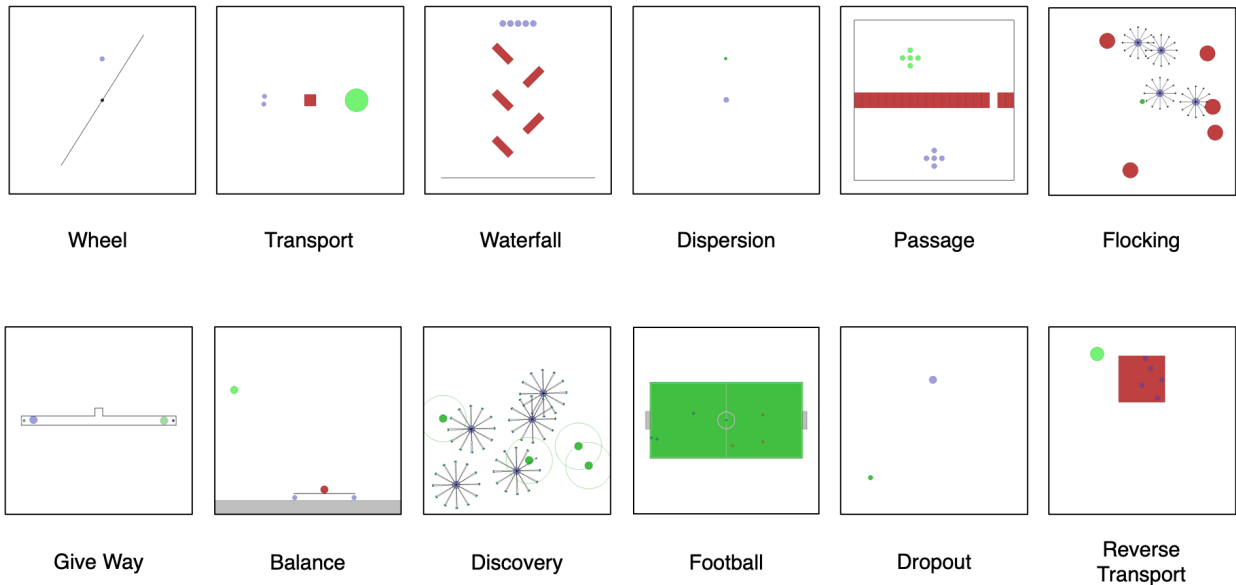


VectorizedMultiAgentSimulator (VMAS)



Welcome to VMAS!

This repository contains the code for the Vectorized Multi-Agent Simulator (VMAS).

VMAS is a vectorized framework designed for efficient MARL benchmarking.

It is comprised of a vectorized 2D physics engine written in PyTorch and a set of challenging multi-robot scenarios.

Scenario creation is made simple and modular to incentivize contributions.

VMAS simulates agents and landmarks of different shapes and supports rotations, elastic collisions, joints, and custom gravity. Holonomic motion models are used for the agents to simplify simulation. Custom sensors such as LIDARs are available and the simulator supports inter-agent communication.

Vectorization in [PyTorch](#) allows VMAS to perform simulations in a batch, seamlessly scaling to tens of thousands of parallel environments on accelerated hardware.

VMAS has an interface compatible with [OpenAI Gym](#) and with the [RLlib](#) library, enabling out-of-the-box integration with a wide range of RL algorithms.

The implementation is inspired by [OpenAI's MPE](#).

Alongside VMAS's scenarios, we port and vectorize all the scenarios in MPE.

Paper

The arXiv paper can be found [here](#).

If you use VMAS in your research, **cite** it using:

```
@article{bettini2022vmas,  
  title = {VMAS: A Vectorized Multi-Agent Simulator for Collective Robot Learning},  
  author = {Bettini, Matteo and Kortvelesy, Ryan and Blumenkamp, Jan and Prorok, Amanda},  
  year = {2022},  
  journal={The 16th International Symposium on Distributed Autonomous Robotic Systems},  
  publisher={Springer}  
}
```

Video

Watch the presentation video of VMAS, showing its structure, scenarios, and experiments.



[![VMAS Video](https://img.youtube.com/vi/aaDRYfiesAY/0.jpg)](https://www.youtube.com/watch?v=aaDRYfiesAY)

Table of contents

- [VectorizedMultiAgentSimulator \(VMAS\)](#)
 - [Welcome to VMAS!](#)
 - [Paper](#)
 - [Video](#)
 - [Table of contents](#)
 - [How to use](#)
 - [Notebooks](#)
 - [Install](#)
 - [Run](#)
 - [RLlib](#)
 - [Simulator features](#)
 - [Creating a new scenario](#)
 - [Play a scenario](#)
 - [Rendering](#)
 - [Rendering on server machines](#)
 - [List of environments](#)
 - [VMAS](#)
 - [MPE](#)
 - [TODOs](#)

How to use

Notebooks

-  [Open in Colab](#) **Using a VMAS environment.**
Here is a simple notebook that you can run to create, step and render any scenario in VMAS. It reproduces the `use_vmas_env.py` script in the `examples` folder.
-  [Open in Colab](#) **Using VMAS in RLlib.** In this notebook, we show how to use any VMAS scenario in RLlib. It reproduces the `rllib.py` script in the `examples` folder.

Install

To install the simulator, you can use pip:

```
pip install vmas
```

Run

To use the simulator, simply create an environment by passing the name of the scenario you want (from the `scenarios` folder) to the `make_env` function.

The function arguments are explained in the documentation. The function returns an environment object with the OpenAI gym interface:

Here is an example:

```

env = vmas.make_env(
    scenario_name="waterfall",
    num_envs=32,
    device="cpu", # Or "cuda" for GPU
    continuous_actions=True,
    wrapper=None, # One of: None, vmas.Wrapper.RLLIB, and vmas.Wrapper.GYM
    max_steps=None, # Defines the horizon. None is infinite horizon.
    **kwargs # Additional arguments you want to pass to the scenario initialization
)

```

A further example that you can run is contained in `use_vmas_env.py` in the `examples` directory.

RLLib

To see how to use VMAS in RLLib, check out the script in `examples/rllib.py`.

Simulator features

- **Vectorized:** VMAS vectorization can step any number of environments in parallel. This significantly reduces the time needed to collect rollouts for training in MARL.
- **Simple:** Complex vectorized physics engines exist (e.g., Brax~\cite{brax2021github}), but they do not scale efficiently when dealing with multiple agents. This defeats the computational speed goal set by vectorization. VMAS uses a simple custom 2D dynamics engine written in PyTorch to provide fast simulation.
- **General:** The core of VMAS is structured so that it can be used to implement general high-level multi-robot problems in 2D. It can support adversarial as well as cooperative scenarios. Holonomic point-robot simulation has been chosen to focus on general high-level problems, without learning low-level custom robot controls through MARL.
- **Extensible:** VMAS is not just a simulator with a set of environments. It is a framework that can be used to create new multi-agent scenarios in a format that is usable by the whole MARL community. For this purpose, we have modularized the process of creating a task and introduced interactive rendering to debug it. You can define your own scenario in minutes. Have a look at the dedicated section in this document.
- **Compatible:** VMAS has wrappers for [RLLib](#) and [OpenAI Gym](#). RLLib has a large number of already implemented RL algorithms.
Keep in mind that this interface is less efficient than the unwrapped version. For an example of wrapping, see the main of `make_env`.
- **Tested:** Our scenarios come with tests which run a custom designed heuristic on each scenario.
- **Entity shapes:** Our entities (agent and landmarks) can have different customizable shapes (spheres, boxes, lines). All these shapes are supported for elastic collisions.
- **Faster than physics engines:** Our simulator is extremely lightweight, using only tensor operations. It is perfect for running MARL training at scale with multi-agent collisions and interactions.
- **Customizable:** When creating a new scenario of your own, the world, agent and landmarks are highly customizable. Examples are: drag, friction, gravity, simulation timestep, non-differentiable communication, agent sensors (e.g. LIDAR), and masses.
- **Non-differentiable communication:** Scenarios can require agents to perform discrete or continuous communication actions.
- **Gravity:** VMAS supports customizable gravity.
- **Sensors:** Our simulator implements ray casting, which can be used to simulate a wide range of distance-based sensors that can be added to agents. We currently support LIDARs. To see available sensors, have a look at the `sensors` script.
- **Joints:** Our simulator supports joints. Joints are constraints that keep entities at a specified distance. The user can specify the anchor points on the two objects, the distance (including 0), the thickness of the joint, if the joint is allowed to rotate at either anchor point, and if he wants the joint to be collidable. Have a look at the waterfall scenario to see how you can use joints.
- **Agent actions:** Agents' physical actions are 2D forces for holonomic motion. Agent rotation can also be controlled through a torque action (activated by setting `agent.action.u_rot_range` at agent creation time). Agents can also be equipped with continuous or discrete communication actions.

Creating a new scenario

To create a new scenario, just extend the `BaseScenario` class in `scenario.py`.

You will need to implement at least `make_world`, `reset_world_at`, `observation`, and `reward`. Optionally, you can also implement `done`, `info`, `process_action`, and `extra_render`.

You can also change the viewer size, zoom, and enable a background rendered grid by changing these inherited attributes in the `make_world` function.

To know how, just read the documentation of `BaseScenario` in `scenario.py` and look at the implemented scenarios.

Play a scenario

You can play with a scenario interactively! **Just execute its script!**

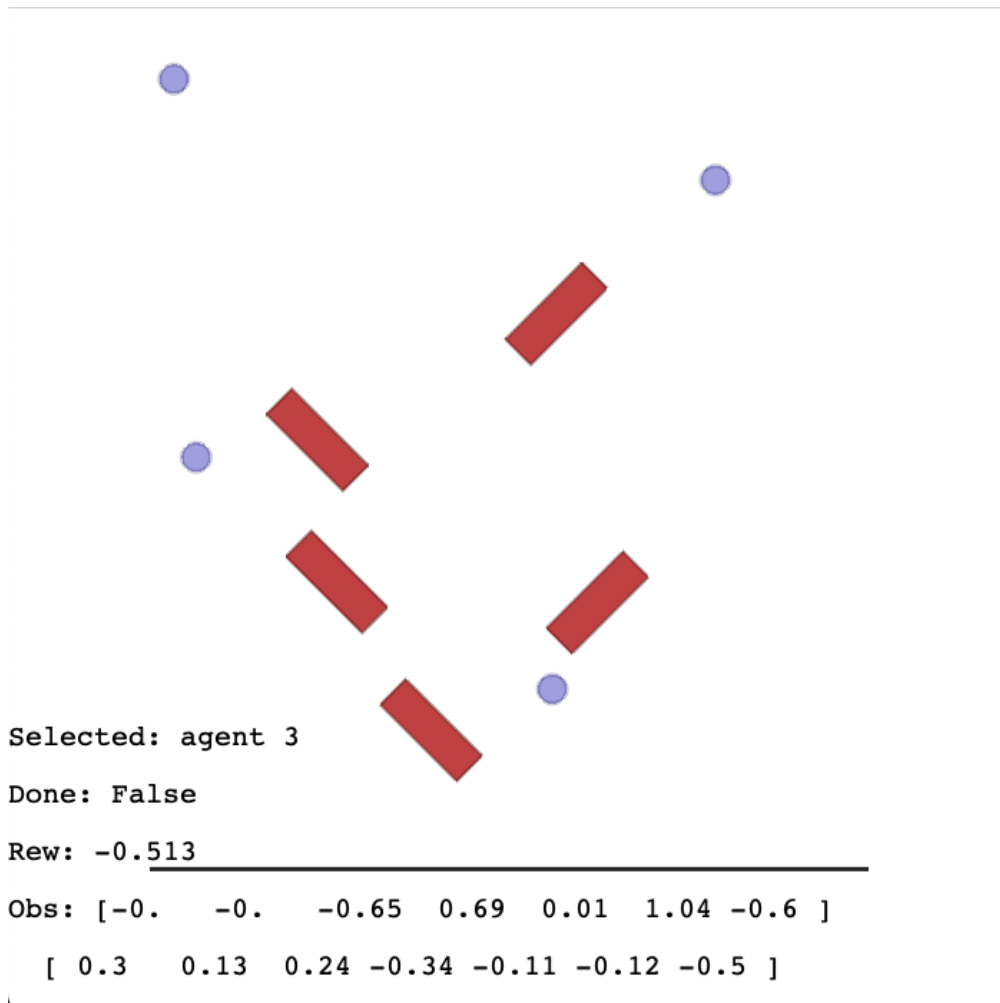
Just use the `render_interactively` function in the `interactive_rendering.py` script. Relevant values will be plotted to screen.

Move the agent with the arrow keys and switch agents with TAB. You can reset the environment by pressing R.

If you have more than 1 agent, you can control another one with W,A,S,D and switch the second agent using LSHIFT. To do this, just set `control_two_agents=True`.

On the screen you will see some data from the agent controlled with arrow keys. This data includes: name, current obs, current reward, total reward so far and environment done flag.

Here is an overview of what it looks like:



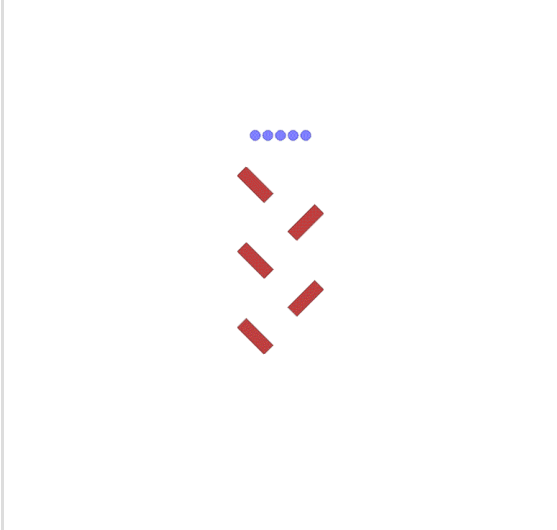
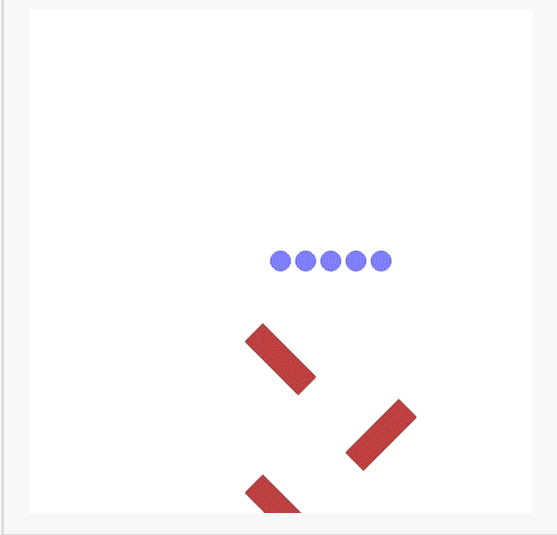
Rendering

To render the environment, just call the `render` or the `try_render_at` functions (depending on environment wrapping).

Example:

```
env.render(  
    mode="rgb_array", # "rgb_array" returns image, "human" renders in display  
    agent_index_focus=4, # If None keep all agents in camera, else focus camera on specific agent  
    index=0, # Index of batched environment to render  
    visualize_when_rgb: bool = False, # Also run human visualization when mode=="rgb_array"  
    plot_position_function=None, # A function to plot under the rendering. This function takes the position (x,y) as in  
)
```

You can also change the viewer size, zoom, and enable a background rendered grid by changing these inherited attributes in the scenario `make_world` function.

Gif	Agent focus
	With <code>agent_index_focus=None</code> the camera keeps focus on all agents
	With <code>agent_index_focus=0</code> the camera follows agent 0

Gif	Agent focus
	With <code>agent_index_focus=4</code> the camera follows agent 4

Rendering on server machines

To render in machines without a display use `mode="rgb_array"` . Make sure you have OpenGL and Pyglet installed.
To use GPUs for headless rendering, you can install the EGL library.
If you do not have EGL, you need to create a fake screen. You can do this by running these commands before the script:

```
export DISPLAY=':99.0'  
Xvfb :99 -screen 0 1400x900x24 > /dev/null 2>&1 &
```


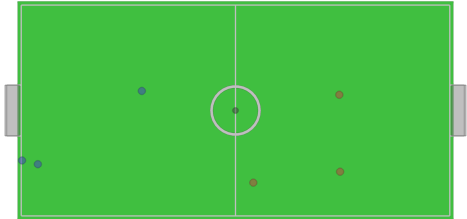
or in this way:

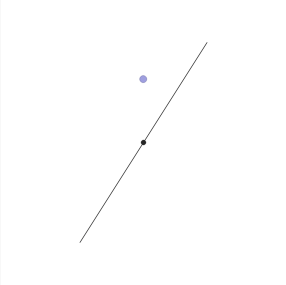
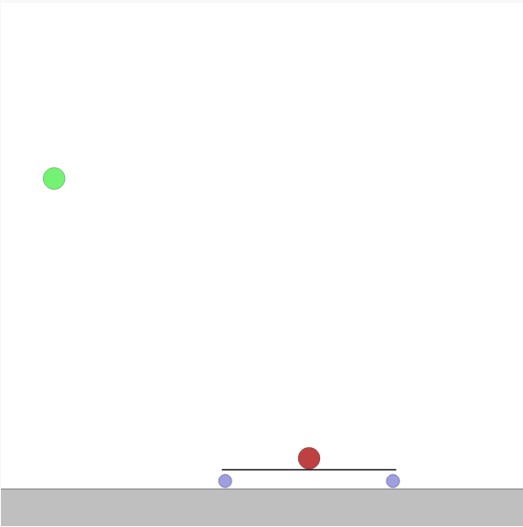
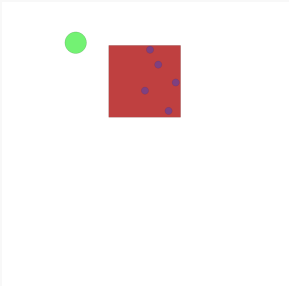

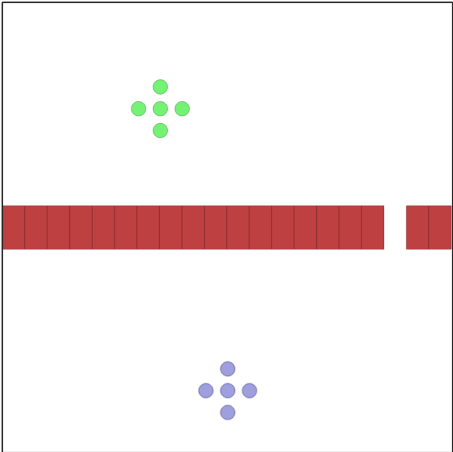

```
xvfb-run -s \"-screen 0 1400x900x24\" python <your_script.py>
```

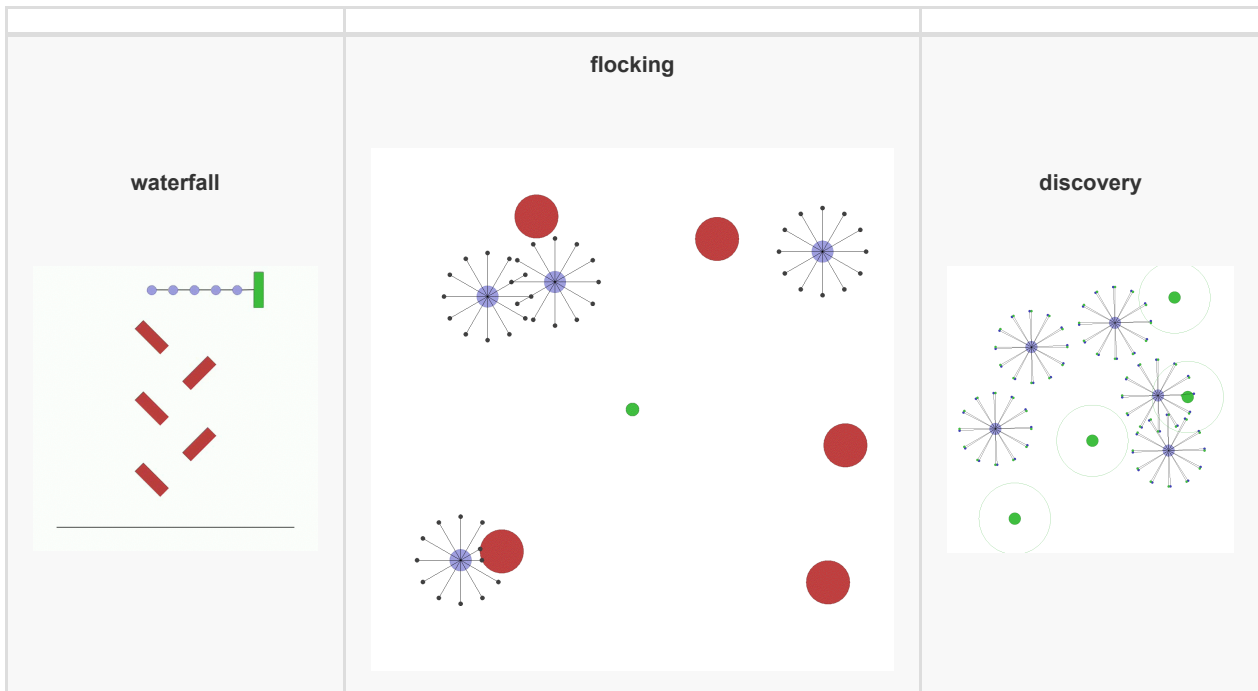
To create a fake screen you need to have `xvfb` installed.




List of environments






VMAS





dropout	football	transport
		

<p>wheel</p> 	<p>balance</p> 	<p>reverse transport</p> 
<p>give_way</p> 	<p>passage</p> 	<p>dispersion</p> 



Env name	Description	GIF
<code>waterfall.py</code>	Debug environment. <code>n_agents</code> agents are spawned in the top of the environment. Each agent is rewarded based on how close it is to the center of the black line at the bottom. Agents have to reach the line and in doing so they might collide with each other and with boxes in the environment.	
<code>dropout.py</code>	In this scenario, <code>n_agents</code> and a goal are spawned at random positions between -1 and 1. Agents cannot collide with each other and with the goal. The reward is shared among all agents. The team receives a reward of 1 when at least one agent reaches the goal. A penalty is given to the team proportional to the sum of the magnitude of actions of every agent. This penalises agents for moving. The impact of the energy reward can be tuned by setting <code>energy_coeff</code> . The default coefficient is 0.02 makes it so that for one agent it is always worth reaching the goal. The optimal policy consists in agents sending only the closest agent to the goal and thus saving as much energy as possible. Every agent observes its position, velocity, relative position to the goal and a flag that is set when someone reaches the goal. The environment terminates when someone reaches the goal. To solve this environment, communication is needed.	
<code>dispersion.py</code>	In this scenario, <code>n_agents</code> agents and goals are spawned. All agents spawn in [0,0] and goals spawn at random positions between -1 and 1. Agents cannot collide with each other and with the goals. Agents are tasked with reaching the goals. When a goal is reached, the team gets a reward of 1 if <code>share_reward</code> is true, otherwise the agents which reach that goal in the same step split the reward of 1. If <code>penalise_by_time</code> is true, every agent gets an additional reward of -0.01 at each step. The optimal policy is for agents to disperse and each tackle a different goal. This requires high coordination and diversity. Every agent observes its position and velocity. For every goal it also observes the relative position and a flag indicating if the goal has been already reached by someone or not. The environment terminates when all the goals are reached.	

Env name	Description	GIF
transport.py	In this scenario, <code>n_agents</code> , <code>n_packages</code> (default 1) and a goal are spawned at random positions between -1 and 1. Packages are boxes with <code>package_mass</code> mass (default 50 times agent mass) and <code>package_width</code> and <code>package_length</code> as sizes. The goal is for agents to push all packages to the goal. When all packages overlap with the goal, the scenario ends. Each agent receives the same reward which is proportional to the sum of the distance variations between the packages and the goal. In other words, pushing a package towards the goal will give a positive reward, while pushing it away, a negative one. Once a package overlaps with the goal, it becomes green and its contribution to the reward becomes 0. Each agent observes its position, velocity, relative position to packages, package velocities, relative positions between packages and the goal and a flag for each package indicating if it is on the goal. By default packages are very heavy and one agent is barely able to push them. Agents need to collaborate and push packages together to be able to move them faster.	
reverse_transport.py	This is exactly the same of transport except with <code>n_agents</code> spawned inside a single package. All the rest is the same.	
give_way.py	In this scenario, two agents and two goals are spawned in a narrow corridor. The agents need to reach the goal with their color. The agents are standing in front of each other's goal and thus need to swap places. In the middle of the corridor there is an asymmetric opening which fits one agent only. Therefore the optimal policy is for one agent to give way to the other. This requires heterogeneous behaviour. If <code>shared_reward</code> is true, each agent gets a reward of one when someone reaches a goal, otherwise each agent gets a reward of 1 for reaching its goal. If <code>dense_reward</code> is True, the agents get dense rewards instead of sparse ones. Each agent observes its position, velocity and the relative position to its goal. The scenario terminates when both agents reach their goals.	
wheel.py	In this scenario, <code>n_agents</code> are spawned at random positions between -1 and 1. One line with <code>line_length</code> and <code>line_mass</code> is spawned in the middle. The line is constrained in the origin and can rotate. The goal of the agents is to make the absolute angular velocity of the line match <code>desired_velocity</code> . Therefore, it is not sufficient for the agents to all push in the extrema of the line, but they need to organize to achieve, and not exceed, the desired velocity. Each agent observes its position, velocity, the current angle of the line module pi, the absolute difference between the current angular velocity of the line and the desired one, and the relative position to the two line extrema. The reward is shared and it is the absolute difference between the current angular velocity of the line and the desired one.	
balance.py	In this scenario, <code>n_agents</code> are spawned uniformly spaced out under a line upon which lies a spherical package of mass <code>package_mass</code> . The team and the line are spawned at a random X position at the bottom of the environment. The environment has vertical gravity. If <code>random_package_pos_on_line</code> is True (default), the relative X position of the package on the line is random. In the top half of the environment a goal is spawned. The agents have to carry the package to the goal. Each agent receives the same reward which is proportional to the distance variation between the package and the goal. In other words, getting the package closer to the goal will give a positive reward, while moving it away, a negative one. The team receives a negative reward of -10 for making the package or the line fall to the floor. The observations for each agent are: its position, velocity, relative position to the package, relative position to the line, relative position between package and goal, package velocity, line velocity, line angular velocity, and line rotation mod pi. The environment is done either when the package or the line fall or when the package touches the goal.	

Env name	Description	GIF
passage.py	In this scenario, a team of 5 robots is spawned in formation at a random location in the bottom part of the environment. A similar formation of goals is spawned at random in the top part. Each robot has to reach its corresponding goal. In the middle of the environment there is a wall with <code>n_passages</code> . Each passage is large enough to fit one robot at a time. Each agent receives a reward which is proportional to the distance variation between itself and the goal. In other words, getting closer to the goal will give a positive reward, while moving it away, a negative one. This reward will be shared in case <code>shared_reward</code> is true. If collisions among robots occur, each robot involved will get a reward of -10. Each agent observes: its position, velocity, relative position to the goal and relative position to the center of each passage. The environment terminates when all the robots reach their goal.	
football.py	In this scenario, a team of <code>n_blue_agents</code> play football against a team of <code>n_red_agents</code> . The boolean parameters <code>ai_blue_agents</code> and <code>ai_red_agents</code> specify whether each team is controlled by action inputs or a programmed AI. Consequently, football can be treated as either a cooperative or competitive task. The reward in this scenario can be tuned with <code>dense_reward_ratio</code> , where a value of 0 denotes a fully sparse reward (1 for a goal scored, -1 for a goal conceded), and 1 denotes a fully dense reward (based on the the difference of the "attacking value" of each team, which considers the distance from the ball to the goal and the presence of open dribbling/shooting lanes to the goal). Every agent observes its position, velocity, relative position to the ball, and relative velocity to the ball. The episode terminates when one team scores a goal.	
discovery.py	In this scenario, a team of <code>n_agents</code> has to coordinate to cover <code>n_targets</code> targets as quickly as possible while avoiding collisions. A target is considered covered if <code>agents_per_target</code> agents have approached a target at a distance of at least <code>covering_range</code> . After a target is covered, the <code>agents_per_target</code> each receive a reward and the target is respawned to a new random position. Agents receive a penalty if they collide with each other. Every agent observes its position, velocity, LIDAR range measurements to other agents and targets (independently). The episode terminates after a fixed number of time steps.	
flocking.py	In this scenario, a team of <code>n_agents</code> has to flock around a target while staying together and maximising their velocity without colliding with each other and a number of <code>n_obstacles</code> obstacles. Agents are penalized for colliding with each other and with obstacles, and are rewarded for maximising velocity and minimising the span of the flock (cohesion). Every agent observes its position, velocity, and LIDAR range measurements to other agents. The episode terminates after a fixed number of time steps.	

MPE

Env name in code (name in paper)	Communication?	Competitive?	Notes
simple.py	N	N	Single agent sees landmark position, rewarded based on how close it gets to landmark. Not a multi-agent environment -- used for debugging policies.

Env name in code (name in paper)	Communication?	Competitive?	Notes
<code>simple_adversary.py</code> (Physical deception)	N	Y	1 adversary (red), N good agents (green), N landmarks (usually N=2). All agents observe position of landmarks and other agents. One landmark is the 'target landmark' (colored green). Good agents rewarded based on how close one of them is to the target landmark, but negatively rewarded if the adversary is close to target landmark. Adversary is rewarded based on how close it is to the target, but it doesn't know which landmark is the target landmark. So good agents have to learn to 'split up' and cover all landmarks to deceive the adversary.
<code>simple_crypto.py</code> (Covert communication)	Y	Y	Two good agents (alice and bob), one adversary (eve). Alice must sent a private message to bob over a public channel. Alice and bob are rewarded based on how well bob reconstructs the message, but negatively rewarded if eve can reconstruct the message. Alice and bob have a private key (randomly generated at beginning of each episode), which they must learn to use to encrypt the message.
<code>simple_push.py</code> (Keep-away)	N	Y	1 agent, 1 adversary, 1 landmark. Agent is rewarded based on distance to landmark. Adversary is rewarded if it is close to the landmark, and if the agent is far from the landmark. So the adversary learns to push agent away from the landmark.
<code>simple_reference.py</code>	Y	N	2 agents, 3 landmarks of different colors. Each agent wants to get to their target landmark, which is known only by other agent. Reward is collective. So agents have to learn to communicate the goal of the other agent, and navigate to their landmark. This is the same as the <code>simple_speaker_listener</code> scenario where both agents are simultaneous speakers and listeners.
<code>simple_speaker_listener.py</code> (Cooperative communication)	Y	N	Same as <code>simple_reference</code> , except one agent is the 'speaker' (gray) that does not move (observes goal of other agent), and other agent is the listener (cannot speak, but must navigate to correct landmark).

Env name in code (name in paper)	Communication?	Competitive?	Notes
<code>simple_spread.py</code> (Cooperative navigation)	N	N	N agents, N landmarks. Agents are rewarded based on how far any agent is from each landmark. Agents are penalized if they collide with other agents. So, agents have to learn to cover all the landmarks while avoiding collisions.
<code>simple_tag.py</code> (Predator-prey)	N	Y	Predator-prey environment. Good agents (green) are faster and want to avoid being hit by adversaries (red). Adversaries are slower and want to hit good agents. Obstacles (large black circles) block the way.
<code>simple_world_comm.py</code>	Y	Y	Environment seen in the video accompanying the paper. Same as <code>simple_tag</code> , except (1) there is food (small blue balls) that the good agents are rewarded for being near, (2) we now have 'forests' that hide agents inside from being seen from outside; (3) there is a 'leader adversary' that can see the agents at all times, and can communicate with the other adversaries to help coordinate the chase.

TODOS

- ☐ Talk about action preprocessing and velocity controller
- ☐ New envs from joint project with their descriptions, gifs, and joint and vel control references
- ☐ New envs from adversarial project
- ☐ Custom actions for scenario
- ☐ Implement 1D camera sensor
- ☐ Make football heuristic efficient
- ☒ Link video of experiments
- ☒ Add LIDAR section
- ☒ Implement LIDAR
- ☒ Rewrite all MPE scenarios
 - ☒ `simple`
 - ☒ `simple_adversary`
 - ☒ `simple_crypto`
 - ☒ `simple_push`
 - ☒ `simple_reference`
 - ☒ `simple_speaker_listener`
 - ☒ `simple_spread`
 - ☒ `simple_tag`
 - ☒ `simple_world_comm`