
Curiosity-driven RL for symbolic equation solving

Kevin P. O’Keeffe
Starling Research Institute
Seattle, WA, USA
kevin.p.okeeffe@gmail.com

Abstract

We explore if RL can be useful for symbolic mathematics. Previous work showed contrastive learning can solve linear equations in one variable. We show model-free PPO Schulman et al. (2017) augmented with curiosity-based exploration and graph-based actions can solve nonlinear equations such as those involving radicals, exponentials, and trig functions. Our work suggests curiosity-based exploration may be useful for general symbolic reasoning tasks.

1 Introduction

Reinforcement learning (RL) has been applied to diverse fields (Ng et al., 2006; Degraeve et al., 2022; Vinyals et al., 2019) but has not yet been widely adopted in symbolic mathematics, where agents perform tasks like solving algebraic equations or evaluating integrals analytically. Traditional approaches to symbolic mathematics rely on hand-engineered systems like Mathematica or Maple. An RL-based approach, where agents learn transformations autonomously, could reduce manual curation and find novel solution techniques.

Applying RL to symbolic math is difficult because the state space is combinatorially large, as each equation can branch into numerous sub-expressions. The action space is also large and dynamic: at every step, the agent can apply various algebraic manipulations to different sub-expressions, so the number and type of actions change with the equation’s form.

We here show RL agents combined with curiosity-based exploration and a graph-based action space can overcome these challenges. The agents learn to solve a wide range of algebraic equations such as those involving radicals and trig functions. Though perhaps simple from a human perspective, solving such equations using RL is non-trivial and to our knowledge has never before been done. Moreover, they are a key building block for more advanced tasks like solving differential equations.

2 Related Work

Linear Equations with Primitive Action Spaces. Poesia et al. (Poesia et al., 2021) pioneered the use of RL on symbolic equations. They showed agents trained with a contrastive loss can solve linear equations with integer coefficients. Their formulation differs from ours in that their action space is at a primitive granularity: states and actions are represented as raw text (strings of symbols), and each algebraic operation (associativity, commutativity, adding a constant, etc.) corresponds to a single low-level action. We represent equations as expression trees and define higher-level actions. They also use sparse rewards (we use dense rewards imbued with curiosity) and do not study nonlinear equations such as rational equations with complex functions (exp, sin, etc). See Appendix.

Q-Learning for Symbolic Equations. Dabelow and Ueda (Dabelow & Ueda, 2024) used Q-learning in a SymPy-based framework, restricting the agent to linear equation of form $a_0 + a_1x = a_2 + a_3x$ and $a_0 + b_0c + (a_1 + b_1c)x = a_2 + b_2c + (a_3 + b_3c)$ with coefficients in $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$. They do not study nonlinear equations.

3 Problem Formulation

Our Markov Decision Process (MDP) formulation is

States: equations like $ax + b = 0$ or $cx + d = -x/b$. We vectorize these using preorder traversal over the equations' expression tree (Figure 1) and pad to a maximum length $L = 50$ (suitable length for the equation sizes we here consider). Then we map operations and symbols to integers $\{\text{add} : 1, \text{sub} : 2, \text{mul} : 3, \dots, x : 5, a : 6, \dots\}$. We encode both the lhs and rhs of the equation in this manner and then concatenate. An example embedding is $x + a \rightarrow [\text{add}, x, a] \rightarrow [1, 5, 8, \text{PAD}, \dots]$

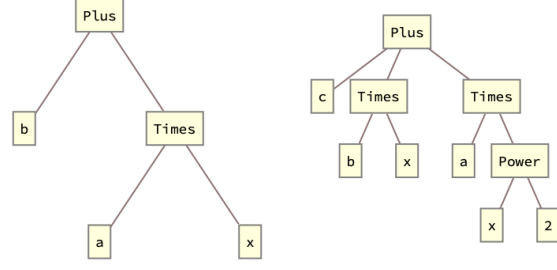


Figure 1: Expression trees for $ax + b$ and $ax^2 + bx + c$.

Actions: represented as $(operation, term)$ pairs, such as (sub, b) or (div, a) . Formally,

$$O = \{\text{add, subtract, mult, divide}\} \cup \{\text{square, sqrt, exp, log, sin, cos, asin, acos}\} \quad (1)$$

$$T = \text{SubExpr}(\text{lhs}) \cup \text{SubExpr}(\text{rhs}), \quad (2)$$

$$A = (O \times T) \cup \{(Expand, None), (collect, x), (multiply, -1)\}. \quad (3)$$

Notice the first set of operations take two arguments, the second one argument. For the terms, we choose the list of sub-expressions in the lhs and rhs expression trees. Example of sub-expressions are

$$ax + b = 0 \Rightarrow \{a, x, ax, b\} \quad (4)$$

$$(ax + b)/(cx + d) + e = 0 \Rightarrow \{a, x, ax, b, c, d, cx, cx + d, ax + b\} \quad (5)$$

This term set is expressive enough to solve rational equations and all other equation types we consider in this paper. It is also dynamic: the list of sub-expressions is derived from the equation/state and thus has variable length. Looping back to the operations, we also include $(mul, -1)$ and

$$\text{expand: } cx + d + e(ax + b) \rightarrow aex + be + cx + d = 0$$

$$\text{collect } x: ax + bx \rightarrow (a + b)x$$

These allow the agent to perform key algebraic steps (Appendix). Finally, we index the action set A serially, cap it at size $|A| = 50$ ¹, and mask illegal actions (e.g., division by zero, see Appendix).

Rewards. Define the *complexity* C of an equation as the total number of nodes and edges in the expression tree². The complexities of the equations in Figure 1 are $C(ax + b) = 5 + 4 = 9$ and $C(ax^2 + bx + c) = 10 + 9 = 19$. The reward is then

$$R(\text{action}) = C(\text{equation}) - C(\text{equation after action}) \quad (6)$$

The intuition here is to encourage the agent to take actions which simplify the equation.

State Transition Function. We wrote our code in Python and used SymPy to apply operations to the equations. At each step, we keep track of a lhs and rhs of an equation e.g. $lhs = (ax + b)$ and $rhs = 0$. We apply actions to both the lhs and rhs. For example, (sub, b) results in $(lhs, rhs) = (ax, -b)$, and then (div, a) results in $(lhs, rhs) = (x, -b/a)$. The terminal condition for the environment is when $lhs = x$ and the substitution of the rhs into the original equation simplifies to 0.

Limitations. Importantly, this MDP formulation only works on equations which are ‘closed,’ in the sense that solving them requires manipulating the terms already present in the equation/sub-expression list. By contrast, solving ‘open’ equations requires adding new, out-of-equation terms or clever substitutions. A classic example is the quadratic equation $ax^2 + bx + c$ which is solved by completing the square – adding $(b/2a)^2$ to each side. This is ‘generative’ reasoning, since the term $(b/2a)^2$ is *not* in the term set we have defined. Equations that require these more exotic actions are beyond the scope of the current work (and were also not studied in all previous works Poesia et al. (2021); Dabelow & Ueda (2024)).

¹We explored other values like $|A| = 40, 70, 100$ and found no changes in performance

²We judged number of nodes+edges correlates better with algebraic complexity than number of nodes alone.

Table 1: Mean success rates $\pm$ s.e. over $N_{\text{trial}} = 10$, $N_{\text{train}} = 3 \times 10^6$.

Equation	A*	A2C	PPO	PPO-ICM	PPO-RIDE	PPO-NGU	PPO-RND
$ax + b$	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
$a/x + b$	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00
$c(ax + b) + d$	1.00 ± 0.00	0.20 ± 0.13	0.10 ± 0.09	0.70 ± 0.14	0.60 ± 0.15	0.40 ± 0.15	1.00 ± 0.00
$c + d/(ax + b)$	0.00 ± 0.00	0.00 ± 0.00	0.20 ± 0.13	0.50 ± 0.16	0.40 ± 0.15	0.50 ± 0.16	0.80 ± 0.13
$(ax + b) + e(cx + d)$	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.20 ± 0.13	0.00 ± 0.00	0.20 ± 0.13	0.40 ± 0.15
$e + (ax + b)/(cx + d)$	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.10 ± 0.09	0.00 ± 0.00	0.30 ± 0.14	0.50 ± 0.16

4 Results

Fixed equation environments. We begin with six simple algebraic equations displayed in the first column of Table 1. The first two, $ax + b$, $a/x + b$, are easily solved, and are included as baselines. The remaining four are more challenging, having a nested structure and more unknowns. Equations of this form are found in elementary textbooks on algebra, making them natural first test cases.

We consider a *fixed equation* environment, where the agent tried to solve a single equation during every episode. We used the A2C and PPO algorithms from Stable Baselines 3 Raffin et al. (2021) and a homemade A^* algorithm as a non-learning baseline (Appendix). We also attached four different curiosity methods to PPO: Intrinsic Curiosity Module (ICM), Rewarding Impact-Driven Exploration (RIDE), Never Give Up (NGU), and Random Network Distillation (RND) Pathak et al. (2017); Burda et al. (2019); Badia et al. (2020); Raileanu & Rocktäschel (2020); Pathak et al. (2019). These are implemented in the python package *rlite* Yuan et al. (2025). We hypothesize these exploration methods are needed to solve complex equations. Each algorithm was trained for $N_{\text{train}} = 3 \times 10^6$ steps over $N_{\text{trial}} = 10$ trials, and performance was evaluated based on success rates.

Table 1 shows all methods solved the first two equations, but performance diverged for the more complex equations. For $c + d/(ax + b)$, PPO-RND led with a 0.80 success rate, followed by PPO-ICM and PPO-NGU at 0.50, while A^* failed. The most challenging equations, $(ax + b) + e(cx + d)$ and $e + (ax + b)/(cx + d)$, saw low success rates with PPO-RND achieving 0.40 and 0.50. Overall, PPO-NGU and PPO-RND were the top performers. The Appendix shows some example solution traces.

The takeaway from this problem is that our MDP formulation based on expression trees works and that curiosity-based exploration is required to solve non-elementary equations.

Random equation environments. Now we task the agent with solving a random equation during each episode. We generate train/test sets by starting with x and applying actions recursively. Recall actions are *(operation, term)* pairs. The operations are as before, but we restrict terms to (a, b, c) (we excluded d, e since we wanted to favor long, rather than wide equations). Applying a single action generated equations like ax , $\log(x)$, $x + b$, applying 2 actions equations like $ax + b$, $\log(x) + d$, $(x + b)/c$ and so on. We threw out any equations that SymPy couldn't solve, and considered multi-root equations solved when just one root was found (e.g. $\sin(x) = 0$ is solved if the agent found $x = 0$ rather than $x = 2n\pi$). We created a small dataset which included all equations of *depth* < 4 (size 3874) and sub-sampled to a $10^3/10^2$ train/test split, and a large dataset of all equations of *depth* < 5 (size 15625) sub-sampled to a $10^4/10^3$ train/test split.

Importantly, this generation process does *not* create rational equations of form $ax + b/(cx + d) + e = 0$. To include these, we augmented the dataset with a subset of such equations generated by combining simpler sub-expressions (Appendix). Below we give examples of equations and their solutions from the large dataset.

$$\begin{aligned}
 a - b + \ln(x) = 0 &\implies x = e^{b-a} & -b + \frac{-c + x/b}{c} = 0 &\implies x = bc + bc^2 \\
 c + \sin\left(\frac{x-a}{c}\right) = 0 &\implies x = a - c \arcsin(c) & \sin(\ln(\sin(x))) = 0 &\implies x = \pi/2 \\
 \sqrt{bx + a} - c = 0 &\implies x = (c^2 - a)/b & b + \log(x^2/b^2) &\implies x = \sqrt{b^2 \exp(-b)}
 \end{aligned}$$

Figure 2 plots the learning curves for both the small dataset (top row) and large dataset (bottom row). We plot $test_{\text{greedy}}$, the fraction of test equation solved following a greedy policy, and $test_{10}$ the

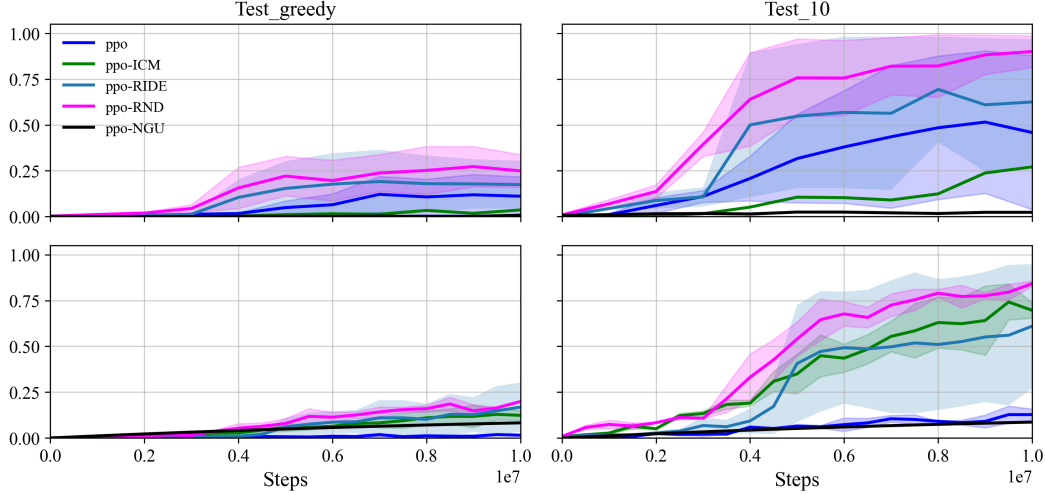


Figure 2: Top: small dataset; Bottom: large dataset. Curves show mean success rate over 3 trials; shading spans min–max range.

fraction that were solved at least once from 10 rollouts following the stochastic policy. PPO-RND and PPO-RIDE are the best performers, achieving good generalization $test_{10} \approx 1.0, 0.8$ on both datasets (although $test_{greedy} \approx 0.25$ was much lower). PPO shows decent performance on the small datasets, but fails on the large one entirely. Surprisingly, PPO-NGU is the worst performer on this random equation environment, showing little progress on both the small and large datasets.

As before, the takeaway is that only curiosity-based exploration can solve the hardest tasks.

5 Discussion

We have shown that curiosity-driven RL agents with action spaces based on expression trees can learn to solve some simple nonlinear equations in one variable, going beyond the linear equations considered in previous work Poesia et al. (2021); Dablow & Ueda (2024). Of the four curiosity types we tested, RND was the best performer. It is lightweight computationally, and solved both the fixed and random equation environments. NGU by contrast has a higher memory footprint, which may explain its poor performance on the random equation environment. Future work could explore this issue.

The main limitation of our work – and all previous work – is that the equations we considered were “closed.”; all manipulations needed to solve them were contained within the original expression; the right *term* in the action tuple (*operation*, *term*) lay in the expression tree; we never had to generate novel terms. Extending our method to handle “open” equations, where new terms, auxiliary expressions, or changes of variables are introduced, will likely require a generative sub-model.

To expand on this a little, Box 1 lists the first three non-trivial polynomial equations as benchmarks for future work. Admittedly, these are challenging problems. Solving the cubic and quartic equations requires an agent to discover Cardano’s and Ferrari’s method which are rather labyrinthine Boyer & Merzbach (2011). But solving the quadratic equation ought to be within reach. We speculate an Alphazero-like algorithm Silver et al. (2017), which excels at long-horizon planning, might here be useful.

Box 1: Open Challenges

Algebraic equations	Solvable with RL
$ax^2 + bx + c = 0$	?
$ax^3 + bx^2 + cx + d = 0$	?
$ax^4 + bx^3 + cx^2 + dx + e = 0$	?

References

- Badia, A. P., Sprechmann, P., Vitvitskyi, A., Guo, D., Piot, B., Kapturowski, S., Tieleman, O., Arjovsky, M., Pritzel, A., Bolt, A., Blundell, C., Zisserman, A., Appleyard, J., Botvinick, M., Vinyals, O., Hassabis, D., Silver, D., Pascanu, R., and Hessel, M. Never give up: Learning directed exploration strategies. In *8th International Conference on Learning Representations*, 2020. URL <https://arxiv.org/abs/2002.06038>.
- Boyer, C. B. and Merzbach, U. C. *A history of mathematics*. John Wiley & Sons, 2011.
- Burda, Y., Edwards, H., Pathak, D., Storkey, A., Darrell, T., and Efros, A. A. Exploration by random network distillation. In *7th International Conference on Learning Representations*, 2019. URL <https://arxiv.org/abs/1810.12894>.
- Dabelow, L. and Ueda, M. Symbolic equation solving via reinforcement learning. *arXiv preprint arXiv:2401.13447*, 2024.
- Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de Las Casas, D., et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.
- Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX: The 9th international symposium on experimental robotics*, pp. 363–372. Springer, 2006.
- Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787. PMLR, 2017.
- Pathak, D., Gandhi, D., and Gupta, A. Self-supervised exploration via disagreement. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 5062–5071. PMLR, 2019. URL <https://arxiv.org/abs/1906.04161>.
- Poesia, G., Dong, W., and Goodman, N. Contrastive reinforcement learning of symbolic reasoning domains. *Advances in neural information processing systems*, 34:15946–15956, 2021.
- Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research*, 22(268): 1–8, 2021.
- Raileanu, R. and Rocktäschel, T. Ride: Rewarding impact-driven exploration for procedurally-generated environments. In *8th International Conference on Learning Representations*, 2020. URL <https://arxiv.org/abs/2002.12292>.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.
- Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.
- Yuan, M., Zhang, Z., Xu, Y., Luo, J. S., Li, B., Jin, X., and Zeng, W. RLLTE: Long-term evolution project of reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2025.

.1 Macroactions

Here we justify why the three macroactions

$$(expand, none), (collect, x), (mul, -1)$$

discussed in the main text are *required* to solve the equations we consider, and are not simple conveniences for the RL agent. Consider the action set *without* these macroactions, which we requote below

$$O = \{ \text{add, subtract, mul, divide} \} \cup \{ \text{square, sqrt, exp, log, sin, cos, asin, acos} \} \quad (7)$$

$$T = \text{SubExpr(lhs)} \cup \text{SubExpr(rhs)}, \quad (8)$$

$$A = (O \times T) \cup \{(Expand, None)\} \cup \{(collect, x)\} \cup \{(multiply, -1)\}. \quad (9)$$

Now consider the equation $dx + c(ax + b) = e$. One must distribute c into the bracketed term $ax + b$. The action set above cannot do this. One needs the *expand* macroaction for this. Moreover, once the term is expanded we get $dx + cax + cb = e$ and one needs to factor the x common to both the first and second terms. *collectx* performs this function. Finally, consider the equation $-x = b$. Since -1 is not included in our term set, one needs $(mul, -1)$ to solve the equation. (One could alternatively add -1 to the term set but we chose not to.)

.2 Dataset generation: Rational Equation

To supplement the recursive datasets, we constructed rational equations of the form

$$\frac{ax + b}{cx + a} + b = 0,$$

where a, b, c are (non-zero) symbolic coefficients. We excluded degenerate cases such as denominators that vanish identically or cancel trivially with numerators.

Examples of retained equations include:

$$\begin{aligned} \frac{x + b}{cx + a} = 0 &\Rightarrow x = -b, \\ \frac{ax + b}{cx + a} + b = 0 &\Rightarrow x = \frac{-b - ab}{a + cb}, \\ \frac{ax + b}{cx + a} - c = 0 &\Rightarrow x = \frac{-b + ac}{a - c^2}. \end{aligned}$$

These rational forms were merged into both the small and large datasets, with their proportion limited to preserve diversity across other functional forms (logarithmic, trigonometric, exponential, etc.).

.3 Illegal actions

The main issue was to avoid illegal division by zero. This was not as simple as precluding $(div, 0)$ from the action set. One had to prohibit ‘hidden’ division by zero, such as prohibiting division by $(x + a)$ or $(x + b)$ for an equation of form $(x + a)(x + b)$. To do this, we checked if the equation of form $P(x)Q(x)...$, where $P(x), Q(x)$ are polynomials in x , and then removed $(div, P, Q, ...)$ from the action set.

.4 Hyperparameters

We used Stable Baselines 3’s default hyperparameters for A2C and PPO (Table 2).

.5 A* Implementation

A simple A* baseline searches the equation graph using complexity C (nodes + edges) as heuristic. States are hashed via SymPy strings; priority queue expands lowest $g(n) + h(n)$ nodes. Depth limited to 20; actions masked for validity. Pseudocode: Initialize queue with start state; expand until solved or empty, updating scores and paths. Serves as non-learning baseline, efficient for shallow trees but limited by heuristic admissibility.

Algorithm	Hyperparameters
A2C	<i>learning_rate</i> = 0.0007, <i>n_steps</i> = 5, <i>gamma</i> = 0.99, <i>gae_lambda</i> = 1.0, <i>ent_coef</i> = 0.0, <i>vf_coef</i> = 0.5, <i>max_grad_norm</i> = 0.5, <i>use_rms_prop</i> = <i>True</i> , <i>device</i> = 'auto'
PPO	<i>learning_rate</i> = 0.0003, <i>n_steps</i> = 2048, <i>batch_size</i> = 64, <i>n_epochs</i> = 10, <i>gamma</i> = 0.99, <i>gae_lambda</i> = 0.95, <i>clip_range</i> = 0.2, <i>ent_coef</i> = 0.0, <i>vf_coef</i> = 0.5, <i>max_grad_norm</i> = 0.5, <i>normalize_advantage</i> = <i>True</i> , <i>device</i> = 'auto'

Table 2: Default hyperparameters for A2C and PPO from Stable Baselines 3.

.6 Solution Traces

Below are the solution traces for each equation in the fixed equation environment. Notice the solution trace is not always optimal. For instance, $ax + b = 0$ could be solved with (sub, b) , (div, a) , but instead the agent selects (sub, ax) and then $(mul, -1)$. "Truediv" is the sympy notation for divide.

1. Equation: $a + x = 0$
Step 1: $a + x = 0$ | subtract, a
Solved: $x = -a$
2. Equation: $ax = 0$
Step 1: $a*x = 0$ | div, a
Solved: $x = 0$
3. Equation: $ax + b = 0$
Step 1: $ax + b = 0$ | subtract, ax
Step 2: $-ax = b$ | multiply, -1
Step 3: $ax = -b$ | div, a
Solved: $x = -b/a$
4. Equation: $a/x + b = 0$
Step 1: $a/x + b = 0$ | subtract, b
Step 2: $a/x = -b$ | truediv, $1/x$
Step 3: $-b*x = a$ | truediv, b
Step 4: $-x = a/b$ | multiply, -1
Solved: $x = -a/b$
5. Equation: $c(ax + b) + d = 0$
Step 1: $c*(ax + b) + d = 0$ | expand, None
Step 2: $acx + bc + d = 0$ | subtract, acx
Step 3: $-acx = bc + d$ | multiply, -1
Step 4: $acx = -bc - d$ | truediv, c
Step 5: $ax = (-bc - d)/c$ | truediv, a
Solved: $x = (-bc - d)/(ac)$
6. Equation: $c + d/(ax + b) = 0$
Step 1: $c + d/(ax + b) = 0$ | subtract, c
Step 2: $d/(ax + b) = -c$ | multiply, $(ax + b)$
Step 3: $d = -c(ax + b)$ | expand, None
Step 4: $d = -cax - cb$ | multiply, -1
Step 5: $-d = cax + cb$ | subtract, cb
Step 6: $-d - cb = cax$ | truediv, c
Step 7: $(-d - cb)/c = ax$ | truediv, a
Step 8: $((-d - cb)/c)/a = x$ | truediv, $1/a$
Solved: $x = (-d - cb)/(ca)$
7. Equation: $cx + d + e(ax + b) = 0$

Step 1: $cx + d + e(ax + b) = 0$ | expand, None
 Step 2: $aex + be + cx + d = 0$ | collect, x
 Step 3: $be + d + x(ae + c) = 0$ | subtract, $x(ae + c)$
 Step 4: $-x(ae + c) = be + d$ | truediv, $ae + c$
 Step 5: $-x = (be + d)/(ae + c)$ | multiply, -1
 Solved: $x = -(be + d)/(a* e + c)$

8. Equation: $e + (ax + b)/(cx + d) = 0$

Step 1: $e + (ax + b)/(cx + d) = 0$ | truediv, $1/(cx + d)$
 Step 2: $(e + (ax + b)/(cx + d))(cx + d) = 0$ | expand, None
 Step 3: $ax + b + cex + de = 0$ | collect, x
 Step 4: $b + de + x(a + ce) = 0$ | subtract, $x(a + ce)$
 Step 5: $-x(a + ce) = b + de$ | expand, None
 Step 6: $-ax - cex = b + de$ | collect, x
 Step 7: $x*(-a - ce) = b + de$ | truediv, $-a - ce$
 Solved: $x = (b + de)/(-a - c*e)$

.7 Comparison of our formulation to that of previous work by Poesia et al. (2021)

Aspect	Poesia et al. (NeurIPS21)	This work
Target domains	Linear 1-var equations (plus toy “CommonCore” envs)	Linear <i>and</i> nonlinear algebraic, trigonometric
State representation	Raw text $\rightarrow$ bi-LSTM (character level)	Expression-tree graph $\rightarrow$ MLP
Action granularity	Very primitive axioms (commute, distribute, add- <i>c</i> ...) $\Rightarrow$ huge branching factor	Macro ops via SymPy (expand, collect x) + arithmetic Dynamic term list = all sub-expressions
Reward signal	Sparse — binary solved / unsolved	Dense — drop in complexity + curiosity
Exploration / search	Beam search during training; Contrastive Policy Learning (InfoNCE) on positives vs. beam negatives	Pure model-free A2C / PPO; (no external search)
Network backbone	2-layer bidirectional LSTM	2-layer MLP
Core strength	Bootstraps from <i>extreme</i> reward sparsity with no shaping	Scales to richer mathematics via structured representation and macro actions

Table 3: Side-by-side comparison of our RL formulation with that of Poesia et al. (2021).