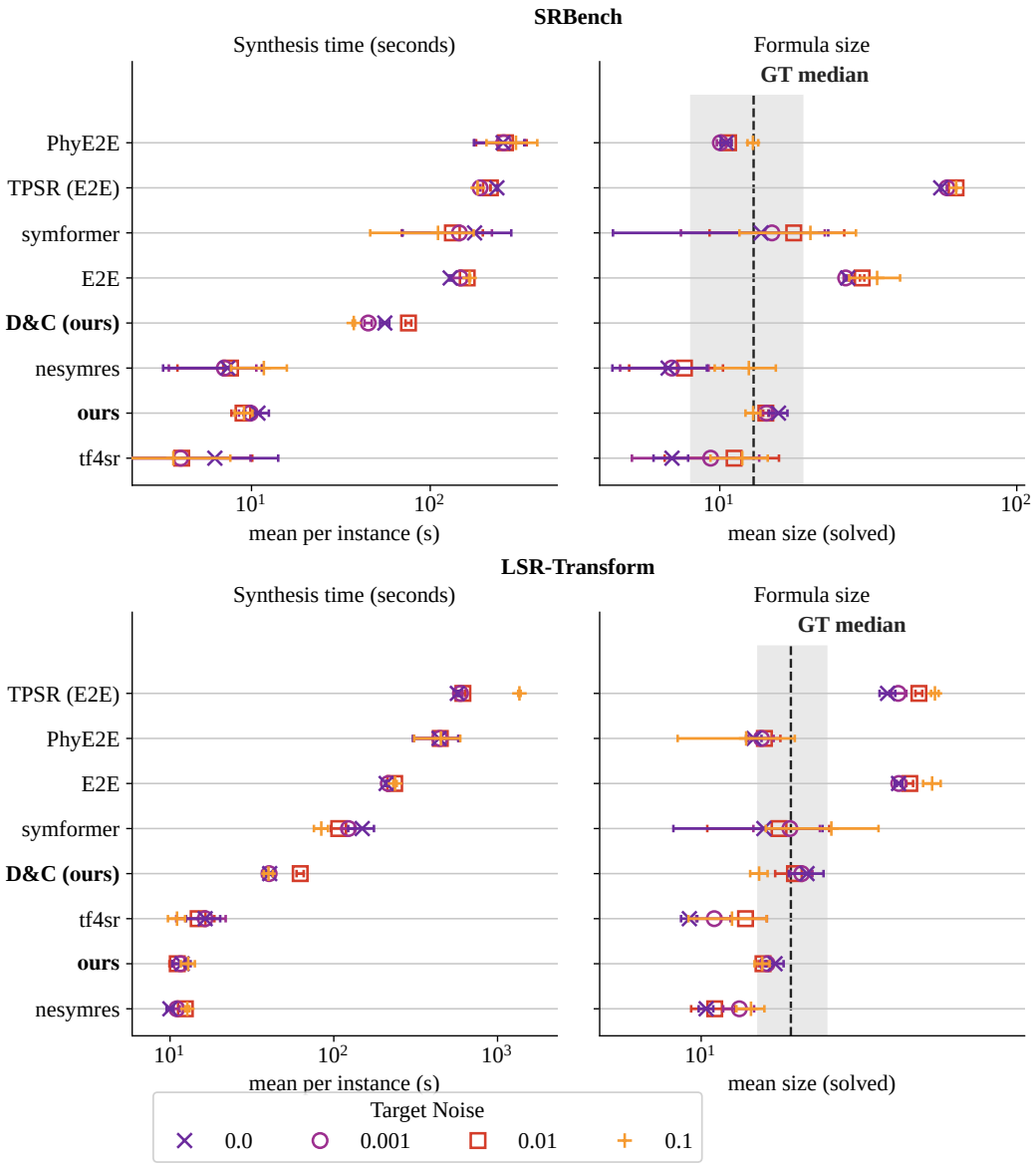



Figure 7: Time per formula and predicted formula complexity for SRBench and LSR-Transform.

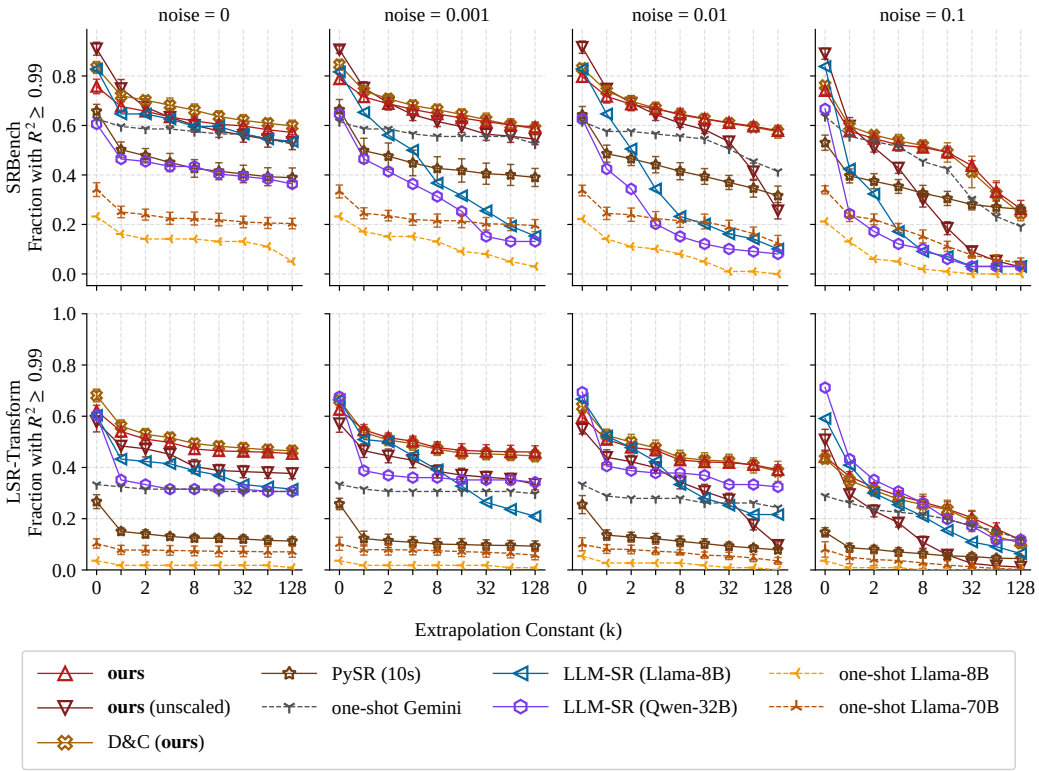



Figure 8: Solution rate $\rho_k$ versus extrapolation constant $k$ on the 99 SRBench formulae and the 111 LSR-Transform formulae, with every LLM arm drawn.

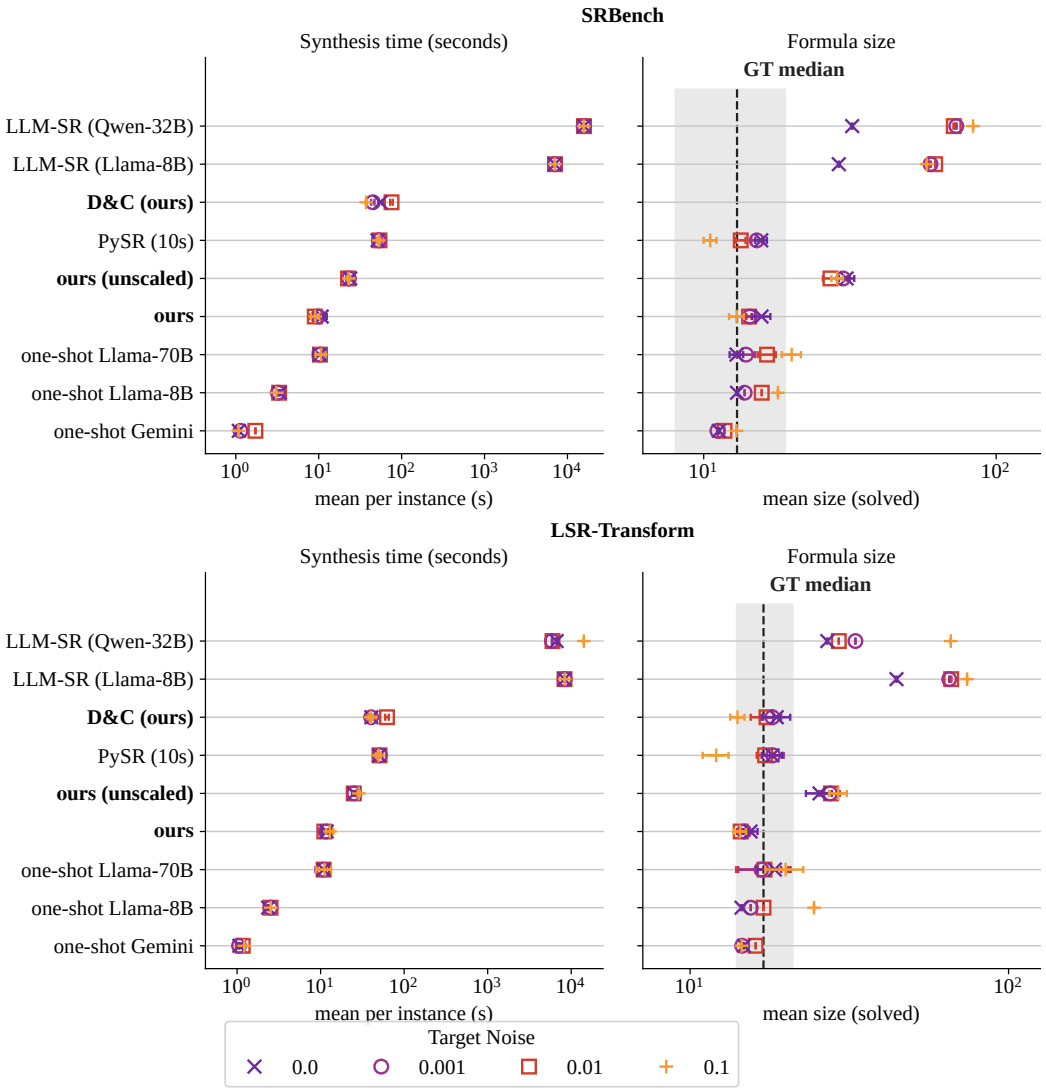



Figure 9: Time per formula and predicted formula complexity for SRBench and LSR-Transform.

Table 3: The control parameters from the main text of the paper and their values.

| Symbol | Value | Meaning |
|---|---|---|
| *Training data generation (Section 4.1)* | | |
| $V_{\text{max}}$ | 10 | largest number of variables in a formula |
| $l$ | 1 | smallest number of nodes in a formula |
| $r$ | 40 | largest number of nodes in a formula |
| $\gamma$ | 0.99 | decay of the formula size distribution |
| $N$ | $\mathcal{U}\{50, \ldots, 400\}$ | data points per training instance |
| *Embedding (Section 4.2)* | | |
| $d_{\text{emb}}$ | 512 | dimension of the data embedding |
| $d_{\text{head}}$ | 32 | query, key and value dimension per attention head |
| *Formula synthesis (Section 4.3)* | | |
| $N$ | 400 | data points per bag |
| $B$ | 100 | bags per task |
| $C$ | 10 | candidate formulae kept per bag |
| $\theta$ | 0.9999 | early-stop threshold on beam-search $R^2$ |
| *Empirical evaluation (Section 5.2)* | | |
| $\tau$ | 0.99 | threshold for a near-exact match |
| $k$ | $\{0, 1, 2, 4, 8, 16, 32, 64, 128\}$ | input-shift constants |
| $n$ | 10,000 | points per extrapolation data set |
| $\epsilon$ | $\{0, 0.001, 0.01, 0.1\}$ | target noise levels |
| $P$ | 53,334 / 20,000 | points pooled per task and seed, SRBench / LLM-SRBench |
| *Further training (Section 4.2)* | | |
| $\epsilon_{\text{max}}$ | 0.1 | largest training-time target noise level |
| $p_{\text{clean}}$ | 0.5 | share of formulae kept noise-free |
| $\eta_{\text{ft}}$ | $5 \times 10^{-5}$ | peak learning rate |
| $W_{\text{ft}}$ | 500 | warmup steps |