Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl

Miles Cranmer[1,2]

[1]*Princeton University, Princeton, NJ, USA*
[2]*Flatiron Institute, New York, NY, USA*

May 5, 2023

PySR is an open-source library for practical *symbolic regression*, a type of machine learning which aims to discover human-interpretable symbolic models. PySR was developed to democratize and popularize symbolic regression for the sciences, and is built on a high-performance distributed backend, a flexible search algorithm, and interfaces with several deep learning packages. PySR's internal search algorithm is a multi-population evolutionary algorithm, which consists of a unique evolve-simplify-optimize loop, designed for optimization of unknown scalar constants in newly-discovered empirical expressions. PySR's backend is an extremely optimized Julia library SymbolicRegression.jl, which can be used directly from Julia. It is capable of fusing user-defined operators into SIMD kernels at runtime, performing automatic differentiation, and distributing populations of expressions to thousands of cores across a cluster. In describing this software, we also introduce a new benchmark, "*EmpiricalBench*, to quantify the applicability of symbolic regression algorithms in science. This benchmark measures recovery of historical empirical equations from original and synthetic datasets.

1 Introduction

Johannes Kepler discovered his famous third law of planetary motion, $(\text{period})^2 \propto (\text{radius})^3$, from searching for patterns in thirty years of data produced by Tycho Brahe's unaided eye. Kepler did not discover this by searching through all conceivable relationships with a genetic algorithm on a computer, but by his own geometrical intuition. "And it was Kepler's Third Law, not an apple, that led Isaac Newton to discover the law of gravitation" [1]. Likewise, Planck's law was not derived from first principles, but was a symbolic form fit to data [2]. This symbolic relationship would inspire the development of Quantum Mechanics.

Now, this first step—discovering empirical relationships from data or based on human intuition—is both difficult and time-consuming, even for low-dimensional data where it has been shown to be NP-hard [3]. In modern, high-dimensional datasets, it seems an impossible task to discover simple symbolic relationships without the use of automated tools.

This brings us to an optimization problem known "symbolic regression," or SR. In this paper, we describe an algorithm for performing SR, and describe software dedicated to equation discovery in the sciences.

Symbolic Regression. SR describes a supervised learning task where the model space is spanned by analytic expressions. This is commonly solved in a multi-objective optimization framework, jointly minimizing prediction error and model complexity. In this

family of algorithms, instead of fitting concrete parameters in some overparameterized general model, one searches the space of simple analytic expressions for accurate and interpretable models. In the history of science, scientists have often performed SR "manually," using a combination of their intuition and trial-and-error to find simple and accurate empirical expressions. These empirical expressions then might lead to new theoretical insights, such as the aforementioned discoveries of Kepler and Planck leading to classical and quantum mechanics respectively. SR algorithms automate this discovery of empirical equations, exploiting the power of modern computing to test many more expressions than our intuition alone could sort through.

SR saw early developments as a tool for science beginning in the 1970s and 1980s, with works such as [4], and the creation of the tool BACON in [5] with further development in, e.g., [6, 7]. Via a combination of heuristic strategies for brute force searching over possible expression trees, BACON and its follow-ups such as FAHRENHEIT [8], demonstrated the ability to discovery a variety of simple physics laws from idealized data. The use of genetic algorithms, which allow for a more flexible search space and fewer prior assumptions about these expressions, was popularized with the work of [9]. Finally, with the development of the user-friendly tools EUREQA in [10, 11, 12], as well as HeuristicLab in [13], SR started to become practical for use on real-world and noisy scientific data and empirical equations.

Equation Discovery for Science. Today EUREQA is a closed-source proprietary tool, and has fallen out of common use in the sciences. In addition to other early work such as [14, 15, 16, 17], there have recently been a significant number of exciting new developments in SR as well as SR for science. These have largely been focused on genetic algorithms on expressions represented as trees, and include core advances to genetic algorithms, such as [18, 19], which propose modified strategies for selecting an individual to mutate and [20, 21] which use an adaptive form of recombination based on emerging statistical patterns. New strategies for SR altogether have also been developed: in particular, our previous work [22, 23, 24] proposes SR as a way of interpreting a neural network. Neural networks can efficiently find low-dimensional patterns

in high-dimensional data, and so this mixed strategy also presents a way of performing SR on high-dimensional or non-tabular data.

Additional techniques include [25, 26] which propose a Markov chain Monte Carlo-like sampling for SR; [27, 28] which trains deep generative models to propose mutated expressions; [29, 30] which pre-train a large transformer model on billions of synthetic examples, in order to autoregressively generate expressions from data; [31, 32, 33] which develop techniques for discovering symmetries with SR; [34] which optimizes a sparse set of coefficients in a very large predefined expression with stochastic gradient descent; and finally [35, 36] which apply genetic algorithms to a flexible space of differential equations.

Apart from trees, a few other data structures are used for representing symbolic expressions, such as imperative representations ("Linear Genetic Programming") [e.g., 37, 38, 39, 40], and sequence representations [e.g., 29, 30, 32]. One of the most popular alternatives to genetic algorithms on trees, especially for the study of PDEs, is "SINDy" (Sparse Identification of Nonlinear Dynamics) [41, 42, 43] which represents expressions as linear combination of a dictionary of fixed nonlinear expressions. Similar ideas of a linear basis of terms is used in the initialization of the "Operon" genetic algorithm [44], as well as in "FFX" [45]. This strategy can also be combined with deep learning [e.g., 43, 46, 47, 48, 49] or other techniques to find expressions on high-dimensional data [e.g., 36]. Additional techniques include mixed integer nonlinear programming [50] and greedy tree searches [51], and combining theory as a type of constraint over the space of models [52].

However, here we argue that many existing algorithms lack certain features which make them practical in a broad setting for discovering interpretable equations in science. The requirements for "scientific SR" are significantly more difficult than for SR applied to synthetic datasets, which many algorithms are implicitly trained on via synthetic benchmarks. We list these below.

1. an empirical equation in science quite often contains unknown real constants[1] The set of

---

[1]One could argue that any constant in a scientific model must depend on a finite number of known constants of physics (e.g., the speed of light) and mathematics (e.g., π). However, this introduces significantly more complexity than is being reduced, as the relationship with fundamental con-

possible expressions considered by some SR algorithms such as [31]: operators, integers, and variables—is discrete and finite up to a given max size. This set is even searchable with brute force, as is used in [31] in combination with a symmetry and dimensional analysis module. However, with arbitrary real constants, the set of equations one must search is uncountably infinite. The difficulty of searching this set is further compounded by several factors:

2. The discovered equation must provide insight into the nature of the problem. The equation which often holds the most insight, and is therefore adopted by scientists, is not always the most accurate (many other ML algorithms could be used instead for accuracy), but is instead an equation which balances accuracy with simplicity.

3. An empirical equation in science is discovered from data containing noise, which often has heteroscedastic structure.

4. Many empirical equations are not differentiable, instead being composed of different expressions active in only parts of parameter space (e.g., $\exp(x)$ for $x < 0$ and $x^2 + 1$ for $x \geq 0$).

5. Discovered expressions in science must satisfy known constraints, e.g., mass conservation.

6. Empirical relations are frequently comprised of operators which are unique to one particular field of science, so the equation search strategy must allow for custom operators.

7. Observational data is often high-dimensional, and the SR algorithm must either manage this internally, or be layered with a feature selection algorithm.

8. The search tool must be capable of finding relationships in non-tabular datasets, such as sequences, grids, and graphs (for example, by using aggregation operators such as $\sum$).

9. Finally, and most importantly, any SR package useful to scientists must be open-source, cross-platform, easy-to-use, and interface with common tools. For more discussion,

the references [53, 54] consider this from the point of view of the astrophysics community.

Contribution. In this paper, we introduce PYSR, a Python and Julia package for Scientific Symbolic Regression. The Python package is available under `pysr` on PyPI and Conda[2] and the Julia library under `SymbolicRegression`[3] on the Julia package registry. Every part of PYSR is written with scientific discovery in mind, as the entire point of creating this package was to enable the authors of this paper to use this tool in their own field for equation discovery. All of the requirements of discovering an empirical equation for science, as listed above, are satisfied by PYSR. We discuss this in detail in the following sections.

2 Methods

First, we give an in-depth discussion of the internal algorithm in PYSR in section 2.1, and then describe the software implementation in section 2.2. We then discuss a few additional implementation features in section 2.3, section 2.4, and section 2.5.

2.1 Algorithm

PYSR is a multi-population evolutionary algorithm with multiple evolutions performed asynchronously. The main loop of PYSR, operating on each population independently, is a classic evolutionary algorithm based on tournament selection for individual selection (first introduced in the thesis of [55] and generalized in [56]), and several mutations and crossovers for generating new individuals. Evolutionary algorithms for SR was first popularized in [9], and has dominated the development of flexible SR algorithms since. PYSR makes several modifications to this classic approach, some of which are motivated by results in recent work.

A simple evolutionary algorithm proceeds as follows:

1. Assume one has a population of individuals, a fitness function, and a set of mutation operators.

---

stants to derived constants might be arbitrarily complex.

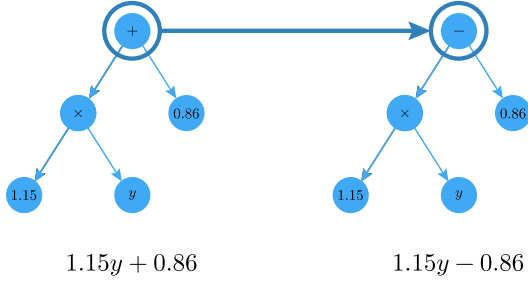$$1.15y + 0.86 \qquad\qquad 1.15y - 0.86$$

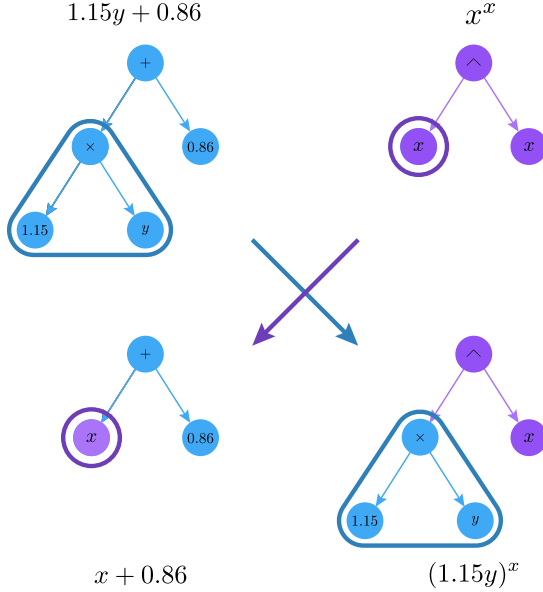Figure 1: A mutation operation applied to an expression tree.

Figure 2: A crossover operation between two expression trees.

2. Randomly select an n_s-sized subset of individuals from the population (e.g., classical tournament selection uses $n_s = 2$, but larger tournaments are also allowed).

3. Run the tournament by evaluating the fitness of every individual in that subset.

4. Select the fittest individual as the winner with probability p. Otherwise, remove this individual from the subset and repeat this step again. If there is one remaining, select it. Thus, the probability will roughly be $p, p(1-p), p(1-p)^2, \dots$ for the first, second, third fittest, and so on.

5. Create a copy of this selected individual,

and apply a randomly-selected mutation from a set of possible mutations.

6. Replace a member of the population with the mutated individual. Here, one would replace the weakest of the population or subset.

This algorithm is computationally efficient and allows for massive parallelization by splitting step 2. into groups of individuals which can evolve independently, with asynchronous "migration" between the groups. For the first of the modifications, we replace the eldest member of a population in step 5., rather than the individual with the lowest fitness, which is known as "age-regularized" evolution [57]. We record the age from the Unix time at which the individual was created during step 5.. Its inclusion in PySR was motivated by the impressive Neural Architecture Search results of [57, 58] — though it is worth noting similar regularizations used earlier in SR, e.g., [59] and [60] — which found this simple modification could prevent early convergence without hurting performance. This prevents the population from specializing too quickly and getting stuck in a local minimum of the search space. The implementation in [58] uses $p = 1$ and explores $4 \leq n_s \leq 64$, though here we allow p to be less than 1, among several other differences.

Modifications. Our algorithm in PySR, first released publicly in 2020 [61], makes three changes to this evolutionary algorithm. We apply simulated annealing [62] to step 4.: given a temperature $T \in [0, 1]$, a mutation is rejected with some probability related to the change in fitness after mutation. The probability for rejection is $p = \exp\left(\frac{L_F - L_E}{\alpha T}\right)$, for L_F and L_E the fitness of the mutated and original individual respectively, and α a hyperparameter. This allows the evolution to alternate between *high temperature* and *low temperature* phases, with the high temperature phases increasing diversity of individuals and the low temperature phases narrowing in on the fittest individuals. α controls the scale of temperature, with $\alpha \to \infty$ being equivalent to regular tournament selection, and $\alpha \to 0$ rejecting all mutations which lower fitness. Since the value of p also controls the diversity of the population, we can also couple the value of p to temperature, setting a larger value of p for a higher temperature, and a smaller value of p for
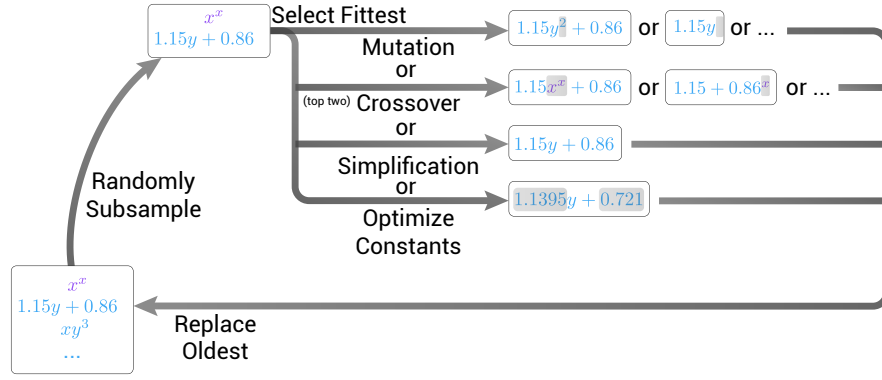
Figure 3: The inner loop of PySR. A population of expressions is randomly subsampled. Among this subsample, a tournament is performed, and the winner is selected for breeding: either by mutation, crossover, simplification, or explicit optimization. Examples of mutation and crossover operations are visualized in figs. 1 and 2.
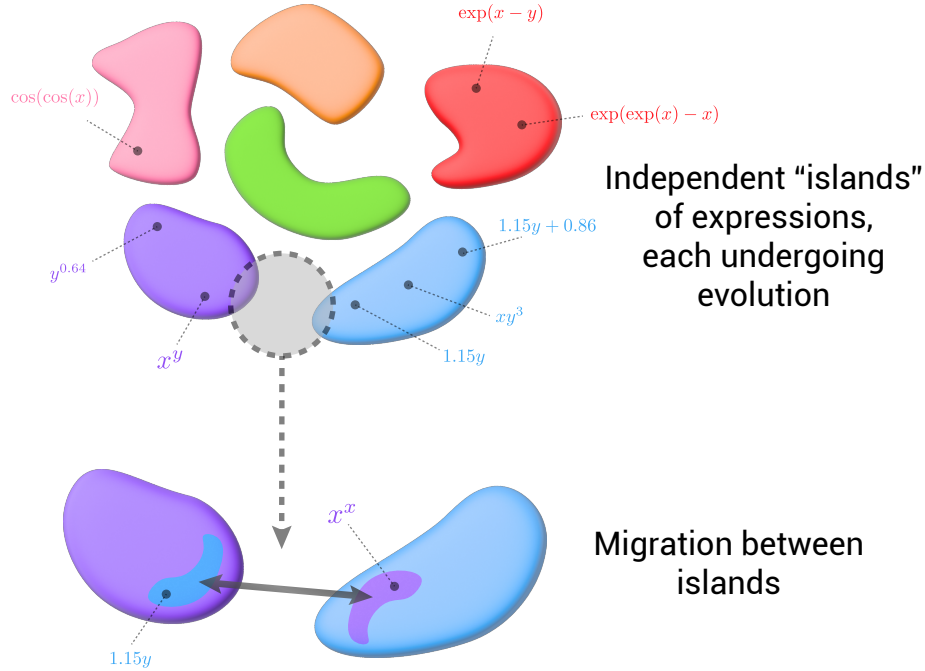

Figure 4: The outer loop of PySR. Several populations evolve independently according to the algorithm described in fig. 3. At the end of a specified number of rounds of evolution, migration between islands is performed.

a lower temperature. Simulated annealing is a simple yet powerful strategy for optimizing over discrete spaces, and we find in section 3 that simulated annealing can significantly speed up the search process.

The second modification we make to tournament selection is specific to the individuals we wish to consider: mathematical expressions. We embed the genetic algorithm inside an *evolve-simplify-optimize loop*. "Evolve" is a repeated application of tournament selection-based evolution for a set number of

mutations. In other words, this is an entire evolution of a population. "Simplify" here refers to equation simplification—equations are simplified to an equivalent form during the loop using a set of algebraic equivalencies (since this happens infrequently enough, it tends to not harm discovery by constraining the search space). The "Optimize" part of this algorithm is a few iterations of a classical optimization algorithm (BFGS [63] is used by default, though any optimizer in the `Optim.jl` package can be used [64]) to optimize constants in every equation explicitly. This part of the algorithm significantly improves the discovery of equations containing real constants, which is essential for practical application of SR to science. It has been shown since at least [65] that performing local gradient searches on numerical constants in expressions can vastly improve the performance of SR—we closely integrate an optimized and stochastic version of this.

The reason why several mutations (as well as crossovers) are performed before proceeding to the *simplify-optimize* stage is because some equations are only accessible via a redundant intermediate state. For example, $x * x - x * x$ would normally simplify to 0, but if the simplification is avoided, the next mutation might change one of the x's to y, resulting in $x * x - x * y$. Simplifying only occasionally is a way of allowing redundant but necessary intermediate expressions while at the same time reducing the size of the search space.

Finally, the third modification we make is the use of a novel *adaptive parsimony* metric. But first, we give a definition of complexity.

> **Complexity**: By default, complexity in PySR is equal to the *number of nodes in an expression tree*, regardless of each node's content.
> However, complexity in PySR is a completely user-configurable. We argue that the best definition for a "simple" expression is the one that is most interpretable by the user. An expression containing a dilogarithm may look commonplace to the particle physicist—and in fact be a great prior over the space of models for certain particle physics data—but extremely unusual to the fluid dynamicist, and perhaps not coincidentally a bad choice of prior for fluids data.

The traditional mechanism for penalizing complexity would be to use a constant *parsimony*, which would involve adding a loss term equal to the complexity of an expression times a constant factor:

$$\ell(E) = \ell_{\text{pred}}(E) + (\text{parsimony}) \cdot C(E), \quad (1)$$

for $C(E)$ the complexity of an expression E and ℓ_{pred} the predictive loss. Instead, we adaptively tune a per-complexity penalty such that the number of expressions at each complexity is approximately the same. This is expressed roughly as:

$$\ell(E) = \ell_{\text{pred}}(E) \cdot \exp(\text{frecency}[C(E)]), \quad (2)$$

where the "frecency" of $C(E)$ is some combined measure of the frequency and recency of expressions occurring at complexity $C(E)$ in the population. In PySR, we simply count the number of expressions with a moving window in time, and divide by a tunable constant. Thus, this is a simple way to punish complexities adaptively by how frequent they are in the population. This encourages the evolutionary search to explore the problem from simple, less-accurate expressions, as well as from complex, more accurate expressions. This qualitatively seems to help alleviate situations where the search specializes too early in the wrong functional form, and can only seem to make small iterations to it, and is discouraged from starting from scratch.

Pseudocode for the outer loop of PySR is given in algorithm 1, which includes the migration steps shown in fig. 4. Algorithm 2 gives pseudocode for the main *evolve-simplify-optimize loop*, formalizing parts of the cartoon in fig. 3. Algorithm 3 outlines the set of mutations, supplemented with simulated annealing. Finally, algorithm 4 describes the tournament selection strategy.

Additional Features. PySR includes a variety of additional features for various types of data and scenarios:

- *Noisy data.* PySR supports multiple strategies for working with noisy data:

  - First, PySR includes an optional denoising preprocessing step that optimizes a Gaussian process on the input dataset with a kernel of the form $k(x, x') = \sigma^2 \exp\left(-|x - x'|^2/2l^2\right) + \alpha\,\delta(x - x') + C$, a superposition of a Gaussian kernel (the standard kernel for interpolating datasets), a white noise kernel (to account for

Algorithm 1: PySR.

input	:	$\mathcal{X}$, the dataset to find expressions for
output	:	the best expressions at each complexity
param	:	n_p, the number of populations (=40)
param	:	L, the number of expressions in each population (=1000)
param	:	α_H, replacement fraction using expressions from H (=0.05)
param	:	α_M, replacement fraction using expressions from $\cup_i M_i$ (=0.05)

```
 1  function pysr(X)
 2      for i in range(n_p)  // [code]
 3          create set P_i containing L random expressions of complexity 3
 4          // ê.g., (3.2 + x_1) has size 3
 5          create empty set M_i // will store best expressions seen in P_i
 6      end
 7      create empty set H // will store best expressions overall
 8      for n in range(n_iter)  // [code]
 9          // the following loop is parallelized over workers:
10          for i in range(n_p)
11              // evolve-simplify-optimize:
12              P_i ← evolve(P_i, X)
13              for E in P_i
14                  simplify E
15                  optimize constants in E
16                  store updated E in P_i
17              end
18              M_i ← most accurate expression in P_i at each complexity
19              // (In actuality, M_i is updated throughout evolve)
20              H ← most accurate expression in M_i ∪ H at each complexity
21              // migration:
22              for E in P_i
23                  if rand() < α_H
24                      replace E in P_i with a random expression from H
25                  end
26                  if rand() < α_M
27                      replace E in P_i with a random expression from ∪_{j≠i}M_j
28                  end
29              end
30          end
31      end
32      return H
33  end
```

intrinsic noise), and a constant kernel (which modifies the mean of the Gaussian process). After this kernel is optimized, the Gaussian process is used to predict denoised targets for each input point, which are then passed to the main PySR algorithm.

– Second, PySR allows one to specify a set of weights for each input data point. This could be used, for instance, if a user knows the uncertainty in a measurement beforehand to be $\sigma > 0$, for σ the standard deviation of the measurement, and thus weights each data point in the loss by the signal-to-noise $1/\sigma^2$.

– Thirdly, the user can define a custom likelihood to optimize, which can also take into account weights. This is explained in the fol-

lowing point.

• *Custom losses.* PySR supports arbitrary user-defined loss functions, passed as either a string or a function object. These take a scalar prediction and target, and optionally a scalar weight, and should return a real number greater than 0. These per-point losses are then summed over the dataset. With this, the user is capable of defining arbitrary regression objectives, custom likelihoods (defined as log-likelihood of a single data point), and classification-based losses. This also allows one to define implicit objectives.

• *Custom operators.* One of the most powerful features of PySR is the ability to define custom operators. Different domains of science have

Algorithm 2: Evolution.

input	: P, a set of expressions	
input	: $\mathcal{X}$ as in algorithm 1	
output	: P, the evolved set of expressions	
param	: n_c, the number of mutations per evolve() call (=300000)	
param	: p_{cross}, the probability of crossover (=0.01)	

```
 1  function evolve(P, X)
 2      for k in range(n_c)  // [code]
 3          if rand() > p_cross  // [code]
 4              // mutation
 5              E ← tournament(P, X)
 6              T ← 1 - k/n_c  // annealing temperature
 7              E* ← mutate(E, T)
 8              replace oldest expression in P with E*
 9          else
10              do
11                  // crossover
12                  E_1 ← tournament(P, X)
13                  E_2 ← tournament(P, X)
14                  E*_1, E*_2 ← crossover(E_1, E_2)
15                  replace oldest two expressions in P with
                        E*_1 and E*_2
16              until satisfies_constraints(E*_1) and
17                  satisfies_constraints(E*_2)
18          end
19      end
20      return P
21  end
```

Algorithm 3: Mutations.

input	: E, an expression	
input	: T, the annealing temperature $\in [0, 1]$	
output	: mutated version of E	
param	: m_i, for $i = 1, \ldots, 8$, the probability weight of mutation type i	
param	: f, the constant perturbation scale (=1)	
param	: ϵ, the minimum perturbation (=0.1)	
param	: α, the temperature scale (=0.1)	

```
 1  function mutate(E, T)  // [code]
 2      adjust weights w_1, ..., w_8 based on constraints
 3      i ← random choice of 1, ..., 8 weighted by w_1, ..., w_8
 4      do
 5          E* ← copy(E)
 6          switch i
 7              case 1  // mutate constant
 8                  a ← (1 + f × T + ε)^(2×rand()−1)
 9                  if rand() < 0.5
10                      a ← −a
11                  end
12                  multiply random constant in E by a
13              case 2  // mutate operator
14                  randomly replace an operator in E with
15                      an operator of the same degree
16              case 3  // append/prepend node
17                  add random node to root or leaf of E
18              case 4  // insert node
19                  insert a random node inside E
20              case 5  // delete subtree
21                  replace a node and its children from E
22                      with a constant or variable
23              case 6  // simplify tree
24                  simplify E
25              case 7  // new tree entirely
26                  E ← random expression
27              case 8  // no mutation
28                  do nothing
29          end
30      until satisfies_constraints(E*)
31      C, C* ← complexity of expressions E, E*, respectively
32      L, L* ← accuracy of expressions E, E*, respectively
33      q_anneal ← exp(− (L*−L)/(α×T))
34      C ← complexity of expressions E
35      q_parsimony ← frecency[C]/frecency[C]
36      if rand() < q_anneal · q_parsimony
37          return E*
38      else
39          return E
40      end
41  end
```

functions unique to their field which appear in many formulae and hold a specific meaning. It is therefore very important that these operators be available in an SR library, and so PYSR allows custom user-defined unary or binary functions. So long as a function can be defined as either $f : \mathbb{R} \to \mathbb{R}$ or $\mathbb{R}^2 \to \mathbb{R}$, it can be used as a user-defined operator. PYSR does not treat built-in operators any differently than user-defined ones. Apart from simplification strategies, PYSR does not know the difference between the $+$ operator and a Bessel function.

- *Feature selection.* Similar to the Gaussian process preprocessing step for denoising, PYSR also uses a simple dimensionality reduction preprocessing step. Given a user-defined number of features to select, PYSR uses a gradient-boosting tree algorithm to first fit the dataset, then select the most important features. These features are fed into the main search loop.

- *Constraints.* PYSR allows various hard constraints to be specified for discovered expressions. These are enforced at every mutation: if a constraint is violated, the mutation is re-

jected. A few are available with the API:

- The maximum size of an expression (the number of instances of operators, variables, and constants), which gives an overall bound on the search space.

- The maximum depth of an expression, which can be used to control how deeply nested the resultant expression is.

Algorithm 4: Tournament selection.

input	: P, a population of expressions	
input	: $\mathcal{X}$ as in algorithm 1	
output	: a single expression (the winner of the tournament)	
param	: n_s, the tournament size (=12)	
param	: $p_{\text{tournament}}$, the probability of selecting the fittest individual (=0.9)	

```
 1  function tournament(P, X)  // [code]
 2  |     Q ← a random subset of size n_s of P
 3  |     while length(Q) > 1
 4  |     |     E ← get_fittest(Q)
 5  |     |     if rand() < p_tournament
 6  |     |     |     break
 7  |     |     end
 8  |     |     remove E from Q
 9  |     end
10  |     return E
11  end
12  function get_fittest(P)  // [code]
13  |     ℓ_best ← ∞
14  |     for E in P
15  |     |     C ← complexity of E
16  |     |     ℓ ← accuracy of E
17  |     |     // include adaptive parsimony:
18  |     |     ℓ ← ℓ × exp(frecency[C])
19  |     |     if ℓ < ℓ_best
20  |     |     |     ℓ_best ← ℓ
21  |     |     |     E* ← E
22  |     |     end
23  |     end
24  |     return E*
25  end
```

- The maximum size of a subexpression in a specific operator. This specifies the maximum size of an expression within an operator's arguments. For example, specifying that the $\wedge$ operator has maximum argument size of $(-1, 3)$ means that its base can have any size expression (indicated by -1), while the expression in its exponent can only have max size of up to 3. This operator-specific constraint can drastically reduce the complexity of discovered expressions.

- The maximum number of nests of a particular operator combination. For example, the nested constraint {sin: {sin: 0, cos: 1}} would indicate that $\sin(\cos(x))$ is allowed, but $\sin(\cos(\cos(x)))$ and $\sin(\sin(x))$ are not.

- *Additional data structures.* See deep learning interface section 2.3.

- *Integrals and dynamical systems.* See deep learning interface section 2.3.

2.2 Software implementation

The search algorithm itself underlying PYSR, as described in pseudocode in algorithms 1 to 4, is written in pure-Julia under the library name SYMBOLICRE-GRESSION.JL[4]. Julia boasts a combination of a high-level interface[5] with very high performance comparable to languages such as C++ and Rust[6]. However, the key advantage of using Julia for this work is the fact that it is a just-in-time compiled (JIT) language. This allows PYSR to make use of optimizations not possible with a statically compiled library: for instance, compiling user-defined operators and losses into the core functions. The most significant advantage of using JIT compilation for PYSR, in terms of performance, is that operators can be fused together into single compiled operators. For example, if a user declares + and − as valid operators, then PYSR will compile a SIMD-enabled kernel for +, −, as well as their combination: (a − b) + c, and so on. This happens automatically for every single combination of operators up to a depth of two operators, even for user-defined operators. Simply by fusing operators in this way, the expression evaluation code, which remains the bottleneck in PYSR, experiences a significant speedup. Because Julia is JIT compiled, these operators need not be pre-defined in the library: they will be just as performant as if they were.

PYSR exposes a simple Python frontend API to this pure-Julia backend search algorithm. This frontend takes the popular style of the scikit−learn machine learning library [66]:

---

[4]https://github.com/MilesCranmer/SymbolicRegression.jl
[5]docs.julialang.org/en/v1/
[6]julialang.org/benchmarks/

```python
from pysr import PySRRegressor

# Declare search options:
model = PySRRegressor(
    model_selection="best",
    unary_operators=["cos", "sin"],
    binary_operators=["+", "-", "/", "*"],
)


# Load the data
X, y = load_data()
# X shape: (n_rows, n_features)
# y shape: (n_rows) or (n_rows, n_targets)

# Run the search:
model.fit(X, y)

# View the discovered expressions:
print(model)

# Evaluate, using the 5th expression along
# the Pareto front:
y_predicted = model.predict(X, 5)
# (Without specify `5`, it will select an
↪   expression
# which balances complexity and error)
```

Parallelization   PYSR is parallelized by populations of expressions which evolve independently, which was described as the basis of allowing large-scale parallelization in [56]. In this framework, a population of expressions is dispatched to a worker (either a thread, if single-node, or a process, if multi-node), where it then evolves over many iterations. Following a batch of iterations, this population is returned to the head worker, which performs several tasks: (1) record the best expressions, at each complexity, found in the population as part of a global "hall of fame;" (2) record the current expressions so they can "migrate" to other populations; and, finally, (3) randomly "migrate" expressions from other populations, and the global hall of fame, to this population. After these steps are finished, this population is once again dispatched to a worker where it undergoes further evolution. This parallelization is asynchronous, and different populations may complete evolution at different speeds without slowing down others. The technique

is largely similar to the method described in [11], although here we have modified the migration step to also migrate based on a global, permanent hall of fame, rather than exclusively migrating current population members. Reintroducing hall of fame members can significantly speed up the search.

2.3 Interfaces

PYSR has export functionality to several popular Python libraries and other formats. PYSR by default will export any discovered expression to SymPy [67], which, by extension, can convert expressions to LaTeX for including in research reports. The mechanism for this functionality is fairly simple: using a string representation of the recovered expression, PYSR replaces symbol names with the SymPy equivalents (e.g., cos would become sympy.cos), and evaluates it with the Python interpreter. As a caveat of this, the user must define any custom operator using SymPy functions, in addition to Julia functions.

A similar technique—evaluating the string representation for an expression—is used for exporting expressions to a callable format in NumPy, PyTorch, and JAX [68, 69, 70]—each of which take vector input. For NumPy, constants in an expression are embedded inside the callable function. However, for PyTorch and JAX, since one may wish to reoptimize the constants in an expression (for example, as is done in [71]), the constants are trainable. In PyTorch, this amounts to creating a PyTorch nn.Parameter for the constants, which causes PyTorch to track gradients with respect to the parameters, so they can be trained via gradient descent. Since JAX is a functional language, the parameters are instead exported as a vector, which the user then will pass into the callable function. This is similar to how deep learning frameworks in JAX are trained.

2.4 Custom Operators

The occurrence rate of a particular mathematical operator in a set of models varies by the scientific field those models describe. For example, models in Epidemiology commonly use exponentials, indicating exponential growth or decay of a contagious disease, but may rarely use a Bessel function which would be more common in in classical physics applications. The entire reason that SR produces interpretable

models is that the generated expressions make use of operators which are common in a particular field, and a domain scientist may be able to see connections with existing models. In some ways, this process relates to language: black box machine learning models represent functions in a language uninterpretable to a scientist, whereas SR uses the language in place. Furthermore, due to the modular nature of scientific modelling—many complex models are built on-top of existing simple models for particular subsystems—it also makes sense from a standpoint of improving model accuracy to make use of common operators (for an interesting explicit example of this, see [26]).

PySR does not, per se, have any built-in operators. Due to the just-in-time compiled nature of Julia, any real scalar function from the entirety of the Julia Base language is available, and can be compiled into the search code at runtime. This includes commonly-used operators such as +, −, *, /, ∧, exp, log, sin, cos, tan, abs, and many others.

Furthermore, since many domains of science have operators that are unique to their field, it is possible to pass an arbitrary Julia function as an operator, whether it be a binary operator (with two arguments) or a unary operator (with one argument). Any function of the form $f : \mathbb{R} \to \mathbb{R}$ or $\mathbb{R}^2 \to \mathbb{R}$, whether continuous or not, can be used as a user-defined operator. The function need only be designed for scalar input and output, and PySR will use Julia to automatically compile and vectorize it, generating SIMD (Single Instruction, Multiple Data) instructions when possible. In PySR, an example of this would be:

```
op = "special(x, y) = cos(x) * (x + y)"
model = PySRRegressor(binary_operators=[op])
```

where the string `special(x, y) = cos(x) * (x + y)` is Julia code giving a function definition. This would define a binary operator `special` that would be compiled into the search code. To enable custom operators to be defined in the various export functionality, the user must also define equivalent operators in SymPy (here, `lambda x, y: sympy.cos(x) * (x + y)`), as well as JAX or PyTorch versions if the user wishes to export to those frameworks as well.

As an example in science, it is very common in astrophysics to see "broken power laws": power laws whose exponent takes on different values in the parameter space. This could be defined in PySR by enabling the power law operator ∧, and then giving the string `cond(x, y) = x < 0 ? 0 : y`, which defines a conditional branch of an expression given some expression defined in the `x` variable (using the ternary operator `condition ? value1 : value2`). For example, `pow(x, cond(x – 5, 3.4) – 2.1)` would define the broken power law:

$$\begin{cases} x \geq 5, & x^{1.3} \\ x < 5, & x^{4.5}. \end{cases}$$

The importance of defining custom operators is that there is no standard set of operators which the library is specifically tuned for; any operator common in a particular field is feasible to implement (so long as it can take 1-2 scalar arguments).

2.5 Custom Losses

In many machine learning toy datasets for benchmarking regression algorithms, Mean-Square-Error (MSE or L_2 loss) is typically used as a learning objective [72]. In a Bayesian framework, MSE is equivalent to assuming every data point is Gaussian distributed, with equal variance per point. Minimizing MSE is equivalent to maximizing the Gaussian log-likelihood. However, in science, one typically works with a likelihood that is very specific to a particular problem, and this is often non-Gaussian. Therefore, it is important for an SR package to allow for custom loss functions. PySR implements this in a way that is very similar to that of custom operators (see section 2.4). Given a string such as "`loss(x, y) = abs(x – y)`", PySR will pass this to the Julia backend, which will automatically vectorize it and use it as a loss function throughout the search process. In a Bayesian context, this would allow one to define arbitrary likelihoods, even for very complex branching logic. This also works for weighted losses, such as "`loss(x, y, w) = abs(x – y) * w`"

3 Evaluation

In section 1, we discussed practical issues with discovering symbolic expressions in the sciences. Here, we present a comparison of PySR with existing tools in terms of addressing these concerns.

3.1 High Level Comparison

First, we present a high level comparison of the various SR tools available. These qualitative points are made using the desirable features given in section 1.

		PySR	Eureqa	GPLearn	AI Feynman	Operon	DSR	PySINDy	EQL	QLattice	SR-Transformer	GP-GOMEA	Symbolic Distillation*
Scalability	Compiled	✓	✓	✗	✗	✓	✗	✗	✓	✓	✓	✓	-
	Multi-core	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
	Multi-node	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-
	GPU-capable	✗	✗	✗	*I	✗	✗	✓	✓	✗	✓	✗	✓
Practicality	No pre-training	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	-
	Denoising	✓	✓	✗	✗	✗	✗	*II	✗	?	✗	✗	✓
	Feature selection	✓	✓	✗	✓	✗	✓	*II	✗	✓	✗	✗	✓
	Differential equations	✗	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
	High-dimensional	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓
	Full Pareto curve	✓	✓	✓	✗	✓	✓	*II	✗	✓	✗	✓	✗
Interfacing	API	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	-
	SymPy Interface	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	-
	Deep Learning export	✓	✗	✗	✗	✗	✗	✗	*III	✗	*III	✗	-
Extensibility	Expressivity score	4	5	4	3	3	3	1b	2	3	1a	3	6
	Open-source	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
	Real Constants	✓	✓	✓	✗	✓	✓	*II	✓	✓	✓	✓	-
	Custom operators	✓	✗	✓	✗	✗	✗	*II	✗	✗	✗	✗	-
	Discontinuous operators	✓	✓	✓	✗	✗	✗	*II	✗	✗	✗	✗	-
	Custom losses	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓
	Symbolic Constraints	✓	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓
	Custom complexity	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	-
	Custom types	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
-	Citation	[self]	[11]	-	[73]	[44]	[27]	[74]	[34]	[75]	[30]	[21]	[23]
	Code	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓

Expressivity scores: (1a) Inflexible basis of fixed expressions, combined in a linear sum. (1b) Pre-trained on equations generated from limited prior. (2) Flexible basis of expressions, with variable internal coefficients. (3) Any scalar tree, with binary and unary operators. (4) Any scalar tree, with custom operators allowed. (5) Any scalar tree, with n-ary operators. (6) Scalar/vector/tensor expressions of any arity.

*   Note that the "Symbolic Distillation" method from [23] is not an algorithm itself; it can be applied to any SR technique. Applying this general method to a specific technique will inherit a ✓ from the Symbolic Distillation column, if given. However, in general, this technique is easiest with those methods which have deep learning export.

*I   Only the symmetry discovery module is GPU-capable.

*II   Conceptually different, as is a linear basis of static nonlinear expressions.

*III   Is itself a neural network.

Table 1: Note that many instances of ✗ are **purely software limitations**. For example, most non-compiled algorithms *could* support custom losses and operators, but few make this easily configurable via an API, which is important for practical use in science. Open source code can be found by clicking on each 🔓 icon.

3.2 EmpiricalBench: Empirical Science Symbolic Regression Benchmark

Existing SR benchmarks have certain simplifications compared to the datasets used to historically discover well-known empirical equations. When discovering a new expression, one does not actually know the physical constants in the expression, and one must have to learn real constants. For example, in the Feynman benchmark dataset of [73], all expressions are listed with the associated physical constants—but these constants originally had to be discovered along with the equation. Similarly, in the commonly-used synthetic "Nguyen" benchmark [76], which includes expressions such as $F_9 = \sin(x) + \sin(y^2)$, there are no non-integral constants. The "SRBench" competition of [77] was meticulous in its comparisons of different methods, but only contained a single real-world task (with an unknown ground truth), with all other expressions being synthetic. This benchmark improved on others in that it included tunable amounts of noise. However, the noise models are very simple: perfectly Gaussian and including no heteroscedasticity. Although such synthetic benchmarks have their uses in SR research, these types of benchmarks are often not entirely indicative of the challenges faced in real-world scientific discovery. Therefore, here we introduce a new benchmark which attempts to accurately portray the empirical equation discovery step of science. Every equation in this dataset is a real empirical expression that a scientist has at one point discovered from experimental, noisy, and imperfect data.

If an algorithm is to be considered to be useful for equation discovery in the sciences, it should be able to discover such relations. Data for this benchmark is shown in fig. 5, with the equations and citations given in table 2. Where original data was used, it was either: (a) taken from an original dataset available publicly, or (b) digitized from a table or plot using `WebPlotDigitizer` [78]. For example, the data for Hubble's Law was manually extracted from the original 1929 paper [79] Where original data was not available, data was generated from the equation with realistic ranges of variables, with noise applied. The code and datasets used to generate this benchmark and evaluate are available in the paper repository, and are based on a fork of the SRBench competition [77, 80].

As noted in the "Regression Target" column, many of these laws which are easily expressed in logarithmic units are instead posed as a problem in observed units. The search algorithm must discover this better unit by itself.

Many existing benchmarks for physical expression searches are based on theoretically-derived equations rather than empirically-observed and formulated. While these are likely useful for benchmarking some parts of the SR landscape, they do not target the aspect which SR would ultimately be used for: empirical discovery. Thus, all the equations in this dataset were first empirically formulated before (and if) they were theoretically-derived. Thus, the physical variables here directly correspond to observables. Furthermore, this dataset does not include any relevant physical constants—the algorithm must find these automatically, as did the scientist who discovered the equation.

3.2.1 Results on EmpiricalBench

We fork the SRBench competition repository [77], as the authors have managed to perform the impressive task of aggregating a large variety of existing methods into a single repository with a common API. However, here we aim to study the entire Pareto front of each algorithm, rather than only a single expression, as was done in SRBench. By reviewing the discovered expressions over the entire Pareto front, we are not sensitive to the somewhat arbitrary choice of selection criteria, which differs by method. We can more accurately gauge performance of the algorithms themselves. Thus, we modify the interface of every algorithm to return a Pareto front of expressions, or, in the cases where it was unavailable, instead we return a list of several candidate expressions. We were able to successfully test the algorithms PySR, Operon [44], DSR [27], EQL [34], QLattice [75], and SR-Transformer [30] on EmpiricalBench. Other codes were either not included in SRBench already, incompatible with our tests, or were otherwise unable to be configured on our system after significant effort.

We allow every method to use 8 cores on an AMD Rome CPU running Rocky Linux, and allow them to search for up to 1 hour. These are similar constraints to the SRBench competition, which had authors individually tune their codes for such a setting, and thus
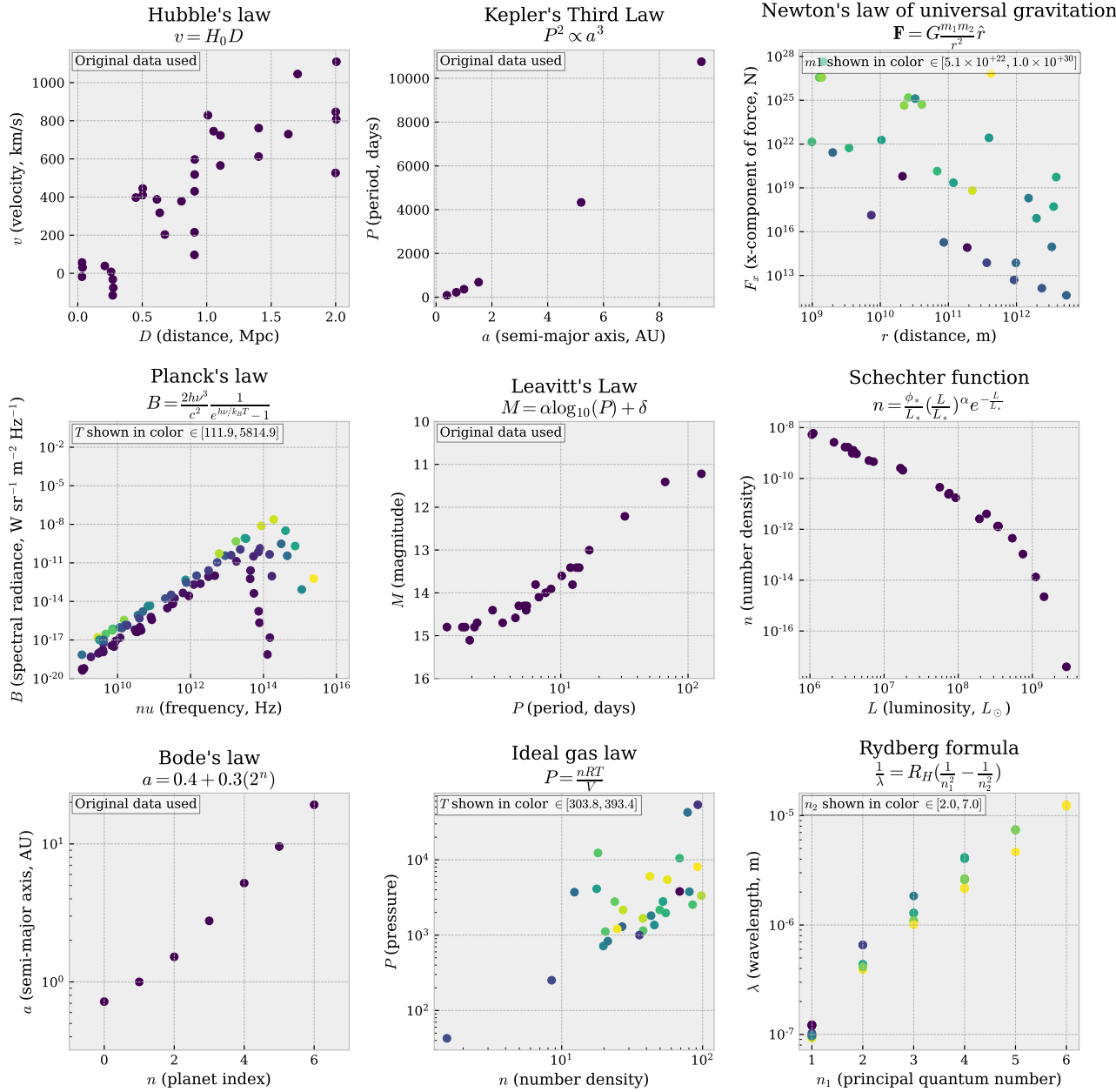
Figure 5: Visualization of all the data in EmpiricalBench, an SR benchmark for science. Color is used to denote additional variables in the cases of relations which depend on more than two inputs. Original data in the discovery of each law is used where easily available. Otherwise, data is generated from the formula with realistic ranges of variables, with a level of noise applied.

we consider these settings most fair.

For each of the 9 equations in EmpiricalBench, we run 5 trials for each of the 6 algorithms, and record the Pareto front of each trial. Each algorithm is fed the entire dataset for training. This is because our *test is the output expression*, rather than a separate test dataset. Following this, each of the Pareto fronts was analyzed, by eye, to see whether the true expression was contained within it. This was performed manually, as oftentimes automated equality checking with `sympy` produced incorrect results. Furthermore, checking expressions by eye allows for small errors in

Name	Law	Early Citation
Hubble's law	$v = H_0 D$	[79]
Kepler's Third Law	$P^2 \propto a^3$	[81]
Newton's law of universal gravitation	$\mathbf{F} = G\frac{m_1 m_2}{r^2}\hat{r}$	[82]
Planck's law	$B = \frac{2h\nu^3}{c^2}\frac{1}{e^{h\nu/k_B T}-1}$	[83]
Leavitt's Law	$M = \alpha \log_{10}(P) + \delta$	[84]
Schechter function	$n = \frac{\phi_*}{L_*}(\frac{L}{L_*})^\alpha e^{-\frac{L}{L_*}}$	[85]
Bode's law	$a = 0.4 + 0.3(2^n)$	[86]
Ideal gas law	$P = \frac{nRT}{V}$	[87]
Rydberg formula	$\frac{1}{\lambda} = R_H(\frac{1}{n_1^2} - \frac{1}{n_2^2})$	[88]

Table 2: Expressions in the EmpiricalBench and associated with the datasets in fig. 5. Each of these expressions was originally empirically discovered.

	PySR	Operon	DSR	EQL	QLattice	SR-Transformer
Hubble	5/5 (5, 0, 0, 0)	0/5 (0, 5, 0, 0)	1/5 (1, 0, 4, 0)	0/5 (0, 0, 0, 5)	0/5 (0, 5, 0, 0)	0/5 (0, 0, 0, 5)
Kepler	5/5 (5, 0, 0, 0)	0/5 (0, 5, 0, 0)	4/5 (4, 1, 0, 0)	0/5 (0, 0, 2, 3)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Newton	5/5 (5, 0, 0, 0)	1/5 (1, 2, 0, 2)	1/5 (1, 0, 4, 0)	0/5 (0, 0, 5, 0)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Planck	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 1, 4)	0/5 (0, 0, 5, 0)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Leavitt	5/5 (5, 0, 0, 0)	0/5 (0, 0, 0, 5)	5/5 (5, 0, 0, 0)	0/5 (0, 0, 5, 0)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Schechter	5/5 (5, 0, 0, 0)	5/5 (5, 0, 0, 0)	5/5 (5, 0, 0, 0)	0/5 (0, 0, 4, 1)	5/5 (5, 0, 0, 0)	0/5 (0, 0, 0, 5)
Bode	5/5 (5, 0, 0, 0)	3/5 (3, 0, 0, 2)	1/5 (1, 0, 3, 1)	0/5 (0, 0, 4, 1)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Ideal Gas	5/5 (5, 0, 0, 0)	0/5 (0, 0, 0, 5)	5/5 (5, 0, 0, 0)	0/5 (0, 0, 4, 1)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)
Rydberg	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 5, 0)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)	0/5 (0, 0, 0, 5)

Table 3: Results of each algorithm on EmpiricalBench. The fraction given is the number of correct expressions rediscovered, divided by the number of total trials. In parentheses, a detailed itemization of the five trials is given, in the order: 1) the number of correct rediscoveries, 2) the number of nearly-correct rediscoveries, 3) the number of runs which failed to produce a well-defined expression, and 4) the number of incorrect rediscoveries.

recovered expressions to be ignored, so long as the overall functional form is correct. It is also superior to numerically checking expressions, as overcomplicated but accurate expressions can be correctly penalized.

For this analysis, each Pareto front was assigned to one of four categories: (1) *correct* if the Pareto front contained the true expression, (2) *almost* if one of the expressions in the Pareto front was off of the true expression by a constant factor somewhere, (3) *failed*, if the search produced no results (e.g., due to numerical instability, segmentation fault, domain errors, etc.), and (4) *incorrect* if the true expression was not found in the Pareto front. All raw data with all discovered Pareto fronts for this analysis is available online at github.com/MilesCranmer/pysr_paper.

We emphasize that the stability of an SR algorithm and software implementation itself has central importance to practical use in science, and therefore, unstable runs which either produce an undefined expression, or crash, are *included* in the results dataset and the primary score. However, one should also note that because we are the authors of PySR, we are therefore more likely to be running in a stable environment than, say, a package which we are unfamiliar with. For example, as seen in table 3, DSR sees improved results if one ignores the failed runs, although there were still some expressions it struggled on. Thus, detailed itemization of these results is given in table 3, to allow for a more nuanced understanding of the results. Also note that this benchmark does not explicitly measure other metrics which are considered in other benchmarks, such as number of evaluations (where, e.g., genetic algorithms are very inefficient), nor does it consider the numerical accuracy of the expressions.

3.3 Discussion

It is also very important to note that these results are from the datasets specific to this competition. Many of these algorithms may do very well against synthetic datasets generated for each expression. For example, Hubble's law is simply a linear relation, so it could be surprising that many algorithms are not able to find it. However, to perform well at the benchmark, each algorithm must find these expressions from the noisy and sometimes biased datasets shown in fig. 5.

What is also interesting is that the two pure deep learning approaches, EQL and SR-Transformer, recovered the fewest expressions out of all tested algorithms. One of these, EQL, learns the expression in an online fashion, while SR-Transformer performs fast inference based on pre-trained weights. SR-Transformer, in particular, is pre-trained on billions of randomly-generated synthetic expressions on a cluster of GPUs for weeks. However, the "untrained" algorithms written using classic heuristics: PySR, Operon, DSR, and QLattice, all out-performed these modern deep learning models. Perhaps this is an insight into the difficulty of preparing deep learning systems for real-world data and unexpected "long-tail" examples, whereas handwritten standard algorithms will often perform just as well on unexpected examples. In SR especially, the space of potential expressions is so massive and nonlinear that interpolation techniques might suffer; consider that x/y and $x \times y$ are just a single mutation away, but infinitely distinct in their data representation. In spite of this, while pure deep learning strategies appears to require improvements in architecture or training to be useful in a real world SR setting, its success in, e.g., [23, 27, 30] shows there is indeed strong potential long-term for such hybrid methods, and these directions should continue to be pursued.

4 Conclusion

In this paper, we have presented PySR, an open-source library for practical symbolic regression. We have developed PySR with the aim of democratizing and popularizing symbolic regression in the sciences. By leveraging a powerful multi-population evolutionary algorithm, a unique evolve-simplify-optimize loop, a high-performance distributed backend, and integration with deep learning packages, PySR is capable of accelerating the discovery of interpretable symbolic models from data.

Furthermore, through the introduction of a new benchmark, EmpiricalBench, we have provided a means to quantitatively evaluate the performance of symbolic regression algorithms in scientific applications.

Our results indicate that, despite advances in deep learning, classic algorithms build on evolution and other untrained symbolic techniques such as

PySR, Operon, DSR, and QLattice, still outperform pure deep-learning-based approaches, EQL and SR-Transformer, in discovering historical empirical equations from original and synthetic datasets. This highlights the challenges faced by deep learning methods when applied to real-world data with its biases and heteroscedasticity, and the nonlinearity of the space of expressions.

However, we emphasize that deep learning techniques—inclusive of generative models like [30], reinforcement learning methods such as [27], and symbolic distillation [23]—all hold potential for improving symbolic regression, and encourage further exploration of such hybrid methods.

Over the past several years since the release of PySR in 2020, there have been a number of exciting PySR applications to discover new models in various subfields. There are too many to list here in detail, but we list some examples: [89] use PySR to discover a new symbolic parameterization for cloud cover formation; [90] use PySR alongside other ML techniques to discover electron transfer rules in various materials; [91, 92, 93] combine PySR with dimensional analysis to discover new astrophysical relations; [94] use PySR in economics, to find effective rules governing international trade; [95] demonstrates how to use PySR to extract dynamical equations learned by a neural differential equation; and, finally, [96] use PySR to discover interpretable population models of gravitational wave sources.

In some ways this list of recent applications provides the strongest validation yet of the science and user-focused approach we argued for in section 1, as PySR has already been applied successfully to model discovery in a variety of fields. It is our hope that PySR will continue to grow as a community tool, and provide value to researchers, helping discover interpretable symbolic relationships in data and ultimately leading to new insights, theories, and advancements in their respective fields.

References

[1] Stephen Hawking. *On the Shoulders of Giants: The Great Works of Physics and Astronomy*. Running, Philadelphia, Pa.; London, 2004.

[2] Max Planck. *Über eine Verbesserung der*

Wien'schen Spectralgleichung. Friedr. Vieweg & Sohn, 1900.

[3] Marco Virgolin and Solon P. Pissis. Symbolic Regression is NP-hard, July 2022.

[4] Donald Gerwin. Information processing, data inferences, and scientific generalization. *Behavioral Science*, 19(5):314–325, 1974.

[5] Pat Langley. BACON: A production system that discovers empirical laws. In *IJCAI*, 1977.

[6] Pat Langley. Rediscovering physics with BACON.3. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'79, pages 505–507, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc.

[7] Pat Langley, Gary L. Bradshaw, and Herbert A. Simon. BACON.5: The discovery of conservation laws. In *IJCAI*, 1981.

[8] Pat Langley and Jan M. Zytkow. Data-driven approaches to empirical discovery. *Artificial Intelligence*, 40(1):283–312, 1989.

[9] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, June 1994.

[10] Josh Bongard and Hod Lipson. From the Cover: Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Science*, 104(24):9943–9948, June 2007.

[11] Michael Schmidt and Hod Lipson. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324(5923):81–85, April 2009.

[12] M. Schmidt and H. Lipson. Symbolic regression of implicit equations. *Genetic Programming Theory and Practice VII*, pages 73–85, 2010.

[13] S. Wagner and M. Affenzeller. HeuristicLab: A Generic and Extensible Optimization Environment. In Bernardete Ribeiro, Rudolf F. Albrecht, Andrej Dobnikar, David W. Pearson, and Nigel C. Steele, editors, *Adaptive and Natural Computing Algorithms*, pages 538–541, Vienna, 2005. Springer.

[14] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature PPSN VI*, Lecture Notes in Computer Science, pages 849–858, Berlin, Heidelberg, 2000. Springer.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[16] J.W. Davidson, D.A. Savic, and G.A. Walters. Symbolic and numerical regression: Experiments and applications. *Information Sciences*, 150(1):95–117, 2003.

[17] Kyle Cranmer and R. Sean Bowman. PhysicsGP: A Genetic Programming approach to event selection. *Computer Physics Communications*, 167(3):165–176, May 2005.

[18] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-Lexicase Selection for Regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 741–748, New York, NY, USA, July 2016. Association for Computing Machinery.

[19] William La Cava, Thomas Helmuth, Lee Spector, and Jason H. Moore. A probabilistic and multi-objective analysis of lexicase selection and epsilon-lexicase selection, April 2018.

[20] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. 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*, GECCO '17, pages 1041–1048, New York,

NY, USA, July 2017. Association for Computing Machinery.

[21] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter A. N. Bosman. Improving Model-based Genetic Programming for Symbolic Regression of Small Expressions. *Evolutionary Computation*, 29(2):211–237, June 2021.

[22] Miles D. Cranmer, Rui Xu, Peter Battaglia, and Shirley Ho. Learning Symbolic Physics with Graph Networks. *ML4Physics Workshop @ NeurIPS 2019*, November 2019.

[23] Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering Symbolic Models from Deep Learning with Inductive Biases. *NeurIPS*, June 2020.

[24] Miles Cranmer, Can Cui, Drummond B Fielding, Shirley Ho, Alvaro Sanchez-Gonzalez, Kimberly Stachenfeld, Tobias Pfaff, et al. Disentangled Sparsity Networks for Explainable AI. *Workshop on Sparse Neural Networks*, page 7, July 2021.

[25] Ying Jin, Weilin Fu, Jian Kang, Jiadong Guo, and Jian Guo. Bayesian Symbolic Regression. January 2020.

[26] Roger Guimerà, Ignasi Reichardt, Antoni Aguilar-Mogas, Francesco A. Massucci, Manuel Miranda, Jordi Pallarès, and Marta Sales-Pardo. A Bayesian machine scientist to aid in the solution of challenging scientific problems. *Science Advances*, 6(5), 2020.

[27] Brenden K. Petersen, Mikel Landajuela, T. Nathan Mundhenk, Claudio P. Santiago, Soo K. Kim, and Joanne T. Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients, April 2021.

[28] Li Li, Minjie Fan, Rishabh Singh, and Patrick Riley. Neural-guided symbolic regression with asymptotic constraints. *arXiv preprint arXiv:1901.07714*, 2019.

[29] Stéphane d'Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep Symbolic Regression for Recurrent Sequences, June 2022.

[30] Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. April 2022.

[31] Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. In *Advances in Neural Information Processing Systems*, volume 33, pages 4860–4871. Curran Associates, Inc., 2020.

[32] Ziming Liu and Max Tegmark. AI poincaré: Machine learning conservation laws from trajectories. *arXiv e-prints*, page arXiv:2011.04698, November 2020.

[33] Sebastian J. Wetzel, Roger G. Melko, Joseph Scott, Maysum Panju, and Vijay Ganesh. Discovering symmetry invariants and conserved quantities by interpreting siamese neural networks. *Physical Review Research*, 2(3):033499, September 2020.

[34] Subham Sahoo, Christoph Lampert, and Georg Martius. Learning Equations for Extrapolation and Control. volume 80 of *Proceedings of Machine Learning Research*, pages 4442–4450, Stockholmsmässan, Stockholm Sweden, July 2018. PMLR.

[35] Steven Atkinson, Waad Subber, Liping Wang, Genghis Khan, Philippe Hawi, and Roger Ghanem. Data-driven discovery of free-form governing differential equations. *arXiv preprint arXiv:1910.05117*, 2019.

[36] Andrew Slavin Ross, Ziwei Li, Pavel Perezhogin, Carlos Fernandez-Granda, and Laure Zanna. Benchmarking of machine learning ocean subgrid parameterizations in an idealized model, October 2022.

[37] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Trans-*

actions on Evolutionary Computation, 5(1):17–26, February 2001.

[38] Aytac Guven. Linear genetic programming for time-series modelling of daily flow rate. *Journal of Earth System Science*, 118(2):137–146, April 2009.

[39] He Ma, Arunachalam Narayanaswamy, Patrick Riley, and Li Li. Evolving symbolic density functionals. *Science Advances*, 8(36):eabq0279, September 2022.

[40] Douglas Mota Dias and Marco Aurélio C. Pacheco. Describing Quantum-Inspired Linear Genetic Programming from symbolic regression problems. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, June 2012.

[41] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[42] Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science Advances*, 3(4):e1602614, 2017.

[43] Kathleen Champion, Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Data-driven discovery of coordinates and governing equations. *arXiv e-prints*, page arXiv:1904.02107, March 2019.

[44] Bogdan Burlacu, Gabriel Kronberger, and Michael Kommenda. Operon C++: An efficient genetic programming framework for symbolic regression. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, GECCO '20, pages 1562–1570, New York, NY, USA, July 2020. Association for Computing Machinery.

[45] Trent McConaghy. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. In Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors, *Genetic Programming Theory and Practice IX*, Genetic and Evolutionary Computation, pages 235–260. Springer, New York, NY, 2011.

[46] Bethany Lusch, J. Nathan Kutz, and Steven L. Brunton. Deep learning for universal linear embeddings of nonlinear dynamics. *Nature Communications*, 9:4950, November 2018.

[47] Gert-Jan Both, Subham Choudhury, Pierre Sens, and Remy Kusters. DeepMoD: Deep learning for Model Discovery in noisy data. 2019.

[48] Zhao Chen, Yang Liu, and Hao Sun. Deep learning of physical laws from scarce data. 2020.

[49] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, and Ali Ramadhan. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[50] Alison Cozad and Nikolaos V. Sahinidis. A global MINLP approach to symbolic regression. *Mathematical Programming*, 170(1):97–119, July 2018.

[51] Fabricio Olivetti de Franca. A Greedy Search Tree Heuristic for Symbolic Regression. *Information Sciences*, 442–443:18–32, May 2018.

[52] Cristina Cornelio, Sanjeeb Dash, Vernon Austel, Tyler Josephson, Joao Goncalves, Kenneth Clarkson, Nimrod Megiddo, et al. AI Descartes: Combining Data and Theory for Derivable Scientific Discovery. *arXiv:2109.01634 [cs]*, October 2021.

[53] A. M. Price-Whelan, B. M. Sip'ocz, H. M. G"unther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, et al. The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *aj*, 156:123, September 2018.

[54] Astropy Collaboration, Adrian M. Price-Whelan, Pey Lian Lim, Nicholas Earl, Nathaniel Starkman, Larry Bradley, David L.

Shupe, et al. The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package. *The Astrophysical Journal*, 935:167, August 2022.

[55] Anne Brindle. *Genetic Algorithms for Function Optimization*. PhD thesis, 1980.

[56] David E. Goldberg and Kalyanmoy Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In GREGORY J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, volume 1, pages 69–93. Elsevier, January 1991.

[57] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized Evolution for Image Classifier Architecture Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4780–4789, July 2019.

[58] Esteban Real, Chen Liang, David So, and Quoc Le. AutoML-Zero: Evolving Machine Learning Algorithms From Scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR, November 2020.

[59] Gregory S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 815–822, New York, NY, USA, July 2006. Association for Computing Machinery.

[60] Michael D. Schmidt and Hod Lipson. Age-fitness pareto optimization. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 543–544, New York, NY, USA, July 2010. Association for Computing Machinery.

[61] Miles Cranmer. PySR: Fast & Parallelized Symbolic Regression in Python/Julia. Zenodo, September 2020.

[62] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, May 1983.

[63] C. G. Broyden. The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, March 1970.

[64] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24):615, 2018.

[65] Alexander Topchy and W. F. Punch. Faster genetic programming based on local gradient search of numeric leaf values. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 155–162, San Francisco, CA, USA, July 2001. Morgan Kaufmann Publishers Inc.

[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[67] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[68] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[70] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, et al. JAX: composable transformations of Python+NumPy programs, 2018.

[71] Pablo Lemos, Niall Jeffrey, Miles Cranmer, Peter Battaglia, and Shirley Ho. Rediscovering Newton's gravity and Solar System properties using deep learning and inductive biases. *In submission*, 2022.

[72] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on tabular data?, July 2022.

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

[74] Alan A. Kaptanoglu, Brian M. de Silva, Urban Fasel, Kadierdan Kaheman, Andy J. Goldschmidt, Jared L. Callaham, Charles B. Delahunt, et al. PySINDy: A comprehensive Python package for robust sparse system identification. November 2021.

[75] Kevin René Broløs, Meera Vieira Machado, Chris Cave, Jaan Kasak, Valdemar Stentoft-Hansen, Victor Galindo Batanero, Tom Jelen, and Casper Wilstrup. An Approach to Symbolic Regression Using Feyn. April 2021.

[76] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael ONeill, R. I. McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, June 2011.

[77] F. O. de Franca, M. Virgolin, M. Kommenda, M. S. Majumder, M. Cranmer, G. Espada, L. Ingelse, et al. Interpretable Symbolic Regression for Data Science: Analysis of the 2022 Competition, April 2023.

[78] Ankit Rohatgi. Webplotdigitizer: Version 4.6, 2022.

[79] Edwin Hubble. A relation between distance and radial velocity among extra-galactic nebulae. *Proceedings of the National Academy of Sciences*, 15(3):168–173, March 1929.

[80] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H. Moore. Contemporary Symbolic Regression Methods and their Relative Performance, July 2021.

[81] Johannes Kepler. *Harmonices Mundi*. Lincii Austriæ, 1619.

[82] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. Jussu Societatis Regiæ ac Typis Joseph Streater, London, England, 1687.

[83] Max Planck. *The Theory of Heat Radiation*. P. Blakiston's Son & Co, Philadelphia, 1914.

[84] Henrietta S. Leavitt and Edward C. Pickering. Periods of 25 Variable Stars in the Small Magellanic Cloud. *Harvard College Observatory Circular*, 173:1–3, March 1912.

[85] William H. Press and Paul Schechter. Formation of Galaxies and Clusters of Galaxies by Self-Similar Gravitational Condensation. *The Astrophysical Journal*, 187:425–438, February 1974.

[86] Charles Bonnet. *Contemplation de La Nature*. Amsterdam, 1764.

[87] Émile Clapeyron. Mémoire sur la puissance motrice de la chaleur. *Journal de l'École Polytechnique*, 1835.

[88] Johannes Rydberg. Researches sur la constitution des spectres d'émission des Éléments chimiques. *Proceedings of the Royal Swedish Academy of Science*, 1889.

[89] Arthur Grundner, Tom Beucler, Pierre Gentine, and Veronika Eyring. Data-Driven Equation Discovery of a Cloud Cover Parameterization, April 2023.

[90] Yanzhang Li, Hongyu Wang, Yan Li, Huan Ye, Yanan Zhang, Rongzhang Yin, Haoning Jia, et al. Electron transfer rules of minerals under pressure informed by machine learning. *Nature Communications*, 14(1):1815, March 2023.

[91] Konstantin T. Matchev, Katia Matcheva, and Alexander Roman. Analytical Modeling of Exoplanet Transit Spectroscopy with Dimensional Analysis and Symbolic Regression. *The Astrophysical Journal*, 930(1):33, May 2022.

[92] Digvijay Wadekar, Leander Thiele, J. Colin Hill, Shivam Pandey, Francisco Villaescusa-Navarro, David N. Spergel, Miles Cranmer, et al. The SZ flux-mass (Y-M) relation at low halo masses: Improvements with symbolic regression and strong constraints on baryonic feedback. September 2022.

[93] Digvijay Wadekar, Leander Thiele, Francisco Villaescusa-Navarro, J. Colin Hill, Miles Cranmer, David N. Spergel, Nicholas Battaglia, et al. Augmenting astrophysical scaling relations with machine learning : Application to reducing the SZ flux-mass scatter, January 2022.

[94] Sergiy Verstyuk and Michael R. Douglas. Machine Learning the Gravity Equation for International Trade. *SSRN Electronic Journal*, 2022.

[95] Patrick Kidger. On Neural Differential Equations, February 2022.

[96] Kaze W. K. Wong and Miles Cranmer. Automated discovery of interpretable gravitational-wave population models. *ML4Astro Workshop @ ICML 2022*, July 2022.

[97] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[98] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *arXiv:1411.1607 [cs]*, July 2015.

[99] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[100] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[101] Chris Elrod. LoopVectorization.jl: Macro(s) for vectorizing loops. JuliaHub, 2022.

[102] Michael Innes. Don't unroll adjoint: Differentiating SSA-Form programs. *CoRR*, abs/1810.07951, 2018.

[103] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwózd, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *ACM Communications in Computer Algebra*, 55(3):92–96, January 2022.

[104] Michael Droettboom, John Hunter, Thomas A Caswell, Eric Firing, Jens Hedegaard Nielsen, Phil Elson, Benjamin Root, et al. Matplotlib: Matplotlib v1.5.1, January 2016.

[105] Rodrigo Luger and Contributors. Showyourwork!, 2021.

[106] Charlie Wood. Powerful Machine Scientists Distill the Laws of Physics From Raw Data. *Quanta Magazine*, May 2022.