Quality-Diversity Self-Play: Open-Ended Strategy Innovation via Foundation Models

Aaron Dharna[1,2]
aadharna@gmail.com

Cong Lu[1,2]
conglu@cs.ubc.ca

Jeff Clune[1,2,3]
jclune@gmail.com

[1]University of British Columbia
[2]Vector Institute
[3]Canada CIFAR AI Chair

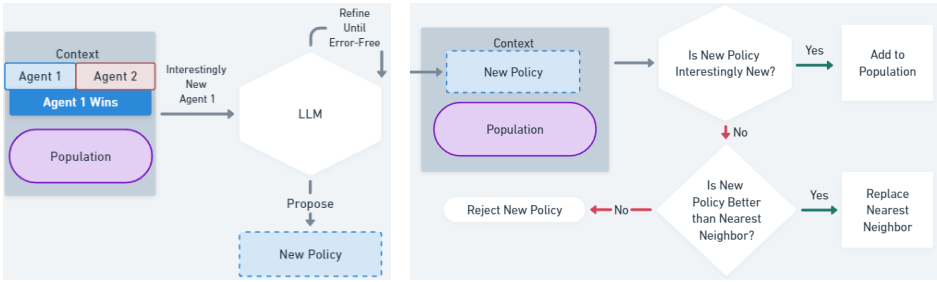
Abstract

Multi-agent dynamics have powered innovation from time immemorial, such as scientific innovations during the space race or predator-prey dynamics in the natural world. The resulting landscape of interacting agents is a continually changing, interconnected, and complex mosaic of opportunities for innovation. Yet, training innovative and adaptive artificial agents remains challenging. Self-Play algorithms bootstrap the complexity of their solutions by automatically generating a curriculum. Recent work has demonstrated the power of foundation models (FMs) as intelligent and efficient search operators. In this paper, we investigate whether combining the human-like priors and extensive knowledge embedded in FMs with multi-agent race dynamics can lead to rapid policy innovation in open-ended Self-Play algorithms. We propose a novel algorithm, Quality-Diversity Self-Play (QDSP) that explores diverse and high-performing strategies in interacting (here, competing) populations. We evaluate QDSP in a two-player asymmetric pursuer-evader simulation with code-based policies and show that QDSP surpasses high-performing human-designed policies. Furthermore, QDSP discovers better policies than those from quality-only or diversity-only Self-Play algorithms. Since QDSP explores new code-based strategies, the discovered policies come from many distinct subfields of computer science and control, including reinforcement learning, heuristic search, model predictive control, tree search, and machine learning approaches. Combining multi-agent dynamics with the knowledge of FMs demonstrates a powerful new approach to efficiently create a Cambrian explosion of diverse, performant, and complex strategies in multi-agent settings.

1 Introduction

Creating algorithms that rival the diversity and ingenuity that evolution has achieved is exceedingly difficult with current methods. In a multi-agent setting, each new policy presents new learning opportunities for other agents. The result is a vast tapestry of agents, each learning in new and diverse ways. We observe this adaptability in the natural world. For example, if a leopard has survived on a nearby hunting ground for several years but a pride of lions migrates into the territory, suddenly the leopard's current strategy has been compromised as the lionesses can gang up on the leopard to steal his kill. To survive, the leopard must learn a new strategy, such as dragging its kill up a tree before eating it. Similar to nature, Self-Play (SP) algorithms have shown remarkable capabilities for generating specialized agents that through competition can learn complex strategies, e.g. AlphaGo [27]. However, replicating nature's innovation is difficult for artificial algorithms

38th Conference on Neural Information Processing Systems (NeurIPS 2024).



(a) QDSP Propose Step (b) QDSP Update Population Step

Figure 1: **Overview of Quality-Diversity Self-Play** a) The **Propose** step takes in competing agents (from two populations p_1 and p_2), and the outcome of their competition. If searching for a new member of p_1, the context is also filled with other members of that population. The FM-based search operator is asked to create an interestingly new policy and refine its implementation until the code works. b) The **Update** step takes the newly proposed policy and checks its novelty against the archive. If the policy is novel, it is added to the population. Otherwise, it competes against its nearest neighbor and replaces it if better in the style of MAP-Elites.

because they lack a general understanding of the richness of the world, and training agents to exhibit new behaviors tends to be slow.

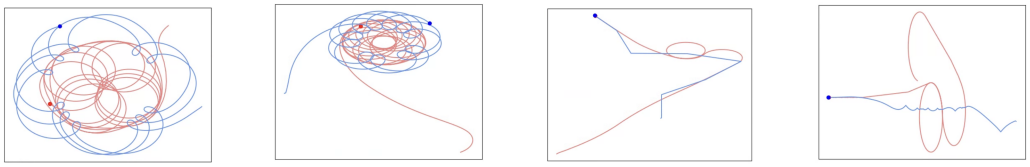
Recent advances in foundation models (FMs) have enabled them to be used as flexible search operators for tasks such as increasing sample efficiency of reinforcement learning algorithms [20] or even making novel scientific discoveries [18, 25]. However, these settings are only "single-agent", e.g. black box optimization of a particular reward signal [8, 13] or even following an intrinsic notion of interestingness [6, 18, 32]. Since multi-agent race dynamics have been shown to spur innovation in humans and the natural world, we believe integrating these dynamics with FM search algorithms will help spark an even bigger explosion of strategic creativity in multi-agent settings. Moreover, integrating prior knowledge from FMs could speed up expensive multi-agent search algorithms.

Our algorithm, Quality-Diversity Self-Play (QDSP) iteratively seeks to either propose novel strategies or improve an existing strategy to be more competitive against the current opposing population. We evaluate QDSP on the (unfortunately named) Homicidal-Chauffeur [9] (HC), a classic asymmetric pursuer-evader game, where the pursuer can move quickly but has a limited turning radius while the evader is slower, but extremely agile. QDSP implements a diverse collection of policies drawn from subfields of computer science and control theory based on tabular Q-learning [31], Monte Carlo tree search [11], evolutionary search, model predictive control [7], linear and quadratic regression [10], simple heuristics and more. Furthermore, QDSP outperforms a collection of human-designed policies as well as multi-agent policy-search algorithms inspired by Eureka [20] and OMNI [32].

2 Background and Related Work

Self-Play: Self-Play (SP) algorithms train an agent to solve a task by having it compete against itself and/or previous versions of itself. Because the opponent set is always expanding, and the algorithm continually seeks to outperform its opponents, Self-Play is an open-ended process [2]. This arms-race dynamic has been explored to great effect in training super-human agents in various matrix, board, and video games [1, 12, 15, 22, 27, 29, 30]. However, these methods often have trouble learning a large diversity of solutions, instead continually refining existing strategies.

Quality-Diversity: On the other hand, Quality-Diversity (QD) algorithms generate and curate an ever-expanding collection of diverse high-performing solutions. A canonical QD algorithm is MAP-Elites [21]. In MAP-Elites, diversity and performance are defined a priori by a collection of functions that quantify characteristics of the solution's behavior or "dimensions of variation" (i.e., how much a robot used each limb) and quality (i.e., how far the robot walked). When MAP-Elites creates a new solution, it is categorized according to its behavior and compared according to its quality. If this solution is the first to display that behavior, it is added to the archive. Otherwise, it must compete against the similar agent. The two are scored according to the performance function and only the better agent is kept [5, 21]. As a result, MAP-Elites produces a diverse collection of policies that have each been competing within their local niche to get as good as possible while keeping their unique characteristics. The MAP-Elites algorithm has been applied to fields such as robotics [5, 21] and evolving cooperative rule-based game-playing agents [3]. Quality-Diversity Self-Play combines the competitive dynamics and curriculum generation of SP with the diversity-preservation of QD.



(a) ObstacleAvoidance vs MCTSPursuit (b) ObstacleAvoidance vs MPCPursuit (c) RandomDirection-Change vs GAPursuit (d) TerrainEvasion vs MCTSPursuit

Figure 2: Selected Generated Policies from Quality-Diversity Self-Play in the HC game. Red lines represent the pursuer trajectories, while blue lines are from the evaders. The underlined agent wins. Explored algorithms include: Q-Learning, MPC, evolutionary search, heuristics, and physics-inspired attractive-repulsive policies (more in Appendix C and Appendix D). The algorithmic diversity shows that many potential solutions exist and would be difficult to manually search for and optimize.

FMs for Search: FMs are generative models trained on internet-scale repositories of a) code and b) general human-written text. Therefore, they achieve general coding competency [4] and also approximate human-like notions such as novelty [6, 32]. Therefore, FMs can be used as "intelligent" search operators when incorporated into stochastic optimization algorithms. Searching over the space of code using FMs has seen great success in single-agent problems [6, 8, 14, 16, 17, 19, 25, 32].

3 Quality-Diversity Self-Play

We propose Quality-Diversity Self-Play, to the best of our knowledge, the first algorithm integrating Quality-Diversity and Self-Play algorithms. Additionally, we believe our method is the first to leverage an FM search operator within a Self-Play loop, thereby increasing its sample efficiency. The FM searches for code-based policies that map states to actions: $\pi(s) = a$. This choice follows recent work that used FM code policies to control robots [16] and simulated agents [8, 20]. Writing control policies as code provides policies that are interpretable, extensible, and amenable to post-hoc safety-critical transformations. Since we automatically execute FM-generated policies, rigorous sandboxing is necessary to ensure safe execution. Our algorithm iterates between policy generation and population improvement for two competing populations, as illustrated in Figure 1. QDSP has a directive to seek out novel policies, and an update rule that improves policy quality.

Policy Generation: We seed QDSP with one simple human-designed policy per population. When generating new policies, e.g., a new member of p_1, QDSP provides the FM-search-operator with context including a randomly sampled member of both populations and the score of how well they perform against each other, as well as additional nearby members of p_1 to inform already explored strategies. The FM generates a new policy conditioned on the provided context and task. The generated policy goes through a refinement period to remove implementation bugs [26].

Population Improvement: The new policy is then embedded, and its k nearest neighbors are retrieved. QDSP asks the FM if the new policy is truly novel. If yes, it adds the new policy to the population; otherwise, the algorithm compares the new policy's performance against its nearest neighbor and keeps the better policy. This update pattern results in a *dimensionless*[1] MAP-Elites-style Self-Play algorithm where novel policies are added to the archive as stepping stones for future innovation while known behaviors are incrementally improved.

4 Evaluation

Problem Setting: We evaluate our algorithm on HC [9], a classic 2-body evader-pursuer game implemented as a finite discrete-time system. The chase takes place on the XY-plane and each agent outputs a continuous heading value to determine their next movement direction. The game is asymmetric because the pursuer moves more quickly but has a limited turning radius, while the evader is slower but has no such restriction. For more details, see Appendix A. QDSP and all baselines are seeded with a single simple human-written policy for the evader (psiRandom) and

[1]While the original MAP-Elites pre-defined dimensions of variation, our algorithm lets the FM automatically cluster discovered policies into groups.

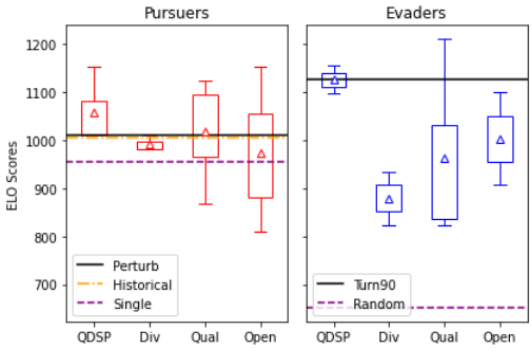


Figure 3: For each run of each algorithm, we select the highest ELO pursuer and evader to create a population of the best agents and the human-designed pursuer and evader policies (SingleStatePursuit, HistoricalPursuit, PerturbPursuit, RandomEvasion, Turn90Evasion). These agents compete in 100 round-robin tournaments against the matching opponent populations to calculate an ELO score. The mean ELO of QDSP's discovered top-end policies in both populations is as good or better than the high-quality human-designed policies (Perturb, Historical, and Turn90).

pursuer (phiSingleState) and run for 250 generations using GPT-4o as a search operator that outputs a code file implementing a policy class. Each algorithm is run three (independent) times.

Baselines: The baselines take inspiration from OMNI [32] and Eureka [20] to design diversity-only and quality-only algorithms. The **Quality-Only (Eureka-inspired) baseline**, is a Self-Play loop where the FM observes the current policies for each player and their competition outcomes. The FM suggests improvements for each policy, after which these new policies compete and the cycle repeats. Furthermore, to test the FM's base knowledge, we implement an **Open-Loop** baseline that is identical to Quality-Only, except the FM is not allowed to observe how well its policies score or any opponent policies. For the **Diversity-Only (OMNI-inspired) baseline**, we maintain two populations of players. To design a new e.g. evader, we sample an evader and pursuer from each population to evaluate them against each other and get the evader's k-nearest neighbors (using policy text embeddings) from the archive. The sampled policies, their competition outcome, and the evader's k-nn are provided to the FM, like OMNI-EPIC [6]. Then the FM designs an interestingly new policy that is added to the evader population, and the cycle repeats.

The Quality-Only (Eureka-inspired) and Open-Loop baselines only maintain the most recent policies for each role and only seek to constantly improve that current strategy whereas QDSP innovates from the entire set of stepping stones in a population's archive. While the Diversity-Only (OMNI-inspired) control maintains populations like QDSP, there is no explicit hill-climbing step to continually improve older no-longer-novel policies. Thus QDSP is a Quality-Diversity algorithm unlike the Diversity-Only (OMNI-inspired) control that focuses on creating interesting diversity only.

Strategy Evaluation and Visualizations: We include a tournament-based analysis quantifying QDSP's and the baselines' generated policies via ELO scores in Figure 3 and Appendix B. Among the generated policies, both QDSP and the Diversity-Only (OMNI-inspired) control explored policies from across computer science and control theory as shown in Appendix D. For example, both pursuers and evaders often used Kalman filters [28] and linear regression models to predict the opponent's future position. Meanwhile, some evasion policies attempted strategies that created imaginary targets on the XY-plane that it had to avoid or tried to hide next to. QDSP is the only algorithm that generates strong policies for both evader and pursuer populations. Even more impressively, when compared against strong human-designed policies, QDSP's policies are as good as the pursuer baseline and better than the evader baseline (Figure 3). Interestingly, the highest-performing pursuers included Monte Carlo tree-search and genetic algorithm policies that implemented their own reward and forward models for finding optimal heading angles. The optimal evader, which achieved higher ELO than the human-designed Turn90 policy (Figure 3b), was a simple heuristic policy that calculates the pursuer's approach vector and moves away from its next location. Figure 2 shows a sample of competitions between various QDSP-generated pursuers (red) and evaders (blue). Additional policies are visualized in Appendix C with example codes in Appendix D.

Measuring Quality and Diversity: To measure the quality of QDSP and the controls, we create shared evaluation populations of pursuers and evaders. Given the competitive nature of the game and the fact that each algorithm bootstraps its populations of policies, there is no shared benchmark to compare each run against (besides the human-designed policies). Therefore we do a post-hoc quality analysis where we construct a shared population from each experiment that is made up of the: human-designed policies, the top-3 policies from each experiment (calculated using ELO scores via an intra-experiment tournament), and then 15 random policies from each experiment. Each algorithm's generated policies compete against this shared population's opposing policies. This

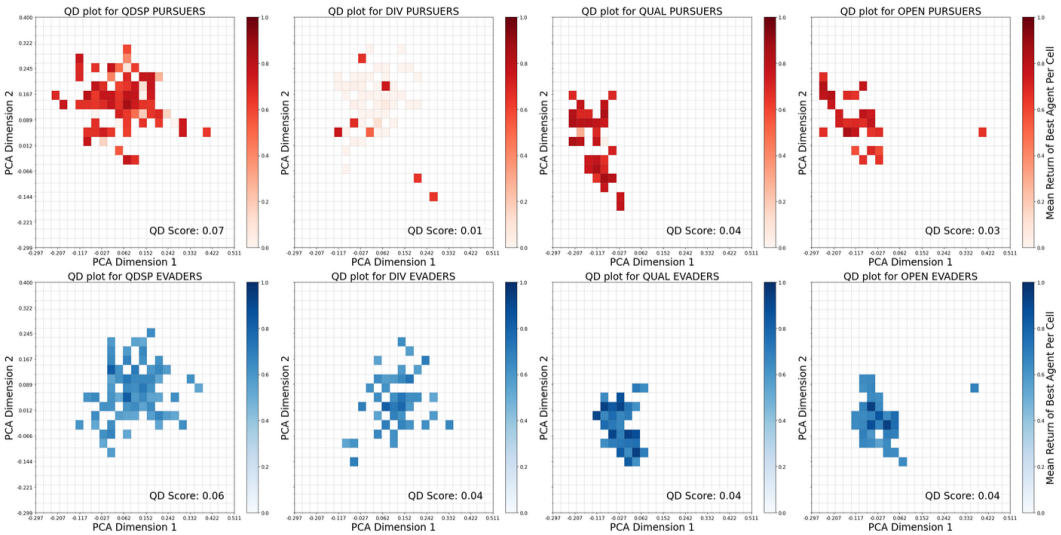


Figure 4: Example QD Plots for each algorithm that show the diversity and quality of the solutions found. The QD Plots are defined using policy embedding vectors. We take all of the policy embeddings from all the experiments and compress them to 2 dimensions via Principle Component Analysis [23]. The resulting space is discretized into 25x25 bins and each policy is put into its appropriate category. The QD map maintains only the highest-performing policy per cell. The QD Score then calculates the average of the Map combining how much and how well the space has been explored into a shared metric. For further QD-Score analysis see Figure 5 in Appendix B.

generates a shared objective score can be compared across algorithms. We incorporate their quality information into the QD-Plots seen in Figure 4 and discussed below.

Which algorithm created the best policies? To answer that, we take a champion from each run and have them compete against each other. To measure just the quality of the high-end policies, we subsample the evaluation population to the top-1 policy from each algorithm's run and the human-designed policies and have them compete against each other. This removes as many confounding variables as possible when determining which algorithm generated the best policy. QDSP's top-end policies achieve the highest mean ELO scores of the top-1 policies in Figure 3.

Measuring the diversity of QDSP and the controls is a more involved process. To measure each algorithm's diversity, we take all of the policy embeddings and create a shared 2-dimensional space across the experiments. This space is created by doing a 2-dimensional PCA-transformation [23] of all the generated policies which is then chunked into 625 equal bins (25 x 25) defining a Map. Each policy can be categorized into the Map based on where the PCA transform places the embedding vector. The Map uses the quality-information to keep the best policy for each cell in the space. The diversity of each algorithm can then be calculated as the number of filled cells in the Map. Furthermore, we derive the QD-Score of the algorithm to show both the achieved diversity of the solutions found along with their quality (discussed more in Appendix B), as the mean of the Map (including not-filled-in cells as zeroes in the mean calculation) [24]. As seen in Figure 5 (Appendix B), QDSP achieves the highest QD-Score.

5 Conclusion

In this paper, we have introduced the first Quality-Diversity Self-Play algorithm. Furthermore, we demonstrate the power of foundation models as powerful search operators in (already powerful) Self-Play algorithms. We demonstrate that Quality-Diversity Self-Play discovers a wide variety of high-performing strategies in the competitive multi-agent HC game, spanning many disciplines of CS and control. However, the principles here extend beyond code-as-policies; FMs could also define strategies in terms of reward functions for an RL agent to optimize [20] within or beyond Self-Play, which could enable future versions of QDSP to discover and train neural network strategies that are hard to define programmatically.

References

[1] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and IGOR Mordatch. Emergent tool use from multi-agent interaction. *Machine Learning, Cornell University*, 2019.

[2] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. In *International Conference on Machine Learning*, pages 434–443. PMLR, 2019.

[3] Rodrigo Canaan, Julian Togelius, Andy Nealen, and Stefan Menzel. Diverse agents for ad-hoc cooperation in hanabi. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019. doi: 10.1109/CIG.2019.8847944.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021. URL `https://arxiv.org/abs/2107.03374`.

[5] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can Adapt like Animals. *Nature*, 521(7553):503–507, 2015.

[6] Maxence Faldor, Jenny Zhang, Antoine Cully, and Jeff Clune. OMNI-EPIC: Open-endedness via Models of human notions of Interestingness with Environments Programmed in Code, 2024.

[7] Carlos E. Garcia, David M. Prett, and Manfred Morari. Model Predictive Control: Theory and practice – A Survey. *Automatica*, 25(3):335–348, 1989. ISSN 0005-1098. doi: https://doi.org/10.1016/0005-1098(89)90002-2. URL `https://www.sciencedirect.com/science/article/pii/0005109889900022`.

[8] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

[9] R. Isaacs and Rand Corporation. *Games of Pursuit: P-257. 17 November 1951*. Contributions to the theory of games. Rand Corporation, 1951. URL `https://books.google.ca/books?id=Z2t70AEACAAJ`.

[10] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[11] Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. pages 282–293. Springer Berlin Heidelberg, 2006. doi: 10.1007/11871842_29. URL `https://doi.org/10.1007/11871842_29`.

[12] Marc Lanctot, Vinicius Zambaldi, Audrūnas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4193âĂŞ4206, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[13] Robert Tjarko Lange, Yingtao Tian, and Yujin Tang. Large language models as evolution strategies, 2024. URL `https://arxiv.org/abs/2402.18381`.

[14] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pages 331–366. Springer, 2023.

[15] Joel Z. Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research, 2019.

[16] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *arXiv preprint arXiv:2209.07753*, 2022.

[17] Chris Lu, Samuel Holt, Claudio Fanconi, Alex J Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. Discovering preference optimization algorithms with and for large language models. *arXiv preprint arXiv:2406.08414*, 2024.

[18] Chris Lu, Cong Lu, Robert Lange, Jakob N Foerster, Jeff Clune, and David Ha. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. *arXiv preprint arXiv:2408.06292*, 2024.

[19] Cong Lu, Shengran Hu, and Jeff Clune. Intelligent Go-Explore: Standing on the Shoulders of Giant Foundation Models, 2024.

[20] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2024.

[21] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[22] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, PrzemysÅĆaw DÄŹbiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal JÃşzefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[23] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559âĂŞ572, November 1901. ISSN 1941-5990. doi: 10.1080/14786440109462720. URL `http://dx.doi.org/10.1080/14786440109462720`.

[24] Justin K. Pugh, L. B. Soros, Paul A. Szerlip, and Kenneth O. Stanley. Confronting the challenge of quality diversity. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO âĂŹ15. ACM, July 2015. doi: 10.1145/2739480.2754664. URL `http://dx.doi.org/10.1145/2739480.2754664`.

[25] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2023. doi: 10.1038/s41586-023-06924-6.

[26] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Neural Information Processing Systems*, 2023. URL `https://api.semanticscholar.org/CorpusID:258833055`.

[27] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[28] Dan Simon. *Optimal State Estimation: Kalman, H∞, and Nonlinear Approaches*. Wiley, January 2006. ISBN 9780470045343. doi: 10.1002/0470045345. URL `http://dx.doi.org/10.1002/0470045345`.

[29] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994. doi: 10.1162/neco.1994.6.2.215.

[30] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[31] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3âĂŞ4): 279âĂŞ292, May 1992. ISSN 1573-0565. doi: 10.1007/bf00992698. URL `http://dx.doi.org/10.1007/BF00992698`.

[32] Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. OMNI: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023.

Supplementary Material

Table of Contents

A Homicidal Chauffeur Update Equations **10**

B Evaluation Results **10**

C Additional Visualizations **12**

D Code Policies **16**

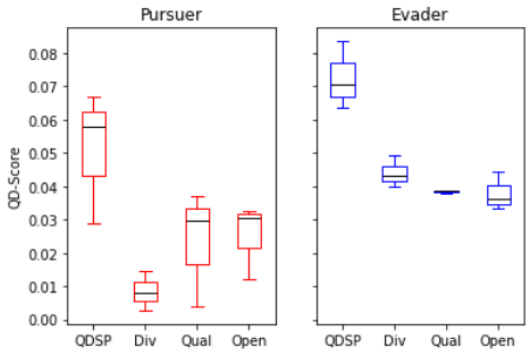


Figure 5: Using QD Plots, like Figure 4, we derive the QD-Score of each algorithm. QDSP achieves the highest QD-Score across all our experiments showing that QDSP both explores many different solution types and improves their quality. The Diversity-Only baseline achieves high coverage of the search space, but does not achieve high quality while the Quality-Only and Open-Loop baselines achieve high quality but low coverage thus bringing down their respective scores.

A Homicidal Chauffeur Update Equations

The problem used in Section 4 is a 2-dimensional Homicidal Chauffeur problem where we have two agents that are navigating around the XY-plane: one evader, e, and one pursuer, p. The pursuer has a minimum turn radius of R and moves at speed s_1 according to heading-angle ϕ while the evader moves at speed s_2 according to heading-angle ψ with $s_1 > s_2$. The next xy-locations for each agent are defined as follows:

$$\dot{\theta} = \frac{s_1}{R}\phi_t \tag{1}$$

$$x_{p,t+1} = x_{p,t} + s_1 \sin(\phi_{t-1} + \dot{\theta}) \tag{2}$$

$$y_{p,t+1} = y_{p,t} + s_1 \cos(\phi_{t-1} + \dot{\theta}) \tag{3}$$

$$x_{e,t+1} = x_{e,t} + s_2 \sin(\psi_t) \tag{4}$$

$$y_{p,t+1} = y_{e,t} + s_2 \cos(\psi_t) \tag{5}$$

$$\phi_{t+1} = \dot{\theta} \tag{6}$$

B Evaluation Results

In addition to the existing visualization in our main paper Section 4, we present further evaluation details and statistics from our tournament evaluation here.

The human-written seed policies are: SingleStatePursuit which calculates the pursuer's optimal heading angle to minimize the distance between the current positions of the pursuer and evader and RandomEvasion which randomly changes direction every 20 timesteps.

Each algorithm creates candidate policies. As an early measure of their quality, the two populations (augmented to include the hand-written policies) compete in 100 round-robin tournaments with the opposing population. These match outcomes update ELO scores for each policy. While ELO scores are incomparable across experiments (i.e., Quality-Only (Eureka-inspired)-policy ELOs do not correspond to QDSP-policy ELOs), they can be compared within each treatment. A secondary evaluation is then run on the two populations by comparing them against a shared evaluation population described in Section 4. Only QDSP created policies that consistently outperform the high-quality hand-written solutions in *both* populations as shown in Figure 3. While Quality-Only

(Eureka-inspired) managed to find good pursuers, the overall quality of their evaders was low. Similarly, for Diversity-Only (OMNI-inspired) and the Open-Loop control found some good pursuers, but their evaders were weaker than the high-quality human-designed policy. Because each algorithm played against the same human-written policies, they can serve as a shared benchmark of quality across the experiments.

The QD-Score of the policies found by the different algorithms is shown in Figure 5 with QDSP showing the highest QDScore. This indicates that QDSP did the best at balancing fine-tuning existing policies while also exploring new solutions.

C Additional Visualizations

We provide additional visualizations for generated agents discussed in Section 4, in addition to existing figures in Figure 6. We sampled 26 evader and pursuer combinations at random for inclusion. Pursuer trajectories are in red while evader trajectories are in blue.

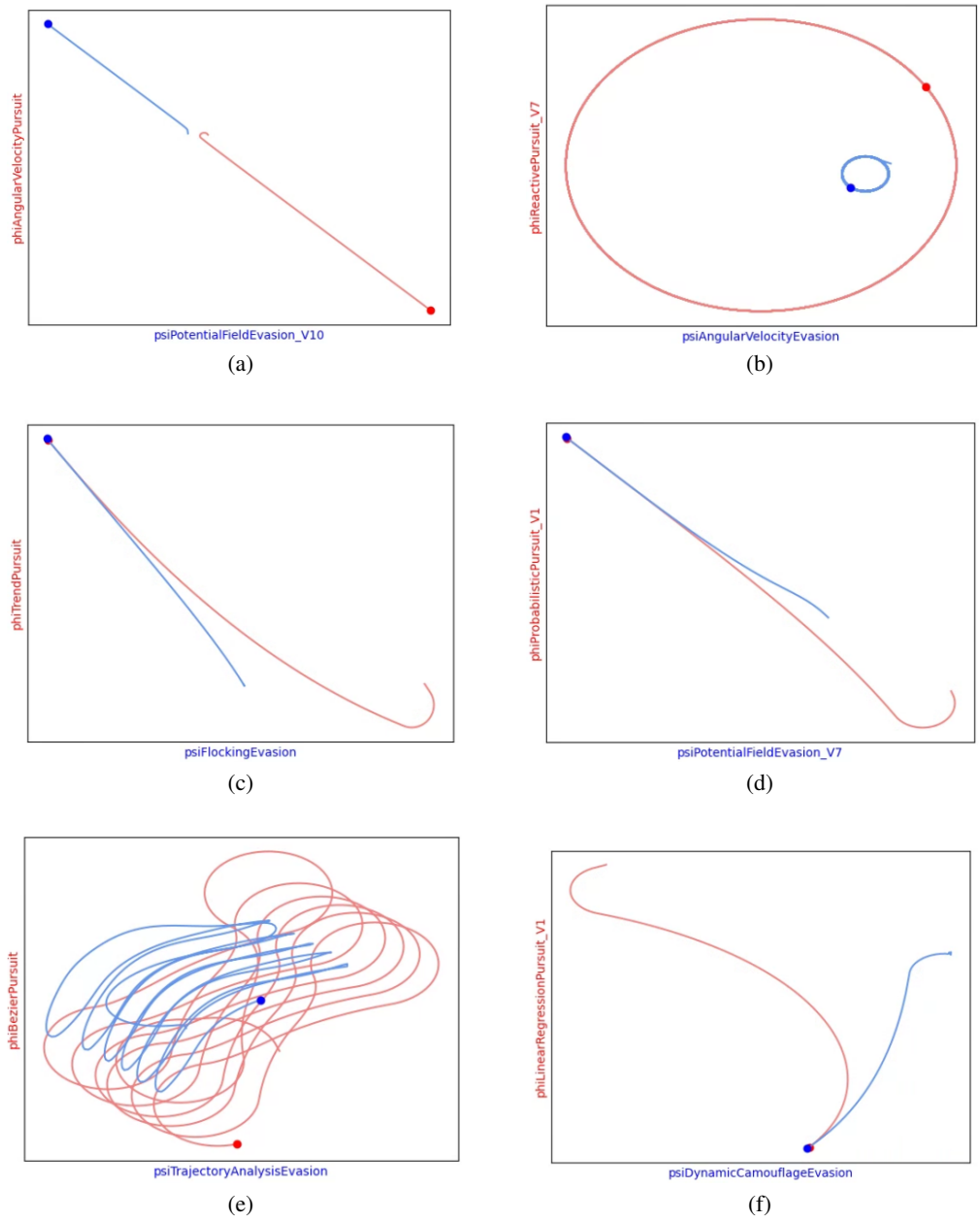
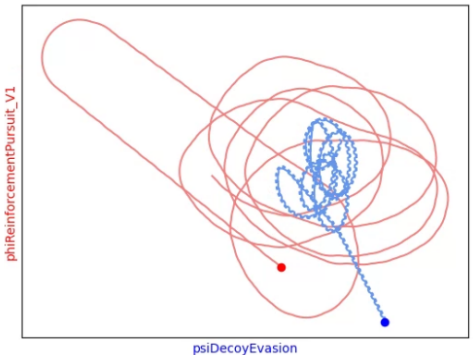
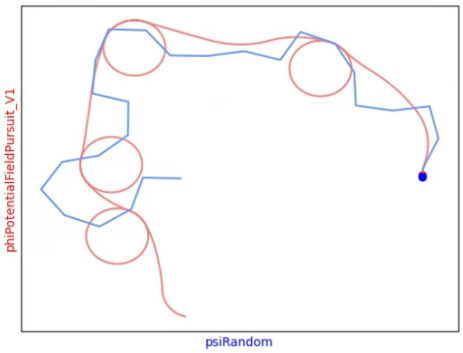


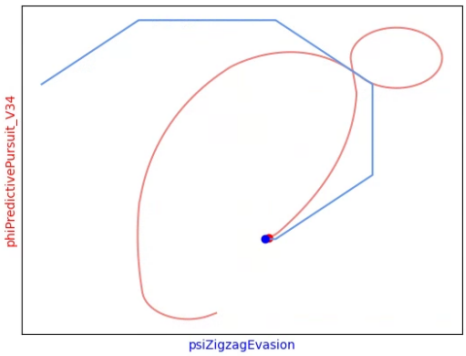
Figure 6: 26 randomly sampled policies from each population. Red trajectories are pursuers and Blue trajectories are from evaders.



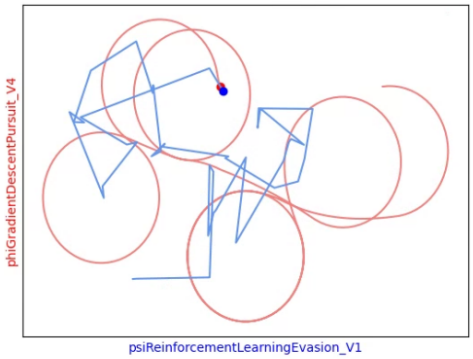
(g)



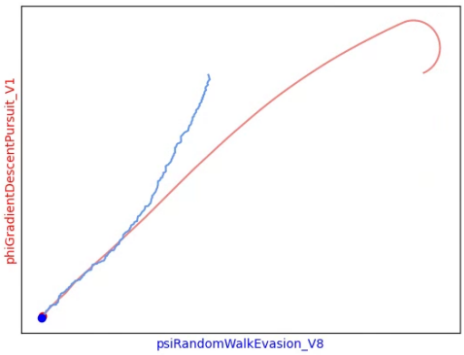
(h)



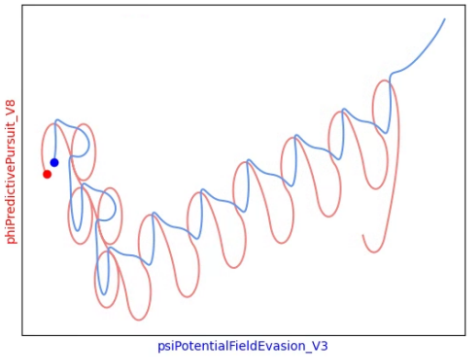
(i)



(j)

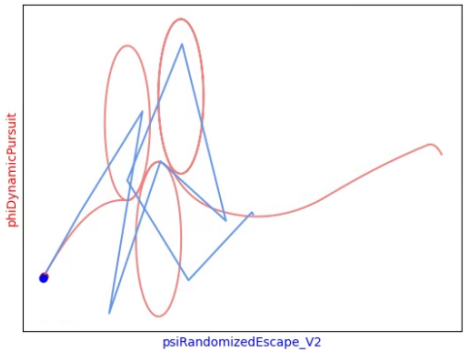


(k)

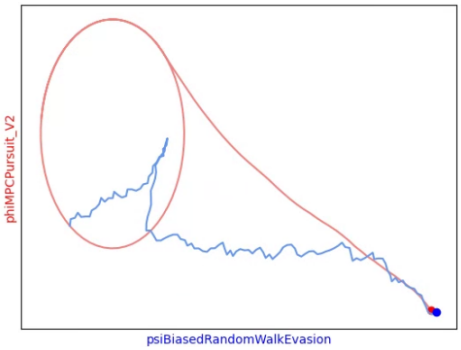


(l)

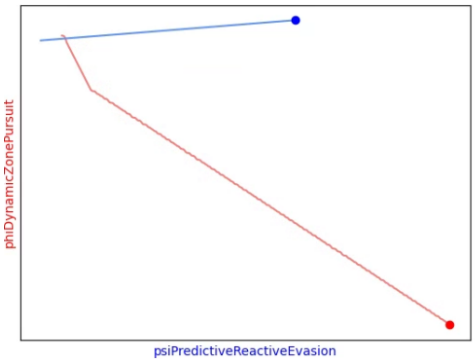
Figure 6: Sample Trajectories (continued).



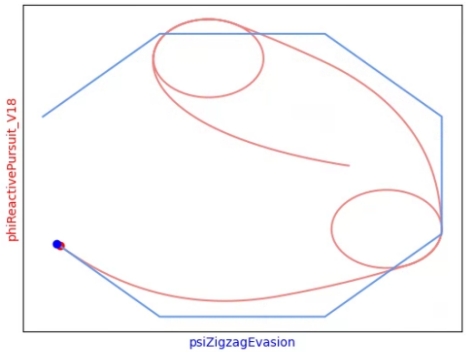
(m)



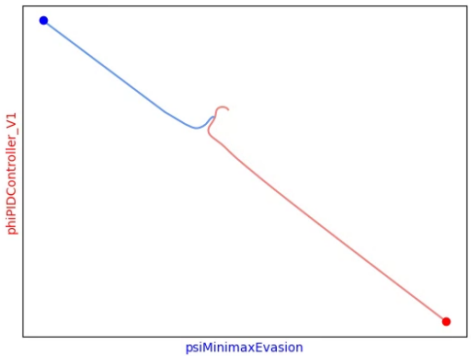
(n)



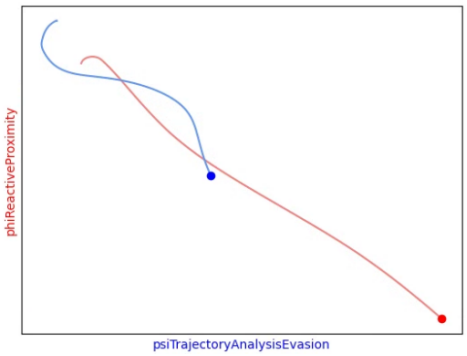
(o)



(p)

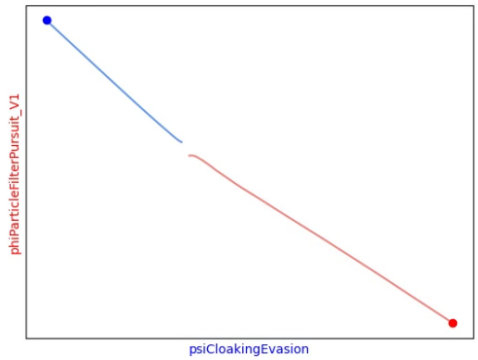


(q)

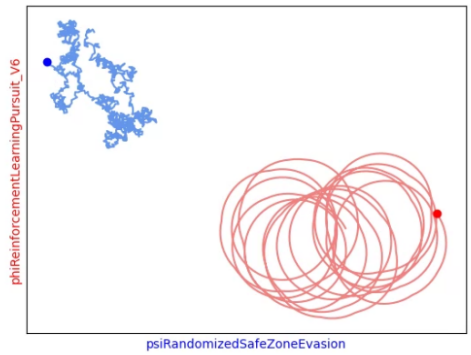


(r)

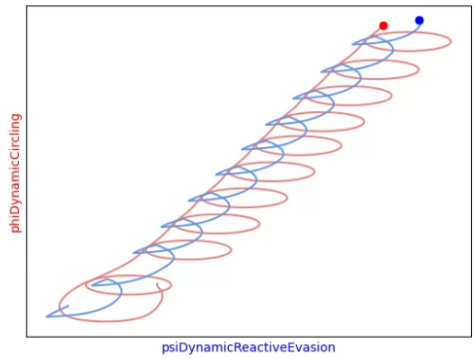
Figure 6: Sample Trajectories (continued).



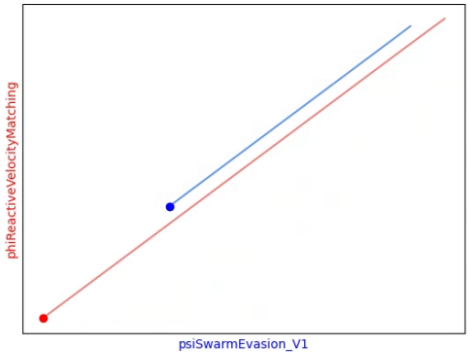
(s)



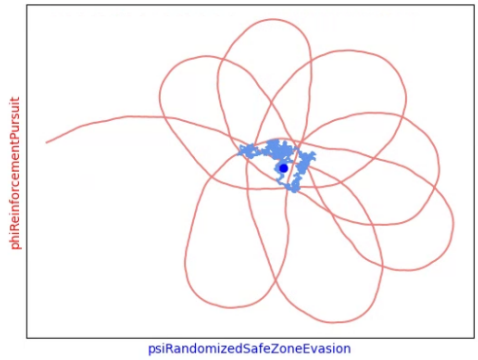
(t)



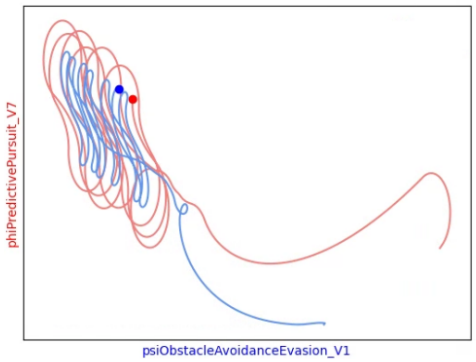
(u)



(v)

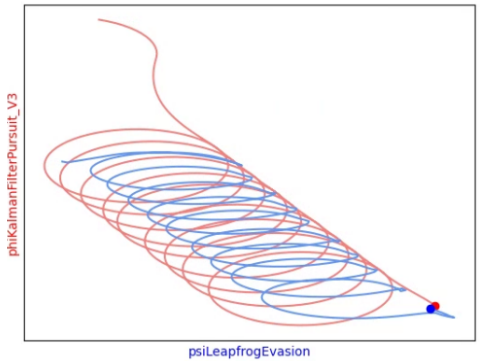


(w)

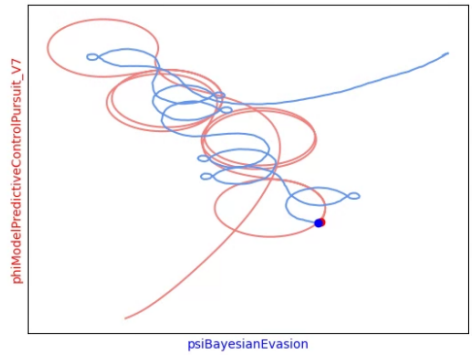


(x)

Figure 6: Sample Trajectories (continued).



(y)



(z)

Figure 6: Sample Trajectories (continued).

D Code Policies

Below are example generated policies. Policies with "phi" in the class name are pursuer policies while policies with "psi" in the class name are evader policies.

Monte-Carlo-Tree-Search Pursuer Policy seen in Figure 2:

```
import numpy as np
import math
import random

class phiMCTSPursuit:
    def __init__(self, consts=(0.01, 0.006, 0.1), simulation_depth=10,
                                        exploration_param=1.4):
        self.description = "phi calculation using Monte Carlo Tree
                                        Search (MCTS) to explore
                                        potential future states and
                                         optimize the pursuer's
                                        heading angle"
        self.__name__ = "phiMCTSPursuit"
        self.consts = consts
        self.simulation_depth = simulation_depth  # Depth of the tree
                                        search
        self.exploration_param = exploration_param  # Exploration
                                        parameter for UCB1

    def ucb1(self, node, total_visits):
        if node['visits'] == 0:
            return float('inf')
        return node['reward'] / node['visits'] + self.
                                        exploration_param * math.
                                        sqrt(math.log(total_visits)
                                         / node['visits'])

    def simulate(self, state, depth):
        if depth == 0:
            return 0
        x = state
        total_reward = 0
        for _ in range(depth):
            action = random.uniform(-1, 1)
            theta_dot = self.consts[0] / self.consts[2] * action
```

```
            x_next = dXdt(x, [action, x[2]])  # Assume evader
                                                maintains same heading
                                                for simplicity
            distance = np.sqrt((x_next[0] - x_next[3]) ** 2 + (x_next[
                                                1] - x_next[4]) ** 2)
            total_reward -= distance
            x = x_next
        return total_reward

    def mcts(self, state):
        tree = {}
        tree[str(state)] = {'reward': 0, 'visits': 0, 'children': {}}
        total_visits = 0

        for _ in range(self.simulation_depth):
            path = []
            current_state = state
            depth = 0

            for depth in range(self.simulation_depth):
                node = tree[str(current_state)]
                if not node['children']:
                    break
                action = max(node['children'], key=lambda a: self.ucb1
                                                    (node['children'][a
                                                    ], node['visits']))
                path.append((current_state, action))
                theta_dot = self.consts[0] / self.consts[2] * action
                current_state = dXdt(current_state, [action,
                                                    current_state[2]])

            if str(current_state) not in tree:
                tree[str(current_state)] = {'reward': 0, 'visits': 0,
                                                    'children': {}}
            reward = self.simulate(current_state, self.
                                                simulation_depth -
                                                depth)

            for state, action in reversed(path):
                node = tree[str(state)]
                if action not in node['children']:
                    node['children'][action] = {'reward': 0, 'visits':
                                                        0}
                node['children'][action]['reward'] += reward
                node['children'][action]['visits'] += 1
                node['reward'] += reward
                node['visits'] += 1
                total_visits += 1

        best_action = max(tree[str(state)]['children'], key=lambda a:
                                            tree[str(state)]['children'
                                            ][a]['reward'])
        return best_action

    def __call__(self, X):
        if len(X) < 2:
            return 0  # Not enough data to perform meaningful
                                                calculation
        state = X[-1]
        return self.mcts(state)

const = (0.01, 0.006, 0.1)

def dXdt(x0, input):
    # theta dot limiter
```

```
    if abs(input[0]) > 1:
        input[0] = 1 * np.sign(input[0])

    x_dot = np.empty(5)

    # simultaneous update of theta
    theta_dot = const[0] / const[2] * input[0]
    x_dot[0] = const[0] * np.sin(x0[2] + theta_dot)
    x_dot[1] = const[0] * np.cos(x0[2] + theta_dot)
    x_dot[2] = theta_dot

    x_dot[3] = const[1] * np.sin(input[1])
    x_dot[4] = const[1] * np.cos(input[1])
    return x0 + x_dot
```

Sample Evader seen in Figure 2

```
import numpy as np

class psiDynamicObstacleAvoidanceEvasion:
    def __init__(self, consts=(0.01, 0.006, 0.1), obstacle_radius=0.15
                                        ):
        self.description = "psi calculation using dynamic obstacle
                                        avoidance to create a
                                        zigzag evasion pattern"
        self.__name__ = "psiDynamicObstacleAvoidanceEvasion"
        self.consts = consts
        self.obstacle_radius = obstacle_radius

    def __call__(self, psi, ii, X):
        if len(X) < 2:
            return psi  # Not enough data to perform meaningful
                                            calculation

        x = X[-1]
        evader_pos = np.array([x[3], x[4]])

        # Calculate the direction to the pursuer
        dx_pursuer = x[3] - x[0]
        dy_pursuer = x[4] - x[1]
        angle_to_pursuer = np.arctan2(dy_pursuer, dx_pursuer)

        # Create dynamic obstacles around the evader's current
                                        position
        obstacle_angle = angle_to_pursuer + np.pi / 4  # 45 degrees
                                        offset from the pursuer
                                        direction
        obstacle_pos = evader_pos + self.obstacle_radius * np.array([
                                        np.sin(obstacle_angle), np.
                                        cos(obstacle_angle)])

        # Calculate the avoidance vector from the obstacle
        dx_obstacle = evader_pos[0] - obstacle_pos[0]
        dy_obstacle = evader_pos[1] - obstacle_pos[1]
        distance_to_obstacle = np.sqrt(dx_obstacle ** 2 + dy_obstacle
                                        ** 2)
        avoidance_vector = np.array([dx_obstacle, dy_obstacle]) / (
                                        distance_to_obstacle + 1e-5
                                        )

        # Calculate the final heading direction for the evader
        final_vector = avoidance_vector + np.array([np.sin(
                                        angle_to_pursuer), np.cos(
                                        angle_to_pursuer)])
```

```
        new_psi = np.arctan2(final_vector[1], final_vector[0])

        # Normalize psi to be within [-pi, pi]
        psi = (new_psi + np.pi) % (2 * np.pi) - np.pi

        return psi
```

Model-Predictive Control Pursuer Policy:

```
import numpy as np
from scipy.optimize import minimize

class phiModelPredictiveControlPursuit:
    def __init__(self, consts=(0.01, 0.006, 0.1), horizon=10,
                                    control_weight=0.1):
        self.description = "phi calculation using Model Predictive
                                        Control to optimize the
                                        pursuer's heading angle
                                        over a finite horizon"
        self.__name__ = "phiModelPredictiveControlPursuit"
        self.consts = consts
        self.horizon = horizon  # Prediction horizon
        self.control_weight = control_weight  # Weight for control
                                        effort in the cost function

    def predict_evader_positions(self, X, psi):
        evader_positions = []
        evader_x, evader_y = X[-1][3], X[-1][4]
        for _ in range(self.horizon):
            evader_x += self.consts[1] * np.sin(psi)
            evader_y += self.consts[1] * np.cos(psi)
            evader_positions.append((evader_x, evader_y))
        return evader_positions

    def cost_function(self, phi, X, evader_positions):
        pursuer_x, pursuer_y, pursuer_theta = X[-1][:3]
        cost = 0
        for i in range(self.horizon):
            theta_dot = self.consts[0] / self.consts[2] * phi
            pursuer_theta += theta_dot
            pursuer_x += self.consts[0] * np.sin(pursuer_theta)
            pursuer_y += self.consts[0] * np.cos(pursuer_theta)
            evader_x, evader_y = evader_positions[i]
            distance = np.sqrt((pursuer_x - evader_x) ** 2 + (
                                            pursuer_y - evader_y)
                                            ** 2)
            cost += distance + self.control_weight * np.abs(phi)
        return cost

    def __call__(self, X):
        if len(X) < 2:
            return 0  # Not enough data to perform meaningful
                                            calculation

        psi = X[-1][2]  # Use the current heading angle of the evader
                                        as the prediction base
        evader_positions = self.predict_evader_positions(X, psi)
        result = minimize(self.cost_function, x0=0, args=(X,
                                        evader_positions), bounds=[
                                        (-1, 1)])
        return result.x[0]
```

A Genetic Algorithm Policy for selecting direction headings:

```python
import numpy as np

class phiGeneticAlgorithmPursuit:
    def __init__(self, consts=(0.01, 0.006, 0.1), population_size=30,
                                        generations=50, mutation_rate=0
                                        .1):
        self.description = "phi calculation using Genetic Algorithm (
                                            GA) to evolve the best
                                            pursuit strategy over
                                            multiple generations"
        self.__name__ = "phiGeneticAlgorithmPursuit"
        self.consts = consts
        self.population_size = population_size  # Number of
                                            individuals in the
                                            population
        self.generations = generations  # Number of generations to
                                            evolve
        self.mutation_rate = mutation_rate  # Probability of mutation
        self.population = np.random.uniform(-1, 1, population_size)  #
                                             Initialize population with
                                             random phi values

    def evaluate_fitness(self, X):
        x = X[-1]
        pursuer_x, pursuer_y, pursuer_theta, evader_x, evader_y = x
        fitness = np.zeros(self.population_size)
        for i in range(self.population_size):
            phi = self.population[i]
            theta_dot = self.consts[0] / self.consts[2] * phi
            new_pursuer_x = pursuer_x + self.consts[0] * np.sin(
                                                pursuer_theta +
                                                theta_dot)
            new_pursuer_y = pursuer_y + self.consts[0] * np.cos(
                                                pursuer_theta +
                                                theta_dot)
            distance_to_evader = np.sqrt((new_pursuer_x - evader_x) **
                                                 2 + (new_pursuer_y -
                                                evader_y) ** 2)
            fitness[i] = -distance_to_evader  # Negative distance for
                                                maximization problem
        return fitness

    def select_parents(self, fitness):
        probabilities = fitness - fitness.min() + 1e-6  # Avoid
                                            division by zero
        probabilities /= probabilities.sum()  # Normalize to make a
                                            probability distribution
        parents_indices = np.random.choice(self.population_size, size=
                                            self.population_size, p=
                                            probabilities)
        return self.population[parents_indices]

    def crossover(self, parents):
        offspring = np.empty(self.population_size)
        crossover_point = np.random.randint(1, self.population_size -
                                            1)
        for i in range(0, self.population_size, 2):
            parent1, parent2 = parents[i], parents[i + 1]
            offspring[i] = np.concatenate((parent1[:crossover_point],
                                                parent2[crossover_point
                                                :]))
            offspring[i + 1] = np.concatenate((parent2[:
                                                crossover_point],
                                                parent1[crossover_point
                                                :]))
```

```
        return offspring

    def mutate(self, offspring):
        for i in range(self.population_size):
            if np.random.rand() < self.mutation_rate:
                mutation_value = np.random.uniform(-1, 1)
                offspring[i] += mutation_value
                offspring[i] = np.clip(offspring[i], -1, 1)  # Ensure
                                                              phi values are
                                                              within [-1, 1]
        return offspring

    def __call__(self, X):
        if len(X) < 2:
            return 0  # Not enough data to perform meaningful
                                                  calculation

        for _ in range(self.generations):
            fitness = self.evaluate_fitness(X)
            parents = self.select_parents(fitness)
            offspring = self.crossover(parents)
            self.population = self.mutate(offspring)

        best_individual_index = np.argmax(self.evaluate_fitness(X))
        return self.population[best_individual_index]
```

A physics-inspired attraction-based policy:

```
    import numpy as np

class phiStochasticAttractionPursuit:
    def __init__(self, consts=(0.01, 0.006, 0.1), attraction_coeff=1.0
                                        , randomness_coeff=0.5):
        self.description = "phi calculation using a combination of
                                        deterministic attraction to
                                         the evader and random
                                        exploration"
        self.__name__ = "phiStochasticAttractionPursuit"
        self.consts = consts
        self.attraction_coeff = attraction_coeff  # Coefficient for
                                        attractive force towards
                                        evader
        self.randomness_coeff = randomness_coeff  # Coefficient for
                                        random exploration

    def __call__(self, X):
        if len(X) < 2:
            return 0  # Not enough data to perform meaningful
                                                  calculation

        x = X[-1]
        pursuer_x, pursuer_y, pursuer_theta, evader_x, evader_y = x

        # Calculate attractive force towards the evader
        dx = evader_x - pursuer_x
        dy = evader_y - pursuer_y
        distance_to_evader = np.sqrt(dx ** 2 + dy ** 2)
        attraction_heading = np.arctan2(dy, dx)
        attraction_error = attraction_heading - pursuer_theta
        attraction_error = (attraction_error + np.pi) % (2 * np.pi) -
                                        np.pi  # Normalize to [-pi,
                                         pi]

        # Add random exploration component
```

```
        random_exploration = np.random.uniform(-1, 1) * self.
                                        randomness_coeff

        # Combine the deterministic attraction and random exploration
        phi = self.attraction_coeff * attraction_error +
                                        random_exploration

        # Clip phi to be within [-1, 1]
        phi = np.clip(phi, -1, 1)

        return phi
```

Q-Learning Evader Policy!

```
import numpy as np
import random

class psiQlearningEvasion:
    def __init__(self, consts=(0.01, 0.006, 0.1), learning_rate=0.1,
                                    discount_factor=0.9, epsilon=0.
                                    1):
        self.description = "psi calculation using Q-learning to
                                        adaptively learn the
                                        optimal evasion strategy"
        self.__name__ = "psiQlearningEvasion"
        self.consts = consts
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = {}
        self.prev_state = None
        self.prev_action = None

    def state_to_key(self, x):
        # Discretize the state for the Q-table
        state = (int(x[0] * 10), int(x[1] * 10), int(x[3] * 10), int(x
                                        [4] * 10))
        return state

    def choose_action(self, state):
        if state not in self.q_table:
            self.q_table[state] = np.zeros(8)  # Initialize Q-values
                                                for 8 possible actions
                                                (angles)

        if random.uniform(0, 1) < self.epsilon:
            return random.randint(0, 7)  # Explore: choose a random
                                                action
        else:
            return np.argmax(self.q_table[state])  # Exploit: choose
                                                the best action based
                                                on Q-values

    def update_q_table(self, reward, new_state):
        if self.prev_state is not None and self.prev_action is not
                                        None:
            prev_q_value = self.q_table[self.prev_state][self.
                                                prev_action]
            max_future_q = np.max(self.q_table[new_state]) if
                                                new_state in self.
                                                q_table else 0
            new_q_value = prev_q_value + self.learning_rate * (reward
                                                + self.discount_factor
                                                * max_future_q -
                                                prev_q_value)
```

```
            self.q_table[self.prev_state][self.prev_action] =
                                                new_q_value

    def __call__(self, psi, ii, X):
        if len(X) < 2:
            return psi  # Not enough data to perform meaningful
                                                calculation

        x = X[-1]
        current_state = self.state_to_key(x)
        action = self.choose_action(current_state)
        angle = action * (2 * np.pi / 8) - np.pi  # Convert action
                                            index to angle

        # Simulate one step to get the new state and calculate reward
        x_dot = np.empty(5)
        x_dot[3] = self.consts[1] * np.sin(angle)
        x_dot[4] = self.consts[1] * np.cos(angle)
        new_x = x.copy()
        new_x[3] += x_dot[3]
        new_x[4] += x_dot[4]
        new_state = self.state_to_key(new_x)
        reward = -np.sqrt((x[0] - new_x[3])**2 + (x[1] - new_x[4])**2)
                                              # Negative distance to
                                            pursuer

        self.update_q_table(reward, new_state)
        self.prev_state = current_state
        self.prev_action = action

        return angle
```