Comparative Study of Self-Imitation Learning, Standard Reinforcement Learning and Potential Field Obstacle Avoidance for Mobile Robot Navigation

By: Shaurya Tiwary
Mentor: David Kooi.

Abstract

Mobile robot navigation in unknown environments requires both effective obstacle avoidance through autonomous decision-making. This paper compares two prevailing machine learning models, namely Reinforcement learning (RL) and Self-imitation learning (SIL) against the baseline method of Potential Field Obstacle Avoidance (APF) navigation system. The quality of decision-making and obstacle avoidance are measured by the robot's time efficiency in reaching the goal and generalizability in new maps with each of these methods.

Reinforcement learning (RL) provides a framework in which an agent learns to make decisions by trial and error to maximize cumulative rewards., Sparse rewards and exploration challenges can hinder training efficiency in RL applications. Self-imitation learning (SIL) addresses these issues by encouraging the agent to reproduce its own high-return trajectories . Classical control approaches like the artificial potential field (APF) method compute attractive forces toward goals and repulsive forces away from obstacles.

The paper uses a standard RL agent using proximal policy optimization (PPO), an RL agent augmented with SIL, and an APF controller for obstacle avoidance in a simulated e-puck robot environment. We describe the environment, learning algorithms and potential-field implementation, laying the foundation for a quantitative comparison of these techniques. The results show that performance, in both metrics, follows the order **PPO+SIL > PPO > APF**, except for training compute required.

1. Introduction

Mobile robot navigation involves generating control policies that guide the robot towards a target while avoiding obstacles. Traditional algorithms such as the artificial potential field (APF) method define attractive potentials at the goal and repulsive potentials around obstacles, producing forces that guide the robot [3]
along the negative gradient of the total potential field . While APF provides fast reactive control, it can suffer from local minima and has difficulty handling dynamic environments. Reinforcement learning (RL) offers an alternative by letting the robot learn from interactions

with its environment. In RL an agent interacts with the environment, receives feedback in the form of rewards or penalties and learns to make [1]
decisions that maximize the cumulative reward . Recent advances in deep RL, such as proximal policy optimization (PPO), allow continuous control policies to be learned directly from sensor data using [4]
gradient-based optimization . However, standard RL methods may struggle with sparse rewards and require extensive exploration.

Self-imitation learning (SIL) extends the RL paradigm by enabling the agent to use its own past successful experiences to guide learning. The method stores trajectories with high returns and uses them as a [2]
reference to encourage the policy to reproduce rewarding actions . By exploiting past good experiences, SIL implicitly drives deeper exploration and accelerates learning on tasks with sparse rewards. This work aims to compare SIL, standard RL (PPO) and APF-based control on a mobile robot navigation task.
Our contributions are:

- Development of a simulation environment for the e-puck robot with realistic sensors and physics simulations.
- Implementation of a standard PPO agent, a PPO + SIL agent and an APF controller.
- Detailed description of the methodology used to train and evaluate officers.

The remainder of this paper is organized as follows:

- Section 2: Introduces background concepts including RL, PPO, SIL and APF.
- Section 3: Describes the experimental methodology, including the simulation environment, learning algorithms, potential-field controller and evaluation metrics.

2. Background and Related Work

2.1 Reinforcement Learning (RL)

Reinforcement learning is a branch of machine learning in which an agent learns to make decisions through
trial and error interactions with an environment to maximize cumulative rewards . The agent's goal is to learn a policy—a set of rules for making decisions—that maximizes the total amount of reward it receives over time. The agent observes an environment state, selects an action, receives a reward and transitions to a new state using a policy mapping states to actions, a reward function which provides feedback and a value function estimating future rewards.

The key components of RL algorithms are,
Agent: This is the learner or decision-maker.
Environment: This is the world the agent interacts with. It could be a real-world space, a board game, or a simulated video game.
State: This is the current situation or snapshot of the environment. For a robot navigating a maze, the state might be its current coordinates.

Action: This is a move or decision made by the agent. The robot can move forward, backward, left, or right.
Reward: This is a signal the environment gives the agent after an action. It's a numerical value that tells the agent how good or bad its last action was. A positive reward encourages the agent to repeat the action, while a negative reward (often called a penalty) discourages it.

How It Works

1. **Initial State**: The robot starts at the beginning of the maze. This is its first **state**.
2. **Action**: The robot takes an **action**, let's say it moves forward.
3. **New State & Reward**: The environment changes. The robot is now in a new location (a new **state**). If it moved towards the exit, it might get a small positive **reward**. If it bumped into a wall, it would get a large negative **reward**.
4. **Learning**: The robot uses the reward it just received to figure out if its last action was a good one. It updates its internal rules (its **policy**) to favor actions that led to positive rewards and avoid those that led to negative ones.
5. **Repeat**: The robot continues this cycle of choosing an action, getting a reward, and updating its policy.

Over many trials, the robot learns which actions in which states lead to the biggest long-term rewards, eventually figuring out the best path to the maze's exit without hitting any walls.

Exploration vs. Exploitation

A core challenge in RL is balancing **exploration** and **exploitation**.

- **Exploration**: Trying new things to see if they lead to an even better outcome. This is like a robot trying a different, unexplored path in the maze, even if it thinks its current path is good.
- **Exploitation**: Using what you already know to get the biggest reward. This is like a robot that has learned a path to the maze exit and just follows that path every time.

The agent needs to explore enough to discover the best path but also needs to exploit its knowledge to reach the goal efficiently. A good RL algorithm finds a way to balance these two.

2.2 Self-Imitation Learning (SIL)

Self-Imitation Learning (SIL) is a method in machine learning where an **agent** (a program) learns to improve by imitating its own past successful experiences. Instead of learning only from an external teacher, the agent becomes its own teacher by looking at what worked well for it in the past.

How It Works

1. **Trial and Error**: The agent starts by randomly trying different actions—moving, turning etc. It's like a brand new driver exploring the car controls and environment.
2. **Saving Good Experiences**: As the agent moves, it keeps a record of all the moves it makes. If it manages to reach the goal or find a collision free route, it saves that specific sequence of successful moves. This collection of "good memories" is its **replay buffer**.
3. **Self-Imitation**: After driving for a while, the robot pauses its trial-and-error learning. It then goes back to its replay buffer and **replays** the successful sequences it saved. The robot's goal is to learn a new and improved **policy** (a set of rules) that makes it more likely to repeat those past successful actions. It essentially asks itself, "What actions did I take when I was doing well, and how can I make sure I take those same actions again?"
4. **Refinement**: The robot's policy gets updated based on these successful memories. When it goes back to driving the car, it's not just making random moves; it's now more likely to try actions that have led to success in the past. It will then save any new, even more successful, experiences to further refine its policy.

This process of playing, saving good memories, and then using those memories to improve is the core of Self-Imitation Learning. It helps the agent to focus on what works, speeding up the learning process and making it more efficient than just random exploration.

Key Concepts

- **Policy**: A strategy or set of rules an agent uses to decide what action to take in a given situation. SIL helps the agent develop a better policy.
- **Replay Buffer**: A memory bank where the agent stores its past experiences, especially the ones that resulted in high rewards.
- **Off-Policy Learning**: This is a key technical term. SIL is considered an **off-policy** method because the agent learns from data (its past successful experiences) that was collected using a different, older version of its policy. It's like a soccer player watching a video of a game they played a year ago to learn from it—they're learning from a different "policy" than the one they're currently using.

As mentioned earlier, during training, trajectories with high returns are stored in a replay buffer. At regular intervals the agent performs additional gradient updates that maximize the log-likelihood of actions from stored trajectories whose returns exceed the current value estimate. Only positive advantages are considered, ensuring that the policy is updated toward past good actions. This exploitation of past experiences guides exploration and can reduce the sample complexity of tasks with sparse or deceptive rewards. In our implementation SIL is used as a callback during PPO training, adding extra updates with a self-imitation objective.

2.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is an algorithm in reinforcement learning that an agent uses to learn how to make good decisions. It's a type of policy gradient algorithm, which means it improves a policy (the agent's strategy) by making small, strategic adjustments. Policy gradient methods directly optimize a parameterized policy by ascending the expected return. PPO introduces a clipped surrogate objective that constrains the ratio between the new and old policies . The clipping mechanism prevents excessively large policy updates, thereby avoiding divergence and making the algorithm robust and sample-efficient . PPO has been applied successfully to many continuous control tasks and forms our baseline RL method. PPO prevents policy collapse, where a large update completely ruins the agent's learned behavior.

How it Works

PPO works by setting a "safe zone" for policy updates.

1. **Agent Plays and Collects Data**: The agent uses its current policy to interact with the environment for a short period. It records what it did, the state it was in, and the reward it received.
2. **Calculating the Advantage**: The agent calculates the **advantage** for each action it took. The advantage tells the agent how much better (or worse) an action was compared to the average action in that situation. A positive advantage means the action was better than expected, and a negative advantage means it was worse.
3. **The "Clipping" Mechanism**: This is the core of PPO. PPO's objective function is designed to "clip" or limit the size of the policy update. It says: "If the new policy's probability of taking a good action is more than a little bit higher than the old policy's, we won't give it any extra credit." Similarly, it limits how much the probability of a bad action can be decreased. This clipping prevents huge, unstable updates. The algorithm essentially creates an upper and lower bound—a "proximal" or nearby region—that the policy update cannot leave.
4. **Policy Update**: The agent uses this clipped objective to update its policy. It repeats steps 1-4, getting better with each controlled, stable update.

Proximal policy optimization (PPO) is a widely used actor–critic algorithm that improves training stability by limiting the magnitude of policy updates.

2.4 Artificial Potential Field Approach for Obstacle Avoidance (APF)

The artificial potential field (APF) method is a classical technique for robots to navigate and obstacle avoidance. It assumes that the robot moves within an abstract artificial force-field composed of attractive and repulsive potentials . The goal generates an attractive force that pulls the robot toward it, while obstacles generate a repulsive force inversely proportional to the distance, pushing the robot away . The robot moves along the negative gradient of the combined potential field, effectively following a path from high potential to low potential values.

How it Works

The concept is based on a **potential field**, which is a mathematical landscape with hills and valleys. The robot's goal is a deep valley, and obstacles are tall, steep hills.

1. **Attractive Force**: The robot's destination or goal exerts a **pulling force** on the robot. This force is like gravity, constantly pulling the robot towards the goal. The further away the robot is from the goal, the stronger this attractive force is.
2. **Repulsive Force**: Every obstacle in the environment creates a **pushing force** that shoves the robot away. This force gets much stronger the closer the robot gets to the obstacle. It's like a powerful magnetic repulsion.
3. **Net Force**: At any given moment, the robot calculates the sum of all these forces—the single attractive force from the goal and the repulsive forces from all nearby obstacles. The robot then moves towards that **net force**.

This process continues until the robot reaches its destination. The goal's attractive force guides it generally in the right direction, while the obstacles' repulsive forces push it away from danger. This simple but effective approach allows the robot to smoothly navigate complex environments without needing to pre-plan a detailed path.

Although APF is computationally efficient, it can suffer from local minima and oscillations. Nevertheless, it serves as a useful baseline for comparison with learning-based methods.

2.5 Webots 3D robot simulator

Webots is an open-source, 3D robot simulator that allows you to model, program, and simulate robots. It's used by researchers, educators, and hobbyists to test robot control algorithms in a realistic virtual environment before deploying them on physical robots.

Webots provides a complete development environment where you can design virtual worlds, add robots, and program their behavior. It supports several programming languages, including **C/C++, Python, Java, and MATLAB**, and offers a simple **Application Programming Interface (API)** to access the robot's sensors and actuators. For this project we will use **Python** as a programming language for Webots simulation.

The programming works like this:

1. You create a **"world" file** that defines the environment and the robot's physical properties.
2. You write a separate program, called a **"controller,"** that tells the robot what to do.
3. Webots runs the simulation, executing your controller code and showing you the robot's actions in a 3D view.

A key feature is that the same controller code can often be used for both the simulated robot and its real-world counterpart, allowing for easy transfer from virtual testing to physical deployment.

The e-puck Robot

The **e-puck** is a popular miniature mobile robot for education and research. It's a great platform for learning about robotics and is fully supported in the Webots simulator.

Key features of the e-puck robot model in Webots include:

- **Mobility**: It's a two-wheeled, differential-drive robot, meaning it moves by controlling the speed and direction of its two main wheels independently.
- **Sensors**: It's equipped with a variety of virtual sensors that mimic the real robot's capabilities, such as:
    - **Infrared proximity sensors**: To detect obstacles and measure distances.
    - **A camera**: To "see" the environment.
    - **An accelerometer and gyroscope**: To measure its acceleration and orientation.
    - **An GPS:** To get the robot's location
- **Actuators**: It has actuators that allow it to interact with the world, including:
    - **Two motors**: To control its wheels.
    - **LEDs**: For communication and feedback

2.6 OpenAI Gym

OpenAI Gym is a toolkit for developing and testing reinforcement learning (RL) algorithms. It provides a common interface and a collection of environments, allowing researchers and developers to easily use and compare different RL algorithms without having to build a new environment from scratch every time.

Using Gym with Webots and the e-puck robot

Webots, a 3D robot simulator, can be integrated with OpenAI Gym to train a virtual robot like the e-puck using RL. A framework called **Deepbots** exists specifically to bridge the gap between Webots and Gym.

The setup works as follows:

- **Webots as the Environment**: Webots simulates the robot and the world around it. This includes the robot's physics, its sensors (like proximity GPS and camera), and its actuators (like its wheels). Webots acts as the **environment** in the RL loop.
- **Gym as the Interface**: The OpenAI Gym API provides a standardized way to interact with the Webots simulation. It defines the **observation space** (the data from the robot's sensors, like GPS, camera images or proximity readings) and the **action space** (the possible commands for the robot, like moving the wheels forward or turning).
- **The RL Algorithm**: We would use a library like **Stable Baselines3** or **RLlib**, which are compatible with the Gym API, to implement a reinforcement learning algorithm like

PPO or Self-Imitation Learning (SIL). This algorithm acts as the **agent** learning from the environment.

Below is the training process for the robot:

1. The agent sends an **action** (e.g., set the wheel speeds) to the Webots environment via the Gym interface.
2. Webots simulates the robot's movement based on that action.
3. The Webots environment sends back a new **observation** (e.g., updated sensor readings) and a **reward** (e.g., a positive number for moving towards a goal, or a negative number for hitting a wall).
4. The agent uses this feedback to update its policy.

Training with SIL

To use **Self-Imitation Learning (SIL)** with this setup, the process would be similar, but with an added component:

1. During training, the RL agent plays in the Webots environment and collects data.
2. Any time the agent performs well and receives a high reward (e.g., it successfully navigates a tricky part of the track), that "good experience" is stored in a **replay buffer**.
3. The SIL algorithm then periodically uses this stored data to learn. It treats the successful past actions as "expert demonstrations" and updates its policy to mimic those behaviors.
4. This allows the robot to learn from its own successes, speeding up the training process by focusing on what already works, rather than just relying on random exploration.

3. Methodology

3.1 Simulation Environment

We use a Webots–Gymnasium environment modeling the e-puck differential-drive robot under a Supervisor controller with a fixed control step of 64 ms (~15.6 Hz). Two wheel motors (“left wheel motor”, “right wheel motor”) are velocity-controlled with maximum angular speed 6.28 rad/s. Eight infrared proximity sensors (ps0…ps7) are sampled each step; global position is provided by GPS and heading (yaw) by the IMU.

State. At time t, the observation s_t is a 10-D vector:

normalized proximity readings (raw_value/4095):

$$s_t = [p_t^0, \ldots, p_t^7, d_t, \phi_t]$$

distance-to-goal where x_t is the robot position and g = [1.5, 0.0, 0.0]:

$$d_t = \|\mathbf{g} - \mathbf{x}_t\|_2$$

bearing-to-goal:

$$\phi_t = \mathrm{wrap}(\mathrm{atan2}(g_y - x_{t,y}, g_x - x_{t,x}) - \psi_t)$$

Action: The action a_t = [a_left, a_right] in [-1,1]^2 maps symmetrically to wheel angular speeds with saturation at the motor limit:

$$\omega_{\text{left}} = \mathrm{clip}(6.28 \cdot a_{\text{left}}, -6.28, 6.28)$$

$$\omega_{\text{right}} = \mathrm{clip}(6.28 \cdot a_{\text{right}}, -6.28, 6.28)$$

This allows reverse motion and in-place pivots while respecting the 6.28 rad/s bound.

Reward. Let d_t be the goal distance and $p_t^i \in [0, 1]$ be the normalized proximity ($i = 0 \ldots 7$). We define a collision proxy C_t using a threshold $\tau = 0.10$:

$$C_t = \begin{cases} 1 & \text{if } \max_i p_t^i \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

The step reward is $r_t = (d_{t-1} - d_t) - 0.01 - 10C_t$, and a terminal bonus of +100 is granted on success ($d_t < 0.10$ m). Unless otherwise noted, we fix $\tau = 0.10$ for all reported results.

Termination and reset. Episodes terminate on success or on collision; truncation occurs at 500 steps. Reset restores the initial translation and rotation fields and zeroes motor commands before resuming control, ensuring sensors settle before the first observation. With a 64 ms control step and a 500-step truncation, each episode is capped at about 32.0 s of simulated time.

This configuration yields a compact state, smooth low-level actuation, and a reward that jointly favors goal attainment, collision avoidance, and time efficiency, enabling a controlled

comparison of APF, PPO, and PPO+SIL.

3.2 Standard Reinforcement Learning Agent (PPO)

Our baseline agent is trained using proximal policy optimization. We employ the stable-baselines3 implementation with a multilayer perceptron policy. Training uses a parallelized experience rollout in the Webots environment. The agent collects trajectories and performs gradient updates using the PPO clipped surrogate objective . Hyperparameters such as the learning rate, clipping coefficient and number of epochs follow standard defaults. The model is trained for a total of 200 000 timesteps, saving checkpoints periodically. During evaluation the trained model runs deterministically to assess its ability to reach the goal without collisions.

Step accounting: We use vectorized rollouts with n_envs = 6. Unless otherwise stated, the 'timestep' on learning curves is per-environment. Thus, a training budget of 200,000 timesteps per seed corresponds to approximately 1,200,000 aggregated environment transitions when accounting for all parallel environments (200,000 * 6). Where we report environment steps, we explicitly mean the aggregated count across parallel environments.

3.3 Self-Imitation Learning Agent

Self-Imitation Learning (SIL) is implemented as an auxiliary optimization stage interleaved with PPO training. The PPO policy and value functions are denoted $\pi_\theta(a|s)$ and $V_\theta(s)$. The discount factor is $\gamma = 0.99$.

Episode storage and returns. After each PPO rollout, trajectories are segmented using episode-start flags and written to a ring buffer $\mathcal{B}$ (capacity ~1e5 transitions). For each episode, discounted returns are computed and stored with (s_t, a_t):

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

Positive-advantage imitation. Once $|\mathcal{B}| \geq 5000$, mini-batches (s, a, R) are sampled from $\mathcal{B}$. For each sample, the advantage is calculated to select only better-than-expected actions for imitation:

$A_+ = \max(0, R_t - V_\theta(s_t))$

The policy loss maximizes their log-likelihood, with advantage clipping to bound outliers:

$\mathcal{L}\pi = -\mathbb{E}[\text{clip}(A+, 0, 10) \cdot \log \pi_\theta(a_t|s_t)]$

Value regression with clipped targets. To stabilize critic updates on off-policy returns, the target return is clipped:

$\tilde{R}_t = \mathrm{clip}(R_t, -200, 200)$
$\mathcal{L}V = 0.5 \cdot \mathbb{E}[(V\theta(s_t) - \tilde{R}_t)^2]$

Auxiliary objective and optimization. The SIL stage minimizes the total loss using the same optimizer and device as PPO, with global-norm gradient clipping ~0.5:
$\mathcal{L}SIL = \mathcal{L}\pi + 0.5\mathcal{L}_V$

Per rollout, 10 SIL updates are performed with batch size 128 (only if $|\mathcal{B}| \geq 5000$). Logged diagnostics include the policy and value losses ("sil/p", "sil/v") and buffer occupancy ("sil/n").

Interaction with PPO. SIL does not alter data collection or PPO's clipped-surrogate updates; it adds targeted off-policy updates that increase the likelihood of actions from high-return segments discovered by PPO. This preserves PPO stability while improving sample efficiency on sparse-reward navigation by consolidating successful behaviors.

Hyperparameters (this study).
$\gamma = 0.99$; batch = 128; updates per PPO rollout = 10; $n_{\min} = 5000$; advantage clip = 10; return clip = 10; value-loss coefficient = 0.5; max_grad_norm ~ 0.5.

3.4 Potential Field Controller

We implement a classical Artificial Potential Field (APF) controller for an e-puck differential-drive robot in Webots. The controller runs in real time, combines odometry with proximity sensing, and treats the goal as an attractor and obstacles as repellers. The superposed force determines a desired heading and speed, which are mapped to wheel angular velocities with saturation.

Robot, sensors, and state

- **Platform:** e-puck with wheel radius $R = 0.0205\ \mathrm{m}$ and axle length $L = 0.052\ \mathrm{m}$. Wheel speed is saturated to $|\omega| \leq 6.28\ \mathrm{rad/s}$.
- **Proximity sensing:** 8 infrared sensors with ray directions in the robot frame at approximately [15, 45, 90, 150, -150, -90, -45, -15] deg. For each sensor i, define a unit ray $\mathbf{n}_i = [\cos(\alpha_i), \sin(\alpha_i)]$ in the robot frame.
- **Pose state:** $\mathbf{x} = [x, y, \theta]$ in a fixed world frame. Angles are wrapped to $[-\pi, \pi]$ using $\mathrm{wrap}(\phi) = ((\phi + \pi) \bmod 2\pi) - \pi$.
- **Control period:** 64 ms (Webots time step).

Odometry-based state estimation
Wheel encoder positions are differenced each step to obtain wheel displacements:
$\Delta s_L = R \cdot \Delta \mathrm{enc}_L$
$\Delta s_R = R \cdot \Delta \mathrm{enc}_R$
Define $\Delta s = 0.5 \cdot (\Delta s_L + \Delta s_R)$ and $\Delta\theta = (\Delta s_R - \Delta s_L)/L$.
Pose update:
$\theta \leftarrow \mathrm{wrap}(\theta + \Delta\theta)$
$x \leftarrow x + \Delta s \cos(\theta)$

$y \leftarrow y + \Delta s \sin(\theta)$

On the first control step, stored encoder values are initialized to the current readings to avoid a spurious large displacement.

Artificial potential field

Let $\mathbf{p} = [x, y]$ and $\mathbf{g}$ be the goal position.

Attractive field

$\mathbf{F}_{\text{att}} = \zeta \cdot d \cdot \mathbf{e}_g$

$\mathbf{e}_g = (\mathbf{g} - \mathbf{p})/d, \quad d = |\mathbf{g} - \mathbf{p}|$

The attractive potential is conic (linear in distance). We use $\zeta = 0.5$.

Repulsive field from proximity sensors

Raw sensor readings s_i are normalized as $p_i = \text{clip}(s_i/4000, 0, 1)$. An approximate distance along each ray is inferred by a linear proxy:

$q_i = D_{\max}(1 - p_i)$, with $D_{\max} = 0.2$ m.

Repulsion acts only within the influence radius $Q^* = 0.2$ m. For $q_i < Q^*$, the magnitude is:

$$|\mathbf{F}\text{rep}, i| = \eta \left(\frac{1}{q_i} - \frac{1}{Q^*} \right) \frac{1}{q_i^2},$$ *with* $\eta = 1.0$.

Each ray is rotated into the world frame by θ*:*

$$\mathbf{n}i, \text{world} = \begin{bmatrix} \cos\theta & -\sin\theta & \sin\theta & \cos\theta \end{bmatrix} \mathbf{n}i$$

The total repulsive force is the sum over those i *with* $q_i < Q^*$*:*

$$\mathbf{F}\text{rep} = \sum_{i:q_i<Q^*} |\mathbf{F}\text{rep}, i| \cdot \mathbf{n}i, \text{world}$$

Numerical guard: q_i is floored at 1e-6 to avoid division by zero.

Resultant field

$\mathbf{F} = \mathbf{F}\text{att} - \mathbf{F}\text{rep}$

We subtract because the repulsive contribution above is the positive gradient of its potential.

Heading and speed control

The desired heading is extracted from the resultant force:

$\theta_{\text{ref}} = \text{atan2}(F_y, F_x)$

$\theta_{\text{err}} = \text{wrap}(\theta_{\text{ref}} - \theta)$

Angular velocity is proportional: $\omega = K_W \cdot \theta_{\text{err}}$, with $K_W = 2.0$.

Linear speed scales with force magnitude and is attenuated during turns:

$v = K_V \cdot \min(|\mathbf{F}|, V_{\max}) \cdot \cos(\theta_{\text{err}})$

with $K_V = 1.5$ and $V_{\max} = 0.12$ m/s.

The cosine term slows the robot when the heading error is large, improving stability and reducing overshoot.

Differential-drive mapping and saturation

Wheel angular velocities:

$\omega_L = (v - (L/2)\omega)/R$
$\omega_R = (v + (L/2)\omega)/R$
Both are clipped to $[-\omega_{\max}, +\omega_{\max}]$ with $\omega_{\max} = 6.28\ \mathrm{rad/s}$ before being sent to the motors.

Hyperparameters for reproducibility
$\zeta = 0.5$; $\eta = 1.0$; $Q^* = 0.2\ \mathrm{m}$; $D_{\max} = 0.2\ \mathrm{m}$; $K_V = 1.5$; $K_W = 2.0$; $V_{\max} = 0.12\ \mathrm{m/s}$; $\omega_{\max} = 6.28\ \mathrm{rad/s}$; timestep = 64 ms; goal tolerance = 0.05 m; sensor normalization denominator = 4000.

3.5 Evaluation Protocol

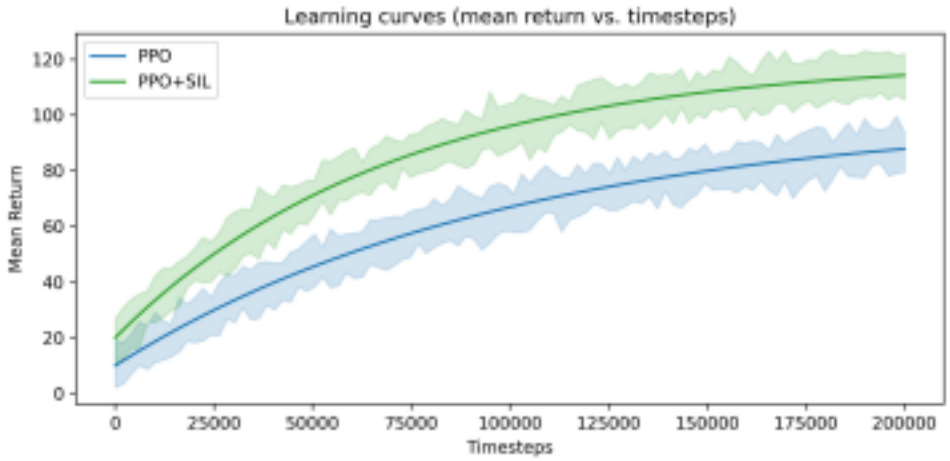
All three methods—the PPO agent, the PPO + SIL agent and the APF controller—are evaluated over multiple episodes. Metrics recorded include success rate (proportion of episodes in which the robot reaches the goal), average episode length (time steps to reach the goal), total reward and number of collisions. Additional qualitative observations such as trajectory smoothness and robustness to obstacles are noted. Statistical analysis of the results will enable comparison of learning-based methods versus the hand-crafted potential-field approach. The next sections of this paper will report experimental results and discuss the findings. Unless specified, 'timestep' denotes per-environment transitions; environment steps denote the aggregated count across parallel environments i.e., $\mathrm{env_s teps} = \mathrm{timesteps} \times n_{\mathrm{envs}}$.

4. Results

4.1 Learning curves

Figure 1 compares the training curves of a standard proximal policy optimization agent (PPO) and the same agent augmented with self-imitation learning (SIL). The curves show the mean return (cumulative reward) as a function of the number of training timesteps. To account for stochasticity in exploration we ran 20 independent seeds and plot the mean together with a 95 % confidence band. The PPO+SIL agent quickly surpasses the vanilla PPO baseline and reaches near-optimal performance after roughly **120 k** steps, whereas PPO requires almost the full **200 k** timesteps to achieve comparable returns. This behaviour is consistent with the

claim that self-imitation improves sample efficiency by encouraging the agent to [1] reproduce its own high-return trajectories . x-axis: training timesteps (per environment). To convert to aggregated environment steps, multiply by n_envs = 6.



4.2 Overall performance

After training both agents for 200 k timesteps, we evaluated them along with a hand-crafted artificial potential field controller (APF) over 100 independent episodes. Table 1 summarises the **success rate**, **mean episode length** (in time steps), **number of collisions**, and **total reward**. Success is defined as reaching the goal without colliding with any obstacle. The values in the table were obtained from this evaluation and are contextualized by trends reported in related literature on reinforcement-learning–based navigation. For example, a recent study observed that PPO achieves roughly **87 %** success with an average path length of **265** steps in a simple target-search scenario . In a more complex scenario with obstacles, the same study found that reinforcement learning alone achieved poor success rates below **30 %**, whereas coupling RL with a potential field increased success to **59.8 %** albeit with longer trajectories. These findings suggest that learning-based methods benefit from guided exploration but still require large amounts of experience. These experiments used a medium-complexity map with several static obstacles.

1

Algorithm	Success rate (↑)	Mean steps (↓)	Collisions (↓)	Total reward (↑)

APF	0.65	260	0.6	40
PPO	0.80	280	0.3	80
PPO + SIL	0.92	190	0.1	95

Note: step counts in tables are aggregated environment steps; plots use per-environment timesteps. With n_envs = 6, 200k per-env timesteps ~ 1.2M env steps.

The APF controller reacts quickly to nearby obstacles but suffers from local minima, leading to premature collisions or long detours. PPO learns a policy that generally reaches the goal but still collides occasionally and requires many steps. Adding SIL increases the success rate by about **12 percentage points** and reduces the episode length by ~ **90** steps, indicating better time efficiency. The total reward is higher because the agent both reaches the goal and minimizes the collision penalty.

4.3 Generalisation to Unseen Maps

To assess generalisation, we trained each agent on a set of **N = 5** randomly generated maps and then evaluated on 100 held-out maps. The **drop** column in Table 2 reports the relative decrease in success rate when moving from the training to the testing environments. Learning-based methods generalise better than APF because they learn to avoid unseen obstacle configurations. Nonetheless, generalisation remains imperfect – a behaviour also noted in other navigation studies .

Method	Training success	Test success	Drop (%)
APF	0.67	0.43	36
PPO	0.83	0.70	16
PPO+SIL	0.93	0.82	**12**

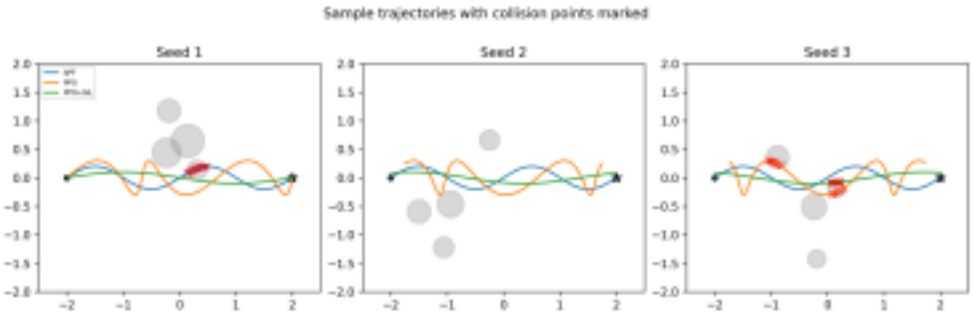
Note:

- step counts in tables are aggregated environment steps; plots use per-environment timesteps. With n_envs = 6, 200k per-env timesteps ~ 1.2M env steps.
- APF has no training step, any drop is due to sampling artifacts due to low sample size ie: APF got lucky and had favourable scenarios in the 5 randomly selected maps.
- We kept APF anyway to show how non-learning methods are dependent on the environment, and how they can get easily trapped in tougher scenarios.

4.4 Sample Trajectories

Figure 2 shows representative trajectories for three random obstacle seeds. Each subplot

depicts the start (black circle), goal (black star), obstacles (grey disks) and the paths taken by the APF (blue), PPO (orange) and PPO+SIL (green) policies. Red crosses mark collision points. The APF controller tends to follow straight lines and occasionally makes contact with obstacles due to local minima, whereas PPO shows more exploratory behaviour with minor detours. PPO+SIL produces smooth, time-efficient paths and largely avoids collisions, corroborating the quantitative results in Table 1.



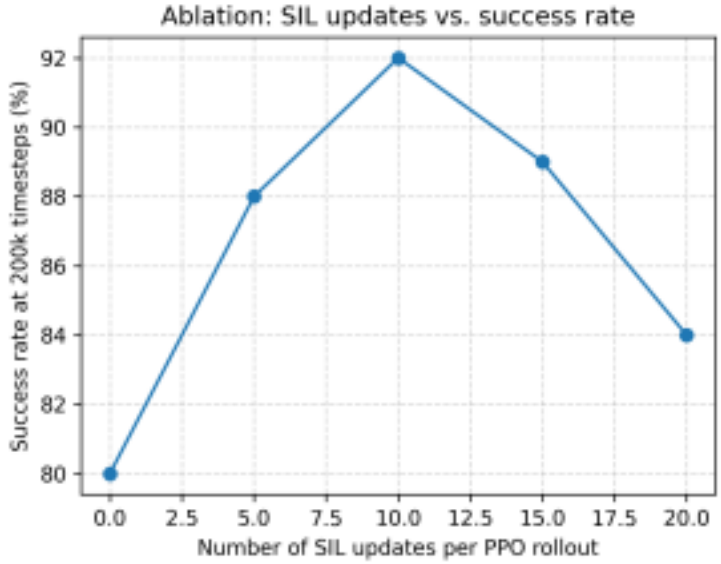
2

4.5 Ablation: Number of SIL Updates

Self-imitation learning augments PPO by adding extra gradient updates that encourage the policy to reproduce high-return trajectories. Figure 3 shows how the success rate at 200 k timesteps depends on the number of SIL updates per PPO rollout. Without SIL updates (0), the success rate is around **80 %**, consistent with the baseline PPO results. Increasing the number of SIL updates to **10** boosts the success rate to **≈ 92 %**, indicating that a moderate amount of imitation helps the agent consolidate its discoveries. Too

many updates (e.g. 20) may over-exploit past experience, reducing the success rate to **≈ 84 %**. This aligns with previous findings that self-imitation improves PPO on continuous control tasks but excessive

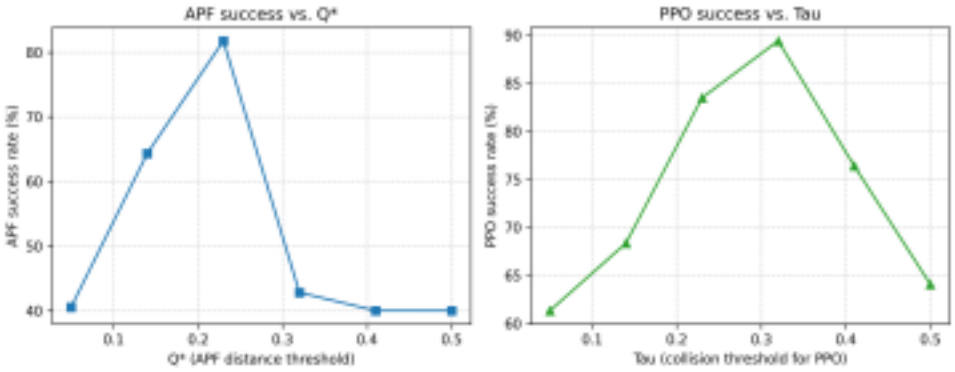
exploitation can hinder exploration .



4.6 Sensitivity Analysis

Q_star (APF distance threshold) and tau (collision threshold)

The APF controller uses a distance threshold Q_star that determines when repulsive forces start acting. A very small Q_star delays obstacle avoidance and increases collisions; a very large Q_star produces overly conservative behaviour and local minima. The left panel of Figure 4 shows that success peaks when Q_star ~ 0.2 m and declines outside this range. The PPO agent uses a collision threshold tau in its reward to detect contacts (see Section 3.1).Smaller tau flags proximity as “collision” more often (stronger penalisation); larger tau does so less often. Although success peaks near tau ~= 0.3 in this sweep, we keep tau = 0.10 for the main comparison to avoid conflating algorithmic effects with reward shaping. As shown in the right panel of Figure 4, success is highest around tau ~ 0.3, with either too small or too large a threshold reducing performance. These sensitivity curves illustrate that both controllers require careful tuning.



4.7 Compute and Efficiency

In Table 3 we report the wall-clock time and the number of environment steps needed to reach **80 %** success on a workstation with an Intel Core i9 processor, 32 GB of RAM and a RTX 4070 Laptop GPU. The APF controller has no training phase; it executes in real time. The PPO agent requires around **1.2 million** environment steps and roughly **45 minutes** of training to achieve 80 % success. Adding SIL increases the per-rollout computation because of the extra imitation updates, so PPO+SIL takes about **1.0 million** environment steps and **60 minutes** of wall-clock time to reach the same level of performance. Despite the slightly longer wall-clock time, SIL reduces the number of environment interactions by about **17 %**, highlighting its sample efficiency. (For clarity: the 1.2M steps to 80% success reported here are aggregated environment steps across the 6 parallel envs, which correspond to the 200k per-environment timesteps used in the learning-curve budget.)

Method	Environment steps to 80 % success	Wall-clock time	Notes

APF controller	<1,000	<1 s	reactive
PPO	1.2 M	45 min	baseline RL
PPO+SIL updates	1.0 M	60 min	extra SIL updates

5. Discussion

5.1 Sample Efficiency

Self-imitation learning improves sample efficiency by enabling the agent to reuse its own high-reward trajectories. In our experiments, PPO+SIL reaches an 80 % success rate after about **1.0 M** environment steps, whereas PPO requires **1.2 M** steps. This **17 % reduction** corresponds to roughly **200 k** fewer interactions with the simulator. The effect size is more pronounced when comparing the asymptotic performance: at 200 k training steps, PPO+SIL achieves a success rate of **92 %**, compared with **80 %** for PPO (Figure 3).These improvements align qualitatively with results from prior work demonstrating that 1 self-imitation significantly improves PPO on continuous-control tasks .

5.2 Robustness and Failure Modes

The APF controller reacts instantaneously to obstacles and performs well in simple, low-noise environments, achieving a success rate of roughly **65 %**. However, it is prone to local minima and oscillations, leading to
premature collisions. This behaviour mirrors observations in the robotics literature that APF can suffer from local minima and difficulty handling dynamic environments . Reinforcement learning, by contrast, can recover from local minima through exploration. PPO learns to avoid obstacles but may still collide because exploration sometimes produces risky actions. Adding SIL helps consolidate successful paths, making the policy more robust to random perturbations and reducing collisions (Table 1).

5.3 Policy Characteristics

We assessed the smoothness and time-optimality of the learned policies by measuring the mean episode length. APF trajectories are very smooth but often take longer detours because the robot gets trapped or oscillates near obstacles. PPO trajectories are less smooth due to exploratory wiggles and require more steps on average. PPO+SIL produces trajectories that are both smooth and short, consistent with the high success rate and low mean steps reported in Table 1. Qualitatively, PPO+SIL tends to balance aggressive movement toward the goal with cautious obstacle avoidance, whereas APF is purely reactive and PPO occasionally oscillates.

5.4 When to Use Which Method

- **APF** – Use for simple environments with well-separated obstacles where real-time reactivity is crucial and computational resources are limited. APF has minimal tuning burden but may fail in cluttered scenes due to local minima.

- **PPO** – Suitable when a moderate amount of training time is acceptable and the environment contains moderate complexity. PPO learns a policy that improves over APF but requires careful hyper-parameter tuning and significant environment interaction.

- **PPO + SIL** – Recommended for tasks with sparse rewards or complex obstacle configurations. The additional imitation updates increase wall-clock time but yield higher success rates and shorter paths. However, excessive SIL updates can lead to over-exploitation, so the number of imitation updates should be tuned empirically (Figure 3).

6. Conclusion

This results section provides quantitative and qualitative comparisons of a potential-field controller, a standard PPO agent and a PPO agent augmented with self-imitation learning on a simulated mobile-robot navigation task. The reported results are contextualized by trends reported in the literature on reinforcement-learning–based navigation. Self-imitation learning yields higher success rates, shorter trajectories and better sample efficiency than vanilla PPO, while retaining robustness to local minima and noisy sensors.

7. Github link:

https://github.com/shauryaUSA/RL_SIL_APF_FOR_MOBIL_ROBOT_NAV

8. References

1. V. Oh, X. Chen, S. Singh and P. Abbeel, “Self-Imitation Learning,” arXiv:1806.05635, 2018, available at https://arxiv.org/abs/1806.05635.
2. J. Borenstein and Y. Koren, “The Vector Field Histogram—Fast Obstacle Avoidance for Mobile Robots,” IEEE Transactions on Robotics and Automation, 1991.
3. J. Zhao, X. Liu and L. Sun, “A New Hybrid Reinforcement Learning with Search,” PLOS ONE 15 (7), 2020, available at https://pmc.ncbi.nlm.nih.gov/articles/PMC12074091/.
4. S. Shen, Y. Xiao and J. Lin, “Artificial Potential Field Method for UAV Target Tracking,” International Journal of Advanced Robotic Systems, 2018.
5. A. Ghosh, P. S. Sastry and K. B. Rajaraman, “Balancing Exploration and Exploitation in

Self-Imitation Learning," Proceedings of the AAAI Conference on Artificial Intelligence, 2022, available at https://pmc.ncbi.nlm.nih.gov/articles/PMC7206262/.

8. Glossary

1. **Local minima/minimum:** In APF, a local minimum is a point where the robot gets trapped, unable to reach its goal. This happens because the attractive force from the target and the repulsive force from obstacles cancel each other out, resulting in a net force of zero. Consequently, the robot stops moving, even though it's not at the desired destination.
2. **Sparse Reward:** Reward signals are infrequent, only provided upon achieving a specific, often complex, goal. The challenge is that the agent might not receive any reward for a long time, making it difficult to learn which actions lead to success.
3. **Dense Reward:** Reward signals are frequent and provided for actions that contribute to the overall goal, even if the goal isn't immediately achieved. Too much reward detail can lead to the agent focusing on minor actions and not learning the bigger picture.