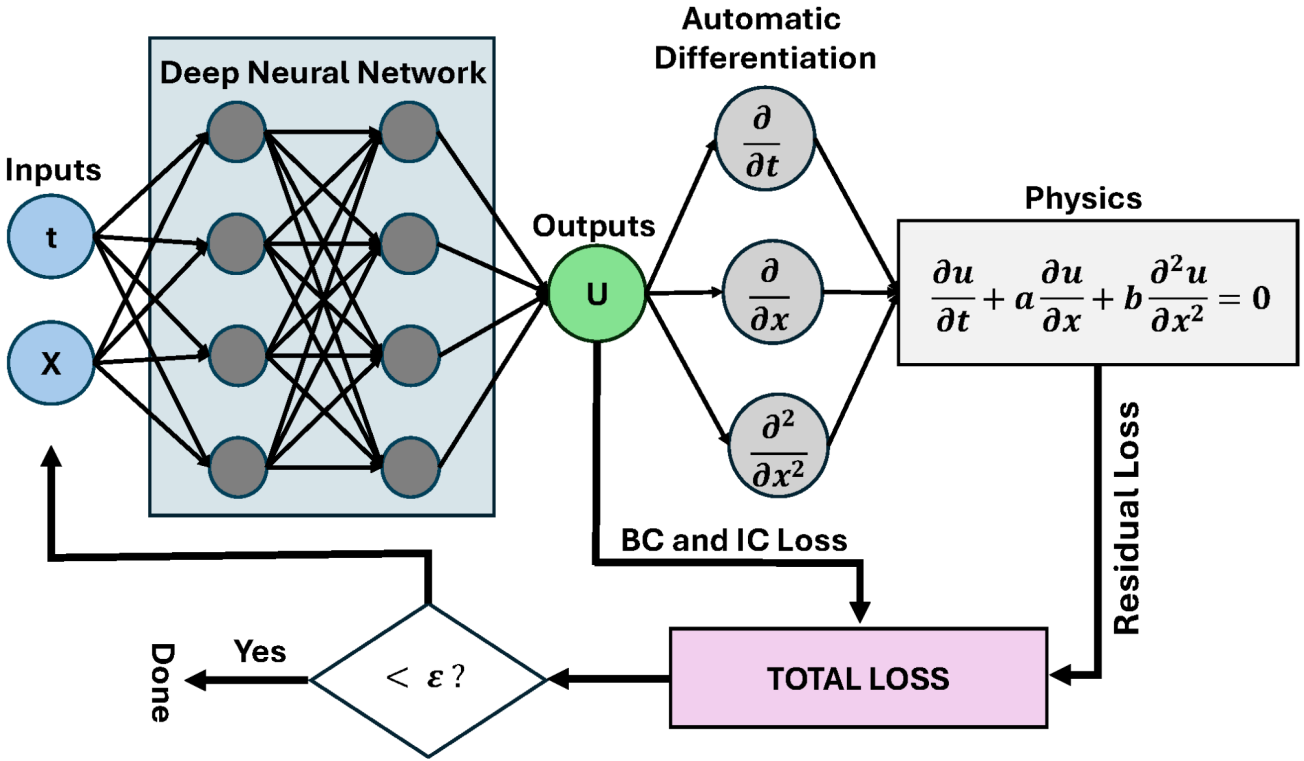
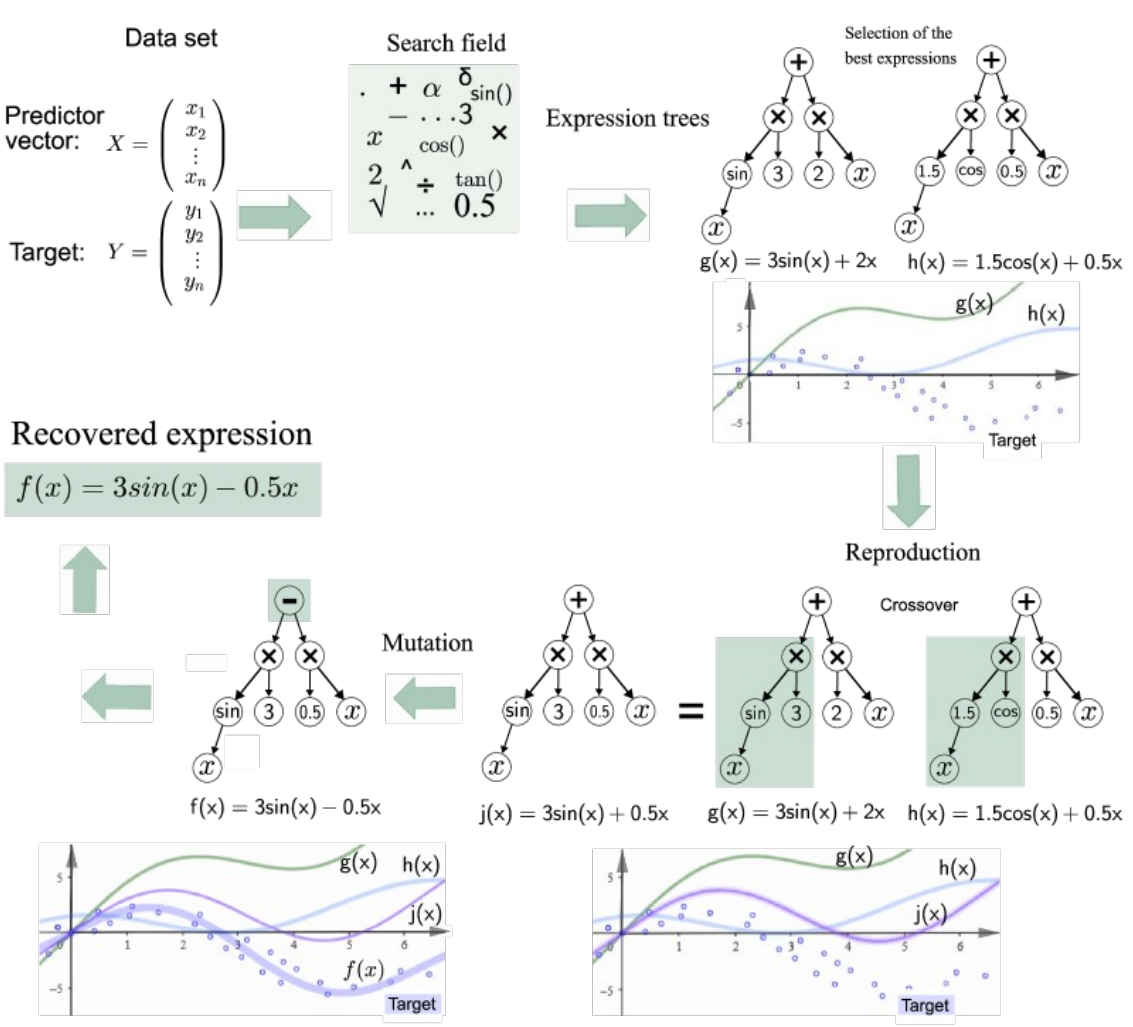
Symbolic regression

Last class


Deep Neural Network
Automatic Differentiation
Inputs
t
X
Outputs
U
$\frac{\partial}{\partial t}$
$\frac{\partial}{\partial x}$
$\frac{\partial^2}{\partial x^2}$
Physics
$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} + b\frac{\partial^2 u}{\partial x^2} = 0$
Residual Loss
BC and IC Loss
TOTAL LOSS
$< \varepsilon$?
Yes
Done

Next step

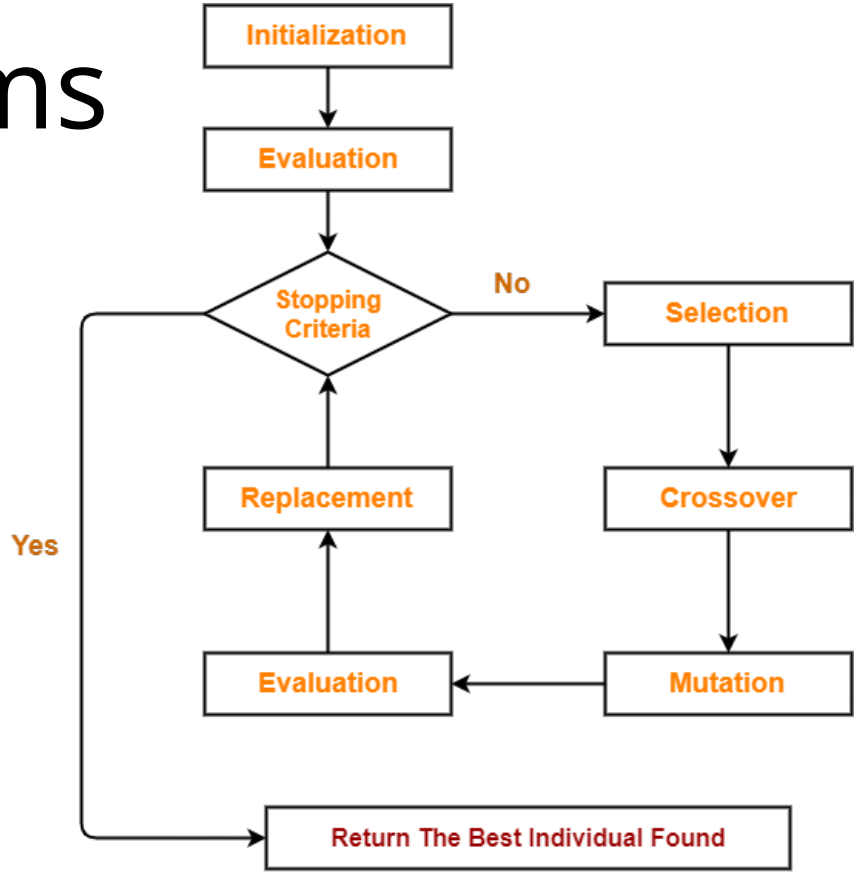
Knowledge distillation is a machine learning technique that aims to transfer the learnings of a large pre-trained model, the “teacher model,” to a smaller “student model.” It’s used in deep learning as a form of model compression and knowledge transfer, particularly for massive deep neural networks.



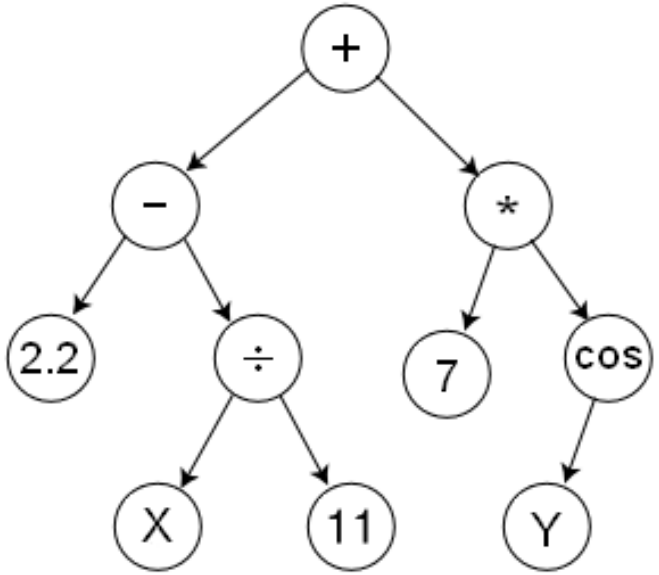
Genetic algorithms

Simple Genetic Algorithm

STEP 1. INITIALIZATION
Generate initial population $\mathcal{P}$ at random or with prior knowledge
STEP 2. FITNESS EVALUATION
Evaluate the fitness for all individuals in $\mathcal{P}$
STEP 3. SELECTION
Select a set of promising candidates $\mathcal{S}$ from $\mathcal{P}$
STEP 4. CROSSOVER
Apply crossover to the mating pool $\mathcal{S}$ for generating a set of offspring $\mathcal{O}$
STEP 5. MUTATION
Apply mutation to the offspring set $\mathcal{O}$ for obtaining its perturbed set $\mathcal{O}'$
STEP 6. REPLACEMENT
Replace the current population $\mathcal{P}$ with the set of offspring $\mathcal{O}'$
STEP 7. TERMINATION
If the termination criteria are not met, go to STEP 2

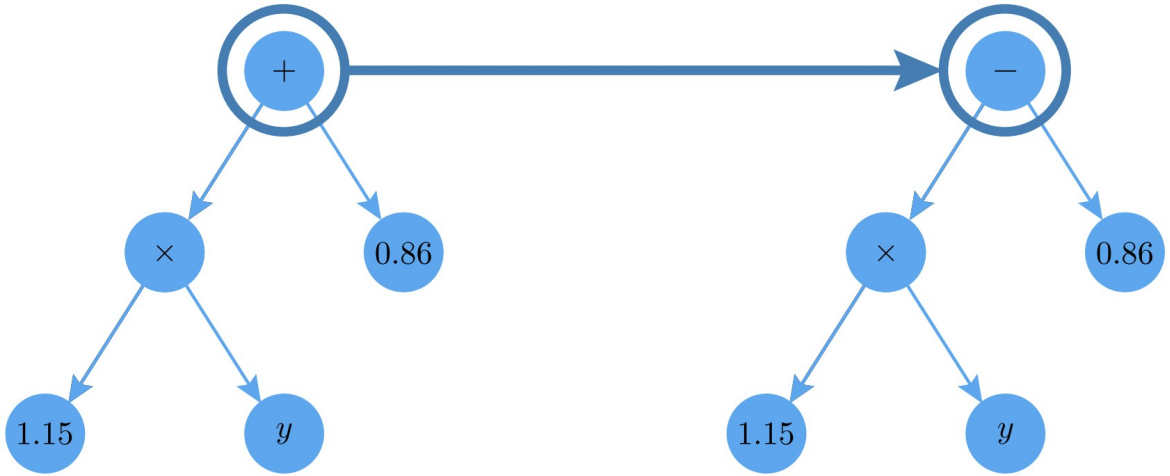


Arithmetic trees



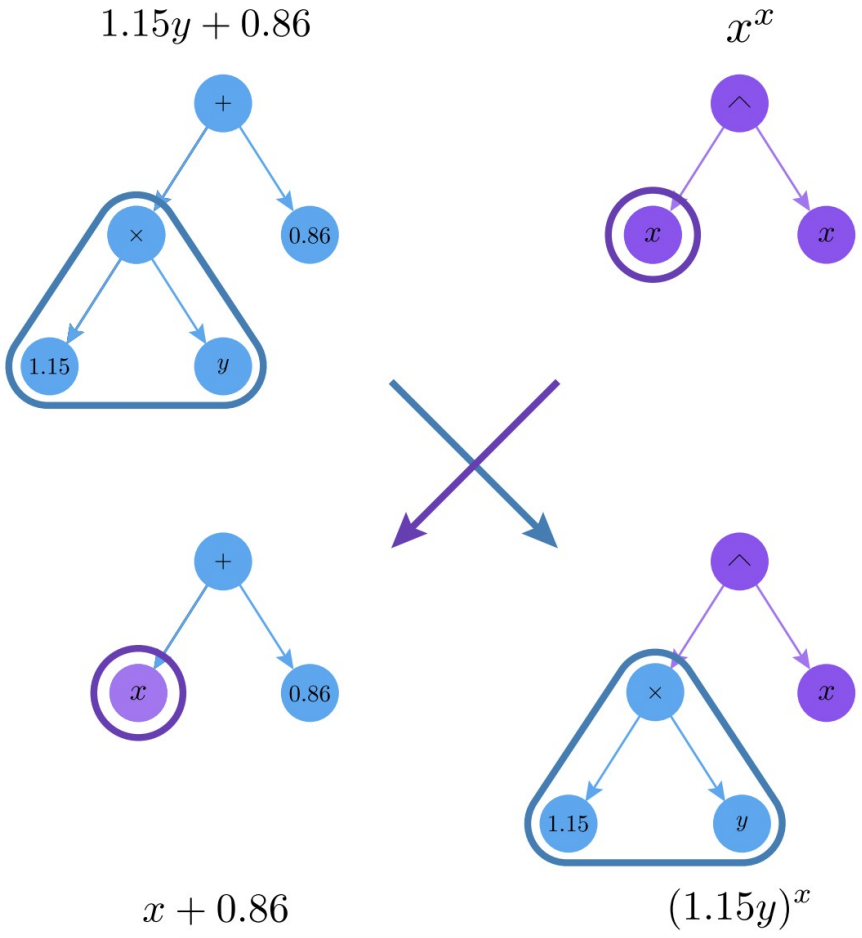
$$\left(2.2 - \left(\frac{X}{11}\right)\right) + \left(7 * \cos(Y)\right)$$

Mutation

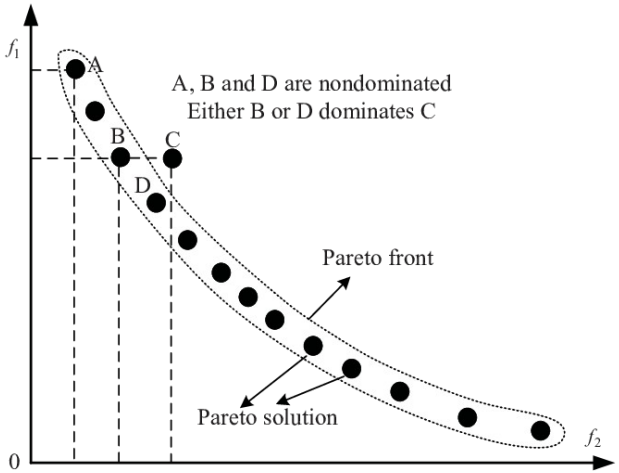


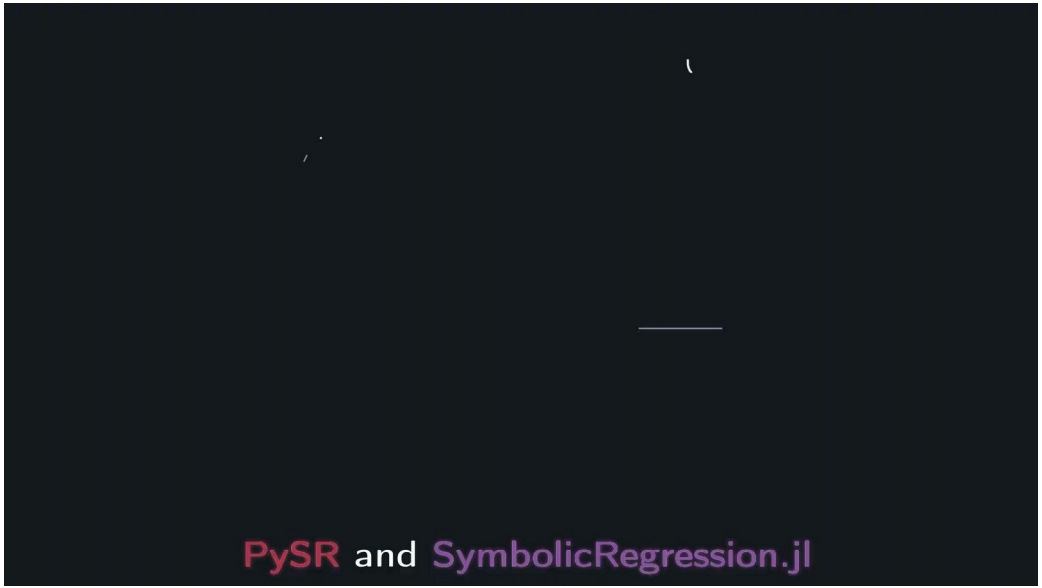
$1.15y + 0.86$ $1.15y - 0.86$

Crossover



Fitness function





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

PySR

PySR & SymbolicRegression.jl

github.com/MilesCranmer/pysr_paper

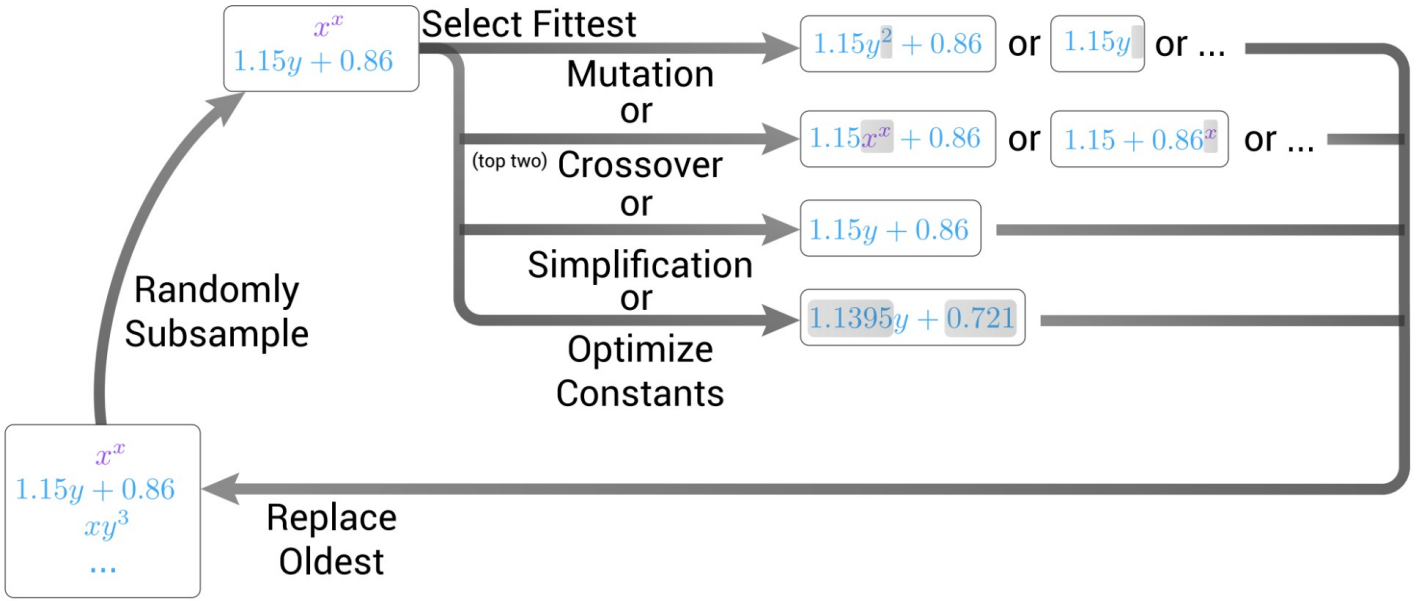
Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl

Miles Cranmer[1,2]

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

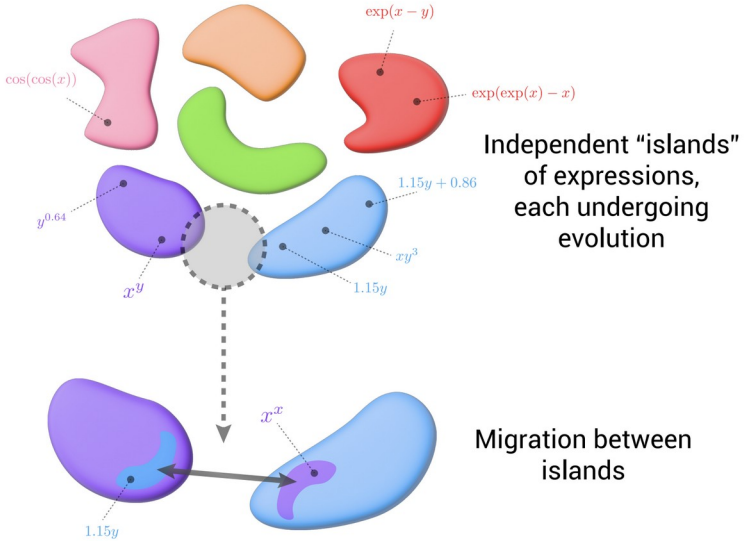
May 2, 2023

The symbolic regression loop



Algorithm

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)



Independent "islands" of expressions, each undergoing evolution

Migration between islands

```
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          // e.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
```

Algorithm

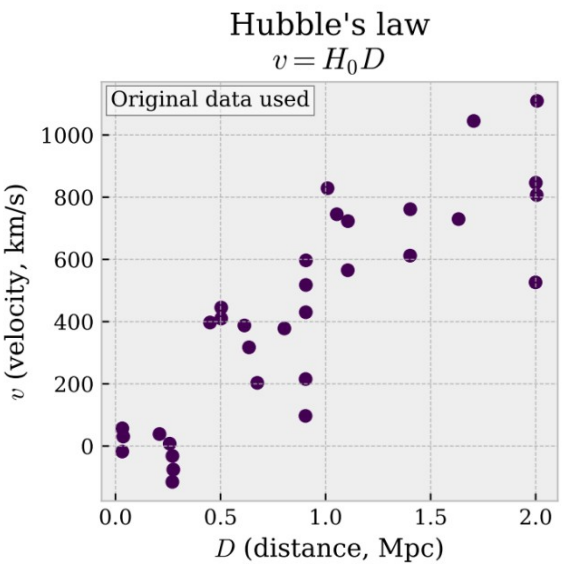
input : P, a set of expressions
input : $\mathcal{X}$ as in algorithm 1
output : P, the evolved set of expressions
param : n_c, the number of mutations per `evolve()` call (=300000)
param : p_{cross}, the probability of crossover (=0.01)

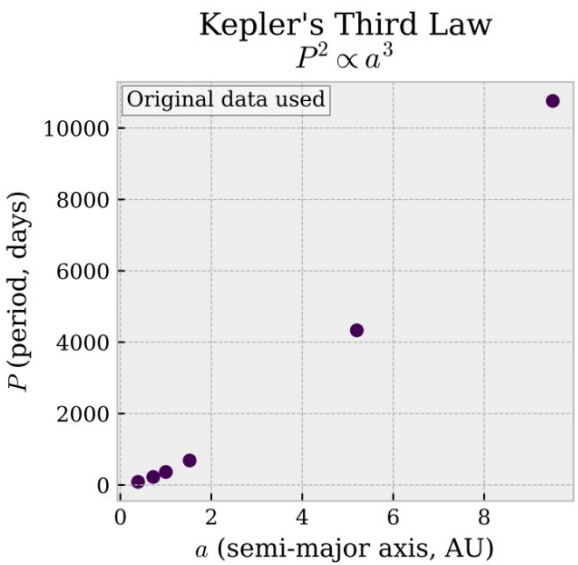
```
1  function evolve(P, X)
2      for k in range(n_c) // [code]
3          if rand() > p_cross // [code]
4              // mutation
5              E ← tournament(P, X)
6              T ← 1 − k/n_c // annealing temperature
7              E* ← mutate(E, T)
8              replace oldest expression in 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
```

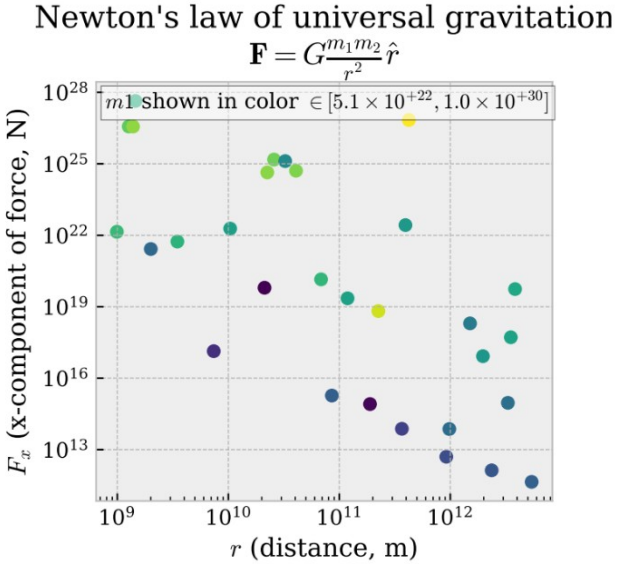
input : P, a population of expressions
input : $\mathcal{X}$ as in algorithm 1
output : a single expression (the winner of the tournament)
param : n_s, the tournament size (=12)
param : $p_{\text{tournament}}$, the probability of selecting the fittest individual (=0.9)

```
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
```

Use cases


Hubble's law
$v = H_0 D$
Original data used
v (velocity, km/s)
D (distance, Mpc)


Kepler's Third Law
$P^2 \propto a^3$
Original data used
P (period, days)
a (semi-major axis, AU)


Newton's law of universal gravitation
$\mathbf{F} = G\frac{m_1 m_2}{r^2}\hat{r}$
$m1$ shown in color $\in [5.1 \times 10^{+22}, 1.0 \times 10^{+30}]$
F_x (x-component of force, N)
r (distance, m)

Benchmark

	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)

Benchmark

		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	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓

Code example

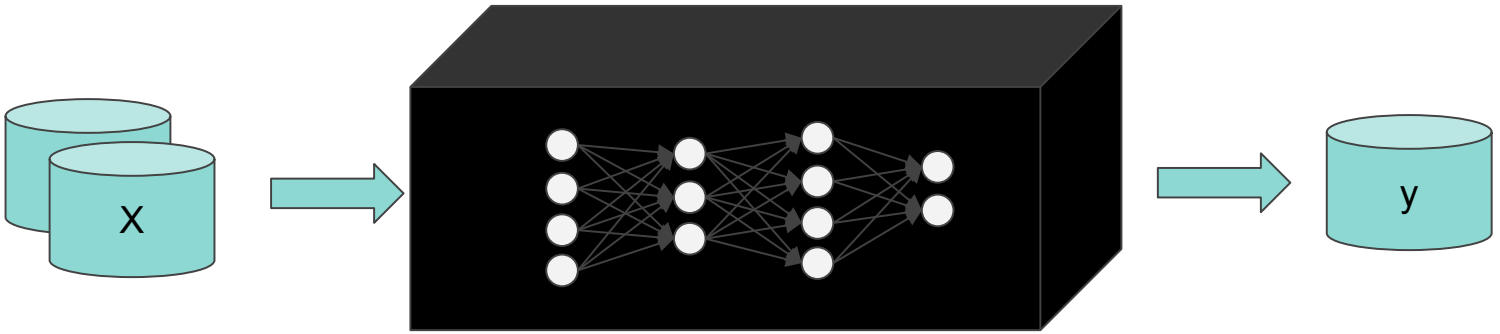
We can use custom operators!

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

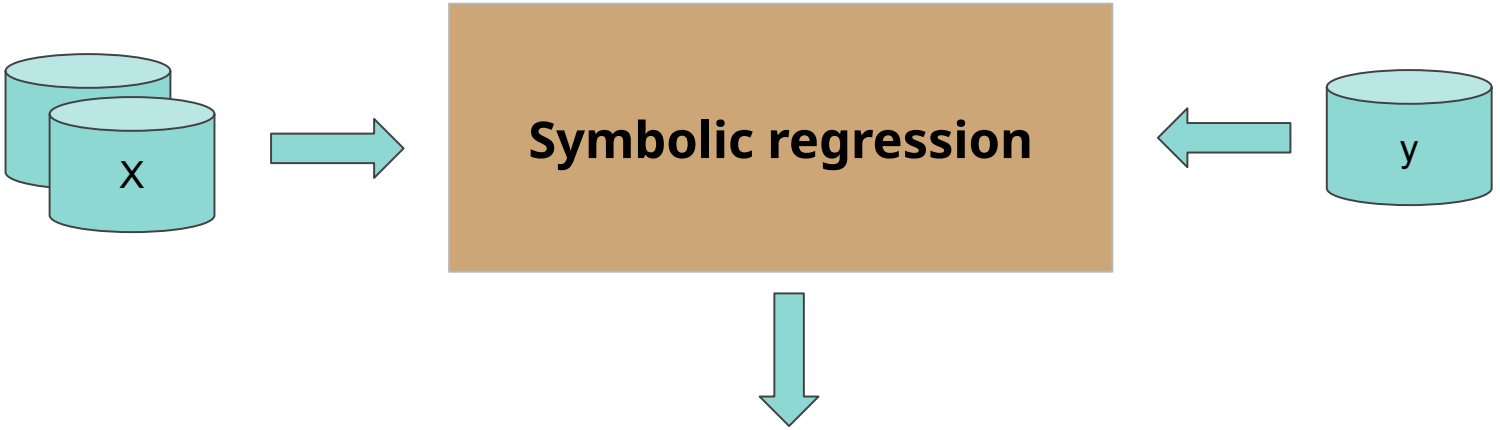
PySRRegressor Reference - PySR

```
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)
```

Coupling with neural networks



Coupling with neural networks



complexity	pick	score	equation	loss
1	0	0.000000	4.4324794	42.354317
3	1	1.255691	(x0 * x0)	3.437307
5	2	0.011629	((x0 * x0) + -0.28087974)	3.358285
	3	0.897855	((x0 * x0) + cos(x3))	1.368308

Exercises

1. Add Gaussian noise to `y_sr` before passing it to PySR. How robust is the symbolic regression?
2. Change the PINN to solve the Burgers' equation (from Day 1) and try to distill the shockwave equation.
3. Remove "sin" from the unary_operators list. Can PySR approximate the sine wave using Taylor expansion terms (polynomials)?

pedrogasparetto@ymail.com