

Gutenberg School of Management and Economics

& Research Unit “Interdisciplinary Public Policy”

Discussion Paper Series

TSGP-2: On the Non-Linearity of Symbolic Regression Problems and How to Exploit It with Model-based Semantic Genetic Programming

Philipp Anthes, Alina Geiger, Martin Briesch, Franz Rothlauf

13th November 2026

Discussion paper number 2608

All papers listed in this series reflect the views of the authors only and do not necessarily represent those of the University of Mainz and/or the research unit “Interdisciplinary Public Policy”. The authors are solely responsible for the content. Working papers in this series are not peer reviewed. They present preliminary research results and are circulated to encourage discussion and comment.

Johannes Gutenberg University Mainz
Gutenberg School of Management and Economics
Jakob-Welder-Weg 9
55128 Mainz
Germany
https://wiwi.uni-mainz.de/

Contact details

Philipp Anthes
Department of Information Systems and Business Administration
Johannes Gutenberg University
Jakob-Welder-Weg 9
55128 Mainz
Germany
anthes@uni-mainz.de

Alina Geiger
Information Systems and Business Administration
Johannes Gutenberg University
Jakob-Welder-Weg 9
55128 Mainz
Germany
geiger@uni-mainz.de

Martin Briesch
Department of Information Systems and Business Administration
Johannes Gutenberg University
Jakob-Welder-Weg 9
55128 Mainz
Germany
briesch@uni-mainz.de

Franz Rothlauf
Department of Information Systems and Business Administration
Johannes Gutenberg University
Jakob-Welder-Weg 9
55128 Mainz
Germany
rothlauf@uni-mainz.de

All discussion papers can be downloaded from http://wiwi.uni-mainz.de/DP

TSGP-2: On the Non-Linearity of Symbolic Regression Problems and How to Exploit It with Model-based Semantic Genetic Programming

PHILIPP ANTHES, Johannes Gutenberg University, Germany
ALINA GEIGER, Johannes Gutenberg University, Germany
MARTIN BRIESCH, Johannes Gutenberg University, Germany
FRANZ ROTHLAUF, Johannes Gutenberg University, Germany

Model-based variation operators such as the one used in Transformer Semantic Genetic Programming (TSGP) are affected by possible biases of their training data. TSGP is trained, in particular, on programs generated by Genetic Programming (GP) runs on linear target functions, which leads to a bias toward linear offspring. This bias can hinder search performance on problems that require non-linear solutions. This work has two goals: first, it investigates the linearity of standard test problems such as the Feynman and real-world benchmark problems. While most of the structure of real-world problems can be well approximated by linear structures, the Feynman problems exhibit varying levels of non-linear structure. Thus, a proper variation operator of a search method should be able to create both types of programs, linear as well as non-linear ones. Motivated by this observation, we propose TSGP-2, which follows the functionality of TSGP but uses as training data programs from GP runs on both linear and non-linear target functions. As a result, the training data includes a lower share of programs that behave linearly (the share drops from 83% to 37%). With a higher share of non-linear structures, TSGP-2 outperforms TSGP on problems with non-linear structure, while performing similarly on linear problems. In addition, TSGP-2 is less biased toward linear offspring and is able to propose a higher number of meaningful non-linear offspring earlier during search, which coincides with better solutions. The findings underscore the importance of training data for a learned variation operator and emphasize that the biases of search operators should be aligned with the problems they are applied to.

CCS Concepts: • **Computing methodologies** → **Genetic programming**; *Neural networks.*

Additional Key Words and Phrases: genetic programming, symbolic regression, learned variation operator, transformer, training data, linearity

1 Introduction

The goal of symbolic regression (SR) is to find a mathematical function whose behavior, its *semantics*, fits a given target behavior as closely as possible. In genetic programming (GP) these functions are evolved in an evolutionary search process through selection and variation. In particular, the variation operators have a major influence on which behavior is found and how quickly the search converges to a good solution [22, 33, 35].

Standard operators such as subtree mutation and subtree crossover [13, 25] randomly vary parts of the function. They thereby ignore how a variation affects a function's semantics [22]. Consequently, variation often produces offspring that are either semantically identical to their parents or substantially different from them, both of which harm search efficiency [4, 19, 32]. Semantic GP addresses this with variation operators that control the semantic similarity between offspring and parents [35]. Some approaches such as Geometric Semantic GP (GSGP) [22] and SLIM_GSGP [34] work with pre-defined syntactic rules that define how a parent should be varied to obtain an offspring that is semantically similar.

Instead of relying on fixed rules, the recently proposed Transformer Semantic GP (TSGP) [2] uses a trained machine learning model as a semantic variation operator. This model is trained using

Authors' Contact Information: Philipp Anthes, anthes@uni-mainz.de, Johannes Gutenberg University, Mainz, Germany; Alina Geiger, Johannes Gutenberg University, Mainz, Germany; Martin Briesch, Johannes Gutenberg University, Mainz, Germany; Franz Rothlauf, Johannes Gutenberg University, Mainz, Germany.

pairs of semantically similar functions to learn how a syntactic change affects the semantics of a function. TSGP is able to produce meaningful offspring, effectively guiding the search process. As a result, it finds better solutions for a wide range of SR problems compared to standard GP, SLIM_GSGP, GSGP, and other model-based approaches [1, 3]. An open and unsolved question is why the performance improvements of TSGP are not consistent across all problems [3].

Recent work on pretrained SR models has shown that learned models are strongly biased toward their training distribution [29, 42]. In TSGP, the model is trained on programs generated during GP runs that search for linear target functions [3]. It is still unclear whether and how the choice of training data affects the offspring proposed by the model, and, ultimately, search performance. Understanding these effects is crucial if the model is to be applied to a wide range of problems with different characteristics. While a linear function might already explain most of the target behavior for some problems, others have shapes that a linear function cannot sufficiently approximate. Instead, non-linear functions are required to solve these problems [31]. In case the variation operator of TSGP is indeed biased toward linear offspring, this would harm search performance on non-linear problems.

Therefore, we analyze in this work the problem structure of synthetic as well as real-world benchmark problems regarding the linearity of their behavior. Motivated by these findings, we generate training data based on both linear and non-linear target functions. With this training data, we train a novel model-based variation operator which we call TSGP-2. We compare the search performance of TSGP-2 with TSGP and standard GP across a wide range of synthetic and real-world problems and trace the differences back to the search behavior and the variation operators.

We find that while synthetic Feynman problems exhibit clear non-linear structures, most of the analyzed real-world problems can largely be explained with linear approximations. Only four of the 13 real-world problems exhibit clear non-linear structures. Furthermore, we confirm that the training data of TSGP consists mainly of programs that behave linearly (83%), compared to TSGP-2 (37%).

With this less linear training data, TSGP-2 achieves the best mean rank on both the Feynman problems (1.19, compared to 1.88 for TSGP and 2.94 for standard GP) and the real-world problems (1.46, compared to 1.92 and 2.62). On the Feynman problems, it significantly outperforms TSGP on 7 of the 16 problems, while TSGP is never significantly better than TSGP-2. This gap in performance between TSGP and TSGP-2 is evident on problems with clear non-linear structures, whereas on problems without them, both reach comparable solution quality.

Our analysis traces this advantage back to the training data. On the Feynman problems, the best programs of TSGP-2 leave the linear region earlier than those of TSGP (generation 11 vs. 16), after which its error advantage over TSGP grows. During the search, TSGP-2 proposes non-linear offspring earlier and more often, and selection keeps them, whereas TSGP keeps proposing linear offspring that selection discards. Standard GP proposes the most strongly non-linear offspring, but selection rarely keeps them due to their low quality. Thus, the learned TSGP and TSGP-2 operators propose non-linear programs that are more relevant for search.

Section 2 describes related work on semantic GP and model-based approaches and discusses training-data bias in learned models. Section 3 defines our linearity measure λ and analyzes the linearity of the benchmark problems. Section 4 describes the TSGP approach, introduces TSGP-2, and compares the linearity of their training data. Section 5 compares the search performance of TSGP-2, TSGP, and standard GP, and Section 6 traces the differences back from the best programs and the offspring to the operators. Section 7 concludes this paper.

2 Related Work

2.1 Semantic GP

Standard variation operators apply syntactic changes to an expression with unpredictable semantic effects [35]. Therefore, semantic variation operators have been proposed that control the semantic effect of a variation to better guide the search process. We focus on semantic mutation operators, which can be divided into indirect and direct approaches [35].

Indirect approaches keep the standard mutation and filter its offspring by their semantics. Semantically driven mutation [4] rejects offspring that are semantically identical to their parent, and semantic similarity-based mutation [32] accepts only offspring whose semantic distance to the parent lies within a given range. In contrast, direct approaches construct offspring with a controlled semantic effect. GSGP [22] follows a pre-defined rule by adding a random substructure to the parent. The output of this substructure is scaled by a mutation step, so that the semantic change is bounded. As a result, the offspring lies within a controlled semantic distance to its parent. However, because every offspring contains its entire parent, expressions grow quickly. SLIM_GSGP [34] adds a deflate mutation that removes a previously added substructure so that individuals can also shrink. Subsequent work explores further variants [36], the choice of the mutation step [6], and the extension of SLIM_GSGP with crossover [24].

TSGP [2, 3] is also a direct approach, but replaces pre-defined rules with a trained machine learning model that proposes, for a given parent, an offspring of similar semantics.

2.2 Model-based GP

Model-based GP replaces standard variation operators with a model learned from data with the goal to pass on promising substructures to the offspring rather than disrupting them through random variation [12, 40]. These include estimation-of-distribution algorithms (EDAs), which estimate a probability distribution over the selected individuals in every generation and sample the next population from it [9, 17]. They differ mainly in the dependencies their model can express. PIPE [28] gives each node position of a prototype tree an independent distribution over the instruction set, stochastic grammar-based GP [27] learns probabilities for the production rules of a grammar, and N-gram GP [26] conditions each instruction of a linear program on the two preceding ones. DAE-GP [45] uses an LSTM denoising autoencoder that generates offspring by corrupting a parent and reconstructing it, with the corruption strength controlling the trade-off between exploration and exploitation [43, 44]. In contrast, GP-GOMEA [39, 40] does not sample from its model. Instead, it learns which node positions should be varied together and uses this linkage model to mix these groups of nodes between solutions.

TSGP differs from these approaches in two ways. First, its model is fitted once on separate training data before any search, rather than on the population of the current run, and second, the model learns how to generate semantically similar offspring.

2.3 Bias in Learned Models

In symbolic regression, pretrained models [5, 11, 37] are trained on millions of expressions sampled from a random expression generator [16], each paired with its evaluation on data. Given a new dataset, they predict an expression directly by decoding from the model instead of using an evolutionary search and either propose it as a skeleton function whose constants are fitted afterwards [5] or include the constants directly [11]. This makes them fast and accurate on problems that resemble their training data, but inaccurate for problems that are not contained in the training set. Sato and Sato [29] call this reproduction bias and measure it as the share of generated expressions whose structure already occurs in the training data. Voigt et al. [41, 42] report that such models hold their

accuracy inside the pre-training distribution and degrade outside it. In other areas of supervised learning, such as computer vision and natural language processing, it is also well studied that a model inherits the biases of its training data [7, 8, 18, 30].

Like these models, TSGP is trained via supervised learning, but it operates as a variation operator that conditions each proposed offspring on a parent. Consequently, any bias inherited from its training data influences the direction in which an offspring is moved away from its parent within the semantic space.

3 Linearity of Solutions

This section suggests a measure for linearity and applies it to programs and standard test problems. Section 3.1 introduces λ, which measures the linearity by how much of its behavior a linear function explains. We use λ throughout this work to measure the linearity of the training data, of the offspring, and of the best programs of a search. Section 3.2 applies the same idea to the targets of common SR benchmark problems, which shows whether solving them requires linear or non-linear solutions.

3.1 A Measure of Linearity

Since solutions in SR are represented as programs in symbolic form, a naive way to measure their linearity would be to check whether a program uses only linear operations. However, this approach fails for syntactically non-linear programs as such programs can still behave nearly linearly on the inputs on which they are evaluated. For example, $(0.2\,x_1 + 0.5)^2$ is a non-linear polynomial, which behaves almost like a linear function within the considered input range (see Fig. 1).

Therefore, we measure the linearity of a solution by its semantics instead of its syntax. Following the usual convention in semantic GP [14, 22, 35], the *semantics* of a program f is the vector of its outputs $s(f) = \big(f(\mathbf{x}_1), \ldots, f(\mathbf{x}_m)\big) \in \mathbb{R}^m$ on a fixed set of inputs $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m]$, where each $\mathbf{x}_i \in \mathbb{R}^d$ and d is the dimensionality of the problem. The linearity of $s(f)$ is quantified by an ordinary least-squares regression of $s(f)$ on X. We define $\lambda(f)$ as its adjusted R^2, which corrects R^2 for the number of inputs d. This corresponds to an approach often used to characterize the linearity of a fitness landscape [20].

For λ close to 1, f behaves linearly on X. The lower λ, the less of the behavior of f a linear function explains. For the subsequent analysis, we distinguish between four linearity levels. We call a program *linear* if $\lambda \geq 0.99$, *nearly linear* if $0.9 \leq \lambda < 0.99$, *moderately non-linear* if $0.5 \leq \lambda < 0.9$ and *strongly non-linear* if $\lambda < 0.5$, meaning a linear fit explains less than half of the variance. Since $s(f)$, and thus λ, depends on the inputs X, the same program can behave almost linearly on a narrow input range and clearly non-linearly on a wide one. Throughout this work, we therefore compute λ on standardized inputs with mean 0 and standard deviation 1, as used in the GP searches (Sections 4.2 and 5.1).

Fig. 1 gives some examples with discrepancies between the syntax class and the semantics of a program. It lists 10 one-dimensional example functions from five syntactic classes, ranging from constant or linear through polynomial, rational, and algebraic to transcendental. For the constant and linear example functions, the choice of constants has no effect on λ. In contrast, for the examples from non-linear classes, two functions with the same skeleton but different constants can differ strongly in λ. For example, the program $(x_1 + 0.5)^2$ has $\lambda = 0.34$, whereas $(0.2\,x_1 + 0.5)^2$ has $\lambda = 0.92$, and 0.2^{x_1} has $\lambda = 0.30$, whereas $0.5^{0.3\,x_1}$ has $\lambda = 0.98$. The examples illustrate that the linearity of a program can only be assessed by looking at its semantics and not its syntax. Thus, λ is a proper measure for assessing the linearity of solutions as well as training data or offspring generated by search operators.

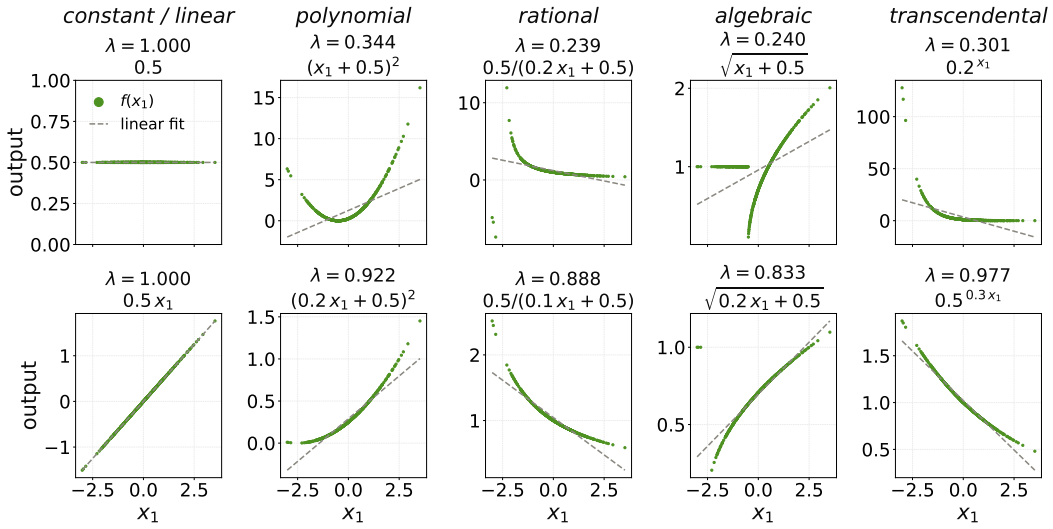


Fig. 1. Each plot shows the output of an example function on standardized inputs together with the least-squares linear fit. λ is the adjusted R^2 of that fit. Each column shows two examples of one syntactic class. Constant and linear functions always have $\lambda = 1$. Although the non-linear examples differ only in their constants, the example functions in the bottom row behave more linearly (higher λ). As in the experiments, division and power are protected and return 1 for undefined results (Table 2).

3.2 Linearity of Benchmark Problems

We now analyze the linearity of common SR benchmark problems. In particular, we consider synthetic and real-world problems of PMLB [23], the collection that underlies SRBench [15], with $d = 2$ to $d = 5$ input variables, which is the range covered by TSGP in this work (Section 4.2). We take all 13 real-world regression datasets with unknown ground truth with at least 100 samples, where n denotes the number of samples of a dataset. In addition, we use the Feynman equations [31], whose targets follow a known physical formula. As the Feynman equations define many test problems, we draw 4 per dimensionality at random, which gives us 16 problems. Feynman datasets are randomly subsampled to $n = 10{,}000$ for computational efficiency. Table 1 lists all 29 problems.

To measure the linearity of a problem, we apply λ to its target values. Unlike the semantics of a program, however, the target of a benchmark problem can contain noise, so a low λ does not tell whether the variance the linear fit leaves unexplained is non-linear structure or noise. We therefore also fit a neural network (NN) with two hidden layers of 128 and 64 units. Since an NN can approximate any continuous function [10], the variance it cannot explain comes from noise, insufficient training, or limited capacity. The difference $R^2_{\text{NN}} - \lambda$ is the variance the NN explains beyond the linear model (0: no additional structure; close to 1: almost entirely non-linear), which we call the *exploitable non-linearity* of a problem. The remaining $1 - R^2_{\text{NN}}$ is the variance neither model explains, which usually comes from noise.

Since an in-sample evaluation would credit the NN for fitting the noise, we evaluate both models on held-out data. We fit them on four random 75/25 splits and report the median. In each split, inputs and target are standardized to mean 0 and standard deviation 1 with the statistics of the training split. For problems, λ is therefore the R^2 of the linear model on the test split instead of the adjusted R^2. On held-out data, both values can be negative if a model predicts worse than the mean of the held-out targets. A problem is perfectly linear if $\lambda = 1$ and has no linear structure if $\lambda \leq 0$.

Table 1. The synthetic and real-world benchmark problems, ordered by their exploitable non-linearity $R^2_{\text{NN}} - \lambda$. d is the dimensionality and n the number of samples. λ and R^2_{NN} are the held-out R^2 of a linear model and of a neural network fitted to the target.

Problem	**Ground truth**	d	n	λ	R^2_{NN}	$R^2_{\text{NN}} - \lambda$
Feynman equations						
I_8_14	$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$	4	10,000	0.00	1.00	1.00
III_17_37	$\beta(1 + \alpha \cos\theta)$	3	10,000	0.13	1.00	0.87
I_50_26	$x_1(\cos\omega t + \alpha \cos^2 \omega t)$	4	10,000	0.34	1.00	0.66
II_13_17	$2I/(4\pi\epsilon c^2 r)$	4	10,000	0.47	1.00	0.53
II_2_42	$\kappa(T_2 - T_1)A/d$	5	10,000	0.60	1.00	0.40
II_34_29b	$g\,\mu B J_z/(h/2\pi)$	5	10,000	0.68	1.00	0.32
III_4_32	$1/(e^{(h/2\pi)\omega/(k_b T)} - 1)$	4	10,000	0.69	1.00	0.31
II_4_23	$q/(4\pi\epsilon r)$	3	10,000	0.72	1.00	0.28
II_10_9	$\sigma/(\epsilon(1 + \chi))$	3	10,000	0.79	1.00	0.21
II_11_3	$qE_f/(m(\omega_0^2 - \omega^2))$	5	10,000	0.82	1.00	0.18
I_39_11	$pV/(\gamma - 1)$	3	10,000	0.83	1.00	0.17
I_25_13	q/C	2	10,000	0.83	1.00	0.17
II_8_31	$\epsilon E_f^2/2$	2	10,000	0.88	1.00	0.12
II_27_18	ϵE_f^2	2	10,000	0.88	1.00	0.12
I_34_27	$(h/2\pi)\,\omega$	2	10,000	0.93	1.00	0.07
test_7	$\sqrt{8\pi G\rho/3 - \alpha c^2/d^2}$	5	10,000	0.98	1.00	0.02
Real-world datasets						
analcatdata_apnea1	–	3	475	0.01	0.89	0.88
analcatdata_apnea2	–	3	475	0.01	0.82	0.81
nikuradse_1	–	2	362	0.62	1.00	0.37
visualizing_galaxy	–	4	323	0.90	0.96	0.07
rabe_266	–	2	120	0.98	1.00	0.02
ESL	–	4	488	0.83	0.84	0.01
LEV	–	4	1,000	0.52	0.52	0.00
analcatdata_neavote	–	2	100	0.94	0.94	0.00
vinnie	–	2	380	0.77	0.76	-0.01
ERA	–	4	1,000	0.41	0.40	-0.01
visualizing_environmental	–	3	111	0.26	0.24	-0.02
chscase_geyser1	–	2	222	0.76	0.73	-0.03
cloud	–	5	108	0.76	0.68	-0.08

Both the Feynman and the real-world problems cover a wide range of λ. Among the Feynman problems, λ ranges from 0.98 for test_7 to 0.00 for I_8_14, and among the real-world problems from 0.98 for rabe_266 to 0.01 for the two apnea datasets.

However, the two problem sets differ heavily in their exploitable non-linearity. On every Feynman problem, the NN explains the target perfectly ($R^2_{\text{NN}} = 1.00$), which shows that the model's training and capacity are sufficient for these noise-free targets. All variance the linear model leaves unexplained is therefore exploitable. On the real-world problems, R^2_{NN} ranges from 1.00 for rabe_266 and nikuradse_1 to 0.24 for visualizing_environmental. Only four of the 13 problems show a clear exploitable non-linearity: the two apnea datasets (0.88 and 0.81), nikuradse_1 (0.37) and visualizing_galaxy (0.07). On rabe_266 and ESL, it is marginal (0.02 and 0.01), and on the remaining seven problems it lies between −0.08 (cloud) and 0.00. On these problems, the NN does not predict better than the linear model.

To sum up, many Feynman problems have a substantial exploitable non-linearity, whereas there are only a few real-world problems with non-linear structures. An operator intended to solve a wide

range of problems should therefore be able to generate programs with both linear and non-linear structures.

4 TSGP-2

The training data of the original TSGP [2, 3] consists of programs generated by GP searches for linear target functions. As this might hurt search performance due to a potential linearity bias (Section 3.2), we propose TSGP-2. TSGP-2 follows the model-building pipeline of TSGP (Section 4.1), but generates both linear and non-linear target functions (Section 4.2). Section 4.3 studies whether this leads to less linear training data.

4.1 The TSGP Approach

TSGP is a model-based GP approach whose variation operator is a trained model that proposes, for a given program, another program with similar semantics. The model is trained once on separate training data (model building) and can then serve as variation operator for a standard GP search on different problems (model inference).

Model Building. Training a model to propose semantic variations requires examples of program pairs that are semantically similar. Since the programs generated along a single GP run are variants of one another, TSGP obtains its training pairs from standard GP runs on synthetic problems [2]. The pipeline is identical for TSGP and TSGP-2 and has five stages. For more details on the pipeline, we refer to the original work [2]:

(1) In a first step, synthetic datasets are generated for the GP searches. TSGP draws a random target function of dimensionality d from a function generator (Section 4.2).[1] From that function, a synthetic regression dataset is created by drawing random inputs, computing their outputs and adding Gaussian noise to them.
(2) A standard GP run searches for a program that fits this dataset. Every program f_i that is generated along the run is saved, together with its semantics $s(f_i)$, in an archive of that run. The Gaussian noise added to the function prevents the GP run from converging too fast and thus broadens the set of found programs.
(3) The archived programs are normalized into a canonical form with SymPy [21]. This removes syntactically distinct but semantically equivalent duplicates and sharpens the signal the model receives about which syntactic changes actually alter the semantics.
(4) Each archived program is paired with semantically similar programs, which are selected based on their *semantic distance* (SD), i.e., the Euclidean distance between their semantic vectors [22]

$$\mathrm{SD}(f_i, f_j) \;=\; \|s(f_i) - s(f_j)\|_2. \tag{1}$$

As a result, each program f_i forms pairs with its k nearest neighbors $f_o \in N_k(f_i)$ within the same archive. Steps one to four are repeated several times, creating a set of archives with pairs of semantically similar programs.
(5) An encoder-decoder transformer [38] is trained from scratch on the pairs of all archives to generate f_o given f_i. In addition to f_i, the transformer also receives two conditioning values: the semantic distance $\mathrm{SD}(f_i, f_o)$ of the pair and the dimensionality d of the problem the pair was drawn from. SD provides the model with a learning signal of how semantically similar two programs are, and d conditions the model to propose programs that use at most d variables.

[1] To ensure higher comparability, TSGP uses the same function generator as TSGP-2, restricted to the operator $\{+\}$ so that it produces only linear targets. The reported TSGP results are similar to those published in Anthes et al. [3].

Model Inference. Once trained, the model serves as the variation operator during search. For each parent, the conditioning values are set to a target semantic distance SD_t and the dimensionality d of the problem. SD_t controls the desired semantic distance between parent and offspring (step size). Decoding is constrained by syntax control [43], which at each step assigns zero probability to every token that would make the sequence infeasible. Consequently, the operator always proposes a syntactically valid program.

4.2 TSGP versus TSGP-2: The Target Functions make the Difference

In this work, TSGP as well as TSGP-2 generate their target functions using a random expression generator based on the probabilistic tree sampler of [5, 16]. For a target function of dimensionality d, the generator first draws the number of operators uniformly from $\{d-1, d, d+1\}$. It then samples the tree shape, determining where unary and binary operators are placed [16]. The leaves are filled with the input variables, drawn without replacement until each variable appears once and uniformly thereafter, so that every target depends on all d variables. Each variable leaf is then wrapped as $a\,x + b$, with a and b taken from a fixed set of constants $\{0, 0.1, \ldots, 1\}$ and $a \neq 0$.

The difference between TSGP and TSGP-2 lies in the set from which the operators are drawn: TSGP draws its operators only from $\{+\}$, so all target functions are linear by design, as in [2, 3]. In contrast, TSGP-2 uses a larger operator set and draws the operators of the target function from the set $\{+, -, \times, \div, \sin, \sqrt{\cdot}\}$ with equal weights.

From each target function, a synthetic regression problem is derived, which serves as objective for a GP search. In particular, the function is evaluated on 1,000 input points sampled uniformly from $[1, 5]$; the corresponding outputs are perturbed with Gaussian noise of 0.1 times their standard deviation. Then, both inputs and outputs are standardized. For each dimensionality $d \in \{2, 3, 4, 5\}$, we generate 100 problems and run one standard GP search on each, with a population size of 2,000, until 100,000 unique programs are archived or 300 generations are reached.

We want to emphasize that for both variants, TSGP and TSGP-2, the GP searching for solutions uses the same primitive set $\{+, -, \times, \%, \text{pow}\}$. Thus, neither model can propose a primitive the other cannot, and any difference between their training data, and between the learned operators, lies only in how these primitives are combined by the GP to solve the different target functions.

Each archived program is paired with its $k = 3$ nearest neighbors. From the resulting pairs, we draw 2.5 million pairs per dimensionality, resulting in 10 million pairs in total, and hold out 10% of them for validation. This is half as many pairs as in [3], whose ablation studies showed that the training budget can be reduced in the number of pairs without a significant loss in performance. We train the TSGP and TSGP-2 models for 8 epochs. The remaining pairs, which are not drawn for training or validation, form a separate test set.

4.3 Linearity of the Training Data

We analyze whether the non-linear targets used for TSGP-2 indeed lead to less linear training data. This is not guaranteed, since the targets enter the training data only indirectly through the programs GP generates while searching for them. To analyze the linearity of the training data generated by the GP, we randomly draw 50,000 of the 10 million training pairs of each model. For each pair, we compute λ (Section 3.1) for both programs on 500 input points drawn from a standard normal distribution, resulting in 100,000 evaluated programs per model.

Fig. 2 compares the linearity of the training data for TSGP (left) and TSGP-2 (right). We find that the linearity of the targets carries over to the training data. In the TSGP training data, 82.7% of the programs behave linearly, while only 7.2%, 4.3%, and 5.8% of the programs are nearly linear, moderately non-linear, and strongly non-linear, respectively. The situation is different for TSGP-2 training data, where we observe a lower share of programs that behave linearly. Only 36.8% of

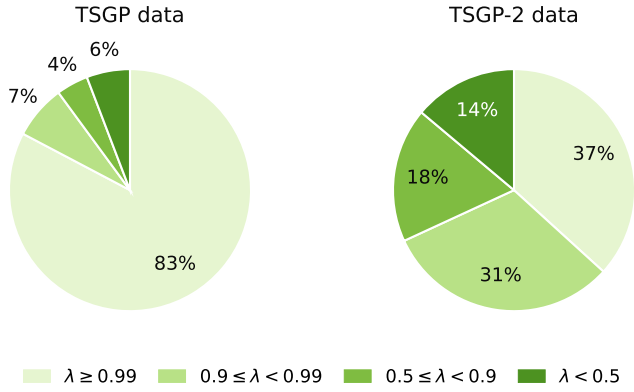


Fig. 2. Share of linearity level among 100,000 sampled training programs of TSGP (left) and TSGP-2 (right).

the programs behave linearly, while 31.4% are nearly linear, 18.0% moderately and 13.9% strongly non-linear. Thus, a non-linear behavior of the programs is far more frequent than in the TSGP data: about a third of the TSGP-2 programs are moderately or strongly non-linear, compared to only a tenth of the TSGP programs. Thus, the training data of TSGP is strongly biased toward linear behavior, whereas the use of non-linear targets in TSGP-2 reduces this bias considerably. The following Section 5 examines whether this difference improves search performance, and Section 6 analyzes the behavior of the variation operator.

5 Search Performance

To analyze the effect of the training data on solution quality, we compare the performance of TSGP-2 to TSGP and standard GP.

5.1 Experimental Setting

All three methods (TSGP, TSGP-2, and standard GP) use the parameters in Table 2 and differ only in their variation operator. Each search evolves a population of 100 programs over 100 generations. The initial population is created with ramped half-and-half and depths from 2 to 5, and parents are selected by tournament selection of size 5 without elitism. The primitive set is the same as in model building. It consists of the problem variables, ephemeral random constants (ERCs) from -0.5 to 0.5 in steps of 0.1, and the operators $+$, $-$, $\times$, protected division (%) and protected power (pow), which return 1 for undefined results.

Standard GP applies subtree crossover with a probability of 0.9 and subtree mutation with a probability of 0.1, which inserts a new subtree of depth 0 to 2. Following Koza [13], the maximum tree depth is 17. TSGP and TSGP-2 generate one offspring per selected parent, both with a target semantic distance of $\mathrm{SD}_t = 1$.

For each run, we split the data of a problem at random into 75% training and 25% test cases and standardize inputs and target with the mean and standard deviation of the training cases. As the solution of a run, we report the program with the lowest training RMSE over all generations and evaluate it on the test cases. On each problem, we compare every pair of methods by their test errors over the 30 runs with a two-sided Mann-Whitney-U-test. Within each problem set, we rank the methods on each problem by their median test error and compare the ranks of every pair with a Wilcoxon signed-rank test. Both tests are Holm-corrected over the three pairwise comparisons at a significance level of 0.05.

Table 2. Parameters of the searches. % and pow are protected operators, which return 1 for undefined results.

Parameter	Value
Population size	100
Generations	100
Initialization	Ramped half-and-half, depth $[2, 5]$
Selection	Tournament, size 5
Primitive set	$\{x_1, \ldots, x_d, \text{ERC}, +, -, \times, \%, \text{pow}\}$
ERC	$[-0.5, 0.5]$, step size 0.1
Fitness	RMSE on the training cases
Data split	75/25 training/test
Runs	30 per method and problem

5.2 Solution Quality

We test whether the difference in constructing the training data between TSGP and TSGP-2 affects solution quality, and whether this effect depends on the exploitable non-linearity of a problem (compare Table 1).

Table 3 reports the median test error of the solutions over the 30 runs of each problem. On the Feynman problems, TSGP-2 achieves the best mean rank (1.19), which is significantly better than those of TSGP (1.88) and standard GP (2.94). Both TSGP variants significantly outperform standard GP on 13 problems, and their errors vary considerably less between runs. TSGP-2 finds the best solution on 13 of the 16 problems (and ties on two) and is significantly better than TSGP on 7 of them, whereas TSGP is never significantly better than TSGP-2. The performance gains of TSGP-2 are mostly present for problems with medium or high exploitable non-linearity (Table 1). Six of the seven problems on which TSGP-2 is significantly better have an exploitable non-linearity between 0.17 and 0.32. For problems with very high linearity, where a linear fit explains most of the structure, TSGP and TSGP-2 reach almost identical errors. On III_17_37 and I_50_26, two of the most non-linear problems, no method is significantly better than another. We assume that this is due to the cosine structure of the problem, which is not part of the primitive set of the methods (Table 2).

On the real-world problems, TSGP-2 achieves a mean rank of 1.46 and significantly outperforms standard GP (2.62) but not TSGP (1.92). This is expected, as most real-world problems have little exploitable non-linearity. However, TSGP-2 still achieves better results than TSGP for the four problems with exploitable non-linearity (for visualizing_galaxy even significantly better). On the two apnea datasets, which carry the most exploitable non-linearity, standard GP finds the best solutions, but TSGP-2 reduces the error of TSGP from 0.86 to 0.74 and from 0.85 to 0.64. Due to the strong variation between runs (interquartile ranges of 0.33 to 0.60), however, this difference is not significant. On the nine problems without exploitable non-linearity, TSGP finds the better solution on four and TSGP-2 on five, and none of these differences is significant.

In summary, TSGP-2 achieves the best mean rank on both problem sets. TSGP-2 outperforms TSGP on problems with exploitable non-linearity, achieving significantly better results on 8 of them. For problems without exploitable non-linearity, both achieve a similar solution quality.

6 Analysis of the Linearity Bias

Section 5.2 shows that TSGP-2 finds better solutions than TSGP, mainly on problems with exploitable non-linearity. To understand where this advantage comes from, we analyze the linearity of the best program of each run (Section 6.1), the linearity of the offspring proposed and selected during the search (Section 6.2), and the linearity of the offspring created by the search operator (Section 6.3).

Table 3. Median test RMSE of the solutions over 30 runs, with the interquartile range over the runs in parentheses. Lower is better and the best method per problem is bold. For each problem, superscripts indicate all other methods that are significantly worse (Mann-Whitney per problem and Wilcoxon signed-rank over the per-problem ranks; both are Holm-corrected). Problems are ordered as in Table 1, from the highest exploitable non-linearity to the lowest.

Problem	**Standard GP** *(a)*	**TSGP** *(b)*	**TSGP-2** *(c)*
Feynman equations			
I_8_14	0.774 (0.096)	[a] 0.696 (0.122)	[a] **0.633** (0.223)
III_17_37	0.578 (0.187)	0.500 (0.135)	**0.488** (0.336)
I_50_26	0.803 (0.037)	0.793 (0.020)	**0.792** (0.022)
II_13_17	0.543 (0.263)	[a] **0.470** (0.111)	[a] 0.484 (0.114)
II_2_42	0.622 (0.178)	[a] 0.402 (0.214)	[a] **0.326** (0.173)
II_34_29b	0.503 (0.155)	[a] 0.339 (0.089)	[ab] **0.312** (0.079)
III_4_32	0.451 (0.250)	[a] 0.359 (0.096)	[ab] **0.314** (0.076)
II_4_23	0.394 (0.242)	[a] 0.318 (0.060)	[ab] **0.272** (0.104)
II_10_9	0.466 (0.301)	[a] 0.256 (0.070)	[ab] **0.234** (0.066)
II_11_3	0.468 (0.240)	[a] 0.288 (0.080)	[ab] **0.250** (0.048)
I_39_11	0.267 (0.221)	[a] 0.190 (0.050)	[ab] **0.140** (0.058)
I_25_13	0.175 (0.171)	0.186 (0.038)	**0.160** (0.091)
II_8_31	0.264 (0.221)	[a] **0.104** (0.033)	[a] **0.104** (0.042)
II_27_18	0.240 (0.260)	[a] 0.103 (0.026)	[a] **0.099** (0.033)
I_34_27	0.186 (0.233)	[a] **0.015** (0.013)	[a] **0.015** (0.006)
test_7	0.152 (0.336)	[a] 0.120 (0.030)	[ab] **0.097** (0.008)
mean rank	2.94	[a] 1.88	[ab] 1.19
Real-world datasets			
analcatdata_apnea1	**0.532** (0.600)	0.855 (0.385)	0.737 (0.527)
analcatdata_apnea2	**0.541** (0.424)	0.845 (0.331)	0.635 (0.567)
nikuradse_1	0.385 (0.205)	0.373 (0.107)	**0.333** (0.125)
visualizing_galaxy	0.312 (0.087)	[a] 0.284 (0.042)	[ab] **0.270** (0.048)
rabe_266	0.235 (0.117)	[a] 0.094 (0.026)	[a] **0.088** (0.024)
ESL	0.497 (0.171)	[a] 0.391 (0.049)	[a] **0.390** (0.047)
LEV	0.722 (0.128)	[a] **0.675** (0.038)	[a] 0.679 (0.036)
analcatdata_neavote	0.271 (0.082)	**0.265** (0.096)	**0.265** (0.116)
vinnie	0.525 (0.051)	0.523 (0.048)	**0.520** (0.044)
ERA	0.818 (0.064)	[a] **0.791** (0.036)	[a] 0.793 (0.048)
visualizing_environmental	0.901 (0.184)	0.909 (0.214)	**0.874** (0.169)
chscase_geyser1	0.508 (0.076)	**0.499** (0.057)	0.501 (0.062)
cloud	0.635 (0.308)	**0.523** (0.289)	0.588 (0.316)
mean rank	2.62	1.92	[a] 1.46

6.1 Error and Linearity of the Best Programs

We analyze how the linearity and error of the best programs evolve over the search. Fig. 3 shows the training error (top) and the linearity λ (bottom) of the *best program* (i.e., the program with the lowest training error so far) over the generations. Linearity is measured on the training cases of each run. We plot the median over the 30 runs for each problem and over all problems in each set (Feynman versus real-world). Appendix A shows results for each problem.

On the Feynman problems, both TSGP variants reduce the error much faster than standard GP from the start, while they barely differ from each other in the first generations. From about generation 15 on, the error difference increases. At generation 40, the median error of TSGP-2 is

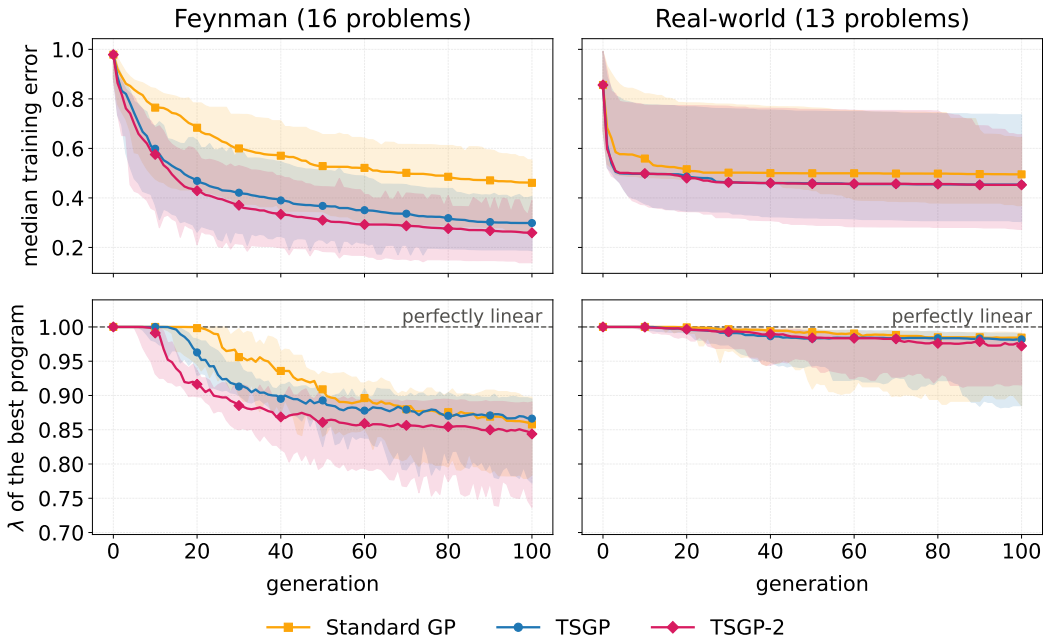


Fig. 3. Training error (top) and linearity λ (bottom) of the best program over the run, for the Feynman (left) and real-world problems (right); we show the median over runs and problems. Bands are bootstrap intervals over problems.

0.33 compared to 0.39 for TSGP, and it remains lower until the end of the run (0.26 compared to 0.30).

The best programs of TSGP-2 also leave the linear region first. Starting from a median λ of 1.00 for all methods, the median λ of TSGP-2 drops below 0.99 at generation 11, that of TSGP at generation 16 and that of standard GP at generation 25. This happens shortly before the error advantage of TSGP-2 over TSGP increases. After around 20 generations, the difference in linearity is clearly visible ($\lambda = 0.92$ for TSGP-2, $\lambda = 0.96$ for TSGP and $\lambda = 1.00$ for standard GP), before the difference in error becomes large. We conclude that TSGP takes longer than TSGP-2 to find good programs outside the linear region, and TSGP-2 keeps the resulting error advantage until the end of the search.

The comparison with standard GP shows that more non-linearity alone does not automatically lead to better solutions. From about generation 70 on, the best programs of standard GP are as non-linear as those of TSGP (0.86 vs. 0.87 at generation 100), but their training error is considerably higher (0.46 vs. 0.30). The non-linear programs that standard GP finds are worse than those of TSGP and TSGP-2. This indicates that the learned operators propose non-linear programs that are more relevant to the target.

For the real-world problems, both TSGP variants also reduce the error faster than standard GP, from 0.86 to about 0.50 within 10 generations, whereas standard GP reaches 0.56. Afterward, all three methods improve only slowly, and by the end of the run the TSGP variants reach a median error of 0.45 and standard GP of 0.50. For all three methods, the best programs found remain nearly linear throughout the run, with a median λ above 0.97. TSGP and TSGP-2 barely differ in error and linearity throughout the run. This is expected, since nine of the 13 real-world problems carry no

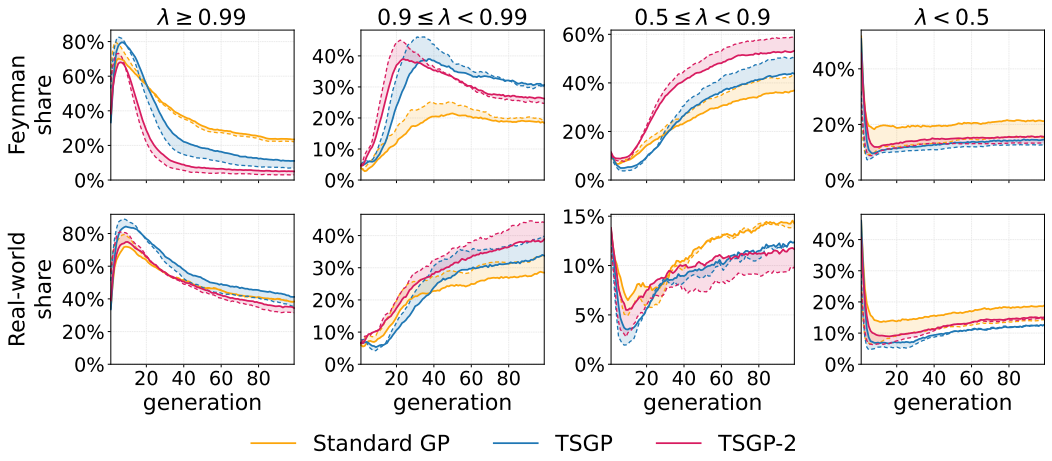


Fig. 4. Percentage of offspring that belong to a linearity level over the number of generations. Results are for all offspring (solid) and selected offspring (dashed) and for Feynman (top) and real-world problems (bottom). If a dashed line is above the corresponding solid one, selection favors solutions of this linearity level. If it is below, selection disfavors it, indicating low fitness.

exploitable non-linearity. On the real-world problems with the highest exploitable non-linearity, the two apnea datasets, the best programs found by TSGP are much more linear than those of TSGP-2; TSGP-2 finds solutions with lower error, whereas TSGP reduces the error only slightly over a run (Figs. 7 and 9 in Appendix A).

In summary, due to the linearity bias of TSGP, its best programs leave the linear region later. In contrast, the best programs of TSGP-2 reach the non-linear region earlier on problems with exploitable non-linearity. After they leave the linear region, the error advantage of TSGP-2 over TSGP grows, and both differences persist until the end of the search. On the Feynman problems, standard GP also finds non-linear solutions, but they have higher errors, indicating that the non-linearity induced by both TSGP variants is more beneficial for search performance.

6.2 Linearity of Populations over the Number of Generations

The previous subsection shows that TSGP-2 is able to find non-linear solutions with high performance faster than TSGP. To understand why, we study how the linearity of the created solutions changes over the number of generations. For the four different linearity levels (Section 3.1), Fig. 4 plots the percentage of solutions that belong to a given linearity level over the number of generations. We show results for all offspring before selection (solid line) and after selection (dashed line). Results are for Feynman (top) and real-world problems (bottom). The percentages are averaged over all runs and problems.

We start our analysis with the Feynman problems. At the start of the search, all three methods mainly generate linear solutions. Within a few generations, the share of linear offspring in a population reaches its maximum of about 70% for standard GP and TSGP-2 and 80% for TSGP (top, left plot). After this peak, TSGP-2 quickly reduces the number of linear solutions and at generation 20 only 29% of its offspring are still linear, compared with about 55% for TSGP and standard GP. Although selection in TSGP clearly disfavors linear structures (there is a large gap between the blue solid and dashed line), TSGP populations still contain many more linear structures than those of TSGP-2.

TSGP-2 is fastest in creating nearly linear structures (second column) as well as moderately non-linear structures (third column). For the nearly linear structures, TSGP-2 already reaches a maximum of 39% of the population at generation 24; TSGP needs 37 generations until it reaches the same level, whereas standard GP stays below 21%. For all methods, the percentage of nearly linear programs (until about generation 50) and of moderately non-linear programs (from about generation 25 until the end of the search) is higher in the population after selection than before. Thus, these programs have higher fitness and are preferred by selection.

For the strongly non-linear structures (top, right), standard GP creates about 20% strongly non-linear offspring in each generation throughout the search, but make up only 11% to 15% of the selected offspring. For the TSGP variants, the percentage is lower and a higher share of these solutions is kept by selection. Thus, standard GP often proposes strongly non-linear offspring, but these solutions have low fitness and are often discarded by selection. In contrast, the two TSGP variants generate fewer non-linear structures; however, the created solutions usually have high fitness and survive selection.

For the real-world problems, the situation is different, since optimal solutions have a much more linear structure (compare Table 1). Over the number of generations, the percentage of linear offspring (bottom, left) remains much higher for all three search approaches. After a few generations, the number of nearly linear structures starts increasing until the end of the search. The number of moderately non-linear as well as non-linear structures slightly increases over time; however, their absolute numbers remain low. For example, non-linear structures created by TSGP-2 usually have lower fitness and thus selection reduces the percentage of such structures in a population. This is different for nearly linear structures, which are favored by selection (dashed line is higher than the solid line). Thus, for the real-world problems, which have only little exploitable non-linearity, the creation of non-linear structures gives no advantage to the search, as those structures usually have lower fitness in comparison to more linear structures. These findings are consistent with the performance of the search methods (compare Section 5.2).

In summary, search usually starts with creating linear structures. As the Feynman problems have a higher level of exploitable non-linearity, search methods move towards creating more non-linear structures. TSGP-2 is fastest in moving from high-quality linear structures to high-quality non-linear structures. In contrast, standard GP creates many non-linear structures, but most of them have low quality and are discarded by selection. The situation is different for the real-world problems, which have much less exploitable non-linearity and can often be solved with more linear programs. For these problems, it is not relevant to generate high-quality non-linear structures. Thus, all three methods are able to create high-quality linear or nearly linear structures.

6.3 Analyzing the Linearity Bias of the Search Operator

The previous section studies how the linearity of populations changes over the number of generations. We found that TSGP has a stronger linearity bias than TSGP-2 and is slower in generating high-quality non-linear solutions. This section goes deeper and looks at the search operator. We want to understand whether the used search operator has a bias towards creating more linear or non-linear offspring.

Consequently, we apply the search operator to one parent and measure the change in linearity of the offspring. As parents, we use the input programs of held-out test pairs from the TSGP and TSGP-2 training data. We use only programs that are not included in any training or validation pair of any model, neither as input nor as output. From the test pairs of each model (TSGP and TSGP-2), we randomly draw 20,000 parents. To study the bias of the different operators, we apply each operator once to each of these 40,000 parents. TSGP as well as TSGP-2 are conditioned on

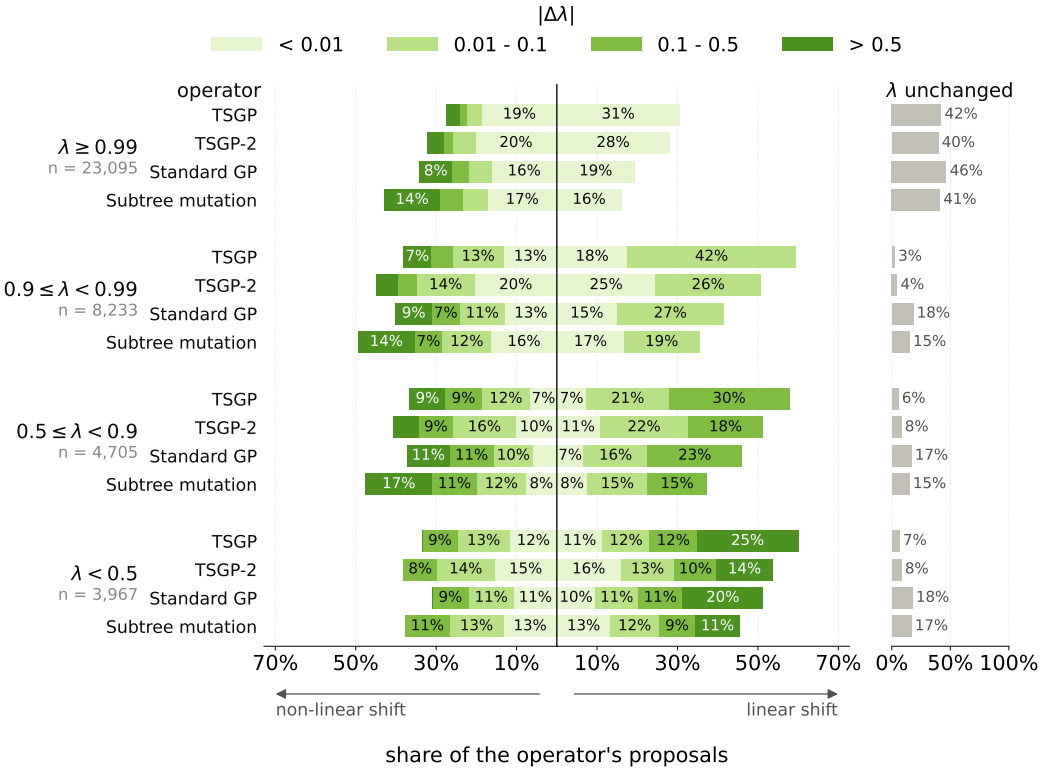


Fig. 5. Change in linearity $\Delta\lambda$ from parent to offspring, per linearity level of the parent and operator. Bars left of zero lower λ, bars right of zero raise it, and the shade indicates $|\Delta\lambda|$. The right panel shows the share of offspring with unchanged λ. All operators vary the same 40,000 parents from test pairs held out from the TSGP and TSGP-2 training data.

$SD_t = 1$. As baselines, we use the variation operators of standard GP with the settings described in Section 5.1.

For each parent, we measure the change in linearity $\Delta\lambda = \lambda(\text{offspring}) - \lambda(\text{parent})$. A positive $\Delta\lambda$ means the operator creates an offspring that is more linear than its parent; a negative $\Delta\lambda$ indicates a more non-linear offspring. Offspring with $|\Delta\lambda| \leq 10^{-6}$ count as unchanged. We report results separately for the four different linearity levels of the parents, since $\Delta\lambda$ depends on the linearity of the parent, e.g., if a parent is linear, it is not possible to obtain an offspring with $\Delta\lambda > 0.01$.

For the four different linearity levels of the parent, Fig. 5 shows $\Delta\lambda$ for the different types of operators (TSGP, TSGP-2, standard GP, and subtree mutation). Below the linearity level of the parent, we plot the number n of parents that belong to this level. The vertical line separates all offspring that are more linear than their parents (linear shift) from the offspring that are more non-linear than their parents (non-linear shift). For example, the first line of Fig. 5 shows that when applying the TSGP operator to the 23,095 linear parents ($\lambda \geq 0.99$), 42% of the offspring are unchanged ($|\Delta\lambda| \leq 10^{-6}$), 31% are slightly more linear ($10^{-6} < \Delta\lambda < 0.01$), and 19% are slightly more non-linear ($-0.01 < \Delta\lambda < -10^{-6}$).

For linear parents ($\lambda \geq 0.99$), the offspring remain mostly linear independently of the used operator. TSGP-2 creates more structures with lower $\lambda(\text{offspring})$ in comparison to TSGP (32%

versus 27%). TSGP as well as TSGP-2 lower λ less often than standard GP and subtree mutation (34% and 43%). However, as we have seen in Section 6.2, the strongly non-linear offspring produced by standard GP often have low fitness and do not survive the selection step.

For all four linearity levels of the parents, we find that the TSGP operator has the strongest bias towards linear structures. 31%, 59%, 58%, and 60% of the offspring are more linear than the parent for linear, nearly linear, moderately non-linear, and strongly non-linear parents, respectively. The linearity bias of the search operator is lower for TSGP-2. TSGP not only moves offspring toward linearity more often, but also by larger steps, which is most visible for nearly linear parents. TSGP moves 42% of them close to or onto the linear level, compared to only 26% for TSGP-2, 27% for standard GP, and 19% for subtree mutation.

7 Conclusions

This work analyzed the linearity of SR benchmark problems and proposed TSGP-2, which reduces the linearity bias that TSGP inherits from its training data. While most real-world problems can be well solved by linear structures (although not all of them), solving Feynman problems requires non-linear structures. The training data of TSGP, however, consists mainly of programs with linear behavior.

We therefore proposed TSGP-2, which trains a model on data that consists of both linear and non-linear program behavior, reducing the share of linear programs from 83% to 37%. We find that TSGP-2 achieves the best mean rank across the studied problems (Feynman and real-world) and clearly outperforms TSGP on problems with exploitable non-linearity. On the Feynman problems, TSGP-2 is significantly better on 7 out of 16 problems compared to TSGP, and never significantly worse. On the real-world problems, TSGP-2 is the only operator that significantly outperforms standard GP and improves especially on the problems with exploitable non-linearity.

On the remaining problems without exploitable non-linearity, TSGP-2 and TSGP reach comparable solution quality. On these problems, selection filters out non-linear structures that are not relevant for the search. We conclude that the linearity bias mostly harms performance on problems where non-linear structures are needed and, in contrast to other learned models, a learned variation operator therefore needs to be studied in combination with search dynamics, such as selection.

However, proposing non-linear structures during search does not automatically lead to high-quality solutions. Across all problems, standard GP proposes the most strongly non-linear offspring, although many of these solutions are discarded by selection. The final best programs found with GP are as non-linear as those of TSGP but perform considerably worse. What matters is therefore not how much non-linearity an operator proposes, but whether these structures are relevant for the search. The operator of TSGP-2 proposes non-linear structures that are more frequently selected during search, indicating that the variations are more meaningful (Section 6.2).

Our findings highlight the importance of training data for a learned variation operator. Since a learned model reproduces the biases of its training data, an learned operator that is meant to be applied to a wide range of problems requires training data that comprehensively covers the relevant problem properties. As linearity is only one problem property, future work should investigate further kinds of biases in training data. This concerns for example other program behavior (e.g., quadratic or sinusoidal shapes) or other properties a learned operator can be biased in.

References

[1] Philipp Anthes. 2026. Syntactic Flexibility Enables Compact Solutions in Transformer Semantic GP. In *Genetic Programming (EuroGP 2026)*. Springer, 51–66. doi:10.1007/978-3-032-23005-8_4

[2] Philipp Anthes, Dominik Sobania, and Franz Rothlauf. 2025. Transformer Semantic Genetic Programming for Symbolic Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '25)*. ACM, New York, NY,

USA, 952–960. doi:10.1145/3712256.3726412

[3] Philipp Anthes, Dominik Sobania, and Franz Rothlauf. 2026. Transformer Semantic Genetic Programming for d-dimensional Symbolic Regression Problems. *ACM Transactions on Evolutionary Learning and Optimization* (2026). doi:10.1145/3845808

[4] Lawrence Beadle and Colin G. Johnson. 2009. Semantically driven mutation in genetic programming. In *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1336–1342. doi:10.1109/CEC.2009.4983099

[5] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. 2021. Neural Symbolic Regression that Scales. In *Proceedings of the 38th International Conference on Machine Learning (PMLR, Vol. 139)*. 936–945.

[6] Davide Farinati, Gloria Pietropolli, and Leonardo Vanneschi. 2025. Exploring the Impact of Data Scale on Mutation Step Size in SLIM-GSGP. Springer-Verlag, Berlin, Heidelberg, 35–51. doi:10.1007/978-3-031-89991-1_3

[7] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. 2019. ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations (ICLR)*.

[8] Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel R. Bowman, and Noah A. Smith. 2018. Annotation Artifacts in Natural Language Inference Data. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 107–112. doi:10.18653/v1/N18-2017

[9] Mark Hauschild and Martin Pelikan. 2011. An introduction and survey of estimation of distribution algorithms. *Swarm and evolutionary computation* 1, 3 (2011), 111–128. doi:10.1016/j.swevo.2011.08.003

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

[11] Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. 2022. End-to-end symbolic regression with transformers. *Advances in Neural Information Processing Systems* 35 (2022), 10269–10281. doi:10.52202/068431-0746

[12] Kangil Kim, Yin Shan, Xuan Hoai Nguyen, and Robert I McKay. 2014. Probabilistic model building in genetic programming: a critical review. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 115–167. doi:10.1007/s10710-013-9205-x

[13] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

[14] Krzysztof Krawiec and Pawel Lichocki. 2009. Approximating geometric crossover in semantic space. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 987–994. doi:10.1145/1569901.1570036

[15] William La Cava, Bogdan Burlacu, Marco Virgolin, Michael Kommenda, Patryk Orzechowski, Fabrício Olivetti de França, Ying Jin, and Jason H Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, Vol. 1.

[16] Guillaume Lample and François Charton. 2020. Deep Learning for Symbolic Mathematics. In *International Conference on Learning Representations (ICLR)*. arXiv:1912.01412.

[17] Pedro Larrañaga and Jose A. Lozano. 2002. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Boston, MA. doi:10.1007/978-1-4615-1539-5

[18] Tom McCoy, Ellie Pavlick, and Tal Linzen. 2019. Right for the Wrong Reasons: Diagnosing Syntactic Heuristics in Natural Language Inference. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*. 3428–3448. doi:10.18653/v1/P19-1334

[19] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 2008. Semantic building blocks in genetic programming. In *European Conference on Genetic Programming*. Springer, 134–145. doi:10.1007/978-3-540-78671-9_12

[20] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Mike Preuss, Claus Weihs, and Günter Rudolph. 2011. Exploratory Landscape Analysis. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 829–836. doi:10.1145/2001576.2001690

[21] 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. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103. doi:10.7717/peerj-cs.103

[22] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature (PPSN XII) (LNCS, Vol. 7491)*. Springer, 21–31. doi:10.1007/978-3-642-32937-1_3

[23] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. 2017. PMLB: A Large Benchmark Suite for Machine Learning Evaluation and Comparison. *BioData Mining* 10, 36 (2017), 1–13. doi:10.1186/s13040-017-0154-4

[24] Gloria Pietropolli, Davide Farinati, Luca Manzoni, Mauro Castelli, Sara Silva, and Leonardo Vanneschi. 2025. Introducing crossover in slim-gsgp. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, 103–119. doi:10.1007/978-3-031-89991-1_7

[25] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises. Freely available at http://www.gp-field-guide.org.uk.

[26] Riccardo Poli and Nicholas Freitag McPhee. 2008. A linear estimation-of-distribution GP system. In *European Conference on Genetic Programming*. Springer, 206–217. doi:10.1007/978-3-540-78671-9_18

[27] Alain Ratle and Michele Sebag. 2001. Avoiding the bloat with stochastic grammar-based genetic programming. In *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer, 255–266. doi:10.1007/3-540-46033-0_21

[28] Rafal Salustowicz and Jürgen Schmidhuber. 1997. Probabilistic incremental program evolution. *Evolutionary computation* 5, 2 (1997), 123–141. doi:10.1162/evco.1997.5.2.123

[29] Shun Sato and Issei Sato. 2026. Can Test-time Computation Mitigate Reproduction Bias in Neural Symbolic Regression? *arXiv preprint arXiv:2505.22081* (2026).

[30] Antonio Torralba and Alexei A. Efros. 2011. Unbiased look at dataset bias. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1521–1528. doi:10.1109/CVPR.2011.5995347

[31] Silviu-Marian Udrescu and Max Tegmark. 2020. AI Feynman: A physics-inspired method for symbolic regression. *Science advances* 6, 16 (2020), eaay2631. doi:10.1126/sciadv.aay2631

[32] Nguyen Quang Uy, Nguyen Xuan Hoai, and Michael O'Neill. 2009. Semantics based mutation in genetic programming: The case for real-valued symbolic regression. In *15th International Conference on Soft Computing (Mendel'09)*. Brno, Czech Republic, 73–91.

[33] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I. McKay, and Edgar Galván-López. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (2011), 91–119. doi:10.1007/s10710-010-9121-2

[34] Leonardo Vanneschi. 2024. SLIM_GSGP: The Non-bloating Geometric Semantic Genetic Programming. In *Genetic Programming (EuroGP 2024)*. Springer, 125–141. doi:10.1007/978-3-031-56957-9_8

[35] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2014. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines* 15, 2 (2014), 195–214. doi:10.1007/s10710-013-9210-0

[36] Leonardo Vanneschi, Davide Farinati, Diogo Rasteiro, Liah Rosenfeld, Gloria Pietropolli, and Sara Silva. 2025. Exploring Non-bloating Geometric Semantic Genetic Programming. In *Genetic Programming Theory and Practice XXI*. Springer, Singapore, 237–258. doi:10.1007/978-981-96-0077-9_12

[37] Martin Vastl, Jonáš Kulhánek, Jiří Kubálík, Erik Derner, and Robert Babuška. 2024. SymFormer: End-to-End Symbolic Regression Using Transformer-Based Architecture. *IEEE Access* 12 (2024), 37840–37849. doi:10.1109/ACCESS.2024.3374649

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30.

[39] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. 2017. Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1041–1048. doi:10.1145/3071178.3071287

[40] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. 2021. Improving model-based genetic programming for symbolic regression of small expressions. *Evolutionary Computation* 29, 2 (2021), 211–237. doi:10.1162/evco_a_00278

[41] Henrik Voigt, Michael Habeck, and Joachim Giesen. 2026. Beyond the Training Distribution: Mapping Generalization Boundaries in Neural Program Synthesis. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '26 Companion)*. ACM, New York, NY, USA, 1520–1528. doi:10.1145/3795101.3814681

[42] Henrik Voigt, Paul Kahlmeyer, Kai Lawonn, Michael Habeck, and Joachim Giesen. 2025. Analyzing Generalization in Pre-Trained Symbolic Regression. *arXiv preprint arXiv:2509.19849* (2025).

[43] David Wittenberg. 2022. Using Denoising Autoencoder Genetic Programming to Control Exploration and Exploitation in Search. In *Genetic Programming (EuroGP 2022) (LNCS, Vol. 13223)*. Springer, 102–117. doi:10.1007/978-3-031-02056-8_7

[44] David Wittenberg, Franz Rothlauf, and Christian Gagné. 2023. Denoising autoencoder genetic programming: strategies to control exploration and exploitation in search. *Genetic Programming and Evolvable Machines* 24, 2 (2023), 17. doi:10.1007/s10710-023-09462-2

[45] David Wittenberg, Franz Rothlauf, and Dirk Schweim. 2020. DAE-GP: denoising autoencoder LSTM networks as probabilistic models in estimation of distribution genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. 1037–1045. doi:10.1145/3377930.3390180

A Results per Problem

Figs. 6 to 9 show the training error and the linearity λ of the best program over the run for each problem. The problems are ordered by their exploitable non-linearity (Table 1). Lines are medians over the 30 runs of a problem and bands the interquartile range over these runs. For λ, the vertical axis is scaled to the values from generation 5 on, since the best programs of the first generations scatter widely between runs.

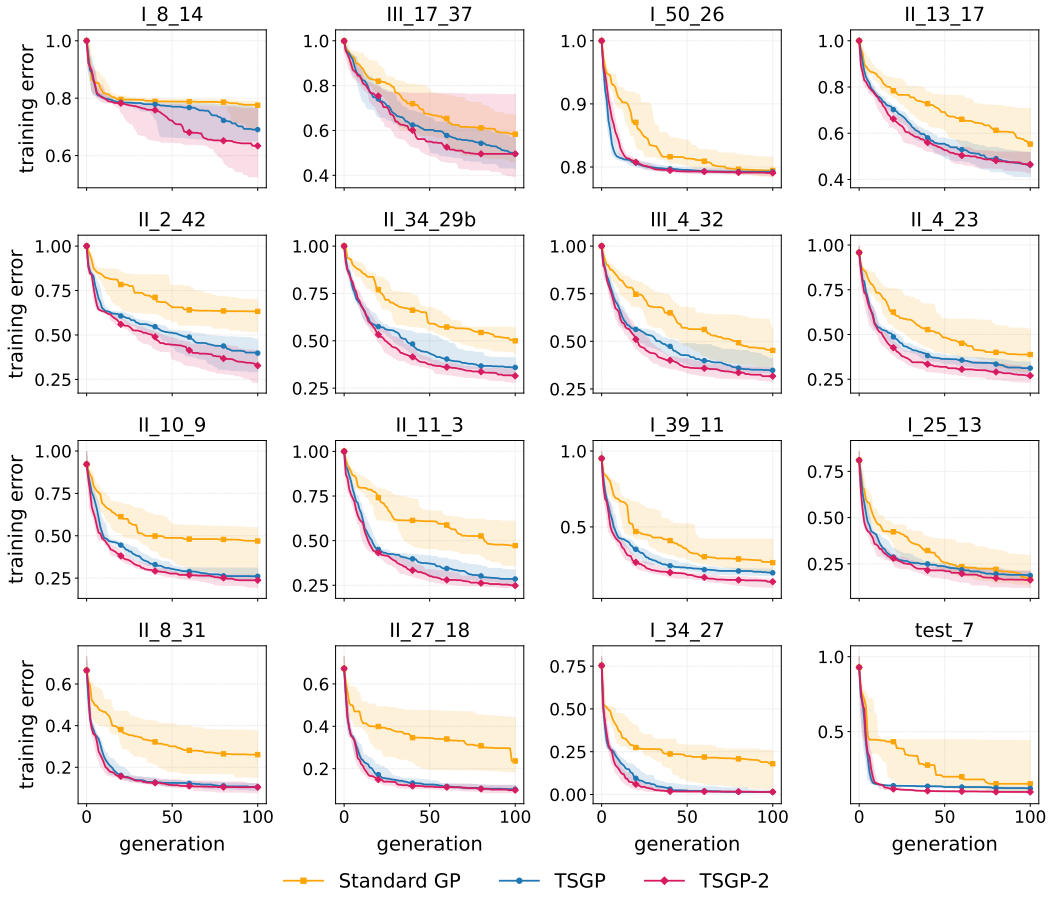


Fig. 6. Training error of the best program over the run on each Feynman problem. Lines are medians over the 30 runs, bands the interquartile range.

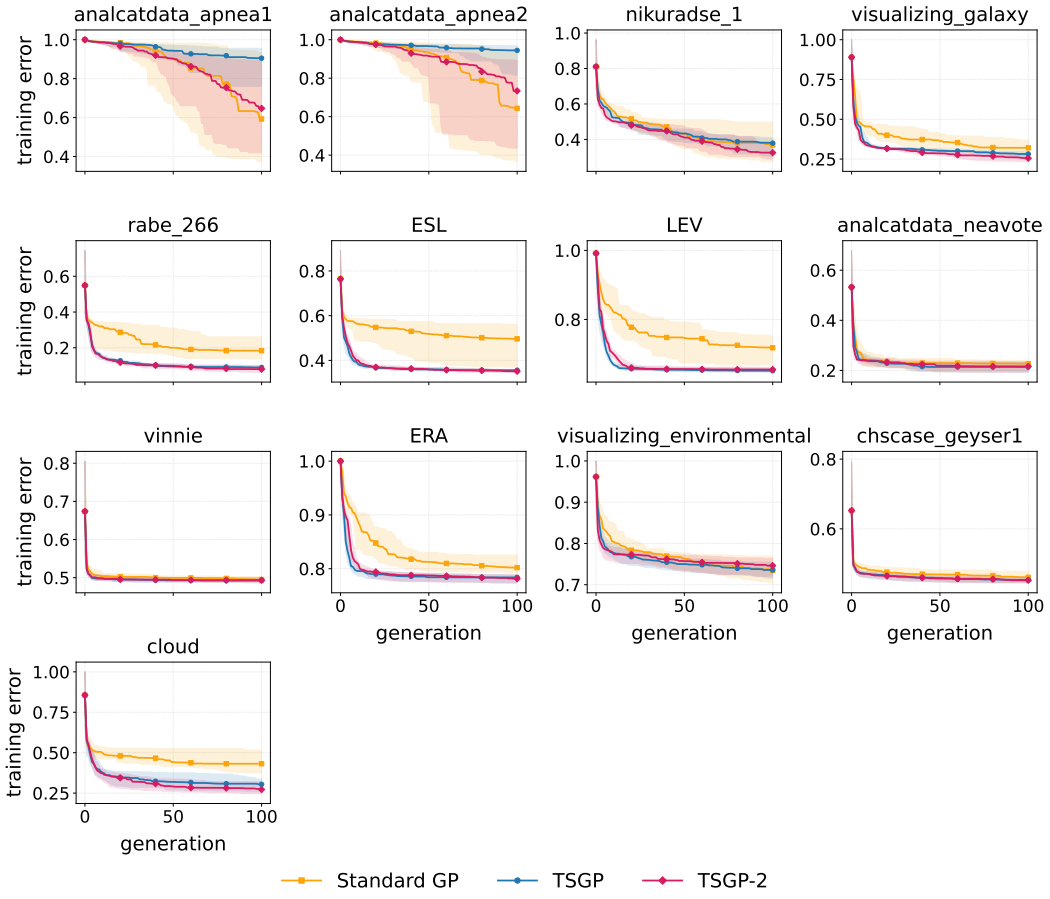


Fig. 7. Training error of the best program over the run on each real-world problem. Lines are medians over the 30 runs, bands the interquartile range.

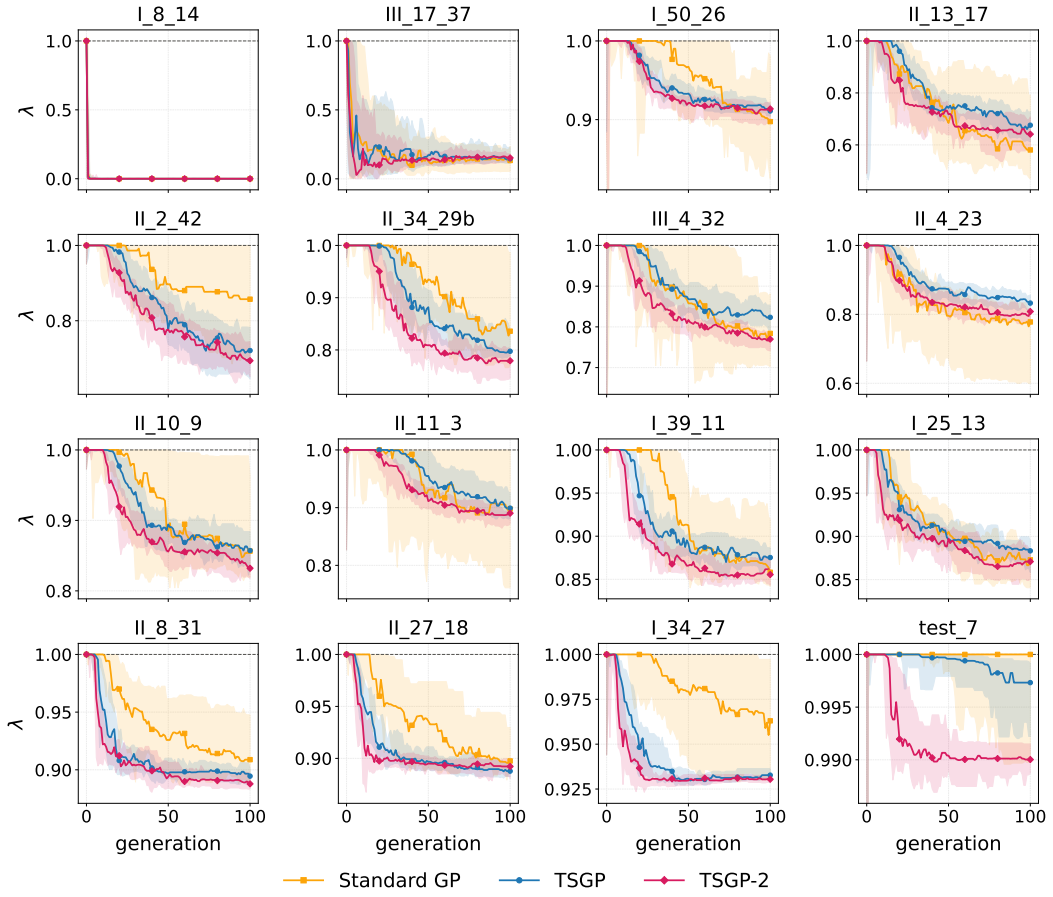


Fig. 8. Linearity λ of the best program over the run on each Feynman problem. Lines are medians over the 30 runs, bands the interquartile range.

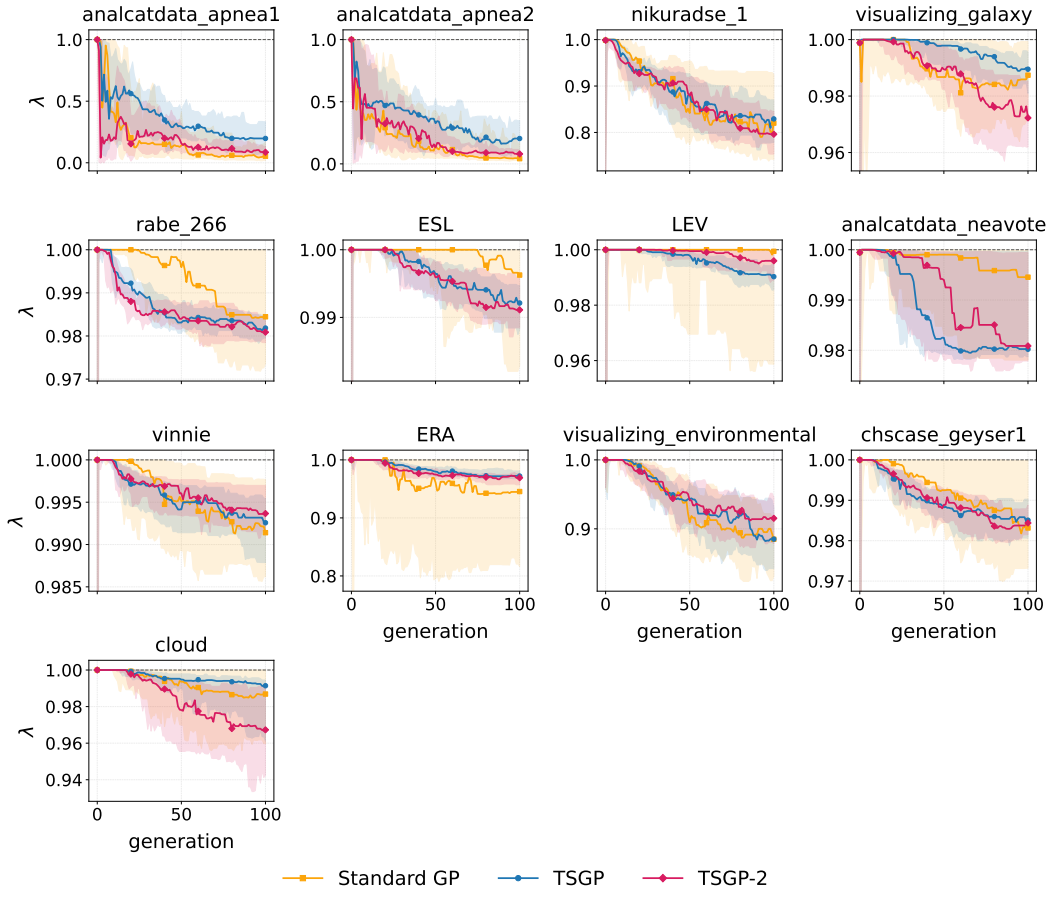


Fig. 9. Linearity λ of the best program over the run on each real-world problem. Lines are medians over the 30 runs, bands the interquartile range.