
SWARMRL: BUILDING THE FUTURE OF SMART ACTIVE SYSTEMS

Samuel Tovey[†], Christoph Lohrmann[†], Tobias Merkt

David Zimmer, Konstantin Nikolaou, Simon Koppenhöfer

Anna Bushmakina, Jonas Scheunemann, Christian Holm

Institute for Computational Physics

University of Stuttgart

70569, Stuttgart, Germany

{stovey, clohrmann, holm}@icp.uni-stuttgart.de

ABSTRACT

This work introduces **SwarmRL**, a Python package designed to study intelligent active particles. **SwarmRL** provides an easy-to-use interface for developing models to control microscopic colloids using classical control and deep reinforcement learning approaches. These models may be deployed in simulations or real-world environments under a common framework. We explain the structure of the software and its key features and demonstrate how it can be used to accelerate research. With **SwarmRL**, we aim to streamline research into micro-robotic control while bridging the gap between experimental and simulation-driven sciences. **SwarmRL** is available open-source on GitHub at <https://github.com/SwarmRL/SwarmRL>.

Keywords Active Matter, Deep Reinforcement Learning, HPC, Micro-robotics

1 Introduction

Mastering control of microrobots at the microscopic scale has the potential for insights and the development of new technologies capable of changing the world. Whether it be an improved understanding of bacterial navigation strategies or direct control over microscopic robots, numerous fields, including construction [Hsu et al., 2016], plant pollination and ecosystem defense [Stefanec et al., 2022], and search and rescue [Murphy et al., 2008], will be enhanced by advancements in micro-scale control research. One field in particular that will see significant changes is medicine, where it is expected that micro robotic agents will be capable of advanced treatment strategies including targeted cancer therapies [Schmidt et al., 2020], assisted and drug delivery [Nelson and Pané, 2023, Felfoul et al., 2016, Hosseinidoust et al., 2016, Zhuang et al., 2015], assisted fertilisation [Medina-Sánchez et al., 2016, Khalil et al., 2014], amongst many others [Nelson et al., 2010, Li et al., 2022]. The two most significant challenges in realising this future are developing the tools and materials to build such capable microscopic agents and programming them to achieve complex tasks with minimal observed input. These agents should not only be capable of performing complex actions such as swimming, pushing, and perhaps, communications, but they should do so without constant input from a supervisor. The first of these challenges has been, and is being, extensively investigated. Currently, there are many approaches for designing mobile micro robots [Dabbagh et al., 2022]. Initial approaches included the light-driven Janus particles [Su et al., 2019] and coils [Wang et al., 2019] and magnetically driven devices taking on numerous shapes [Shen et al., 2023, Dhatt-Gauthier et al., 2023]. Recent approaches have also included using living cells to produce motion on this microscopic scale, the Xenobots being a prime example [Kriegman et al., 2020, Blackiston et al., 2021]. However, the control of these agents, once built and deployed, is a complicated problem. Due to the size of these agents, directly

²These authors contributed equally

installing computational processing power onto them is, thus far, intractable. Therefore, simplistic algorithms that aim to encode reactionary behaviour into the agents are often constructed, something that would require limited on-board processing. Examples of this include the use of microrobots to perform chemotaxis [Hartl et al., 2021] by changing their shapes upon exposure to varying fields, object manipulation [Kim et al., 2016] through the use of an external magnetic field, and reproducing swarming behaviour by applying simple rotations and translations to individual agents by a set of rules triggered by changes in local environment [B  uerle et al., 2020, Lavergne et al., 2019]. More recent approaches have utilised reinforcement learning approaches to further push in the direction of learned control strategies. Qin et al. [2023] utilised a Q-learning algorithm to learn swimming strategies in gated microswimmers, Borra et al. [2022] used an actor-critic framework to learn competitive capture-evasion strategies from hydrodynamic cues, and Mu   os-Landin et al. [2021] investigated actor-critic learned navigation strategies under the influence of stochastic environments. In all cases, demonstrating under what conditions microrobots can perform complex tasks in various environments brings us a step closer to realising the full potential of this technology.

It is clear that research in the direction of micro-robotic control is both promising and excitingly multi-disciplinary; however, due to the nature of the technology required, whether it be complex experimental equipment, deep knowledge of state-of-the-art machine learning algorithms and their implementation, or the expertise to construct physically realistic simulations, the entry barrier can be high for specialists who focus on only one of these areas. For this reason, we have developed the SwarmRL Python package to combine each control workflow element under a common framework and accelerate algorithm development and deployment. SwarmRL enables researchers to apply control strategies to particles in simulation or experiments. These strategies can be built from sets of rules that are rigidly programmed and utilise the environment of the particles, or they can be trained using the actor-critic reinforcement learning approach, offering complete flexibility in how the agents are controlled, what actions they are capable of, and what tasks they must perform. Beyond the customizability, wherever possible, it is built on top of the JAX Bradbury et al. [2018] ecosystem to maximise performance and is suited for deployment on large distributed compute clusters. This paper aims to introduce the SwarmRL software and demonstrate how it can be applied to perform frontier research. We begin with an overview of the theory underpinning the software, explicitly discussing microscale active matter, simulating it in a physically realistic manner, and the reinforcement learning we have built to control it. Afterwards, the architecture of the software is described, detailing how each piece of the software fits together and what can be done with them. Finally, we discuss some of the unique features of SwarmRL, including performance and visualisation capabilities, before concluding with an outlook of the project. With SwarmRL, we hope to enable scientists from diverse backgrounds to contribute to and develop the field of micro-robotics research and realise the potential of this technology.

2 Theory

SwarmRL harnesses tools from a range of fields, the largest of which are active matter simulation and reinforcement learning. In order to understand and better utilize the software infrastructure, it is important to also have an understanding of the theory it implements. In this section, we introduce the important theoretical concepts on which SwarmRL is built, before exploring the architecture of the package.

2.1 Active Matter

Biological and artificial active systems

Active matter refers to systems that consume energy from their surroundings and are thus internally driven out of equilibrium [Romanczuk et al., 2012]. All biological and human-made machines fall under this term. However, in the context of this work, we focus on the branch of active systems that consists of small active particles that use energy to perform persistent, directed motion. This much more narrow definition covers plenty of examples in biology and engineering. Many bacteria like *Escherichia coli* [Wadhwa and Berg, 2022], archaea like *Thermoplasma volcanium* [Jarrell et al., 2021] and small eukaryotes like *Chlamydomonas reinhardtii* [Silflow and Lefebvre, 2001] use molecular motors and organelles like archaella, flagella or pili to self-propel. Artificial microswimmers can be made from colloidal particles that use, e.g., catalytic reactions [Howse et al., 2007], self-generated temperature gradients [Jiang et al., 2010] or magnetic fields [Mandal et al., 2018] to generate propulsion. In some more rare cases, the active particles can self-propel and self-steer, i.e., use torque to actively change their direction. For bacteria, tumbling [Berg and Brown, 1972, Darnton et al., 2007] is a well-known mechanism for change of direction. Some artificial microswimmers can also be steered by, e.g., change in laser focus [B  uerle et al., 2020] or magnetic field direction [Carlsen et al., 2014]. In all these examples, the active particle is of micrometer size and suspended in a liquid, making translational and rotational Brownian motion a non-negligible factor that competes with the deterministic active motion.

Simulation

Active matter systems with self-propelled small particles are usually modelled within the framework of stochastic dynamics of the individual constituents. We use the two- or three dimensional overdamped Langevin equations

$$\dot{\mathbf{r}}_i(t) = \gamma_t^{-1} [F_i^{\text{act}}(t)\hat{\mathbf{e}}_i(t) + \mathbf{F}_i(\mathbf{r}_i, \{\mathbf{r}_j\})] + \sqrt{2k_B T \gamma_t^{-1}} \boldsymbol{\xi}_i^t(t), \quad (1)$$

$$\dot{\hat{\mathbf{e}}}_i(t) = \left[\gamma_r^{-1} [M_i^{\text{act}}(t)\hat{\mathbf{n}}_i(t) + \mathbf{M}_i(\mathbf{r}_i, \{\mathbf{r}_j\})] + \sqrt{2k_B T \gamma_r^{-1}} \boldsymbol{\xi}_i^r(t) \right] \times \hat{\mathbf{e}}_i(t), \quad (2)$$

where $\mathbf{r}_i$ denotes the position of particle i , $\gamma_{t(r)}$ the translational (rotational) friction coefficient, F_i^{act} the active driving force that models self-propulsion, $\hat{\mathbf{e}}$ the unit vector representing the direction of the particle, $\mathbf{F}_i(\mathbf{r}_i, \{\mathbf{r}_j\})$ a conservative force from interactions between particle i and its environment as well as from interactions with other particles j , k_B the Boltzmann constant, T the absolute temperature, $\boldsymbol{\xi}_i^{t,(r)}$ the translational (rotational) noise with $\langle \boldsymbol{\xi}_i^{t,(r)}(t) \rangle = 0$ and $\langle \boldsymbol{\xi}_i^{t,(r)}(t) \otimes \boldsymbol{\xi}_j^{t,(r)}(t') \rangle = \delta_{ij} \delta(t - t') \mathbf{1}$, where $\langle \cdot \rangle$ denotes an ensemble average (expectation value) and $\mathbf{1}$ the unity matrix, M_i^{act} an active steering torque, $\hat{\mathbf{n}}$ a unit vector perpendicular to $\hat{\mathbf{e}}$ that sets the steering direction and $\mathbf{M}_i(\mathbf{r}_i, \{\mathbf{r}_j\})$ an interaction torque analogous to $\mathbf{F}_i$. We stress the time dependence of F_i^{act} and M_i^{act} because SwarmRL is designed to focus on control and decision-making on the level of single particles. Collective behavior and task fulfillment are to be achieved by the particles' actions and what they can control rather than by external fields acting on all particles.

2.2 Reinforcement Learning

As a sub-field of ML, RL is concerned with how intelligent agents can take actions in an environment to maximize a cumulative reward. It allows these agents to learn optimal protocols to achieve a given task or a set of tasks without explicitly telling them how. This is oftentimes useful when a task is too complex for a simple algorithm or control system. It can also be used when the focus of the investigation is the emergence of a strategy for a problem. RL is, however, data-inefficient and is thus used mostly for problems where data exists abundantly and where the action space is relatively small such as autonomous driving [Kiran et al., 2021], video games [Mnih et al., 2015], and robotics [Ibarz et al., 2021]. It reached great popularity in 2016 when Deepmind's AlphaGo won against the top Go player Lee Sedol [Silver et al., 2016]. When using it for specific tasks, however, RL can achieve remarkable results. In 2022, it solved a 50-year-old open question in mathematics how to most efficiently use matrix multiplication [Fawzi et al., 2022]; this discovery likely improves the efficiency of machine learning significantly due to its widespread use of matrix multiplication. It is further increasingly used in Physics research for sophisticated tasks such as quantum error correction [Zeng et al., 2023] and the magnetic control of tokamak plasmas [Degraeve et al., 2022]. This section provides an introduction to RL and explains the training approach currently implemented in SwarmRL, namely, deep Actor-Critic (AC).

2.2.1 Actor-Critic Reinforcement Learning

The RL training paradigm is concerned with iteratively improving an agent's decision-making such that it can achieve a pre-defined task as efficiently as possible. In practice, this is performed by placing agents into an environment with which they can interact by performing actions, a_t . To decide on such an action, the agent receives a state description, s_t , that describes the environment with sufficient information. The model used to decide on the action is referred to as the policy, $\pi_\theta : s_t \rightarrow a_t$, often taken to be a neural network with parameters, θ that is then referred to as an actor. These actors typically produce a probability distribution, which can then be sampled to select an action given the state. After an action is taken, a new state is produced along with a reward, quantifying the agent's progress towards achieving its task, a process outlined in Figure 1. Fundamentally, the goal of training in RL is to find the optimal policy that, throughout a trajectory, τ , maximises a cost function, $J(\pi_\theta)$, i.e.,

$$\pi^* = \arg \max_{\pi} J(\pi_\theta) = \arg \max_{\pi} \int_{\tau} P(\tau|\pi_\theta) \cdot R(\tau) = \arg \max_{\pi} \langle R(\tau) \rangle_{\tau \sim \pi_\theta}, \quad (3)$$

where $R(\tau)$ is the cumulative reward computed from a trajectory, $P(\tau|\pi_\theta)$ is the probability of a trajectory given the current policy, and the angled brackets denote an average over many trajectories sampled under the π_θ policy. This is achieved by encouraging the policy to increase the probability of selecting actions that returned good rewards and discouraging those that didn't. In practice, training is done throughout discrete-time episodes where state, policy probabilities, and reward data are collected and used to update the policy via the gradient ascent algorithm

$$\theta' = \theta + \eta \cdot \nabla_{\theta} J(\pi_\theta), \quad (4)$$

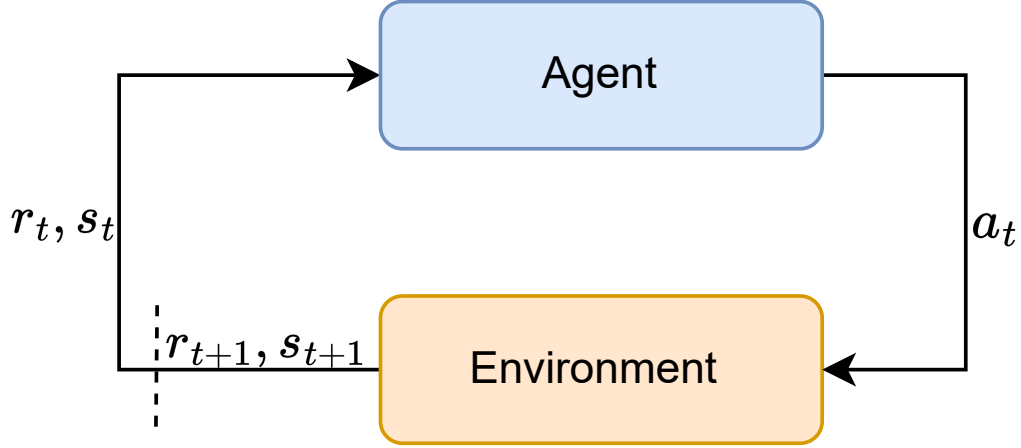


Figure 1: Graphical overview of the RL workflow. The agent selects an action, a_t , based in its current state, s_t , and acts within the environment. This action yields a new state, s_{t+1} and an associated reward, r_{t+1} .

with learning rate η , and

$$\nabla_{\theta} J = \left\langle \sum_t^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot r_t \right\rangle_{\tau \sim \pi_{\theta}} \quad (5)$$

with time step, t , and instantaneous reward, r_t . Typically, to ensure convergence in the long-term rewards and to encourage near-term over long-term progress, an expected returns function, G_t , will be used in the form

$$G_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}, \quad (6)$$

where T is the final time step of the episode, and γ is a discount factor set as a hyperparameter during training influencing the importance of late-time rewards. While this approach can be sufficient in training agents, it can break down in stochastic environments and high-dimensional spaces [Nachum et al., 2017]. To address this, Sutton and Barto [2018] showed that introducing a trainable baseline or value function, V^{π} to compute an advantage function

$$A_t^{\pi} = G_t - V_t^{\pi}, \quad (7)$$

can result in the stabilization of the expected returns and improved convergence time in the training, resulting in the reformulation of Equation 5 to

$$\nabla_{\theta} J = \left\langle \sum_t^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot A_t^{\pi} \right\rangle_{\tau \sim \pi_{\theta}}. \quad (8)$$

When this value function is implemented as a neural network, it is often referred to as a **critic**, thus actor-critic reinforcement learning. The critic's role is to determine the value of the state in which the agent currently finds itself. While several definitions exist, in **SwarmRL**, the value is defined as the theoretical returns possible if one starts in a state, s_t and follows policy π_{θ} . It improves this prediction by training on the true expected returns, Equation 6, computed during episodes through a desired regression algorithm. Therefore, in Equation 8, the advantage function guides training by encouraging selected actions that are better than or equal to those expected from the value function and discouraging those considered below what is theoretically possible. For this reason, AC RL training is often considered adversarial in nature, with the actor learning to beat the critic and the critic, in turn, learning the correct value of the encountered states. The training procedure outlined here is referred to as Vanilla Policy Gradient due to it being the simplest form of actor-critic training. More complex methods extend on the method discussed here, such as trust region policy optimisation [Schulman et al., 2017a] and proximal policy optimisation [Schulman et al., 2017b], the latter of which is also implemented in **SwarmRL**.

2.2.2 Multi-Agent Reinforcement Learning

As **SwarmRL** specializes in controlling microscopic agents, there will be many occasions in which multiple agents must be simultaneously controlled and trained. This is addressed by multi-agent reinforcement learning (MARL) [Zhang

et al., 2021]. In MARL, N autonomous agents interact with the environment and each other. This extension brings several new aspects to the problem, such as cooperation and competition among agents and various information structures. In the following, we will address these aspects briefly.

Cooperative, Competitive and Mixed Setting In a fully cooperative setting, all agents share a common reward, i.e., $r^1 = r^2 = \dots = r^N = r$ or a team-average reward where $r = \frac{1}{N} \sum_{i \in N} r_i$ [Zhang et al., 2018]. The latter is a generalized version of the former. Different reward functions for each agent allow more heterogeneity among agents [Zhang et al., 2021]. A fully competitive setting in MARL is typically modelled as a zero-sum Markov game with $\sum_{i \in N} r^i = 0$ [Littman, 1994]. Most literature focuses on two players where one agent’s reward is the other’s loss, something leveraged in, for example, predator-prey style games. A mixed setting imposes no restrictions on the goal and the relationship among agents [Hu and Wellman, 2003].

Information Structures Another critical challenge in MARL is the question *who knows what* during training and execution. This involves both the decision-making process as well as the optimization process. Three different forms of information structure are generally accepted. The first is the centralized setting with a so-called central controller aggregating joint observations, actions, and rewards and allowing for centralized learning. The execution, on the other hand, can either be centralized or decentralized. A centralized executed policy produces a joint action for all agents, while decentralized produces individually executed policies. The latter is the popular learning scheme of centralized-learning-decentralized-execution [Oliehoek et al., 2016, Hansen et al., 2004]. On the other side, there is the fully decentralized setting in which each agent acts based only on local observation. This setting results in the so-called independent learning scheme [Tan, 1993]. A combination of both is a decentralized setting with networked agents [Zhang et al., 2021]. This allows agents to share local information with their neighbours. The choice of setting depends on the problem at hand.

3 Architecture

SwarmRL has been designed to lower the entry barrier for scientists to access state-of-the-art reinforcement learning algorithms efficiently. It also offers a wide degree of customization and choice of algorithms that can be used during experiments to maximise the experienced user’s flexibility. To achieve both of these goals, SwarmRL is constructed using a modular, object oriented programming structure where almost all pipeline components are written as implementations of an abstract parent that defines the interface through which the components communicate. The Python programming language has been used throughout as it is most common amongst scientific disciplines, however, whenever possible, libraries have been used to improve performance through vectorized maps (vmaps), multi-threading, GPU use, and Just In Time (JIT) compiling [Bradbury et al., 2018]. A coarse-grained flowchart of SwarmRL is presented in Figure 2 where each major component of the architecture is shown with its corresponding connections. The structure of SwarmRL is built around the interaction between the Engine, whether that be simulation or real experiment, and the Agents, the algorithms used to control the particles in the simulation. Connecting the Agents to the Engine is the Force Function. Force Functions are either constructed directly, or through a Trainer in the cases where trainable Agents are in use. Finally, the Agent holds Tasks, Actions, and Observables, each of which provide information and functionality to the algorithm being used. The rest of this section is dedicated to explaining each important component of SwarmRL, where they fit into the different capabilities of the software, and how they can be adapted or extended for individual implementations. It should be noted that, to reduce space and improve readability, all comments, type hints, and methods not directly related to the discussion have been removed from code samples.

3.1 Engine

The Engine class of SwarmRL is one of its most important components. It represents the environment of the Agents and propagates the state according to the given actions. SwarmRL ships with an engine connected to the ESPResSo simulation software [Weik et al., 2019] to solve Equations (1) and (2) numerically. Through ESPResSo, we also offer the underdamped Langevin equation and a coupling to explicit hydrodynamics with lattice Boltzmann as the solver of the Navier-Stokes equations. SwarmRL also ships with rudimentary examples for connecting it directly to an experiment. Direct experimental connections are often kept private out of respect to the institutions that host the experiments. However, most implementations thus far use the TCP communications protocol to send actions and receive state information to hardware. Future work on SwarmRL aims to unify this communication procedure using the ROS ecosystem [Macenski et al., 2022]. In cases where users must connect their own engines, be it simulation or experiment, they inherit from the Engine parent class. They must implement the two methods outline in Code Sample 1.

```
1 class Engine:
2
```

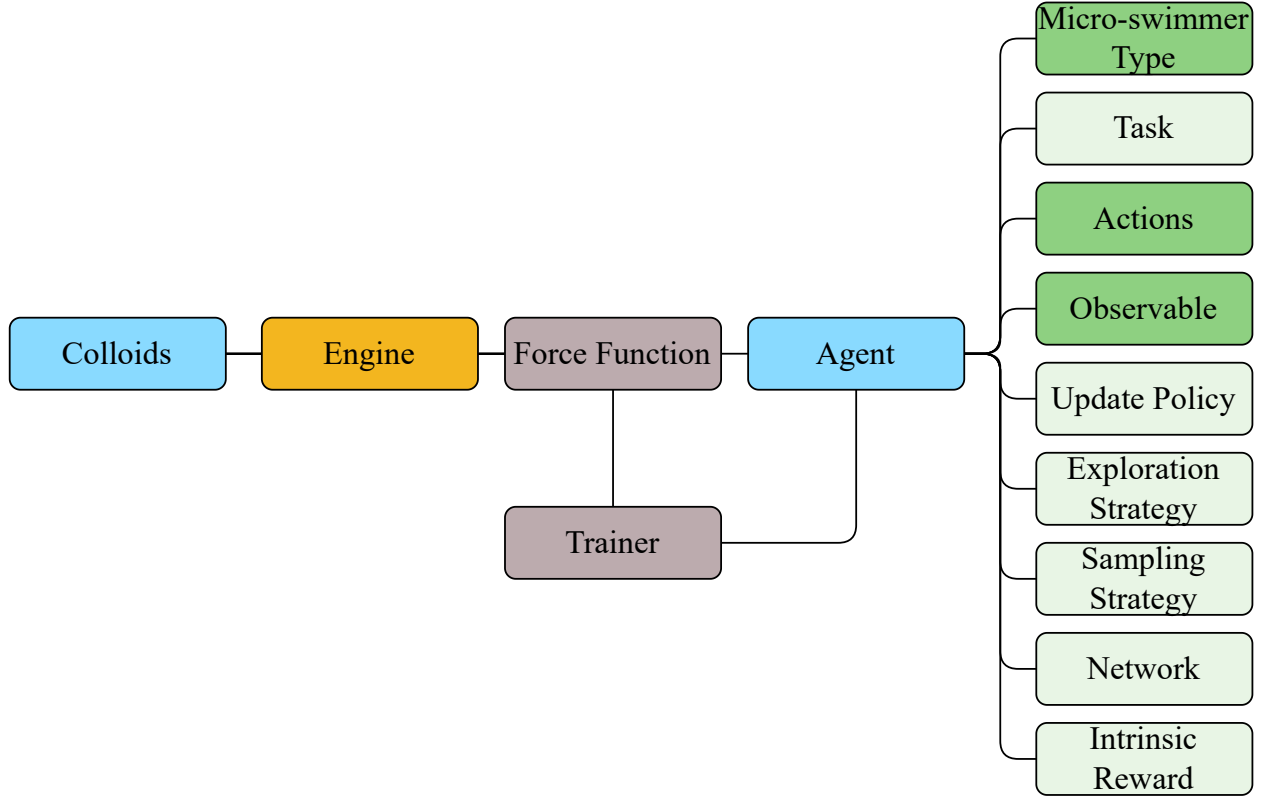


Figure 2: Overview of the SwarmRL software architecture. Light green boxes on the right-hand side of the figure represent modules with default settings that can be adjusted by users but do not need to be, dark green boxes indicate settings that need to be included in a system definition, blue boxes correspond to those directly handling colloid intelligence and properties, the engine is in orange, and grey represents classes that talk to the engine.

```

3  def integrate(
4      self, n_slices, force_model,
5  ):
6      raise NotImplementedError
7
8  def get_particle_data(self):
9      raise NotImplementedError

```

Code Sample 1: SwarmRL Engine parent class.

The `integrate` method is used to step forward either in simulation or in an experiment. It takes as arguments the force model, which holds the control algorithms for the particles in the experiment, see section 3.5, and number of slices to be looped over, that is, the number of times the force model is called for new actions to be applied to the particles. In a simulation, the time in-between successive time slices is filled with integrator steps, applying the chosen action at each time-step. In contrast, in experiment, this would be applying the actions for a fixed period between slices. When instantiated, these parameters, time-step or time between actions, are given to the specific engine. The second method that must be implemented is `get_particle_data`. This method must collect information about the particles in the system and give it back to the user uniformly so that all parts of SwarmRL can work with them. In its current state, SwarmRL defines colloids with properties outlined in Code Sample 2.

```

1  class Colloid:
2      pos
3      director
4      id
5      velocity
6      type

```

Code Sample 2: SwarmRL Colloid dataclass.

Here, all information relevant to the models is assigned to a colloid class and used to predict new actions and to compute rewards.

3.2 Agents

SwarmRL was built to service a variety of simulations and RL studies. A large component in enabling this, is the `Agent` class, which manages how agents in the simulations are controlled, what they see, and what they are tasked with doing. The `Agent` collects the components introduced in the previous sections and represents the full set of sensing, control, and objectives of the colloids in question. This information can then be used by SwarmRL to effectively deploy these agents in the simulation environment. Code samples 3 and 4 demonstrate the construction of two different agents which could both be added to a single simulation within SwarmRL.

```
1 ac_agent = srl.agents.ActorCriticAgent(
2     agent_type=0,
3     network=srl.networks.FlaxNetwork(...),
4     task=srl.tasks.RotateRod(),
5     observable=srl.observables.VisionCones(),
6     actions=action_dict
7 )
```

Code Sample 3: Actor-Critic Agent

```
8 ac_agent = srl.agents.Lymburn(
9     agent_type=1,
10    homing_location=[500., 500., 500.]
11 )
```

Code Sample 4: Classical Agent

In the above examples, the actor-critic reinforcement learning approach controls one group of particles. This means their policy can be trained and adapted to new environments. The other example uses a so-called `ClassicalAgent`, in particular, one based on the swarming agents of [Lymburn et al., 2021].

3.3 Classical Control

SwarmRL, allows users to control the actions of the particles in the system using both RL and classical algorithms. Classical algorithms, in this work, can be implemented without training and typically follow a defined set of rules. Such sets of rules have often been studied in the literature to reproduce behaviour in microswimmers [Bäuerle et al., 2020, Lavergne et al., 2019]. Classical agents are implemented directly into the `Agent` child class on the software side. However, they must follow the same structure, namely, at each time slice, the information of all particles in the system is passed into the class and in turn, it must return an action for each particle. In its current state, SwarmRL offers classical algorithms for the published models of Lymburn et al. [2021], Bäuerle et al. [2020], and Lavergne et al. [2019]. Future work aims at including Model Predictive Control (MPC) algorithms into the SwarmRL ecosystem.

3.4 Machine Learning Control

The machine learning side of agent control is driven by multi-agent reinforcement learning. However, due to the complex nature of this approach, several design decision have been made to balance flexibility with limiting complexity. The main workhorse of the machine learning driven control is the `Network` class, which requires two methods to be implemented, `compute_action` which computes the action of the agent/s under study, and `__call__`, which returns action probabilities and, in the case of actor-critic approaches, also the predicted state values.

Architecture

SwarmRL currently utilizes the Flax neural network library [Heek et al., 2023] exclusively. In order to provide an interface for complex architectures, SwarmRL requires only a single Flax module to be passed to the `FlaxNetwork` class. This module must return both the action probabilities, and the predicted value function in case of actor-critic training. This approach allows implementing very complex neural network architectures such as graph models, with the same API interface as simple dense actors and critics. An example of two commonly used approaches is listed in Code Samples 5 and 6.

Policy Update	Vanilla Gradient Update [Sutton et al., 1999]
	Proximal Policy Optimization [Schulman et al., 2017b]
Returns	Expected Returns
	Generalized Advantage Estimation [Schulman et al., 2018]
Sampling Strategy	Categorical Distribution
	Gumbel Trick [Gumbel, 1954]

Table 1: Overview of RL algorithm options available in SwarmRL

```

12 class DisjointActorCritic:
13
14     def __call__(self, x):
15         # Perform the actor forward pass.
16         actor_values = Dense(12)(x)
17         actor_values = relu(actor_values)
18         actor_output = Dense(4)(actor_values)
19
20         # Perform the critic forward pass.
21         critic_values = Dense(12)(x)
22         critic_values = relu(critic_values)
23         critic_output = Dense(1)(critic_values)
24
25         return actor_output, critic_output

```

Code Sample 5: Disjoint actor-critic

```

26 class CombinedActorCritic:
27
28     def __call__(self, x):
29         # Perform the shared layer computation.
30         shared_layer = Dense(12)(x)
31         shared_layer = relu(shared_layer)
32
33         # Compute the actor output.
34         actor_output = Dense(4)(shared_layer)
35
36         # Compute the critic output.
37         critic_output = Dense(1)(shared_layer)
38
39         return actor_output, critic_output

```

Code Sample 6: Combined actor-critic

These code blocks show a disjoint and combined actor critic approach bundled into the same Flax network structure. In this way, many complex neural network architectures can be constructed under a single module including graph or transformer modules.

Apart from the architecture, several components are essential to training AC models: the policy update algorithm, the approach used in the expectation computation, and the strategy for sampling actions from the learned distribution.

Actions

Actions in SwarmRL are dictated by dataclasses which can be read by an engine and turned into forces, torques, or other values in a simulation or experiment. At the time of this report, SwarmRL has been focused on micro-robotics with standard direction control, therefore, the class is built from a simple set of attributes outlined in Code Sample 7.

```

1 class Action:
2     force = 0.0
3     torque= np.zeros((3,))
4     new_direction = None

```

Code Sample 7: SwarmRL Action dataclass.

Force is applied along the agent's director axis, torque describes the agent's rotation, and new_direction provides a sometimes simpler approach to a torque calculation. From this template, we create an action by specifying the class's force, torque, and new direction attributes and add them to a dictionary to be passed to an agent. For example, if the agent should be capable of translation forwards, translation backwards, clockwise, and counterclockwise rotation, the set of actions can be passed to the Agent as in Code Sample 8.

```

1 agent_action_dictionary = {
2     "Forwards": Action(force=10.0),
3     "Backwards": Action(force=-10.0),
4     "CCW": Action(torque=np.array(0., 0., 10.)),
5     "CW": Action(torque=np.array(0., 0., -10.))
6 }

```

Code Sample 8: Example Action dictionary.

3.5 Force Function

The interaction model in SwarmRL is responsible for consuming data from the simulation and sending back computed actions for each of the agents. SwarmRL can control agents using machine learning and standard control theory approaches. The parent class for the models requires only a single function to be called by the Engine: `calc_action`, which takes a list of agent objects as input and returns a list of actions. This is the only occasion in which the environment communicates with the control algorithms and the communication must adhere to this very 'narrow' interface. This design choice enables novice SwarmRL users to switch out environments without adapting their RL setup and experienced users to easily implement new environments according to their needs.

Agent information is passed through as a dictionary when creating a model, demonstrated in Code Sample 9.

```

1 interaction_model = ForceFunction(
2     agents={
3         "0": ActorCriticAgent,
4         "1": ClassicalAgent
5     }
6 )

```

Code Sample 9: Example ForceFunction instantiation for a classical and actor-critic RL agent.

In the case above, the experiment has at least two types of particles. One is controlled using an actor-critic reinforcement learning approach while the other takes actions computed by a classical, non-trainable algorithm. In this way, SwarmRL users have complete control over the agents in their simulations and experiments.

3.6 Tasks

As the goal of SwarmRL is to be used for the study of micro-scale robotic or biological systems to learn about their behaviour and control, it is often important to measure how well the system is doing. In RL, this is often referred to as the reward which is maximised during training. For classical algorithms, a task can also be used as a measure of how well the approach is doing or passed into the model as feedback. In SwarmRL, system performance measurement is handled by the Task class. The parent class of the task is built up of several methods that need to be implemented in the case of a custom task. The reduced class architecture is displayed in Code Sample 10.

```

1 class Task:
2     @property
3     def kill_switch(self):
4         pass
5     def __call__(self, colloids):
6         raise NotImplementedError("Implemented in child class.")

```

Code Sample 10: SwarmRL Task parent class.

The above class contains two fundamental components, the kill switch and the call method. As its name suggests, the kill switch is used to end the current run. If this property is set to True, SwarmRL will stop the current running engine and either restart it, or just finish. This is used, for example, in episodic training, where the training should stop and the engine should reset after an amount of time or when a criterion is reached. However, the kill switch can also be used for more critical cases, such as in an experiment when it appears that the agents would perform damaging actions. The

second component, the call function, is used to compute a value describing the current state of this task, ideally, large positive values mean the task is being achieved, whereas negative values mean it is actively not being achieved. For example, when training RL agents to rotate an object, the call method would compute the rotation speed and return some positive value if it was increasing or large, and a negative value if it were decreasing.

At times, we want agents to perform more than just one task, in these cases, two options exist. The first is to write a custom task that provides feedback on two different objects such as how fast something is rotating and how far it has been pushed to a desired location. However, these tasks often quickly become convoluted and difficult to reuse. Therefore, SwarmRL provides the `Multitasking` class which takes a list of single tasks as an argument and applies them to the engine. This approach is more modular and allows for faster testing of strategies.

3.7 Observables

A critical component of any RL algorithm and an important part of classical agent control approaches is their state description. Rather than passing all state information (e.g., all positions and directions of the agents) to the decision making model, the observable usually selects from or condenses the information considerably. This mimics the limited sensing capabilities that agents might have. For example, agents might only be able to retrieve information about other particles within a finite sensing radius or might not be able to tell the direction of their neighbours. Investigating the amount of information needed for an agent to still be able to perform its task is very relevant for the design of cost-efficient micro agents. In SwarmRL, the description of the environment as sensed by the agents is computed by `Observables`. All observables implemented in SwarmRL inherit from a simple parent class outlined in Code Sample 11.

```
1 class Observable:
2     def compute_observable(self, colloids):
3         raise NotImplementedError("Implemented in child class.")
```

Code Sample 11: SwarmRL Observable parent class.

The class requires implementing one method, `compute_observable`, which takes a list of particle objects as input and returns the state description vector for each of them. This state vector is then passed on to the models by the force function. As with the `Task` class, we often want to construct agents capable of sensing the world in multiple ways. To achieve this, we also provide a `MultiSensing` class which takes a list of observables and computes a single output.

3.8 Intrinsic Reward

In RL agents can be rewarded intrinsically to either encourage the exploration behavior of an agent or to assist in building its knowledge about the environment [Pathak et al., 2017]. As the name suggests, such a method is intrinsic to the agent and is therefore implemented as a separate sub-module `IntrinsicReward` which the agent can be equipped with.

Random Network Distillation

Random network distillation (RND) was introduced in 2018 by Burda et al. [2018] as an efficient method for exploring the state space of reinforcement learning agents. It has since been studied more broadly for its application and data selection for ML applications [Finkbeiner et al., 2023] and its implications on understanding data requirements for neural networks [Tovey et al., 2023].

SwarmRL provides `RNDReward`, a module implementing RND, defining an intrinsic reward for agents, encouraging them to explore unseen regions of their state space. Going beyond the original version from Burda et al. [2018], SwarmRL provides two implementations of RND, differing in their capability to memorize previous states. For both versions of RND, a default setup is provided in `swarmrl/intrinsic_reward/rnd_configs.py`. Utilizing such configuration the implementation of RND is shown in Code Sample 12.

```
40 rnd_config = RNDConfig(...)
41
42 ac_agent = srl.agents.ActorCriticAgent(
43     ...,
44     intrinsic_reward=RNDReward(rnd_config)
45 )
```

Code Sample 12: Random Network Distillation as Intrinsic Reward

The implementation of the RND backend is based on the `ZnNL` python package [Nikolaou and Tovey, 2021], offering a flexible but simple interface for complex training algorithms for neural networks.

3.9 Exploration Policy

In addition to the sampling strategy which can select actions from its distributions that do not have the largest probability, SwarmRL also provides direct access to exploration policies to further guide the agents into new, unexplored regions of their environment.

Random Exploration

The simplest form of exploration policy is the `RandomExploration` class, which an exploration probability can parameterize, ζ and decay rate, ϵ . The exploration rate dictates how likely it is for an agent to take an exploration action over a policy action, and the decay rate slowly decreases this probability as the training evolves by

$$\zeta' = \exp^{-\epsilon \frac{t}{T}} \zeta, \quad (9)$$

where t is simulation time, and T is the time for a single training episode.

3.10 Trainer

The trainer is module in SwarmRL responsible for handling the updates of the models and their interaction with the environment while developing a policy. In the simplest case, the trainer deploys the models in the environment, collects the rewards over an episode, and performs a network update, as outlined in Code Sample 13. However, in many cases more is demanded during RL training and even how rewards are collected can change depending on the problem. Therefore, in SwarmRL, we have implemented several variations of the Trainer.

Continuous vs Episodic

A user must first choose whether to perform episodic or continuous training. In episodic training, agents are deployed in the environment for a fixed number of steps or until some condition is met. After this time, a reward is computed based on their final state or trajectory during the episode. This reward is then used to update the networks before the environment is reset and the process is started again. In this approach, the episodes should be long enough to allow the agents to achieve their tasks but short enough to maximize efficiency. A similar but alternative approach is semi-episodic training, wherein the agents are updated continuously during the training but after some time, or again, once a condition is met, the environment is reset and the training continues. These options are available through the `EpisodicTrainer` packaged into SwarmRL. The `EpisodicTrainer`, displayed in Code Samples 13 and 14, allows users to define how often the environment should be reset as a function of RL-episodes, or to pass a task which can force an environment reset.

```
1 rl_trainer = srl.trainers.ContinuousTrainer(
2     [protocol_1, protocol_2],
3     loss,
4 )
5 rl_trainer.perform_rl_training(
6     system_runner=srl.Engine(...),
7     n_episodes=5000,
8     episode_length=50,
9 )
```

Code Sample 13: Continuous Training

```
10 rl_trainer = srl.trainers.EpisodicTrainer(
11     [protocol_1, protocol_2],
12     loss,
13 )
14 rl_trainer.perform_rl_training(
15     get_engine=engine_fn,
16     n_episodes=10,
17     reset_frequency=30, # > 1 is semi-episodic
18     episode_length=100,
19 )
```

Code Sample 14: Episodic Training

Continuous training is a paradigm in which agents are deployed into an environment and are updated after a fixed amount of time without resetting the environment. This online training is useful for situations where training must occur in the

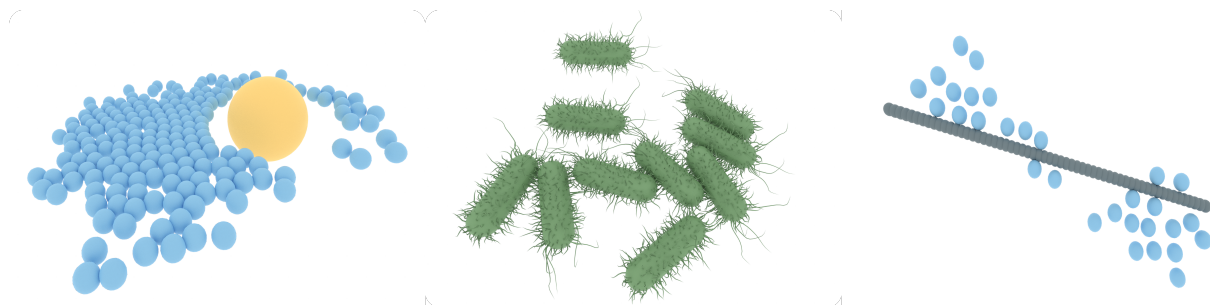


Figure 3: Example renderings using `SwarmRL` and the `ZnVis` visualization engine. (*left*) A snapshot from a reservoir computing experiment using a reinforcement learning controlled swarm. (*center*) Colloids are trained to perform chemotaxis in the same fashion as bacteria. Here an open-source bacteria 3d model file has been used inside of `SwarmRL` to convey realism. The mesh was created by Sketchfab artist andrewfrueh, whose explicit permission was requested and granted before being used in this publication. The model is licensed under creative commons 4.0. (*right*) Demonstration of colloids rotating a rod, controlled using an RL algorithm.

real world or where the time it will take to achieve a task is not well known. `SwarmRL` offers the `ContinuousTrainer` class to perform this kind of training. It should be noted that if a `Task` is provided to the continuous trainer capable of resetting the environment, the training will end. This can also be helpful in cases where an end to training can be strictly identified.

4 Performance

Reinforcement learning typically does not require expensive neural networks, and classical control algorithms are limited in complexity by the number of particles they control. However, speed becomes critical when considering the number of iterations they have to go through during training and deployment. Therefore, `SwarmRL` has been built with a performance-first mentality, leveraging the JAX ecosystem [Heek et al., 2023] heavily. JAX is a Python library that, among other things, re-implements the NumPy library [Harris et al., 2020] with multi-threading, GPU deployment, and XLA-enhanced JIT compile capabilities. Most variables in `SwarmRL` are written using JAX arrays or their PyTree counterparts. PyTrees extend the functionality JAX has for NumPy objects to data structures like classes. Therefore, most of the class objects including the `Colloid` class can be placed onto a GPU or parallelized over. Furthermore, large parts of code are JIT compiled to ensure optimal performance.

5 Software Practices

In order to ensure the longevity and continued usability of the `SwarmRL` software, we adhere to several software practices. `SwarmRL` is thoroughly tested with unit, integration, and functional tests in place. Overall, we have 87 % coverage on the code with all classes having at least one unit test. The code follows the NumPy doc-string format and is extensively documented which is hosted at <https://swarmrl.github.io/SwarmRL.ai/>. Before new code is added to the project, all tests must pass and new tests should be included. The project is hosted on GitHub where all these standards are enforced automatically. Furthermore, we require at least one review from a code-owner, that is, a member of the core `SwarmRL` team, to accept new changes to the software.

6 Visualization

Visualization is an essential part of the scientific process, particularly in simulations where one is searching for emergent strategy. It is not always clear from plots alone if the agents have learned what they should do. For this reason, `SwarmRL` interfaces closely with the `ZnVis` [Tovey, 2021] visualization library. `ZnVis`, by design, handles the output data from `SwarmRL` and is developed alongside the project. The package can use custom mesh files for the agents, export rendered videos, and capture still-frame renderings using the Mitsuba engine [Jakob et al., 2022]. Figure 3 illustrates the visualization capabilities of the package used on `SwarmRL` experiments.

7 Conclusion

We have introduced SwarmRL, a Python package aimed at accelerating research into control and understanding of microscopic active particles, whether biological or artificial. SwarmRL allows researchers to implement state-of-the-art control algorithms driven by reinforcement learning or classical policies, in both simulated and experiment settings. As a Python package, SwarmRL has a familiar interface for many scientists, and is simple enough to use with limited background in software. Beyond its functionality, we have built SwarmRL to be performant, capable of running on many cores, GPUs, and scalable to HPC clusters. Our current focus is extending SwarmRL to handle more control algorithms, e.g., MPC, Q-learning, additional experiment implementations, and additional engines. Overall, our goal is that SwarmRL allow scientists to spend more time creating new ideas, building new technology, and changing the field, rather than implementing and staying up to date with state-of-the-art machine learning methods.

8 Acknowledgements

C.H and S.T acknowledge financial support from the German Funding Agency (Deutsche Forschungsgemeinschaft DFG) under Germany’s Excellence Strategy EXC 2075-390740016. C.H and S.T acknowledge financial support from the German Funding Agency (Deutsche Forschungsgemeinschaft DFG) under the Priority Program SPP 2363, “Utilization and Development of Machine Learning for Molecular Applications - Molecular Machine Learning” Project No. 497249646. C.H and C.L acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Project Number 327154368-SFB 1313. The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG. The authors acknowledge support from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) Compute Cluster grant no. 492175459. S.T would like to acknowledge Annalena Daniels for her detailed review and comments on the manuscript.

References

- T. Bäuerle, R. C. Löffler, and C. Bechinger. Formation of stable and responsive collective states in suspensions of active colloids. *Nature Communications*, 11(1):2547, May 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-16161-4. URL <https://doi.org/10.1038/s41467-020-16161-4>.
- H. C. Berg and D. A. Brown. Chemotaxis in escherichia coli analysed by three-dimensional tracking. *Nature*, 239 (5374):500–504, 1972.
- D. Blackiston, E. Lederer, S. Kriegman, S. Garnier, J. Bongard, and M. Levin. A cellular platform for the development of synthetic living machines. *Science Robotics*, 6(52):eabf1571, 2021. doi: 10.1126/scirobotics.abf1571. URL <https://www.science.org/doi/abs/10.1126/scirobotics.abf1571>.
- F. Borra, L. Biferale, M. Cencini, and A. Celani. Reinforcement learning for pursuit and evasion of microswimmers at low reynolds number. *Phys. Rev. Fluids*, 7:023103, Feb 2022. doi: 10.1103/PhysRevFluids.7.023103. URL <https://link.aps.org/doi/10.1103/PhysRevFluids.7.023103>.
- J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL <http://github.com/google/jax>.
- Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by random network distillation, 2018.
- R. W. Carlsen, M. R. Edwards, J. Zhuang, C. Pacoret, and M. Sitti. Magnetic steering control of multi-cellular bio-hybrid microswimmers. *Lab on a Chip*, 14(19):3850–3859, 2014.
- S. R. Dabbagh, M. R. Sarabi, M. T. Birtek, S. Seyfi, M. Sitti, and S. Tasoglu. 3d-printed microrobots from design to translation. *Nature Communications*, 13(1):5875, Oct 2022. ISSN 2041-1723. doi: 10.1038/s41467-022-33409-3. URL <https://doi.org/10.1038/s41467-022-33409-3>.
- N. C. Darnton, L. Turner, S. Rojevsky, and H. C. Berg. On torque and tumbling in swimming escherichia coli. *Journal of bacteriology*, 189(5):1756–1764, 2007.
- J. Degraeve, F. Felici, J. Buchli, M. Neunert, B. Tracey, F. Carpanese, T. Ewalds, R. Hafner, A. Abdolmaleki, D. de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.
- K. Dhatt-Gauthier, D. Livitz, Y. Wu, and K. J. M. Bishop. Accelerating the design of self-guided microrobots in time-varying magnetic fields. *JACS Au*, 3(3):611–627, 2023. doi: 10.1021/jacsau.2c00499. URL <https://doi.org/10.1021/jacsau.2c00499>.

- A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022. doi: 10.1038/s41586-022-05172-4. URL <https://doi.org/10.1038/s41586-022-05172-4>.
- O. Felfoul, M. Mohammadi, S. Taherkhani, D. de Lanauze, Y. Zhong Xu, D. Loghin, S. Essa, S. Jancik, D. Houle, M. Lafleur, L. Gaboury, M. Tabrizian, N. Kaou, M. Atkin, T. Vuong, G. Batist, N. Beauchemin, D. Radzioch, and S. Martel. Magneto-aerotactic bacteria deliver drug-containing nanoliposomes to tumour hypoxic regions. *Nature Nanotechnology*, 11(11):941–947, Nov 2016. ISSN 1748-3395. doi: 10.1038/nnano.2016.137. URL <https://doi.org/10.1038/nnano.2016.137>.
- J. Finkbeiner, S. Tovey, and C. Holm. Generating minimal training sets for machine learned potentials, 2023.
- E. J. Gumbel. Statistical theory of extreme values and some practical applications. lectures by emit j. gumbel. national bureau of standards, washington, 1954. 51 pp. diagrams. 40 cents. *The Aeronautical Journal*, 58(527):792–793, 1954. doi: 10.1017/S0368393100099958.
- E. A. Hansen, D. S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715, 2004.
- C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, Sep 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-2649-2. URL <https://doi.org/10.1038/s41586-020-2649-2>.
- B. Hartl, M. Hübl, G. Kahl, and A. Zöttl. Microswimmers learning chemotaxis with genetic algorithms. *Proceedings of the National Academy of Sciences*, 118(19):e2019683118, 2021. doi: 10.1073/pnas.2019683118. URL <https://www.pnas.org/doi/abs/10.1073/pnas.2019683118>.
- J. Heek, A. Levskaya, A. Oliver, M. Ritter, B. Rondepierre, A. Steiner, and M. van Zee. Flax: A neural network library and ecosystem for JAX, 2023. URL <http://github.com/google/flax>.
- Z. Hosseinidoust, B. Mostaghaci, O. Yasa, B.-W. Park, A. V. Singh, and M. Sitti. Bioengineered and biohybrid bacteria-based systems for drug delivery. *Advanced Drug Delivery Reviews*, 106:27–44, 2016. ISSN 0169-409X. doi: <https://doi.org/10.1016/j.addr.2016.09.007>. URL <https://www.sciencedirect.com/science/article/pii/S0169409X16302629>. Biologically-inspired drug delivery systems.
- J. R. Howse, R. A. Jones, A. J. Ryan, T. Gough, R. Vafabakhsh, and R. Golestanian. Self-motile colloidal particles: from directed propulsion to random walk. *Physical review letters*, 99(4):048102, 2007.
- A. Hsu, A. Wong-Foy, B. McCoy, C. Cowan, J. Marlow, B. Chavez, T. Kobayashi, D. Shockey, and R. Pelrine. Application of micro-robots for building carbon fiber trusses. In *2016 International Conference on Manipulation, Automation and Robotics at Small Scales (MARSS)*, pages 1–6, 2016. doi: 10.1109/MARSS.2016.7561729.
- J. Hu and M. P. Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4 (Nov):1039–1069, 2003.
- J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4–5):698–721, Jan. 2021. ISSN 1741-3176. doi: 10.1177/0278364920987859. URL <http://dx.doi.org/10.1177/0278364920987859>.
- W. Jakob, S. Speierer, N. Roussel, and D. Vicini. DRJIT. *ACM Transactions on Graphics*, 41(4):1–19, jul 2022. doi: 10.1145/3528223.3530099. URL <https://doi.org/10.1145/3528223.3530099>.
- K. F. Jarrell, S.-V. Albers, and J. N. d. S. Machado. A comprehensive history of motility and archaeellation in archaea. *FEMS microbes*, 2:xtab002, 2021.
- H.-R. Jiang, N. Yoshinaga, and M. Sano. Active motion of a janus particle by self-thermophoresis in a defocused laser beam. *Physical review letters*, 105(26):268302, 2010.
- I. S. M. Khalil, V. Magdanz, S. Sanchez, O. G. Schmidt, and S. Misra. Biocompatible, accurate, and fully autonomous: a sperm-driven micro-bio-robot. *Journal of Micro-Bio Robotics*, 9(3):79–86, Aug 2014. ISSN 2194-6426. doi: 10.1007/s12213-014-0077-9. URL <https://doi.org/10.1007/s12213-014-0077-9>.
- H. Kim, J. Ali, U. K. Cheang, J. Jeong, J. S. Kim, and M. J. Kim. Micro manipulation using magnetic microrobots. *Journal of Bionic Engineering*, 13(4):515–524, 2016. ISSN 1672-6529. doi: [https://doi.org/10.1016/S1672-6529\(16\)60324-4](https://doi.org/10.1016/S1672-6529(16)60324-4). URL <https://www.sciencedirect.com/science/article/pii/S1672652916603244>.
- B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez. Deep reinforcement learning for autonomous driving: A survey, 2021.

- S. Kriegman, D. Blackiston, M. Levin, and J. Bongard. A scalable pipeline for designing reconfigurable organisms. *Proceedings of the National Academy of Sciences*, 117(4):1853–1859, 2020. doi: 10.1073/pnas.1910837117. URL <https://www.pnas.org/doi/abs/10.1073/pnas.1910837117>.
- F. A. Lavergne, H. Wendehenne, T. Bäuerle, and C. Bechinger. Group formation and cohesion of active particles with visual perception-dependent motility. *Science*, 364(6435):70–74, Apr. 2019.
- M. Li, A. Pal, A. Aghakhani, A. Pena-Francesch, and M. Sitti. Soft actuators for real-world applications. *Nature Reviews Materials*, 7(3):235–249, Mar 2022. ISSN 2058-8437. doi: 10.1038/s41578-021-00389-7. URL <https://doi.org/10.1038/s41578-021-00389-7>.
- M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.
- T. Lymburn, S. D. Algar, M. Small, and T. Jüngling. Reservoir computing with swarms. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 31(3):033121, 03 2021. ISSN 1054-1500. doi: 10.1063/5.0039745. URL <https://doi.org/10.1063/5.0039745>.
- S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>.
- P. Mandal, G. Patil, H. Kakoty, and A. Ghosh. Magnetic active matter based on helical propulsion. *Accounts of chemical research*, 51(11):2689–2698, 2018.
- M. Medina-Sánchez, L. Schwarz, A. K. Meyer, F. Hebenstreit, and O. G. Schmidt. Cellular cargo delivery: Toward assisted fertilization by sperm-carrying micromotors. *Nano Letters*, 16(1):555–561, 2016. doi: 10.1021/acs.nanolett.5b04221. URL <https://doi.org/10.1021/acs.nanolett.5b04221>. PMID: 26699202.
- V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236. URL <https://doi.org/10.1038/nature14236>.
- S. Muiños-Landin, A. Fischer, V. Holubec, and F. Cichos. Reinforcement learning with artificial microswimmers. *Science Robotics*, 6(52):eabd9285, 2021. doi: 10.1126/scirobotics.abd9285. URL <https://www.science.org/doi/abs/10.1126/scirobotics.abd9285>.
- R. R. Murphy, S. Tadokoro, D. Nardi, A. Jacoff, P. Fiorini, H. Choset, and A. M. Erkmen. *Search and Rescue Robotics*, pages 1151–1173. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-30301-5. doi: 10.1007/978-3-540-30301-5_51. URL https://doi.org/10.1007/978-3-540-30301-5_51.
- O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans. Bridging the gap between value and policy based reinforcement learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/fac9f743b083008a894eee7baa16469-Paper.pdf.
- B. J. Nelson and S. Pané. Delivering drugs with microrobots. *Science*, 382(6675):1120–1122, 2023. doi: 10.1126/science.adh3073. URL <https://www.science.org/doi/abs/10.1126/science.adh3073>.
- B. J. Nelson, I. K. Kaliakatsos, and J. J. Abbott. Microrobots for minimally invasive medicine. *Annual Review of Biomedical Engineering*, 12(1):55–85, 2010. doi: 10.1146/annurev-bioeng-010510-103409. URL <https://doi.org/10.1146/annurev-bioeng-010510-103409>. PMID: 20415589.
- K. Nikolaou and S. Tovey. ZnNL, July 2021. URL <https://github.com/zincware/ZnNL>.
- F. A. Oliehoek, C. Amato, et al. *A concise introduction to decentralized POMDPs*, volume 1. Springer, 2016.
- D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 488–489, 2017. doi: 10.1109/CVPRW.2017.70.
- K. Qin, Z. Zou, L. Zhu, and O. S. Pak. Reinforcement learning of a multi-link swimmer at low Reynolds numbers. *Physics of Fluids*, 35(3), 03 2023. ISSN 1070-6631. doi: 10.1063/5.0140662. URL <https://doi.org/10.1063/5.0140662>. 032003.
- P. Romanczuk, M. Bär, W. Ebeling, B. Lindner, and L. Schimansky-Geier. Active brownian particles. *The European Physical Journal Special Topics*, 202(1):1–162, Mar 2012. ISSN 1951-6401. doi: 10.1140/epjst/e2012-01529-y. URL <https://doi.org/10.1140/epjst/e2012-01529-y>.

- C. K. Schmidt, M. Medina-Sánchez, R. J. Edmondson, and O. G. Schmidt. Engineering microrobots for targeted cancer therapies from a medical perspective. *Nature Communications*, 11(1):5618, Nov 2020. ISSN 2041-1723. doi: 10.1038/s41467-020-19322-7. URL <https://doi.org/10.1038/s41467-020-19322-7>.
- J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization, 2017a.
- J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, 2017b.
- J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.
- H. Shen, S. Cai, Z. Wang, Z. Ge, and W. Yang. Magnetically driven microrobots: Recent progress and future development. *Materials & Design*, 227:111735, 2023. ISSN 0264-1275. doi: <https://doi.org/10.1016/j.matdes.2023.111735>. URL <https://www.sciencedirect.com/science/article/pii/S0264127523001508>.
- C. D. Silflow and P. A. Lefebvre. Assembly and motility of eukaryotic cilia and flagella. lessons from *chlamydomonas reinhardtii*. *Plant physiology*, 127(4):1500–1507, 2001.
- D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961. URL <https://doi.org/10.1038/nature16961>.
- M. Stefanec, D. N. Hofstadler, T. Krajník, A. E. Turgut, H. Alemdar, B. Lennox, E. Şahin, F. Arvin, and T. Schmickl. A minimally invasive approach towards “ecosystem hacking” with honeybees. *Frontiers in Robotics and AI*, 9, 2022. ISSN 2296-9144. doi: 10.3389/frobt.2022.791921. URL <https://www.frontiersin.org/articles/10.3389/frobt.2022.791921>.
- H. Su, C.-A. Hurd Price, L. Jing, Q. Tian, J. Liu, and K. Qian. Janus particles: design, preparation, and biomedical applications. *Materials Today Bio*, 4:100033, 2019. ISSN 2590-0064. doi: <https://doi.org/10.1016/j.mtbio.2019.100033>. URL <https://www.sciencedirect.com/science/article/pii/S2590006419300596>.
- R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S.olla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.
- M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.
- S. Tovey. ZnVis: A Visualisation Package for Particle Simulations, 2021.
- S. Tovey, S. Krippendorf, K. Nikolaou, and C. Holm. Towards a phenomenological understanding of neural networks: data. *Machine Learning: Science and Technology*, 4(3):035040, sep 2023. doi: 10.1088/2632-2153/acf099. URL <https://dx.doi.org/10.1088/2632-2153/acf099>.
- N. Wadhwa and H. C. Berg. Bacterial motility: machinery and mechanisms. *Nature reviews microbiology*, 20(3): 161–173, 2022.
- X. Wang, X.-Z. Chen, C. C. J. Alcântara, S. Sevim, M. Hoop, A. Terzopoulou, C. de Marco, C. Hu, A. J. de Mello, P. Falcato, S. Furukawa, B. J. Nelson, J. Puigmartí-Luis, and S. Pané. Mofbots: Metal–organic–framework-based biomedical microrobots. *Advanced Materials*, 31(27):1901592, 2019. doi: <https://doi.org/10.1002/adma.201901592>. URL <https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.201901592>.
- F. Weik, R. Weeber, K. Szuttor, K. Breitsprecher, J. de Graaf, M. Kuron, J. Landsgesell, H. Menke, D. Sean, and C. Holm. Espresso 4.0—an extensible software package for simulating soft matter systems. *The European Physical Journal Special Topics*, 227:1789–1816, 2019.
- Y. Zeng, Z.-Y. Zhou, E. Rinaldi, C. Gneiting, and F. Nori. Approximate autonomous quantum error correction with reinforcement learning. *Phys. Rev. Lett.*, 131:050601, Jul 2023. doi: 10.1103/PhysRevLett.131.050601. URL <https://link.aps.org/doi/10.1103/PhysRevLett.131.050601>.
- K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Başar. Fully decentralized multi-agent reinforcement learning with networked agents, 2018.
- K. Zhang, Z. Yang, and T. Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2021.
- J. Zhuang, R. Wright Carlsen, and M. Sitti. ph-taxis of biohybrid microsystems. *Scientific Reports*, 5(1):11403, Jun 2015. ISSN 2045-2322. doi: 10.1038/srep11403. URL <https://doi.org/10.1038/srep11403>.