SRToolkit: Shared Infrastructure for Symbolic Regression Research

Sebastian Mežnar*†‡, Sašo Džeroski* and Ljupčo Todorovski‡*
*Jožef Stefan Institute, Ljubljana, Slovenia
†Jožef Stefan International Postgraduate School, Ljubljana, Slovenia
‡Faculty of Mathematics and Physics, University of Ljubljana, Ljubljana, Slovenia
emails: sebastian.meznar@ijs.si, saso.dzeroski@ijs.si, ljupco.todorovski@fmf.uni-lj.si

***Abstract*—Symbolic regression has seen rapid methodological growth, yet the field remains fragmented at the implementation level. Most approaches are developed as self-contained systems that reimplement core pipeline components, making fair comparison difficult and raising the barrier to entry for new researchers. We present SRToolkit, an open-source, modular Python toolkit for the development and evaluation of symbolic regression methods. The toolkit provides unified infrastructure for key components of the SR pipeline, including expression representation and compilation to callable functions, parameter estimation, dataset management, and experimental workflows. These components are exposed through easy-to-use interfaces, allowing researchers to focus on algorithmic contributions while relying on well-tested implementations of common functionality. SRToolkit further includes built-in benchmark datasets, a flexible experiment orchestration system with support for reproducible and parallel execution, and an extensible interface for integrating new approaches. The toolkit is available on PyPI and is openly developed on GitHub.**

***Index Terms*—symbolic regression, equation discovery, benchmarking, evaluation framework, open-source software.**

I. Introduction

Symbolic regression (SR), or equation discovery, seeks to learn explicit mathematical expressions from data, providing models that are both predictive and directly interpretable. In recent years, the field has experienced rapid growth, driven by advances in evolutionary algorithms, probabilistic modeling, deep reinforcement learning, and, more recently, large language models. These advances substantially broadened the range of applicable techniques and improved performance on increasingly complex, real-world tasks.

Despite this progress, the field remains highly fragmented at the level of implementation and evaluation. Many proposed approaches are developed as self-contained systems, reimplementing core components of the SR pipeline, including expression representation, parsing, numerical evaluation, parameter estimation, and benchmarking. While this design enables flexibility, it also introduces substantial variability across implementations. Seemingly minor differences in low-level design choices—such as numerical precision, optimization routines, stopping criteria, or evaluation strategies—can lead to significant discrepancies in both computational efficiency and predictive performance. Consequently, empirical comparisons between approaches can be misleading, as observed differences may reflect implementation details as much as underlying algorithmic design.

Existing benchmarking frameworks, such as SRBench [1], have taken important steps toward systematic benchmarking, but typically treat approaches as black boxes without standardizing their components or experimental constraints, limiting the ability to isolate the effect of individual design choices.

Furthermore, the lack of modular, reusable infrastructure creates a high barrier to entry for new researchers, who must invest substantial effort into building or adapting complete SR pipelines before being able to test novel ideas.

To address these challenges, we introduce the Symbolic Regression Toolkit (SRToolkit), a modular Python framework for standardizing the development and evaluation of SR approaches. SRToolkit provides unified abstractions for the core components of the SR pipeline — expression representation and manipulation, parameter estimation, dataset and task definition, and experimental workflows. By decoupling these components behind easy-to-use interfaces, the toolkit allows researchers to focus on their core contributions while building on unified implementations of common functionality.

In particular, SRToolkit facilitates (i) rapid prototyping of new SR approaches through a generic, extensible base class interface, (ii) consistent and reproducible benchmarking across diverse approaches and datasets, and (iii) systematic comparison of SR approaches under controlled experimental conditions, including support for repeated runs with independent random seeds. The framework includes built-in Feynman [2], SRSD [3], and Nguyen [4] benchmarks alongside support for user-defined datasets, and provides flexible experiment configuration through an experiment grid abstraction. Experiments can be executed sequentially or distributed across parallel workers via generated shell scripts, making the framework suitable for both local development and HPC environments.

By promoting modularity, standardization, and reproducibility, SRToolkit reduces duplication of effort, improves comparability across methods, and accelerates progress in symbolic regression research.

This research was funded by the Slovenian Research Agency via the research program "Knowledge Technologies" (P2-0103), the Gravity project "AI for science" (GC-0001), project "Probabilistic and explainable data-driven modelling of Solid-oxide fuel cells" (J2-4452), and by the ARRS Grant for young researchers.

II. Related work

Systematic evaluation of symbolic regression methods has gained increasing attention, with frameworks such as SRBench [1] providing large-scale empirical comparisons across different approaches and datasets. More recently, benchmarking efforts have also been extended to large language model-based approaches with LLM-SRBench [5]. However, these benchmarks do not expose the internals of the SR pipeline to the user. SRBench, for instance, hard-codes data partitions and time budgets and only requires outputs to use Sympy operators. LLM-SRBench likewise fixes the prompt and scoring process, with only high-level parameters (e.g. number of samples) adjustable via config. In practice, this means methods are treated as black boxes with user-defined configurations under a predetermined architecture.

As a result, comparisons may be conducted under different experimental conditions. Even when high-level evaluation protocols are aligned, important sources of variability remain. In particular, methods may differ in their effective sets of allowed operators, which induces different hypothesis spaces and therefore different inductive biases. Moreover, evaluation budgets are typically specified in terms of wall-clock time, while implementations differ substantially in computational efficiency, including expression evaluation and parameter optimization. Consequently, methods may explore significantly different numbers of candidate expressions within the same time budget, making it difficult to disentangle algorithmic performance from implementation-specific efficiency.

The benchmark datasets underlying these comparisons vary widely in scope and complexity. Synthetic benchmarks such as the Nguyen [4] functions provide simple and controlled evaluation scenarios, while physics-inspired datasets such as the Feynman benchmark [2] and its extensions (e.g., SRSD [3]) aim to capture more structured relationships that better reflect real-world applications. Despite their widespread adoption, these benchmarks are often used under differing experimental assumptions, rather than fully standardized protocols. In particular, choices such as allowed operator sets, noise levels, and constraints on expression complexity frequently vary across studies. This lack of standardization in evaluation settings makes it difficult to compare results across papers, even when identical benchmark datasets are used.

Beyond evaluation, symbolic regression approaches differ in how core computational components are implemented. Expression evaluation approaches range from symbolic computation (e.g., SymPy [6]) to compiled code generated through tools such as Cython, Julia, or NumExpr [7], with implementations varying in efficiency and numerical stability. Similarly, parameter estimation ranges from random assignment, to closed-form regression and iterative numerical optimization. These choices directly affect computational efficiency, scalability, and numerical behavior, yet they are typically re-implemented within each individual system, leading to duplicated effort and subtle inconsistencies across approaches.

A number of software libraries provide efficient implementations of symbolic regression algorithms, including PySR [8], gplearn [9], Operon [10], and DSO [11]. These focus on specific approaches and offer highly optimized search procedures. However, they are generally designed as self-contained systems, each bundling its own expression handling, evaluation, and parameter estimation. This tight coupling makes it difficult to reuse shared infrastructure across approaches or to ensure that different approaches are evaluated under truly comparable conditions. At the other end of the spectrum, general-purpose evolutionary computation libraries such as DEAP [12] can be used to implement GP-based SR; however, SR is merely one of many applications expressible in these frameworks rather than a design focus, so SR-specific infrastructure must be re-implemented by each user.

Across these lines of work, existing efforts primarily focus either on advancing individual approaches or on evaluating black box systems under loose experimental setups. Less attention has been given to providing unified infrastructure for the components of the SR pipeline in a way that remains flexible enough to accommodate diverse approaches. This gap motivates the development of a modular toolkit that standardizes these components, enabling consistent and reproducible experiments, and integration of new approaches.

III. Symbolic Regression Toolkit (SRToolkit)

This section describes the architecture of SRToolkit, organized by the main components of the toolkit: expression representation and manipulation, evaluation, datasets and benchmarks, the approach interface, and experiment orchestration. Each subsection describes the component's design and illustrates its usage through code examples.

Figure 1 summarizes the main components of SRToolkit and their interactions. At the core, an `SR_approach` (Section III-D) generates candidate expressions—represented as token lists or expression trees (Section III-A)—and submits them to the `SR_evaluator` (Section III-C). The evaluator fits free constants via the `ParameterEstimator`, scores expressions against data provided by an `SR_dataset`, and maintains a list of evaluated candidates. Datasets are organized into `SR_benchmarks` (Section III-B), which define standardized tasks and settings. At a higher level, the `ExperimentGrid` (Section III-E) orchestrates executions over combinations of datasets, approaches, and random seeds, enabling systematic and reproducible evaluation. Built-in implementations of both approaches (e.g., ProGED, EDHiE) and benchmarks (e.g., Nguyen, Feynman, SRSD) instantiate this architecture.

A. Expression Representation and Manipulation

Expressions in SRToolkit can be represented as *lists of tokens* or *binary expression trees*. The former are sequences of strings in infix notation (e.g., `["X_0", "+", "sin", "(", "X_1", ")"]`). Binary expression trees, implemented through the `Node` class, represent expressions as hierarchical structures suited for traversal and manipulation. The two representations are interconvertible: `tokens_to_tree`

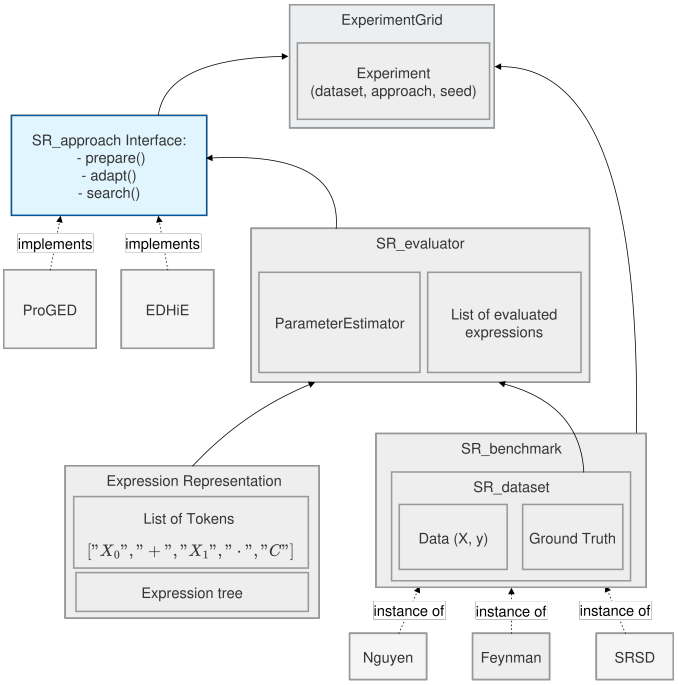

Fig. 1. Overview of the SRToolkit architecture. The blue box highlights the SR_approach Interface that the user must implement, while the gray boxes represent the built-in components provided by the toolkit.

parses token lists into trees, while `Node.to_list` converts trees back into token lists in infix, prefix, or postfix notation.

The meaning of each symbol is defined by the `SymbolLibrary` class, which associates every token with a type, a precedence, an implementation, and a LaTeX template. The toolkit distinguishes five symbol types: *variables* (`X_0`, `X_1`, ...), *free constants* (`C`) whose values are fitted during parameter estimation, *binary operators* (+, −, ·, ...), *unary functions* (`sin`, `cos`, `sqrt`, ...), and *literals* with fixed values (`pi`, `e`). A default library can be constructed via `SymbolLibrary.default_symbols`; custom libraries are built by registering symbols with user-defined properties.

Both expression representations support multiple operations. Expressions can be compiled into vectorized callables via `compile_expr`, rendered as LaTeX strings via `expr_to_latex`, or algebraically simplified via `simplify`, which delegates to SymPy and additionally performs constant folding — collapsing sub-expressions composed entirely of free constants into a single constant. The toolkit also supports random expression generation through `generate_n_expressions`, which samples from a probabilistic context-free grammar (PCFG) that can be manually specified or automatically derived from a symbol library. Additionally, these grammars can incorporate hard constraints that further restrict the space. An automatically derived grammar includes only rules for symbols defined in the symbol library, with probabilities assigned dynamically based on the number of symbols sharing the same type and precedence. Figure 2 illustrates these capabilities.

```python
from SRToolkit.utils import *
import numpy as np

# Define the symbol space with two input
    variables
sl = SymbolLibrary.default_symbols(
    num_variables=2)

# Parse a token list into an expression tree
tree = tokens_to_tree(["X_0", "*", "X_1",
    "+", "C"], sl)

# Convert to prefix and postfix notation
prefix = tree.to_list(sl, notation="prefix")
postfix = tree.to_list(sl, notation="postfix")

# Compile to a function and evaluate
fn = compile_expr(tree, sl)
X = np.array([[1.0, 2.0], [3.0, 4.0]])
C = np.array([0.5])
result = fn(X, C)  # [2.5, 12.5]

# Render as LaTeX
latex = expr_to_latex(tree, sl)

# Simplify: X_0 * X_1 / X_0 -> X_1
simplified = simplify(
    ["X_0", "*", "X_1", "/", "X_0"], sl
)

# Generate 5 unique random expressions from
    the symbol library
exprs = generate_n_expressions(sl,
    num_expressions=5, unique=True)
```

Fig. 2. Expression representation, compilation, and manipulation.

B. Datasets and Benchmarks

SR tasks (problems) are represented at two levels in SRToolkit. An `SR_dataset` encapsulates a single regression task: input data, target values, a symbol library defining the operators and variables, and evaluation settings including the evaluation budget, success threshold, and ranking function. A ground truth expression can be provided as a token list or an expression tree. Each dataset can directly instantiate an `SR_evaluator` (Section III-C) with its configured settings, ensuring consistent evaluation conditions across experiments. A `SR_benchmark` groups multiple datasets into a named collection with support for listing, filtering by number of variables, and lazy loading. Both levels are fully serializable for reproducible sharing of experimental setups.

The toolkit ships with three built-in benchmarks. The **Nguyen** benchmark [4] comprises 10 constant-free equations and serves as a simple, controlled test suite. The **Feynman** benchmark [2] comprises 100 physics equations with up to 9 variables, offering substantially greater structural diversity. The **SRSD** benchmark [3] revisits the Feynman equations with 120 datasets, correcting issues of the original benchmark such as unphysical sampling ranges and physical constants being treated as variables. All benchmarks include predefined

```python
from SRToolkit.dataset import Feynman,
    Nguyen, SR_dataset
import numpy as np

# Load the Feynman benchmark
bm = Feynman()

# List all 2-variable datasets
names = bm.list_datasets(num_variables=2)

# Create a dataset instance
ds = bm.create_dataset(names[0])

# Create an evaluator that evaluates
    expressions against dataset's data
evaluator = ds.create_evaluator()

# Create a custom dataset
custom = SR_dataset(
    X=np.random.randn(100, 2),
    y=np.random.randn(100),
    symbol_library=ds.symbol_library,
    dataset_name="my_dataset",
    max_evaluations=50000
)
```

Fig. 3. Loading benchmark datasets.

symbol libraries and standardized evaluation settings; pre-generated training data is downloaded on first use and cached locally, with per-variable sampling specifications serving as a fallback. The same specifications back the `resample` method, which draws fresh data from the original variable distributions. Users can define custom benchmarks by adding their own datasets. Figure 3 illustrates loading and usage of benchmarks.

C. Evaluation

The `SR_evaluator` is the central interface between SR approaches and the scoring infrastructure. During search, an approach generates candidate expressions and submits them to the evaluator via `evaluate_expr`. The evaluator fits any free constants to the dataset, computes an error score and returns it to guide the search. In addition, it manages evaluation-level concerns such as caching, callbacks, and result aggregation. This separation allows approaches to focus solely on candidate generation, while all scoring and bookkeeping are handled by the evaluator.

Parameter fitting itself is delegated to a dedicated `ParameterEstimator`, which is responsible solely for optimizing free constants. It minimizes the RMSE on the dataset using `scipy.optimize.minimize` (L-BFGS-B by default), with the optimization method, constant bounds, initialization strategy, and convergence tolerances configurable through `EstimationSettings`. Decoupling parameter estimation from the evaluator enables reuse of the same optimization routines across different evaluation strategies and simplifies experimentation with alternative fitting methods. Expressions that exceed limits on the number of free constants or token length are rejected without evaluation. As an alternative to RMSE, the evaluator also supports ranking by Behavioral Expression Distance (BED) [13], which compares output behavior without fitting constants.

```python
from SRToolkit.evaluation import (
    SR_evaluator, ParameterEstimator
)
import numpy as np

# Ground-truth equation y = 2 * x + 3

# Define the dataset
X = np.array([[1.0], [2.0], [3.0], [4.0]])
y = np.array([5.0, 7.0, 9.0, 11.0])

# Create an evaluator and specify the
    evaluation budget
evaluator = SR_evaluator(
    X, y, max_evaluations=1000,
    success_threshold=1e-6
)

# Evaluate a candidate expression
error = evaluator.evaluate_expr(
    ["C", "*", "X_0", "+", "C"]
)

# Collect the top 10 expressions after search
    (other expressions are still available)
results = evaluator.get_results()

# Parameter estimation can also be used
    standalone
estimator = ParameterEstimator(X, y)
rmse, constants =
    estimator.estimate_parameters(
    ["C", "*", "X_0", "+", "C"]
)
# rmse ≈ 0.0, constants ≈ [2.0, 3.0]
```

Fig. 4. Expression evaluation and result collection.

The evaluator tracks the number of evaluated expressions and supports early stopping and monitoring through a callback interface.

After the search completes, `get_results` collects the evaluated expressions into an `SR_results` object. Results can be further enriched by *result augmenters*, which are post-processing plugins that compute additional metrics such as test-set RMSE, R^2, or BED, or apply transformations such as expression simplification and LaTeX rendering. Custom augmenters are defined by subclassing `ResultAugmenter`. Figure 4 illustrates the use of the evaluation workflow.

D. Interface with SR Approaches

New SR approaches are integrated by subclassing `SR_approach` and implementing a `search` method. This method receives an `SR_evaluator` and generates candidate expressions, calling `evaluate_expr` for each one until the evaluator signals that the budget is exhausted or the success threshold has been reached. Hyperparameters are encapsulated

```python
from SRToolkit.approaches import SR_approach,
    ApproachConfig
from SRToolkit.evaluation import SR_evaluator
from SRToolkit.utils import
    generate_n_expressions

class RandomSearch(SR_approach):
    def __init__(self):
        super().__init__(
            ApproachConfig(name="RandomSearch"))

    def prepare(self):
        pass  # stateless

    def search(self, sr_evaluator:
        SR_evaluator, seed=None):
        while not sr_evaluator.should_stop:
            exprs = generate_n_expressions(
                sr_evaluator.symbol_library,
                1)
            sr_evaluator.evaluate_expr(
                exprs[0])
```

Fig. 5. Defining a custom SR approach.

in an `ApproachConfig` dataclass that supports JSON serialization, enabling reproducible experiment reconstruction from configuration files.

Some approaches require a preparation step before search, e.g., training a generative model on the expression space (set of allowed symbols). The toolkit accommodates this through the `adaptation_scope` property: an approach can declare that it needs no adaptation (`"never"`), a single adaptation whose state is cached and reused across experiments (`"once"`), or a fresh adaptation before every run (`"experiment"`). The framework manages the lifecycle automatically via `adapt`, `save_adapted_state`, and `load_adapted_state`, so approaches without an adaptation phase can simply ignore this mechanism.

SRToolkit currently includes two built-in SR approaches. **ProGED** [14] samples expressions from a probabilistic context-free grammar, which can be provided manually or auto-generated from the dataset's symbol library. **EDHiE** [15] trains a hierarchical variational autoencoder on randomly generated expressions and explores its latent space with a genetic algorithm. Both serve as ready-to-use baselines and as reference implementations of the approach interface. Figure 5 shows how to define a minimal custom approach.

Custom components are also shareable. The `bundle` module packs user-defined approaches, constraints, samplers, and other extensions into a single `.srtk` archive that recipients install with one call; saved JSON configurations are then automatically re-bound to the installed code. A complete experimental setup—code and configuration—can thus be reproduced on another machine without manual integration.

E. Experiment Orchestration

The `ExperimentGrid` class manages the full cross-product of datasets, approaches, and repeated runs. Given a list of datasets (or entire benchmarks) and approaches, it generates one job per (dataset, approach, seed) combination, where each seed is derived deterministically from the job index to ensure independent reproducibility.

Jobs can be run sequentially in a single process or distributed to parallel workers. Calling `save_commands` produces a shell script with one CLI command per job, ready for use with GNU Parallel [16] or a cluster scheduler such as SLURM. Because each job writes to its own result file, parallel workers never conflict. For approaches that require one-time adaptation, `adapt_if_missing` pre-computes and caches the adapted state before jobs are dispatched.

Experiments can be monitored and controlled through a callback system. Callbacks are invoked at key events during evaluation (e.g., at the start/end of an experiment, after each expression is scored, or when a new best expression is found), enabling users to implement custom logging, stopping criteria, or progress reporting.

Built-in callbacks include `EarlyStoppingCallback`, which terminates the search when a success threshold or evaluation budget is reached, `LoggingCallback`, which records new best expressions (with file-level locking for concurrent writes), and `ProgressBarCallback`, which provides a live progress indicator. Custom callbacks can be implemented by subclassing `SRCallbacks`.

Once experiments complete, `load_results` merges all per-seed results for a given (dataset, approach) pair into a single `SR_results` object, which can be inspected via `print_results` or accessed programmatically. The `progress` method provides a summary table of completed experiments across the grid. Figure 6 shows a typical workflow.

IV. Availability

SRToolkit is released under the GNU General Public License v3.0 and requires Python 3.10 or later. The stable release can be installed via PyPI (`pip install symbolic-regression-toolkit`) , and the development version is available on GitHub at https://github.com/smeznar/SymbolicRegressionToolkit. Documentation, including API references, guides, and usage examples, is hosted at https://smeznar.github.io/SymbolicRegressionToolkit/.

V. Conclusions and Future Work

We presented SRToolkit, an open-source Python toolkit that provides a unified infrastructure for the development and evaluation of symbolic regression approaches. By standardizing core components of the SR pipeline—including expression representation and compilation, parameter estimation, dataset management, and experimental workflows—the toolkit enables consistent and reproducible comparison of approaches. SRToolkit includes built-in approaches and established benchmark datasets for immediate use, while its modular design

```python
from SRToolkit.dataset import Nguyen, Feynman
from SRToolkit.approaches import ProGED
from SRToolkit.experiments import
    ExperimentGrid
from SRToolkit.evaluation import
    LoggingCallback

# Define the experiment grid
grid = ExperimentGrid(
    datasets=[Nguyen(), Feynman()],
    approaches=ProGED(),
    num_experiments=5,
    results_dir="results/",
    callbacks=LoggingCallback()
)

# Option 1: run sequentially
for job in grid.create_jobs():
    job.run()

# Option 2: generate commands for parallel
    execution
grid.save_commands("commands.sh")
# Then run: cat commands.sh | parallel -j 4

# Check progress and load results
grid.progress()
results = grid.load_results("NG-1", "ProGED")
results.print_results()
```

Fig. 6. Configuring and running an experiment grid.

supports straightforward integration of custom approaches, datasets, and evaluation metrics.

Standardization naturally has limits. SRToolkit assumes a generate-and-test loop; methods that operate outside this paradigm, such as SINDy [17] or one-shot deep learning approaches can still be implemented, but fit less naturally into the comparison framework. Likewise, approaches implemented outside Python (e.g., Operon in C++ or the Julia backend of PySR) can interact with the toolkit through Python bindings, at the cost of crossing language boundaries during evaluation.

Several directions for future development remain. We plan to expand the library of built-in SR approaches to provide stronger and more familiar baselines, including genetic programming and Bayesian-based methods, to further demonstrate the flexibility of the framework. We also aim to introduce tools for result aggregation and visualization to facilitate analysis across experiments. In addition, improving the robustness and accuracy of expression simplification remains an important objective. Finally, we will continue to optimize the performance of core components to better support large-scale benchmarking; in particular, we plan to extend parameter estimation to make it more configurable and support solvers such as Levenberg–Marquardt or gradients-based approaches estimation through automatic differentiation.

References

[1] F. O. de Franca, M. Virgolin, M. Kommenda, M. S. Majumder, M. Cranmer, G. Espada, L. Ingelse, A. Fonseca, M. Landajuela, B. Petersen, R. Glatt, N. Mundhenk, C. S. Lee, J. D. Hochhalter, D. L. Randall, P. Kamienny, H. Zhang, G. Dick, A. Simon, B. Burlacu, J. Kasak, M. Machado, C. Wilstrup, and W. G. La Cava, “”SRBench++ : principled benchmarking of symbolic regression with domain-expert interpretation”,” *”IEEE Trans. Evol. Comput.”*, vol. 29, no. 4, pp. ”1127–1134”, aug 2025.

[2] S.-M. Udrescu and M. Tegmark, “AI Feynman: A physics-inspired method for symbolic regression,” *Science Advances*, vol. 6, no. 16, p. eaay2631, 2020.

[3] Y. Matsubara, N. Chiba, R. Igarashi, and Y. Ushiku, “Rethinking symbolic regression datasets and benchmarks for scientific discovery,” *Journal of Data-centric Machine Learning Research*, 2024. [Online]. Available: https://openreview.net/forum?id=qrUdrXsiXX

[4] N. Q. Uy, N. X. Hoai, M. O’Neill, R. I. McKay, and E. Galván-López, “Semantically-based crossover in genetic programming: application to real-valued symbolic regression,” *Genetic Programming and Evolvable Machines*, vol. 12, no. 2, pp. 91–119, Jun 2011.

[5] P. Shojaee, N.-H. Nguyen, K. Meidani, A. B. Farimani, K. D. Doan, and C. K. Reddy, “LLM-SRBench: A new benchmark for scientific equation discovery with large language models,” in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=SyQPiZJVWY

[6] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, “Sympy: symbolic computing in python,” *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[7] R. McLeod, F. Alted, A. Valentino, G. de Menten, M. Wiebe, cgohlke, A. Bedini, mamrehn, anatoly techtonik, S. Erb, A. Shadchin, A. Bunin, T. Kooij, O. Pavlyk, M. B. Jelloul, L. Garrison, D. M. Hurtado, C. Carey, M. Sarahan, D. Cox, Z. Plesivčak, J. Borgdorff, C. Courbet, M. Dickinson, B. Leitao, A. de Laat, A. Pitrou, A. Portnoy, A. L. Ortega, and A. Böhn. (2018, Dec.) pydata/numexpr: Numexpr v2.6.9. [Online]. Available: https://doi.org/10.5281/zenodo.2483274

[8] M. Cranmer, “Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl,” May 2023. [Online]. Available: http://arxiv.org/abs/2305.01582

[9] T. Stephens, “gplearn: Genetic programming in python, with a scikit-learn inspired api,” https://github.com/trevorstephens/gplearn, 2015.

[10] B. Burlacu, G. Kronberger, and M. Kommenda, “Operon C++: An efficient genetic programming framework for symbolic regression,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1562–1570. [Online]. Available: https://doi.org/10.1145/3377929.3398099

[11] B. K. Petersen, M. L. Larma, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim, “Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients,” in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=m5Qsh0kBQG

[12] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, “DEAP: Evolutionary algorithms made easy,” *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[13] S. Mežnar, S. Džeroski, and L. Todorovski, “Quantifying behavioral dissimilarity between mathematical expressions,” 2025. [Online]. Available: https://arxiv.org/abs/2408.11515

[14] J. Brence, L. Todorovski, and S. Džeroski, “Probabilistic grammars for equation discovery,” *Knowledge-Based Systems*, vol. 224, p. 107077, 2021.

[15] S. Mežnar, S. Džeroski, and L. Todorovski, “Efficient generator of mathematical expressions for symbolic regression,” *Machine Learning*, vol. 112, pp. 4563 – 4596, Sep 2023.

[16] O. Tange. (2023, Nov.) Gnu parallel 20231122 (’grindavík’). [Online]. Available: https://doi.org/10.5281/zenodo.10199085

[17] S. L. Brunton, J. L. Proctor, and J. N. Kutz, “Discovering governing equations from data by sparse identification of nonlinear dynamical systems,” *Proceedings of the National Academy of Sciences*, vol. 113, no. 15, pp. 3932–3937, 2016. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1517384113