highway-env Documentation

Edouard Leurent

Jun 02, 2022
CONTENTS

1 How to cite this work? 3

2 Documentation contents 5
  2.1 Installation ......................................................... 5
  2.2 Getting Started .................................................... 6
  2.3 User Guide .......................................................... 17
  2.4 Frequently Asked Questions ....................................... 61
  2.5 Bibliography ......................................................... 62

Bibliography 63

Python Module Index 65

Index 67
This project gathers a collection of environment for decision-making in Autonomous Driving.

The purpose of this documentation is to provide:

1. a quick start guide describing the environments and their customization options;
2. a detailed description of the nuts and bolts of the project, and how you can contribute.
HOW TO CITE THIS WORK?

If you use this package, please consider citing it with this piece of BibTeX:

```bibtex
@misc{highway-env,
    author = {Leurent, Edouard},
    title = {An Environment for Autonomous Driving Decision-Making},
    year = {2018},
    publisher = {GitHub},
    journal = {GitHub repository},
    howpublished = \url{https://github.com/eleurent/highway-env},
    }  
```
Chapter 1. How to cite this work?
2.1 Installation

2.1.1 Prerequisites

This project requires python3 (>=3.5)

The graphics require the installation of pygame, which itself has dependencies that must be installed manually.

Ubuntu

```
sudo apt-get update -y
sudo apt-get install -y python-dev libSDL-image1.2-dev libSDL-mixer1.2-dev
   libSDL-ttf2.0-dev libSDL1.2-dev libsmpeg-dev python-numpy subversion libportmidi-dev
   ffmpeg libswscale-dev libavformat-dev libavcodec-dev libfreetype6-dev gcc
```

Windows 10

We recommend using Anaconda.

2.1.2 Stable release

To install the latest stable version:

```
pip install highway-env
```

2.1.3 Development version

To install the current development version:

```
pip install --user git+https://github.com/eleurent/highway-env
```


2.2 Getting Started

2.2.1 Making an environment

Here is a quick example of how to create an environment:

```python
import gym
import highway_env
from matplotlib import pyplot as plt
%matplotlib inline

env = gym.make('highway-v0')
env.reset()
for _ in range(3):
    action = env.action_type.actions_indexes['IDLE']
    obs, reward, done, info = env.step(action)
    env.render()

plt.imshow(env.render(mode='rgb_array'))
plt.show()
```

All the environments

Here is the list of all the environments available and their descriptions:

Highway

In this task, the ego-vehicle is driving on a multilane highway populated with other vehicles. The agent’s objective is to reach a high speed while avoiding collisions with neighbouring vehicles. Driving on the right side of the road is also rewarded.
Usage

```python
env = gym.make("highway-v0")
```

Default configuration

```python
{
    "observation": {
        "type": "Kinematics"
    },
    "action": {
        "type": "DiscreteMetaAction",
    },
    "lanes_count": 4,
    "vehicles_count": 50,
    "duration": 40,  # [s]
    "initial_spacing": 2,
    "collision_reward": -1,  # The reward received when colliding with a vehicle.
    "reward_speed_range": [20, 30],  # [m/s] The reward for high speed is mapped linearly from this range to [0, HighwayEnv.HIGH_SPEED_REWARD].
    "simulation_frequency": 15,  # [Hz]
    "policy_frequency": 1,  # [Hz]
    "other_vehicles_type": "highway_env.vehicle.behavior.IDMVehicle",
    "screen_width": 600,  # [px]
    "screen_height": 150,  # [px]
    "centering_position": [0.3, 0.5],
    "scaling": 5.5,
    "show_trajectories": False,
    "render_agent": True,
    "offscreen_rendering": False
}
```

More specifically, it is defined in:

```python
classmethod HighwayEnv.default_config() → dict

Default environment configuration.
Can be overloaded in environment implementations, or by calling configure().
```

Faster variant

A faster (x15 speedup) variant is also available with:

```python
env = gym.make("highway-fast-v0")
```

The details of this variant are described here.
highway-env Documentation

API

class highway_env.envs.highway_env.HighwayEnv(config: Optional[dict] = None)
    A highway driving environment.
    The vehicle is driving on a straight highway with several lanes, and is rewarded for reaching a high speed, staying on the rightmost lanes and avoiding collisions.

    classmethod default_config() -> dict
        Default environment configuration.
        Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict

Merge

In this task, the ego-vehicle starts on a main highway but soon approaches a road junction with incoming vehicles on the access ramp. The agent’s objective is now to maintain a high speed while making room for the vehicles so that they can safely merge in the traffic.

Usage

```python
env = gym.make("merge-v0")
```

Default configuration

```json
{
    "observation": {
        "type": "TimeToCollision"
    },
    "action": {
        "type": "DiscreteMetaAction"
    },
    "simulation_frequency": 15,  # [Hz]
    "policy_frequency": 1,  # [Hz]
    "other_vehicles_type": "highway_env.vehicle.behavior.IDMVehicle",
    "screen_width": 600,  # [px]
    "screen_height": 150,  # [px]
    "centering_position": [0.3, 0.5],
    "scaling": 5.5,
    "show_trajectories": False,
    "render_agent": True,
    "offscreen_rendering": False
}
```

More specifically, it is defined in:

```python
classmethod MergeEnv.default_config() -> dict
    Default environment configuration.
    Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict
```
API

```python
class highway_env.envs.merge_env.MergeEnv(config: Optional[dict] = None)
A highway merge negotiation environment.

The ego-vehicle is driving on a highway and approached a merge, with some vehicles incoming on the access ramp. It is rewarded for maintaining a high speed and avoiding collisions, but also making room for merging vehicles.

classmethod default_config() -> dict
Default environment configuration.

Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict
```

Roundabout

In this task, the ego-vehicle if approaching a roundabout with flowing traffic. It will follow its planned route automatically, but has to handle lane changes and longitudinal control to pass the roundabout as fast as possible while avoiding collisions.

Usage

```python
env = gym.make("roundabout-v0")
```

Default configuration

```json
{
   "observation": {
       "type": "TimeToCollision"
   },
   "action": {
       "type": "DiscreteMetaAction"
   },
   "incoming_vehicle_destination": None,
   "duration": 11
   "simulation_frequency": 15, # [Hz]
   "policy_frequency": 1, # [Hz]
   "other_vehicles_type": "highway_env.vehicle.behavior.IDMVehicle",
   "screen_width": 600, # [px]
   "screen_height": 600, # [px]
   "centering_position": [0.5, 0.6],
   "scaling": 5.5,
   "show_trajectories": False,
   "render_agent": True,
   "offscreen_rendering": False
}
```

More specifically, it is defined in:

2.2. Getting Started
**classmethod** RoundaboutEnv.\texttt{default_config}() $\rightarrow$ dict

Default environment configuration.

Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict

### API

**class** highway_env.envs.roundabout_env.RoundaboutEnv(config: \texttt{Optional[dict]} = \texttt{None})

**classmethod** default_config() $\rightarrow$ dict

Default environment configuration.

Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict

---

**Parking**

A goal-conditioned continuous control task in which the ego-vehicle must park in a given space with the appropriate heading.

---

**Usage**

```
env = gym.make("parking-v0")
```

---

**Default configuration**

```json
{
    "observation": {
        "type": "KinematicsGoal",
        "features": ["x", "y", "vx", "vy", "cos_h", "sin_h"],
        "scales": [100, 100, 5, 5, 1, 1],
        "normalize": False
    },
    "action": {
        "type": "ContinuousAction"
    },
    "simulation_frequency": 15,
    "policy_frequency": 5,
    "screen_width": 600,
    "screen_height": 300,
    "centering_position": [0.5, 0.5],
    "scaling": 7,
    "show_trajectories": False,
    "render_agent": True,
    "offscreen_rendering": False
}
```

More specifically, it is defined in:
classmethod ParkingEnv.default_config() → dict
    Default environment configuration.
    Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict

API

class highway_env.envs.parking_env.ParkingEnv(config: Optional[dict] = None)
    A continuous control environment.
    It implements a reach-type task, where the agent observes their position and speed and must control their acceleration and steering so as to reach a given goal.
    Credits to Munir Jojo-Verge for the idea and initial implementation.

classmethod default_config() → dict
    Default environment configuration.
    Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict

define_spaces() → None
    Set the types and spaces of observation and action from config.

compute_reward(achieved_goal: ndarray, desired_goal: ndarray, info: dict, p: float = 0.5) → float
    Proximity to the goal is rewarded
    We use a weighted p-norm

    Parameters
    • achieved_goal – the goal that was achieved
    • desired_goal – the goal that was desired
    • info (dict) – any supplementary information
    • p – the Lp^p norm used in the reward. Use p<1 to have high kurtosis for rewards in [0, 1]

    Returns
    the corresponding reward

Intersection

An intersection negotiation task with dense traffic.

Warning: It’s quite hard to come up with good decentralized behaviors for other agents to avoid each other. Of course, this could be achieved by sophisticated centralized schedulers, or traffic lights, but to keep things simple a rudimentary collision prediction was added in the behaviour of other vehicles.

This simple system sometime fails which results in collisions, blocking the way for the ego-vehicle. I figured it was fine for my own purpose, since it did not happen too often and it’s reasonable to expect the ego-vehicle to simply wait the end of episode in these situations. But I agree that it is not ideal, and I welcome any contribution on that matter.

2.2. Getting Started
Usage

```python
env = gym.make("intersection-v0")
```

Default configuration

```json
{
    "observation": {
        "type": "Kinematics",
        "vehicles_count": 15,
        "features": ["presence", "x", "y", "vx", "vy", "cos_h", "sin_h"],
        "features_range": {
            "x": [-100, 100],
            "y": [-100, 100],
            "vx": [-20, 20],
            "vy": [-20, 20],
        },
        "absolute": True,
        "flatten": False,
        "observe_intentions": False
    },
    "action": {
        "type": "DiscreteMetaAction",
        "longitudinal": False,
        "lateral": True,
    },
    "duration": 13,  # [s]
    "destination": "o1",
    "initial_vehicle_count": 10,
    "spawn_probability": 0.6,
    "screen_width": 600,
    "screen_height": 600,
    "centering_position": [0.5, 0.6],
    "scaling": 5.5 * 1.3,
    "collision_reward": IntersectionEnv.COLLISION_REWARD,
    "normalize_reward": False
}
```

More specifically, it is defined in:

```python
classmethod IntersectionEnv.default_config() → dict

Default environment configuration.

Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict
```
API

class highway_env.envs.intersection_env.IntersectionEnv(config: Optional[dict] = None)

   @classmethod
   def default_config(cls) -> dict
       Default environment configuration.
       Can be overloaded in environment implementations, or by calling configure().
       :return: a configuration dict

   def step(self, action: int) -> Tuple[np.ndarray, float, bool, dict]
       Perform an action and step the environment dynamics.
       The action is executed by the ego-vehicle, and all other vehicles on
       the road performs their default behaviour for several simulation
       timesteps until the next decision making step.

       Parameters
           action – the action performed by the ego-vehicle

       Returns
           a tuple (observation, reward, terminal, info)

   Racetrack

   A continuous control environment, where the agent has to follow the
   tracks while avoiding collisions with other vehicles.

   Credits and many thanks to @supperted825 for the idea and initial implementation.

Usage

e = gym.make("racetrack-v0")

Default configuration

```json
{
  "observation": {
    "type": "OccupancyGrid",
    "features": ['presence', 'on_road'],
    "grid_size": [[-18, 18], [-18, 18]],
    "grid_step": [3, 3],
    "as_image": False,
    "align_to_vehicle_axes": True
  },
  "action": {
    "type": "ContinuousAction",
    "longitudinal": False,
    "lateral": True
  },
  "simulation_frequency": 15,
  "policy_frequency": 5,
}
```

(continues on next page)
More specifically, it is defined in:

```python
classmethod RacetrackEnv.default_config() → dict
    Default environment configuration.
    Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict
```

## API

```python
class highway_env.envs.racetrack_env.RacetrackEnv(config: Optional[dict] = None)
    A continuous control environment.
    The agent needs to learn two skills: - follow the tracks - avoid collisions with other vehicles
    Credits and many thanks to @supperted825 for the idea and initial implementation. See https://github.com/eleurent/highway-env/issues/231
```

```python
classmethod default_config() → dict
    Default environment configuration.
    Can be overloaded in environment implementations, or by calling configure(). :return: a configuration dict
```

### 2.2.2 Configuring an environment

The *observations*, *actions*, *dynamics* and *rewards* of an environment are parametrized by a configuration, defined as a `config` dictionary. After environment creation, the configuration can be accessed using the `config` attribute.

```python
import pprint

env = gym.make("highway-v0")
pprint.pprint(env.config)
```

```python
{
    'action': {'type': 'DiscreteMetaAction'},
    'centering_position': [0.3, 0.5],
    'collision_reward': -1,
    'controlled_vehicles': 1,
    'duration': 40,
    'o...}
```

(continues on next page)
'ego_spacing': 2,
'high_speed_reward': 0.4,
'initial_lane_id': None,
'lane_change_reward': 0,
'lanes_count': 4,
'manual_control': False,
'observation': {'type': 'Kinematics'},
'offroad_terminal': False,
'offscreen_rendering': True,
'other_vehicles_type': 'highway_env.vehicle.behavior.IDMVehicle',
'policy_frequency': 1,
'real_time_rendering': False,
'render_agent': True,
'reward_speed_range': [20, 30],
'right_lane_reward': 0.1,
'scaling': 5.5,
'screen_height': 150,
'screen_width': 600,
'show_trajectories': False,
'simulation_frequency': 15,
'vehicles_count': 50,
'vehicles_density': 1}

For example, the number of lanes can be changed with:

```python
env.config["lanes_count"] = 2
env.reset()
plt.imshow(env.render(mode="rgb_array"))
plt.show()
```

**Note:** The environment must be `reset()` for the change of configuration to be effective.
2.2.3 Training an agent

Reinforcement Learning agents can be trained using libraries such as `eleurent/rl-agents`, `openai/baselines` or `Stable Baselines3`.

Here is an example of SB3’s DQN implementation trained on `highway-fast-v0` with its default kinematics observation and an MLP model.

```python
import gym
import highway_env
from stable_baselines3 import DQN

env = gym.make("highway-fast-v0")
model = DQN('MlpPolicy', env,
            policy_kwargs=dict(net_arch=[256, 256]),
            learning_rate=5e-4,
            buffer_size=15000,
            learning_starts=200,
            batch_size=32,
            gamma=0.8,
            train_freq=1,
            gradient_steps=1,
            target_update_interval=50,
            verbose=1,
            tensorboard_log="highway_dqn/"
model.learn(int(2e4))
model.save("highway_dqn/model")

# Load and test saved model
model = DQN.load("highway_dqn/model")
while True:
    done = False
    obs = env.reset()
    while not done:
        action, _states = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)
        env.render()
```

A full run takes about 25mn on my laptop (fps=14). The following results are obtained:

**Note:** There are several ways to get better performances. For instance, SB3 provides only vanilla Deep Q-Learning and has no extensions such as Double-DQN, Dueling-DQN and Prioritized Experience Replay. However, `eleurent/rl-agents`’s implementation of DQN does provide those extensions, which yields better results. Improvements can also be obtained by changing the observation type or the model, see the [FAQ](https://highway-env.readthedocs.io/en/latest/faq.html).
2.2.4 Examples on Google Colab

Several scripts and notebooks to train driving policies on highway-env are available on this page. Here are a few of them:

- Highway with image observations and a CNN model
  Train SB3’s DQN on highway-fast-v0, but using image observations and a CNN model for the value function.

- Trajectory Planning on Highway
  Plan a trajectory on highway-v0 using the OPD [HM08] implementation from eleurent/rl-agents.

- A Model-based Reinforcement Learning tutorial on Parking
  A tutorial written for RLSS 2019 and demonstrating the principle of model-based reinforcement learning on the parking-v0 task.

- Parking with Hindsight Experience Replay
  Train a goal-conditioned parking-v0 policy using the HER [AWR+17] implementation from stable-baselines.

- Intersection with DQN and social attention
  Train an intersection-v0 crossing policy using the social attention architecture [LM19] and the DQN implementation from eleurent/rl-agents.

2.3 User Guide

2.3.1 Observations

For all environments, several types of observations can be used. They are defined in the observation module. Each environment comes with a default observation, which can be changed or customised using environment configurations. For instance,
env = gym.make('highway-v0')
env.configure({
    "observation": {
        "type": "OccupancyGrid",
        "vehicles_count": 15,
        "features": ["presence", "x", "y", "vx", "vy", "cos_h", "sin_h"],
        "features_range": {
            "x": [-100, 100],
            "y": [-100, 100],
            "vx": [-20, 20],
            "vy": [-20, 20]
        },
        "grid_size": [[-27.5, 27.5], [-27.5, 27.5]],
        "grid_step": [5, 5],
        "absolute": False
    }
})
env.reset()

**Note:** The "type" field in the observation configuration takes values defined in `observation_factory()` (see source)

### Kinematics

The *KinematicObservation* is a $V \times F$ array that describes a list of $V$ nearby vehicles by a set of features of size $F$, listed in the "features" configuration field. For instance:

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>$x$</th>
<th>$y$</th>
<th>$v_x$</th>
<th>$v_y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-vehicle</td>
<td>5.0</td>
<td>4.0</td>
<td>15.0</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 1</td>
<td>-10.0</td>
<td>4.0</td>
<td>12.0</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 2</td>
<td>13.0</td>
<td>8.0</td>
<td>13.5</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>vehicle V</td>
<td>22.2</td>
<td>10.5</td>
<td>18.0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**Note:** The ego-vehicle is always described in the first row

If configured with `normalize=True` (default), the observation is normalized within a fixed range, which gives for the range [100, 100, 20, 20]:

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>$x$</th>
<th>$y$</th>
<th>$v_x$</th>
<th>$v_y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-vehicle</td>
<td>0.05</td>
<td>0.04</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 1</td>
<td>-0.1</td>
<td>0.04</td>
<td>0.6</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 2</td>
<td>0.13</td>
<td>0.08</td>
<td>0.675</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>vehicle V</td>
<td>0.222</td>
<td>0.105</td>
<td>0.9</td>
<td>0.025</td>
</tr>
</tbody>
</table>

If configured with `absolute=False`, the coordinates are relative to the ego-vehicle, except for the ego-vehicle which
stays absolute.

<table>
<thead>
<tr>
<th>Vehicle</th>
<th>x</th>
<th>y</th>
<th>vx</th>
<th>vy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-vehicle</td>
<td>0.05</td>
<td>0.04</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 1</td>
<td>-0.15</td>
<td>0</td>
<td>-0.15</td>
<td>0</td>
</tr>
<tr>
<td>vehicle 2</td>
<td>0.08</td>
<td>0.04</td>
<td>-0.075</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>vehicle V</td>
<td>0.172</td>
<td>0.065</td>
<td>0.15</td>
<td>0.025</td>
</tr>
</tbody>
</table>

**Note:** The number $V$ of vehicles is constant and configured by the `vehicles_count` field, so that the observation has a fixed size. If fewer vehicles than `vehicles_count` are observed, the last rows are placeholders filled with zeros. The `presence` feature can be used to detect such cases, since it is set to 1 for any observed vehicle and 0 for placeholders.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>presence</td>
<td>Disambiguate agents at 0 offset from non-existent agents.</td>
</tr>
<tr>
<td>$x$</td>
<td>World offset of ego vehicle or offset to ego vehicle on the x axis.</td>
</tr>
<tr>
<td>$y$</td>
<td>World offset of ego vehicle or offset to ego vehicle on the y axis.</td>
</tr>
<tr>
<td>$vx$</td>
<td>Velocity on the x axis of vehicle.</td>
</tr>
<tr>
<td>$vy$</td>
<td>Velocity on the y axis of vehicle.</td>
</tr>
<tr>
<td>heading</td>
<td>Heading of vehicle in radians.</td>
</tr>
<tr>
<td>$\cos_h$</td>
<td>Trigonometric heading of vehicle.</td>
</tr>
<tr>
<td>$\sin_h$</td>
<td>Trigonometric heading of vehicle.</td>
</tr>
<tr>
<td>$\cos_d$</td>
<td>Trigonometric direction to the vehicle’s destination.</td>
</tr>
<tr>
<td>$\sin_d$</td>
<td>Trigonometric direction to the vehicle’s destination.</td>
</tr>
<tr>
<td>$long_{off}$</td>
<td>Longitudinal offset to closest lane.</td>
</tr>
<tr>
<td>$lat_{off}$</td>
<td>Lateral offset to closest lane.</td>
</tr>
<tr>
<td>$ang_{off}$</td>
<td>Angular offset to closest lane.</td>
</tr>
</tbody>
</table>

**Example configuration**

```python
import gym
import highway_env

config = {
    "observation": {
        "type": "Kinematics",
        "vehicles_count": 15,
        "features": ["presence", "x", "y", "vx", "vy", "cos_h", "sin_h"],
        "features_range": {
            "x": [-100, 100],
            "y": [-100, 100],
            "vx": [-20, 20],
            "vy": [-20, 20]
        },
        "absolute": False,
        "order": "sorted"
    }
}
```

(continues on next page)
env = gym.make('highway-v0')
env.configure(config)
obs = env.reset()
print(obs)

Grayscale Image

The GrayscaleObservation is a $W \times H$ grayscale image of the scene, where $W,H$ are set with the observation_shape parameter. The RGB to grayscale conversion is a weighted sum, configured by the weights parameter. Several images can be stacked with the stack_size parameter, as is customary with image observations.
Example configuration

```python
from matplotlib import pyplot as plt
%matplotlib inline
config = {
    "observation": {
        "type": "GrayscaleObservation",
        "observation_shape": (128, 64),
        "stack_size": 4,
        "weights": [0.2989, 0.5870, 0.1140],  # weights for RGB conversion
        "scaling": 1.75,
    },
    "policy_frequency": 2
}
env.configure(config)
obs = env.reset()

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(12, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(obs[i, ...].T, cmap=plt.get_cmap('gray'))
plt.show()
```

Illustration of the stack mechanism

We illustrate the stack update by performing three steps in the environment.

```python
for _ in range(3):
    obs, _, _, _ = env.step(env.action_type.actions_indexes["IDLE"])

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(12, 5))
for i, ax in enumerate(axes.flat):
    ax.imshow(obs[i, ...].T, cmap=plt.get_cmap('gray'))
plt.show()
```
Occupancy grid

The OccupancyGridObservation is a $W \times H \times F$ array, that represents a grid of shape $W \times H$ discretising the space $(X, Y)$ around the ego-vehicle in uniform rectangle cells. Each cell is described by $F$ features, listed in the "features" configuration field. The grid size and resolution is defined by the grid_size and grid_steps configuration fields.

For instance, the channel corresponding to the presence feature may look like this:

Table 1: presence feature: one vehicle is close to the north, and one is farther to the east.

```
   0 0 0 0 0
   0 0 0 0 0
   0 0 0 0 0
   0 0 0 0 1
   0 0 0 0 0
   0 0 0 0 0
```

The corresponding $v_x$ feature may look like this:

Table 2: $v_x$ feature: the north vehicle drives at the same speed as the ego-vehicle, and the east vehicle a bit slower

```
   0 0 0 0 0
   0 0 0 0 0
   0 0 0 0 -0.1
   0 0 0 0 0
   0 0 0 0 0
```

Example configuration

```
"observation": {
   "type": "OccupancyGrid",
   "vehicles_count": 15,
   "features": ["presence", "x", "y", "vx", "vy", "cos_h", "sin_h"],
   "features_range": {
      "x": [-100, 100],
      "y": [-100, 100],
      "vx": [-20, 20],
   }
}
```
"vy": [-20, 20],
"grid_size": [[-27.5, 27.5], [-27.5, 27.5]],
"grid_step": [5, 5],
"absolute": False
}

Time to collision

The TimeToCollisionObservation is a $V \times L \times H$ array, that represents the predicted time-to-collision of observed vehicles on the same road as the ego-vehicle. These predictions are performed for $V$ different values of the ego-vehicle speed, $L$ lanes on the road around the current lane, and represented as one-hot encodings over $H$ discretised time values (bins), with 1s steps.

For instance, consider a vehicle at 25m on the right-lane of the ego-vehicle and driving at 15 m/s. Using $V = 3$, $L = 3$, $H = 10$, with ego-speed of {15 m/s, 20 m/s and 25 m/s}, the predicted time-to-collisions are $\infty$, 5s, 2.5s and the corresponding observation is

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
```

Example configuration

```
"observation": {
    "type": "TimeToCollision"
    "horizon": 10
},
```

API

```python
class highway_env.envs.common.observation.GrayscaleObservation(env: AbstractEnv,
observation_shape: Tuple[int, int], stack_size: int, weights: List[float], scaling: Optional[float] = None,
centering_position: Optional[List[float]] = None,
**kwargs)
```
An observation class that collects directly what the simulator renders.

Also stacks the collected frames as in the nature DQN. The observation shape is C x W x H.

Specific keys are expected in the configuration dictionary passed. Example of observation dictionary in the environment config:

```json
"observation": {
    "type": "GrayscaleObservation", "observation_shape": (84, 84) "stack_size": 4, "weights":
    [0.2989, 0.5870, 0.1140], # weights for RGB conversion,
}
```

**space()** → Space

Get the observation space.

**observe()** → ndarray

Get an observation of the environment state.

```python
```

Observe the kinematics of nearby vehicles.

**space()** → Space

Get the observation space.

**normalize_obs(df: DataFrame) → DataFrame**

Normalize the observation values.

For now, assume that the road is straight along the x axis.

**observe()** → ndarray

Get an observation of the environment state.

```python
```

Observe an occupancy grid of nearby vehicles.
space() → Space
Get the observation space.

normalize(df: DataFrame) → DataFrame
Normalize the observation values.

For now, assume that the road is straight along the x axis. :param DataFrame df: observation data

observe() → ndarray
Get an observation of the environment state.

pos_to_index(position: Union[ndarray, Sequence[float]], relative: bool = False) → Tuple[int, int]
Convert a world position to a grid cell index
If align_to_vehicle_axes the cells are in the vehicle’s frame, otherwise in the world frame.

Parameters
• position – a world position
• relative – whether the position is already relative to the observer’s position

Returns
the pair (i,j) of the cell index

fill_road_layer_by_lanes(layer_index: int, lane_perception_distance: float = 100) → None
A layer to encode the onroad (1) / offroad (0) information
Here, we iterate over lanes and regularly placed waypoints on these lanes to fill the corresponding cells.
This approach is faster if the grid is large and the road network is small.

Parameters
• layer_index – index of the layer in the grid
• lane_perception_distance – lanes are rendered +/- this distance from vehicle location

fill_road_layer_by_cell(layer_index) → None
A layer to encode the onroad (1) / offroad (0) information
In this implementation, we iterate the grid cells and check whether the corresponding world position at the center of the cell is onroad/offroad. This approach is faster if the grid is small and the road network large.

class highway_env.envs.common.observation.KinematicsGoalObservation(env: AbstractEnv, scales: List[float], **kwargs: dict)

space() → Space
Get the observation space.

observe() → Dict[str, ndarray]
Get an observation of the environment state.


Specific to exit_env, observe the distance to the next exit lane as part of a KinematicObservation.
observe() → ndarray
Get an observation of the environment state.

### 2.3.2 Actions

Similarly to observations, several types of actions can be used in every environment. They are defined in the action module. Each environment comes with a default action type, which can be changed or customised using environment configurations. For instance,

```python
import gym
import highway_env

env = gym.make('highway-v0')
env.configure({
  "action": {
    "type": "ContinuousAction"
  }
})
env.reset()
```

#### Continuous Actions

The ContinuousAction type allows the agent to directly set the low-level controls of the vehicle kinematics, namely the throttle $a$ and steering angle $\delta$.

**Note:** The control of throttle and steering can be enabled or disabled through the longitudinal and lateral configurations, respectively. Thus, the action space can be either 1D or 2D.

#### Discrete Actions

The DiscreteAction is a uniform quantization of the ContinuousAction above.

The actions_per_axis parameter allows to set the quantization step. Similarly to continuous actions, the longitudinal and lateral axis can be enabled or disabled separately.

#### Discrete Meta-Actions

The DiscreteMetaAction type adds a layer of speed and steering controllers on top of the continuous low-level control, so that the ego-vehicle can automatically follow the road at a desired velocity. Then, the available meta-actions consist in changing the target lane and speed that are used as setpoints for the low-level controllers.

The full corresponding action space is defined in ACTIONS_ALL:

```python
ACTIONS_ALL = {
    0: 'LANE_LEFT',
    1: 'IDLE',
    2: 'LANE_RIGHT',
    3: 'FASTER',
    4: 'SLOWER'
}
```
Some of these actions might not be always available (lane changes at the edges of the roads, or accelerating/decelerating beyond the maximum/minimum velocity), and the list of available actions can be accessed with `get_available_actions()` method. Taking an unavailable action is equivalent to taking the `IDLE` action.

Similarly to continuous actions, the longitudinal (speed changes) and lateral (lane changes) actions can be disabled separately through the `longitudinal` and `lateral` parameters. For instance, in the default configuration of the `intersection` environment, only the speed is controlled by the agent, while the lateral control of the vehicle is automatically performed by a steering controller to track a desired lane.

**Manual control**

The environments can be used as a simulation:

```python
env = gym.make("highway-v0")
env.configure({
    "manual_control": True
})
env.reset()
done = False
while not done:
    env.step(env.action_space.sample()) # with manual control, these actions are ignored
```

The ego-vehicle is controlled by directional arrows keys, as defined in `EventHandler` API class.

**API**

```python
class highway_env.envs.common.action.ActionType(env: AbstractEnv, **kwargs)
    A type of action specifies its definition space, and how actions are executed in the environment

    space() -> Space
        The action space.

    property vehicle_class: Callable
        The class of a vehicle able to execute the action.
        Must return a subclass of `highway_env.vehicle.kinematics.Vehicle`.

    act(action: Union[int, ndarray]) -> None
        Execute the action on the ego-vehicle.
        Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified `highway_env.envs.common.action.ActionType.vehicle_class`. Must some preprocessing can be applied to the action based on the ActionType configurations.

    Parameters
    action -- the action to execute

    get_available_actions()
        For discrete action space, return the list of available actions.

    property controlled_vehicle
        The vehicle acted upon.
        If not set, the first controlled vehicle is used by default.
```
class highway_env.envs.common.action.ContinuousAction(env: AbstractEnv, acceleration_range: Optional[Tuple[float, float]] = None, steering_range: Optional[Tuple[float, float]] = None, speed_range: Optional[Tuple[float, float]] = None, longitudinal: bool = True, lateral: bool = True, dynamical: bool = False, clip: bool = True, **kwargs)

An continuous action space for throttle and/or steering angle.

If both throttle and steering are enabled, they are set in this order: [throttle, steering]

The space intervals are always [-1, 1], but are mapped to throttle/steering intervals through configurations.

ACCELERATION_RANGE = (-5, 5.0)

Acceleration range: [-x, x], in m/s².

STEERING_RANGE = (-0.7853981633974483, 0.7853981633974483)

Steering angle range: [-x, x], in rad.

space() → Box

The action space.

property vehicle_class: Callable

The class of a vehicle able to execute the action.

Must return a subclass of highway_env.vehicle.kinematics.Vehicle.

act(action: ndarray) → None

Execute the action on the ego-vehicle.

Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified highway_env.envs.common.action.ActionType.vehicle_class. Must some pre-processing can be applied to the action based on the ActionType configurations.

Parameters

action – the action to execute

class highway_env.envs.common.action.DiscreteAction(env: AbstractEnv, acceleration_range: Optional[Tuple[float, float]] = None, steering_range: Optional[Tuple[float, float]] = None, speed_range: Optional[Tuple[float, float]] = None, longitudinal: bool = True, lateral: bool = True, dynamical: bool = True, actions_per_axis: int = 3, **kwargs)

space() → Discrete

The action space.

act(action: int) → None

Execute the action on the ego-vehicle.

Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified highway_env.envs.common.action.ActionType.vehicle_class. Must some pre-processing can be applied to the action based on the ActionType configurations.

Parameters

action – the action to execute

class highway_env.envs.common.action.DiscreteMetaAction(env: AbstractEnv, longitudinal: bool = True, lateral: bool = True, target_speeds: Optional[Union[ndarray, Sequence[float]]] = None, **kwargs)

space() → None

The action space.

act(action: int) → None

Execute the action on the ego-vehicle.

Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified highway_env.envs.common.action.ActionType.vehicle_class. Must some pre-processing can be applied to the action based on the ActionType configurations.

Parameters

action – the action to execute
An discrete action space of meta-actions: lane changes, and cruise control set-point.

```
ACTIONS_ALL = {0: 'LANE_LEFT', 1: 'IDLE', 2: 'LANE_RIGHT', 3: 'FASTER', 4: 'SLOWER'}
```

A mapping of action indexes to labels.

```
ACTIONS_LONGI = {0: 'SLOWER', 1: 'IDLE', 2: 'FASTER'}
```

A mapping of longitudinal action indexes to labels.

```
ACTIONS_LAT = {0: 'LANE_LEFT', 1: 'IDLE', 2: 'LANE_RIGHT'}
```

A mapping of lateral action indexes to labels.

```
space() \rightarrow Space
The action space.
```

```
property vehicle_class: Callable
The class of a vehicle able to execute the action.
Must return a subclass of highway_env.vehicle.kinematics.Vehicle.
```

```
act(action: int) \rightarrow None
Execute the action on the ego-vehicle.
Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified highway_env.envs.common.action.ActionType.vehicle_class. Must some pre-processing can be applied to the action based on the ActionType configurations.

Parameters
action – the action to execute
```

```
get_available_actions() \rightarrow List[int]
Get the list of currently available actions.
Lane changes are not available on the boundary of the road, and speed changes are not available at maximal or minimal speed.

Returns
the list of available actions
```

```
class highway_env.envs.common.action.MultiAgentAction(env: AbstractEnv, action_config: dict, **kwargs)
```

```
space() \rightarrow Space
The action space.
```

```
property vehicle_class: Callable
The class of a vehicle able to execute the action.
Must return a subclass of highway_env.vehicle.kinematics.Vehicle.
```

```
act(action: Union[int, ndarray]) \rightarrow None
Execute the action on the ego-vehicle.
Most of the action mechanics are actually implemented in vehicle.act(action), where vehicle is an instance of the specified highway_env.envs.common.action.ActionType.vehicle_class. Must some pre-processing can be applied to the action based on the ActionType configurations.

Parameters
action – the action to execute
```
get_available_actions()

For discrete action space, return the list of available actions.

2.3.3 Dynamics

The dynamics of every environment describes how vehicles move and behave through time. There are two important sections that affect these dynamics: the description of the roads, and the vehicle physics and behavioral models.

Roads

A *Road* is composed of a *[RoadNetwork](#)* and a list of *Vehicle*.

Lane

The geometry of lanes are described by *AbstractLane* objects, as a parametrized center line curve, providing a local coordinate system.

Conversions between the (longi, lat) coordinates in the Frenet frame and the global x, y coordinates are ensured by the `position()` and `local_coordinates()` methods.

The main implementations are:

- *StraightLane*
- *SineLane*
- *CircularLane*

API

class highway_env.road.lane.AbstractLane

A lane on the road, described by its central curve.

*metaclass__*

alias of ABCMeta

**abstract position**(*longitudinal: float, lateral: float*) → ndarray

Convert local lane coordinates to a world position.

**Parameters**

- *longitudinal* – longitudinal lane coordinate [m]
- *lateral* – lateral lane coordinate [m]

**Returns**

the corresponding world position [m]

**abstract local_coordinates**(*position: ndarray*) → Tuple[float, float]

Convert a world position to local lane coordinates.

**Parameters**

- *position* – a world position [m]

**Returns**

the (longitudinal, lateral) lane coordinates [m]
**abstract heading_at** (longitudinal: float) → float

Get the lane heading at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane heading [rad]

**abstract width_at** (longitudinal: float) → float

Get the lane width at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane width [m]

**classmethod from_config** (config: dict)

Create lane instance from config

**Parameters**
- **config** – json dict with lane parameters

**abstract to_config** () → dict

Write lane parameters to dict which can be serialized to json

**Returns**

dict of lane parameters

**on_lane** (position: ndarray, longitudinal: Optional[float] = None, lateral: Optional[float] = None, margin: float = 0) → bool

Whether a given world position is on the lane.

**Parameters**
- **position** – a world position [m]
- **longitudinal** – (optional) the corresponding longitudinal lane coordinate, if known [m]
- **lateral** – (optional) the corresponding lateral lane coordinate, if known [m]
- **margin** – (optional) a supplementary margin around the lane width

**Returns**

is the position on the lane?

**is_reachable_from** (position: ndarray) → bool

Whether the lane is reachable from a given world position

**Parameters**
- **position** – the world position [m]

**Returns**

is the lane reachable?

**distance** (position: ndarray)

Compute the L1 distance [m] from a position to the lane.

**distance_with_heading** (position: ndarray, heading: Optional[float], heading_weight: float = 1.0)

Compute a weighted distance in position and heading to the lane.
local_angle(heading: float, long_offset: float)
Compute non-normalised angle of heading to the lane.

class highway_env.road.lane.LineType
A lane side line type.

class highway_env.road.lane.StraightLane
A lane going in straight line.

def position(longitudinal: float, lateral: float) → ndarray
Convert local lane coordinates to a world position.

Parameters
• longitudinal – longitudinal lane coordinate [m]
• lateral – lateral lane coordinate [m]

Returns
the corresponding world position [m]

def heading_at(longitudinal: float) → float
Get the lane heading at a given longitudinal lane coordinate.

Parameters
longitudinal – longitudinal lane coordinate [m]

Returns
the lane heading [rad]

def width_at(longitudinal: float) → float
Get the lane width at a given longitudinal lane coordinate.

Parameters
longitudinal – longitudinal lane coordinate [m]

Returns
the lane width [m]

def local_coordinates(position: ndarray) → Tuple[float, float]
Convert a world position to local lane coordinates.

Parameters
position – a world position [m]

Returns
the (longitudinal, lateral) lane coordinates [m]

classmethod from_config(config: dict)
Create lane instance from config

Parameters
config – json dict with lane parameters

def to_config() → dict
Write lane parameters to dict which can be serialized to json

Returns
dict of lane parameters
class highway_env.road.lane.SineLane(
    start: Union[ndarray, Sequence[float]],
    end: Union[ndarray, Sequence[float]],
    amplitude: float, pulsation: float, phase: float,
    width: float = 4,
    line_types: Optional[List[LineType]] = None,
    forbidden: bool = False,
    speed_limit: float = 20,
    priority: int = 0)

A sinusoidal lane.

position(longitudinal: float, lateral: float) → ndarray
    Convert local lane coordinates to a world position.

    Parameters
    • longitudinal – longitudinal lane coordinate [m]
    • lateral – lateral lane coordinate [m]

    Returns
    the corresponding world position [m]

heading_at(longitudinal: float) → float
    Get the lane heading at a given longitudinal lane coordinate.

    Parameters
    longitudinal – longitudinal lane coordinate [m]

    Returns
    the lane heading [rad]

local_coordinates(position: ndarray) → Tuple[float, float]
    Convert a world position to local lane coordinates.

    Parameters
    position – a world position [m]

    Returns
    the (longitudinal, lateral) lane coordinates [m]

classmethod from_config(config: dict)
    Create lane instance from config

    Parameters
    config – json dict with lane parameters

to_config() → dict
    Write lane parameters to dict which can be serialized to json

    Returns
    dict of lane parameters

class highway_env.road.lane.CircularLane(
    center: Union[ndarray, Sequence[float]],
    radius: float,
    start_phase: float, end_phase: float, clockwise: bool = True,
    width: float = 4,
    line_types: Optional[List[LineType]] = None,
    forbidden: bool = False,
    speed_limit: float = 20,
    priority: int = 0)

A lane going in circle arc.

position(longitudinal: float, lateral: float) → ndarray
    Convert local lane coordinates to a world position.

    Parameters
    • longitudinal – longitudinal lane coordinate [m]
• **lateral** – lateral lane coordinate [m]

**Returns**
the corresponding world position [m]

**heading_at** *(longitudinal: float) → float*
Get the lane heading at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane heading [rad]

**width_at** *(longitudinal: float) → float*
Get the lane width at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane width [m]

**local_coordinates** *(position: ndarray) → Tuple[float, float]*
Convert a world position to local lane coordinates.

**Parameters**
- **position** – a world position [m]

**Returns**
the (longitudinal, lateral) lane coordinates [m]

**classmethod from_config** *(config: dict)*
Create lane instance from config

**Parameters**
- **config** – json dict with lane parameters

**to_config** () → dict
Write lane parameters to dict which can be serialized to json

**Returns**
dict of lane parameters

**class** highway_env.road.lane.PolyLaneFixedWidth *(lane_points: List[Tuple[float, float]], width: float = 4, line_types: Optional[Tuple[LineType, LineType]] = None, forbidden: bool = False, speed_limit: float = 20, priority: int = 0)*
A fixed-width lane defined by a set of points and approximated with a 2D Hermite polynomial.

**position** *(longitudinal: float, lateral: float) → ndarray*
Convert local lane coordinates to a world position.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]
- **lateral** – lateral lane coordinate [m]

**Returns**
the corresponding world position [m]
`local_coordinates(position: ndarray) → Tuple[float, float]`
Convert a world position to local lane coordinates.

**Parameters**
- **position** – a world position [m]

**Returns**
the (longitudinal, lateral) lane coordinates [m]

`heading_at(longitudinal: float) → float`
Get the lane heading at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane heading [rad]

`width_at(longitudinal: float) → float`
Get the lane width at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane width [m]

```python
classmethod from_config(config: dict)
Create lane instance from config

**Parameters**
- **config** – json dict with lane parameters

to_config() → dict
Write lane parameters to dict which can be serialized to json

**Returns**
dict of lane parameters
```

**class** `highway_env.road.lane.PolyLane(lane_points: List[Tuple[float, float]], left_boundary_points: List[Tuple[float, float]], right_boundary_points: List[Tuple[float, float]], line_types: Optional[Tuple[LineType, LineType]] = None, forbidden: bool = False, speed_limit: float = 20, priority: int = 0)`
A lane defined by a set of points and approximated with a 2D Hermite polynomial.

`width_at(longitudinal: float) → float`
Get the lane width at a given longitudinal lane coordinate.

**Parameters**
- **longitudinal** – longitudinal lane coordinate [m]

**Returns**
the lane width [m]

```python
to_config() → dict
Write lane parameters to dict which can be serialized to json

**Returns**
dict of lane parameters
```
**Road**

A **Road** is composed of a **RoadNetwork** and a list of **Vehicle**.

The **RoadNetwork** describes the topology of the road infrastructure as a graph, where edges represent lanes and nodes represent intersections. It contains a graph dictionary which stores the **AbstractLane** geometries by their **LaneIndex**. A **LaneIndex** is a tuple containing:

- a string identifier of a starting position
- a string identifier of an ending position
- an integer giving the index of the described lane, in the (unique) road from the starting to the ending position

For instance, the geometry of the second lane in the road going from the "lab" to the "pub" can be obtained by:

```python
lane = road.road_network.graph["lab"["pub"]][1]
```

The actual positions of the lab and the pub are defined in the ```lane``` geometry object.

**API**

```python
```

A road is a set of lanes, and a set of vehicles driving on these lanes.

**act()** → None

Decide the actions of each entity on the road.

**step(dt: float) → None**

Step the dynamics of each entity on the road.

**Parameters**

- **dt** – timestep [s]


Find the preceding and following vehicles of a given vehicle.

**Parameters**

- **vehicle** – the vehicle whose neighbours must be found
- **lane_index** – the lane on which to look for preceding and following vehicles. It doesn’t have to be the current vehicle lane but can also be another lane, in which case the vehicle is projected on it considering its local coordinates in the lane.

**Returns**

its preceding vehicle, its following vehicle
Road regulation

A *RegulatedRoad* is a *Road* in which the behavior of vehicles take or give the right of way at an intersection based on the *priority lane* attribute.

On such a road, some rules are enforced:

- most of the time, vehicles behave as usual;
- however, they try to predict collisions with other vehicles through the *is_conflict_possible()* method;
- when it is the case, right of way is arbitrated through the *respect_priorities()* method, and the yielding vehicle target velocity is set to 0 until the conflict is resolved.

API

class highway_env.road.regulation.RegulatedRoad(
    network: Optional[RoadNetwork] = None,
    vehicles: Optional[List[Vehicle]] = None,
    obstacles: Optional[List[Obstacle]] = None,
    np_random: Optional[RandomState] = None,
    record_history: bool = False)

    step(dt: float) → None
    Step the dynamics of each entity on the road.

    Parameters
    dt – timestep [s]

    enforce_road_rules() → None
    Find conflicts and resolve them by assigning yielding vehicles and stopping them.

    static respect_priorities(v1: Vehicle, v2: Vehicle) → Vehicle
    Resolve a conflict between two vehicles by determining who should yield

    Parameters
    • v1 – first vehicle
    • v2 – second vehicle

    Returns
    the yielding vehicle

Vehicles

Kinematics

The vehicles kinematics are represented in the *Vehicle* class by the *Kinematic Bicycle Model* [PAltcheDAndreaN17].

\[\begin{align*}
    \dot{x} &= v \cos(\psi + \beta) \\
    \dot{y} &= v \sin(\psi + \beta) \\
    \dot{v} &= a \\
    \dot{\psi} &= \frac{v}{l} \sin \beta \\
    \beta &= \tan^{-1}(1/2 \tan \delta),
\end{align*}\]

where
• \((x, y)\) is the vehicle position;
• \(v\) its forward speed;
• \(\psi\) its heading;
• \(a\) is the acceleration command;
• \(\beta\) is the slip angle at the center of gravity;
• \(\delta\) is the front wheel angle used as a steering command.

These calculations appear in the `step()` method.

**API**

class highway_env.vehicle.kinematics.Vehicle(road: Road, position: Union[ndarray, Sequence[float]], heading: float = 0, speed: float = 0, prediction_type: str = 'constant_steering')

A moving vehicle on a road, and its kinematics.

The vehicle is represented by a dynamical system: a modified bicycle model. It’s state is propagated depending on its steering and acceleration actions.

LENGTH: float = 5.0
Vehicle length [m]

WIDTH: float = 2.0
Vehicle width [m]

DEFAULT_INITIAL_SPEEDS = [23, 25]
Range for random initial speeds [m/s]

MAX_SPEED = 40.0
Maximum reachable speed [m/s]

MIN_SPEED = -40.0
Minimum reachable speed [m/s]

HISTORY_SIZE = 30
Length of the vehicle state history, for trajectory display

classmethod create_random(road: Road, speed: Optional[float] = None, lane_from: Optional[str] = None, lane_to: Optional[str] = None, lane_id: Optional[int] = None, spacing: float = 1) → Vehicle

Create a random vehicle on the road.

The lane and/or speed are chosen randomly, while longitudinal position is chosen behind the last vehicle in the road with density based on the number of lanes.

**Parameters**

• `road` – the road where the vehicle is driving
• `speed` – initial speed in [m/s]. If None, will be chosen randomly
• `lane_from` – start node of the lane to spawn in
• `lane_to` – end node of the lane to spawn in
• `lane_id` – id of the lane to spawn in
• **spacing** – ratio of spacing to the front vehicle, 1 being the default

**Returns**
A vehicle with random position and/or speed

classmethod **create_from**(vehicle: Vehicle) → Vehicle
Create a new vehicle from an existing one.
Only the vehicle dynamics are copied, other properties are default.

**Parameters**
vehicle – a vehicle

**Returns**
a new vehicle at the same dynamical state

**act**(action: Optional[Union[dict, str]] = None) → None
Store an action to be repeated.

**Parameters**
action – the input action

**step**(dt: float) → None
Propagate the vehicle state given its actions.
Integrate a modified bicycle model with a 1st-order response on the steering wheel dynamics. If the vehicle is crashed, the actions are overridden with erratic steering and braking until complete stop. The vehicle’s current lane is updated.

**Parameters**
dt – timestep of integration of the model [s]

**Control**

The **ControlledVehicle** class implements a low-level controller on top of a Vehicle, allowing to track a given target speed and follow a target lane. The controls are computed when calling the **act()** method.

**Longitudinal controller**

The longitudinal controller is a simple proportional controller:

\[ a = K_p(v_r - v), \]

where

- \( a \) is the vehicle acceleration (throttle);
- \( v \) is the vehicle velocity;
- \( v_r \) is the reference velocity;
- \( K_p \) is the controller proportional gain, implemented as KP_A.

It is implemented in the **speed_control()** method.
Lateral controller

The lateral controller is a simple proportional-derivative controller, combined with some non-linearities that invert those of the kinematics model.

Position control

\[ v_{\text{lat}, r} = -K_{p, \text{lat}} \Delta \text{lat}, \]
\[ \Delta \psi_r = \arcsin \left( \frac{v_{\text{lat}, r}}{v} \right), \]

Heading control

\[ \psi_r = \psi_L + \Delta \psi_r, \]
\[ \dot{\psi}_r = K_{p, \psi} (\psi_r - \psi), \]
\[ \delta = \arcsin \left( \frac{1}{2} \frac{1}{\psi} \dot{\psi}_r \right), \]

where

- \( \Delta \text{lat} \) is the lateral position of the vehicle with respect to the lane center-line;
- \( v_{\text{lat}, r} \) is the lateral velocity command;
- \( \Delta \psi_r \) is a heading variation to apply the lateral velocity command;
- \( \psi_L \) is the lane heading (at some lookahead position to anticipate turns);
- \( \psi_r \) is the target heading to follow the lane heading and position;
- \( \dot{\psi}_r \) is the yaw rate command;
- \( \delta \) is the front wheels angle control;
- \( K_{p, \text{lat}} \) and \( K_{p, \psi} \) are the position and heading control gains.

It is implemented in the \texttt{steering\_control()} method.

API

\texttt{class highway\_env.\textbf{vehicle.\textbf{controller}.\textbf{ControlledVehicle}(\texttt{road: Road, position: Union[ndarray, Sequence[float]], heading: float = 0, speed: float = 0, target\_lane\_index: Optional[Tuple[str, str, int]] = None, target\_speed: Optional[float] = None, route: Optional[List[Tuple[str, str, int]]] = None)}}

A vehicle piloted by two low-level controller, allowing high-level actions such as cruise control and lane changes.

- The longitudinal controller is a speed controller;
- The lateral controller is a heading controller cascaded with a lateral position controller.
target_speed:: float
Desired velocity.

classmethod create_from(vehicle: ControlledVehicle) → ControlledVehicle
Create a new vehicle from an existing one.

Parameters
vehicle – a vehicle

Returns
a new vehicle at the same dynamical state

plan_route_to(destination: str) → ControlledVehicle
Plan a route to a destination in the road network

Parameters
destination – a node in the road network

act(action: Optional[Union[dict, str]] = None) → None
Perform a high-level action to change the desired lane or speed.

• If a high-level action is provided, update the target speed and lane;
• then, perform longitudinal and lateral control.

Parameters
action – a high-level action

follow_road() → None
At the end of a lane, automatically switch to a next one.

steering_control(target_lane_index: Tuple[str, str, int]) → float
Steer the vehicle to follow the center of a given lane.

1. Lateral position is controlled by a proportional controller yielding a lateral speed command
2. Lateral speed command is converted to a heading reference
3. Heading is controlled by a proportional controller yielding a heading rate command
4. Heading rate command is converted to a steering angle

Parameters
target_lane_index – index of the lane to follow

Returns
a steering wheel angle command [rad]

speed_control(target_speed: float) → float
Control the speed of the vehicle.
Using a simple proportional controller.

Parameters
target_speed – the desired speed

Returns
an acceleration command [m/s²]
get_routes_at_intersection() → List[List[Tuple[str, str, int]]]
Get the list of routes that can be followed at the next intersection.

set_route_at_intersection(_to: int) → None
Set the road to be followed at the next intersection.
Erase current planned route.

Parameters
[to] – index of the road to follow at next intersection, in the road network

predict_trajectory_constant_speed(times: ndarray) → Tuple[List[ndarray], List[float]]
Predict the future positions of the vehicle along its planned route, under constant speed

Parameters
times – timesteps of prediction

Returns
positions, headings

class highway_env.vehicle.controller.MDPVehicle(road: Road, position: List[float], heading: float = 0, speed: float = 0, target_lane_index: Optional[Tuple[str, str, int]] = None, target_speed: Optional[float] = None, target_speeds: Optional[Union[ndarray, Sequence[float]]] = None, route: Optional[List[Tuple[str, str, int]]] = None)
A controlled vehicle with a specified discrete range of allowed target speeds.

target_speed: float
Desired velocity.

act(action: Optional[Union[dict, str]] = None) → None
Perform a high-level action.

• If the action is a speed change, choose speed from the allowed discrete range.
• Else, forward action to the ControlledVehicle handler.

Parameters
action – a high-level action

index_to_speed(index: int) → float
Convert an index among allowed speeds to its corresponding speed

Parameters
index – the speed index []

Returns
the corresponding speed [m/s]

speed_to_index(speed: float) → int
Find the index of the closest speed allowed to a given speed.
Assumes a uniform list of target speeds to avoid searching for the closest target speed

Parameters
speed – an input speed [m/s]

Returns
the index of the closest speed allowed []
classmethod speed_to_index_default(speed: float) → int
    Find the index of the closest speed allowed to a given speed.
    Assumes a uniform list of target speeds to avoid searching for the closest target speed

    Parameters
    speed – an input speed [m/s]

    Returns
    the index of the closest speed allowed []

predict_trajectory(actions: List, action_duration: float, trajectory_timestep: float, dt: float) → List[ControlledVehicle]
    Predict the future trajectory of the vehicle given a sequence of actions.

    Parameters
    • actions – a sequence of future actions.
    • action_duration – the duration of each action.
    • trajectory_timestep – the duration between each save of the vehicle state.
    • dt – the timestep of the simulation

    Returns
    the sequence of future states

Behavior

Other simulated vehicles follow simple and realistic behaviors that dictate how they accelerate and steer on the road. They are implemented in the IDMVehicle class.

Longitudinal Behavior

The acceleration of the vehicle is given by the Intelligent Driver Model (IDM) from [THH00].

\[
\dot{v} = a \left[ 1 - \left( \frac{v}{v_0} \right)^\delta - \left( \frac{d^*}{d} \right)^2 \right]
\]

\[
d^* = d_0 + T v + \frac{v \Delta v}{2ab}
\]

where \(v\) is the vehicle velocity, \(d\) is the distance to its front vehicle. The dynamics are parametrised by:

• \(v_0\) the desired velocity, as target_velocity
• \(T\) the desired time gap, as TIME_WANTED
• \(d_0\) the jam distance, as DISTANCE_WANTED
• \(a, b\) the maximum acceleration and deceleration, as COMFORT_ACC_MAX and COMFORT_ACC_MIN
• \(\delta\) the velocity exponent, as DELTA

It is implemented in acceleration() method.
Lateral Behavior

The discrete lane change decisions are given by the *Minimizing Overall Braking Induced by Lane change* (MOBIL) model from [KTH07]. According to this model, a vehicle decides to change lane when:

- it is **safe** (do not cut-in):
  \[
  \hat{a}_n \geq -b_{safe};
  \]

- there is an **incentive** (for the ego-vehicle and possibly its followers):
  \[
  \hat{a}_c - a_c + p \left( \hat{a}_n - a_n + \hat{a}_o - a_o \right) \geq \Delta a_{th},
  \]

where

- \(c\) is the center (ego-) vehicle, \(o\) is its old follower before the lane change, and \(n\) is its new follower after the lane change
- \(a, \hat{a}\) are the acceleration of the vehicles before and after the lane change, respectively.
- \(p\) is a politeness coefficient, implemented as POLITENESS
- \(\Delta a_{th}\) the acceleration gain required to trigger a lane change, implemented as LANE_CHANGE_MIN_ACC_GAIN
- \(b_{safe}\) the maximum braking imposed to a vehicle during a cut-in, implemented as LANE_CHANGE_MAX_BRAKING_IMPOSED

It is implemented in the `mobil()` method.

**Note:** In the `LinearVehicle` class, the longitudinal and lateral behaviours are approximated as linear weightings of several features, such as the distance and speed difference to the leading vehicle.

### API

```python
class highway_env.vehicle.behavior.IDMVehicle(road: Road, position: Union[ndarray, Sequence[float]],
                                              heading: float = 0, speed: float = 0, target_lane_index:
                                              Optional[int] = None, target_speed: Optional[float] =
                                              None, route: Optional[List[Tuple[str, str, int]]] = None,
                                              enable_lane_change: bool = True, timer:
                                              Optional[float] = None)
```

A vehicle using both a longitudinal and a lateral decision policies.

- Longitudinal: the IDM model computes an acceleration given the preceding vehicle's distance and speed.
- Lateral: the MOBIL model decides when to change lane by maximizing the acceleration of nearby vehicles.

**ACC_MAX = 6.0**

Maximum acceleration.

**COMFORT_ACC_MAX = 3.0**

Desired maximum acceleration.

**COMFORT_ACC_MIN = -5.0**

Desired maximum deceleration.
**DISTANCE_WANTED = 10.0**
Desired jam distance to the front vehicle.

**TIME_WANTED = 1.5**
Desired time gap to the front vehicle.

**DELTA = 4.0**
Exponent of the velocity term.

**DELTA_RANGE = [3.5, 4.5]**
Range of delta when chosen randomly.

**classmethod create_from** *(vehicle: ControlledVehicle) -> IDMVehicle*
Create a new vehicle from an existing one.

The vehicle dynamics and target dynamics are copied, other properties are default.

**Parameters**
vehicle – a vehicle

**Returns**
a new vehicle at the same dynamical state

**act**(action: Optional[Union[dict, str]] = None)
Execute an action.

For now, no action is supported because the vehicle takes all decisions of acceleration and lane changes on its own, based on the IDM and MOBI model.

**Parameters**
action – the action

**step**(dt: float)
Step the simulation.

Increases a timer used for decision policies, and step the vehicle dynamics.

**Parameters**
dt – timestep

**acceleration** *(ego_vehicle: ControlledVehicle, front_vehicle: Optional[Vehicle] = None, rear_vehicle: Optional[Vehicle] = None) -> float*
Compute an acceleration command with the Intelligent Driver Model.

The acceleration is chosen so as to: - reach a target speed; - maintain a minimum safety distance (and safety time) w.r.t the front vehicle.

**Parameters**
- **ego_vehicle** – the vehicle whose desired acceleration is to be computed. It does not have to be an IDM vehicle, which is why this method is a class method. This allows an IDM vehicle to reason about other vehicles behaviors even though they may not IDMs.
- **front_vehicle** – the vehicle preceding the ego-vehicle
- **rear_vehicle** – the vehicle following the ego-vehicle

**Returns**
the acceleration command for the ego-vehicle [m/s²]
desired_gap(ego_vehicle: Vehicle, front_vehicle: Optional[Vehicle] = None, projected: bool = True) → float

Compute the desired distance between a vehicle and its leading vehicle.

Parameters

• ego_vehicle – the vehicle being controlled

• front_vehicle – its leading vehicle

• projected – project 2D velocities in 1D space

Returns

the desired distance between the two [m]

calculate_lane_policy() → None

Decide when to change lane.

Based on: - frequency; - closeness of the target lane; - MOBIL model.

mobil(lane_index: Tuple[str, str, int]) → bool

MOBIL lane change model: Minimizing Overall Braking Induced by a Lane change

The vehicle should change lane only if: - after changing it (and/or following vehicles) can accelerate more; - it doesn’t impose an unsafe braking on its new following vehicle.

Parameters

lane_index – the candidate lane for the change

Returns

whether the lane change should be performed

calculate_from_stop(acceleration: float) → float

If stopped on the wrong lane, try a reversing maneuver.

Parameters

acceleration – desired acceleration from IDM

Returns

suggested acceleration to recover from being stuck

target_speed: float

Desired velocity.

class highway_env.vehicle.behavior.LinearVehicle(road: Road, position: Union[ndarray, Sequence[float]], heading: float = 0, speed: float = 0, target_lane_index: Optional[int] = None, target_speed: Optional[float] = None, route: Optional[List[Tuple[str, str, int]]] = None, enable_lane_change: bool = True, timer: Optional[float] = None, data: Optional[dict] = None)

A Vehicle whose longitudinal and lateral controllers are linear with respect to parameters.

TIME_WANTED = 2.5

Desired time gap to the front vehicle.
act(action: Optional[Union[dict, str]] = None)

Execute an action.

For now, no action is supported because the vehicle takes all decisions of acceleration and lane changes on its own, based on the IDM and MOBIL models.

Parameters
  action – the action

acceleration(ego_vehicle: ControlledVehicle, front_vehicle: Optional[Vehicle] = None, rear_vehicle: Optional[Vehicle] = None) → float

Compute an acceleration command with a Linear Model.

The acceleration is chosen so as to: - reach a target speed; - reach the speed of the leading (resp following) vehicle, if it is lower (resp higher) than ego’s; - maintain a minimum safety distance w.r.t the leading vehicle.

Parameters
  • ego_vehicle – the vehicle whose desired acceleration is to be computed. It does not have to be an Linear vehicle, which is why this method is a class method. This allows a Linear vehicle to reason about other vehicles behaviors even though they may not Linear.
  • front_vehicle – the vehicle preceding the ego-vehicle
  • rear_vehicle – the vehicle following the ego-vehicle

Returns
the acceleration command for the ego-vehicle [m/s2]

steering_control(target_lane_index: Tuple[str, str, int]) → float

Linear controller with respect to parameters.

Overrides the non-linear controller ControlledVehicle.steering_control()

Parameters
  target_lane_index – index of the lane to follow

Returns
a steering wheel angle command [rad]

steering_features(target_lane_index: Tuple[str, str, int]) → ndarray

A collection of features used to follow a lane

Parameters
  target_lane_index – index of the lane to follow

Returns
a array of features

collect_data()

Store features and outputs for parameter regression.

target_speed: float
Desired velocity.

class highway_env.vehicle.behavior.AggressiveVehicle(road: Road, position: Union[ndarray, Sequence[float]], heading: float = 0, speed: float = 0, target_lane_index: Optional[int] = None, target_speed: Optional[float] = None, route: Optional[List[Tuple[str, str, int]]] = None, enable_lane_change: bool = True, timer: Optional[float] = None, data: Optional[dict] = None)
target_speed: float
Desired velocity.

class highway_env.vehicle.behavior.DefensiveVehicle(road: Road, position: Union[ndarray, Sequence[float]], heading: float = 0, speed: float = 0, target_lane_index: Optional[int] = None, target_speed: Optional[float] = None, route: Optional[List[Tuple[str, str, int]]] = None, enable_lane_change: bool = True, timer: Optional[float] = None, data: Optional[dict] = None)

target_speed: float
Desired velocity.

2.3.4 Rewards

The reward function is defined in the _reward() method, overloaded in every environment.

Note: The choice of an appropriate reward function that yields realistic optimal driving behaviour is a challenging problem, that we do not address in this project. In particular, we do not wish to specify every single aspect of the expected driving behaviour inside the reward function, such as keeping a safe distance to the front vehicle. Instead, we would rather only specify a reward function as simple and straightforward as possible in order to see adequate behaviour emerge from learning. In this perspective, keeping a safe distance is optimal not for being directly rewarded but for robustness against the uncertain behaviour of the leading vehicle, which could brake at any time.

Most environments

We generally focus on two features: a vehicle should

- progress quickly on the road;
- avoid collisions.

Thus, the reward function is often composed of a velocity term and a collision term:

\[
R(s, a) = \alpha \frac{v - v_{\text{min}}}{v_{\text{max}} - v_{\text{min}} - b \cdot \text{collision}}
\]

where \(v, v_{\text{min}}, v_{\text{max}}\) are the current, minimum and maximum speed of the ego-vehicle respectively, and \(\alpha, b\) are two coefficients.

Note: Since the rewards must be bounded, and the optimal policy is invariant by scaling and shifting rewards, we choose to normalize them in the \([0, 1]\) range, by convention. Normalizing rewards has also been observed to be practically beneficial in deep reinforcement learning [MKS+15]. Note that we forbid negative rewards, since they may encourage the agent to prefer terminating an episode early (by causing a collision) rather than risking suffering a negative return if no satisfying trajectory can be found.

In some environments, the weight of the collision penalty can be configured through the collision_penalty parameter.
Goal environments

In the Parking environment, however, the reward function must also specify the desired goal destination. Thus, the velocity term is replaced by a weighted p-norm between the agent state and the goal state.

\[ R(s, a) = -\|s - s_g\|_p^{W_p} - b \text{ collision} \]

where \( s = [x, y, v_x, v_y, \cos \psi, \sin \psi] \), \( s_g = [x_g, y_g, 0, 0, \cos \psi_g, \sin \psi_g] \), and \( \|x\|_{W_p} = (\sum_i |W_i x_i|^p)^{1/p} \). We use a p-norm rather than an Euclidean norm in order to have a narrower spike of rewards at the goal.

2.3.5 Graphics

Environment rendering is done with pygame, which must be installed separately.

A window is created at the first call of env.render(). Its dimensions can be configured:

```python
env = gym.make("roundabout-v0")
env.configure(
    "screen_width": 640,
    "screen_height": 480
)
env.reset()
env.render()
```

World surface

The simulation is rendered in a RoadSurface pygame surface, which defines the location and zoom of the rendered location. By default, the rendered area is always centered on the ego-vehicle. Its initial scale and offset can be set with the “scaling” and “centering_position” configurations, and can also be updated during simulation using the O,L keys and K,M keys, respectively.

Scene graphics

- Roads are rendered in the RoadGraphics class.
- Vehicles are rendered in the VehicleGraphics class.

API

```python
class highway_env.envs.common.graphics.EnvViewer(env: AbstractEnv, config: Optional[dict] = None)
    A viewer to render a highway driving environment.

    set_agent_display(agent_display: Callable) → None
    Set a display callback provided by an agent
    So that they can render their behaviour on a dedicated agent surface, or even on the simulation surface.

      Parameters
        agent_display – a callback provided by the agent to display on surfaces
```
set_agent_action_sequence(\texttt{actions: List[Action]}) \rightarrow \texttt{None}

Set the sequence of actions chosen by the agent, so that it can be displayed

\textbf{Parameters}
\begin{itemize}
  \item \texttt{actions} – list of action, following the \texttt{env}'s action space specification
\end{itemize}

\textbf{handle_events()} \rightarrow \texttt{None}

Handle pygame events by forwarding them to the display and environment vehicle.

display() \rightarrow \texttt{None}

Display the road and vehicles on a pygame window.

\textbf{get_image()} \rightarrow \texttt{ndarray}

The rendered image as a rgb array.

OpenAI gym's channel convention is H x W x C

\textbf{window_position()} \rightarrow \texttt{ndarray}

the world position of the center of the displayed window.

close() \rightarrow \texttt{None}

Close the pygame window.

class highway_env.road.graphics.WorldSurface(\texttt{size: Tuple[int, int], flags: object, surf: Surface})

A pygame Surface implementing a local coordinate system so that we can move and zoom in the displayed area.

\textbf{pix}(\texttt{length: float}) \rightarrow \texttt{int}

Convert a distance [m] to pixels [px].

\textbf{Parameters}
\begin{itemize}
  \item \texttt{length} – the input distance [m]
\end{itemize}

\textbf{Returns}
the corresponding size [px]

\textbf{pos2pix}(\texttt{x: float, y: float}) \rightarrow \texttt{Tuple[int, int]}

Convert two world coordinates [m] into a position in the surface [px]

\textbf{Parameters}
\begin{itemize}
  \item \texttt{x} – x world coordinate [m]
  \item \texttt{y} – y world coordinate [m]
\end{itemize}

\textbf{Returns}
the coordinates of the corresponding pixel [px]

\textbf{vec2pix}(\texttt{vec: Union[Tuple[float, float], ndarray]}) \rightarrow \texttt{Tuple[int, int]}

Convert a world position [m] into a position in the surface [px].

\textbf{Parameters}
\begin{itemize}
  \item \texttt{vec} – a world position [m]
\end{itemize}

\textbf{Returns}
the coordinates of the corresponding pixel [px]

\textbf{is_visible}(\texttt{vec: Union[Tuple[float, float], ndarray], margin: int = 50}) \rightarrow \texttt{bool}

Is a position visible in the surface? 
\textbf{Parameters}
\begin{itemize}
  \item \texttt{vec} – a position
  \item \texttt{margin} – margins around the frame to test for visibility
\end{itemize}

\textbf{Returns}
whether the position is visible
move_display_window_to(position: Union[Tuple[float, float], ndarray]) → None
Set the origin of the displayed area to center on a given world position.

Parameters
position – a world position [m]

handle_event(event: Event) → None
Handle pygame events for moving and zooming in the displayed area.

Parameters
event – a pygame event

class highway_env.road.graphics.LaneGraphics
A visualization of a lane.

STRIPE_SPACING: float = 4.33
Offset between stripes [m]

STRIPE_LENGTH: float = 3
Length of a stripe [m]

STRIPE_WIDTH: float = 0.3
Width of a stripe [m]

classmethod display(lane: AbstractLane, surface: WorldSurface) → None
Display a lane on a surface.

Parameters
• lane – the lane to be displayed
• surface – the pygame surface

classmethod striped_line(lane: AbstractLane, surface: WorldSurface, stripes_count: int, longitudinal: float, side: int) → None
Draw a striped line on one side of a lane, on a surface.

Parameters
• lane – the lane
• surface – the pygame surface
• stripes_count – the number of stripes to draw
• longitudinal – the longitudinal position of the first stripe [m]
• side – which side of the road to draw [0:left, 1:right]

classmethod continuous_curve(lane: AbstractLane, surface: WorldSurface, stripes_count: int, longitudinal: float, side: int) → None
Draw a striped line on one side of a lane, on a surface.

Parameters
• lane – the lane
• surface – the pygame surface
• stripes_count – the number of stripes to draw
• longitudinal – the longitudinal position of the first stripe [m]
• side – which side of the road to draw [0:left, 1:right]
**classmethod continuous_line**(`lane: AbstractLane, surface: WorldSurface, stripes_count: int, longitudinal: float, side: int`) → None

Draw a continuous line on one side of a lane, on a surface.

**Parameters**

- `lane` – the lane
- `surface` – the pygame surface
- `stripes_count` – the number of stripes that would be drawn if the line was striped
- `longitudinal` – the longitudinal position of the start of the line [m]
- `side` – which side of the road to draw [0:left, 1:right]

**classmethod draw_stripes**(`lane: AbstractLane, surface: WorldSurface, starts: List[float], ends: List[float], lats: List[float]`) → None

Draw a set of stripes along a lane.

**Parameters**

- `lane` – the lane
- `surface` – the surface to draw on
- `starts` – a list of starting longitudinal positions for each stripe [m]
- `ends` – a list of ending longitudinal positions for each stripe [m]
- `lats` – a list of lateral positions for each stripe [m]

**class** `highway_env.road.graphics.RoadGraphics`

A visualization of a road lanes and vehicles.

**static display**(`road: Road, surface: WorldSurface`) → None

Display the road lanes on a surface.

**Parameters**

- `road` – the road to be displayed
- `surface` – the pygame surface

**static display_traffic**(`road: Road, surface: WorldSurface, simulation_frequency: int = 15, offscreen: bool = False`) → None

Display the road vehicles on a surface.

**Parameters**

- `road` – the road to be displayed
- `surface` – the pygame surface
- `simulation_frequency` – simulation frequency
- `offscreen` – render without displaying on a screen

**static display_road_objects**(`road: Road, surface: WorldSurface, offscreen: bool = False`) → None

Display the road objects on a surface.

**Parameters**

- `road` – the road to be displayed
- `surface` – the pygame surface
highway-env Documentation

• `offscreen` – whether the rendering should be done offscreen or not

class highway_env.road.graphics.RoadObjectGraphics

A visualization of objects on the road.

classmethod display(object_: RoadObject, surface: WorldSurface, transparent: bool = False, offscreen: bool = False)

Display a road objects on a pygame surface.

The objects is represented as a colored rotated rectangle

Parameters

• `object` – the vehicle to be drawn
• `surface` – the surface to draw the object on
• `transparent` – whether the object should be drawn slightly transparent
• `offscreen` – whether the rendering should be done offscreen or not

static blit_rotate(surf: Surface, image: Surface, pos: Union[ndarray, Sequence[float]], angle: float, origin_pos: Optional[Union[ndarray, Sequence[float]]] = None, show_rect: bool = False) → None

Many thanks to https://stackoverflow.com/a/54714144.

2.3.6 The Multi-Agent setting

Most environments can be configured to a multi-agent version. Here is how:

Increase the number of controlled vehicles

To that end, update the environment configuration to increase `controlled_vehicles`

```python
import gym
import highway_env

env = gym.make('highway-v0')
env.seed(0)
env.configure({'controlled_vehicles': 2})  # Two controlled vehicles
env.configure({'vehicles_count': 1})  # A single other vehicle, for the sake of visualisation
env.reset()

from matplotlib import pyplot as plt
%matplotlib inline
plt.imshow(env.render(mode="rgb_array"))
plt.title("Controlled vehicles are in green")
plt.show()
```

2.3. User Guide
Change the action space

Right now, since the action space has not been changed, only the first vehicle is controlled by `env.step(action)`. In order for the environment to accept a tuple of actions, its action type must be set to `MultiAgentAction`. The type of actions contained in the tuple must be described by a standard `action configuration` in the `action_config` field.

```python
env.configure({
    "action": {
        "type": "MultiAgentAction",
        "action_config": {
            "type": "DiscreteMetaAction",
        }
    }
})
env.reset()

_, (ax1, ax2) = plt.subplots(nrows=2)
ax1.imshow(env.render(mode="rgb_array"))
ax1.set_title("Initial state")

# Make the first vehicle change to the left lane, and the second one to the right
action_1, action_2 = 0, 2  # See highway_env.envs.common.action.DiscreteMetaAction.
env.step((action_1, action_2))

ax2.imshow(env.render(mode="rgb_array"))
ax2.set_title("After sending actions to each vehicle")
plt.show()
```
**Change the observation space**

In order to actually decide what \texttt{action}_1 and \texttt{action}_2 should be, both vehicles must generate their own observations. As before, since the observation space has not been changed no far, the observation only includes that of the first vehicle.

In order for the environment to return a tuple of observations – one for each agent –, its observation type must be set to \texttt{MultiAgentObservation} The type of observations contained in the tuple must be described by a standard \texttt{observation configuration} in the \texttt{observation_config} field.

```python
env.configure({
    "observation": {
        "type": "MultiAgentObservation",
        "observation_config": {
            "type": "Kinematics",
        }
    }
})
obs = env.reset()
```

```python
import pprint
pprint.pprint(obs)
```

```python
(array([[ 1., 0.90797305, 0.5 , 0.3125 , 0. ],
        [ 1., 0.10906096, -0.5 , -0.04341291, 0. ],
        [ 1., 0.33000726, -0.5 , 0. , 0. ],
        [ 0. , 0. , 0. , 0. , 0. ]],
       dtype=float32),
array([[1. , 0.90797305, 0.5 , 0.3125 , 0. ],
        [0. , 0. , 0. , 0.3125 , 0. ],
        [0. , 0. , 0. , 0. , 0. ]]),
(continues on next page)
```
Wrapping it up

Here is a pseudo-code example of how a centralized multi-agent policy could be trained:

```python
# Multi-agent environment configuration
env.configure({
    "controlled_vehicles": 2,
    "observation": {
        "type": "MultiAgentObservation",
        "observation_config": {
            "type": "Kinematics",
        }
    },
    "action": {
        "type": "MultiAgentAction",
        "action_config": {
            "type": "DiscreteMetaAction",
        }
    }
})

# Dummy RL algorithm
class Model:
    """ Dummy code for an RL algorithm, which predicts an action from an observation,
    and update its model from observed transitions."""
    def predict(self, obs):
        return 0
    def update(self, obs, action, next_obs, reward, info, done):
        pass
model = Model()

# A training episode
obs = env.reset()
done = False
while not done:
    # Dispatch the observations to the model to get the tuple of actions
    action = tuple(model.predict(obs_i) for obs_i in obs)
    # Execute the actions
    next_obs, reward, info, done = env.step(action)
    # Update the model with the transitions observed by each agent
    for obs_i, action_i, next_obs_i in zip(obs, action, next_obs):
        model.update(obs_i, action_i, next_obs_i, reward, info, done)
    obs = next_obs
```

For example, this is supported by eleurent/rl-agents’s DQN implementation, and can be run with
cd <path/to/rl-agents/scripts>
python experiments.py evaluate configs/IntersectionEnv/env_multi_agent.json \
    configs/IntersectionEnv/agents/DQNAgent/ego_attention_2h.
--json \
    --train --episodes=3000

Fig. 3: Video of a multi-agent episode with the trained policy.

### 2.3.7 Make your own environment

Here are the steps required to create a new environment.

---

**Note:** Pull requests are welcome!

---

**Set up files**

1. Create a new your_env.py file in highway_env/envs/
2. Define a class YourEnv, that must inherit from AbstractEnv

This class provides several useful functions:

- A `default_config()` method, that provides a default configuration dictionary that can be overloaded.
- A `define_spaces()` method, that gives access to a choice of observation and action types, set from the environment configuration
- A `step()` method, which executes the desired actions (at policy frequency) and simulate the environment (at simulation frequency)
- A `render()` method, which renders the environment.

**Create the scene**

The first step is to create a RoadNetwork that describes the geometry and topology of roads and lanes in the scene. This should be achieved in a YourEnv._make_road() method, called from YourEnv.reset() to set the self.road field.

See Roads for reference, and existing environments as examples.

**Create the vehicles**

The second step is to populate your road network with vehicles. This should be achieved in a YourEnv._make_road() method, called from YourEnv.reset() to set the self.road.vehicles list of Vehicle.

First, define the controlled ego-vehicle by setting self.vehicle. The class of controlled vehicle depends on the choice of action type, and can be accessed as self.action_type.vehicle_class. Other vehicles can be created more freely, and added to the self.road.vehicles list.

See vehicle behaviors for reference, and existing environments as examples.
Make the environment configurable

To make a part of your environment configurable, overload the `default_config()` method to define new `{"config_key": value}` pairs with default values. These configurations then be accessed in your environment implementation with `self.config["config_key"]`, and once the environment is created, it can be configured with `env.configure({"config_key": other_value})` followed by `env.reset()`.

Register the environment

In `highway_env/envs/your_env.py`, add the following line:

```python
register(
    id='your-env-v0',
    entry_point='highway_env.envs:YourEnv',
)
```

and import it from `highway_env/envs/__init__.py`:

```python
from highway_env.envs.your_env import *
```

Profit

That's it! You should now be able to run the environment:

```python
import gym
import highway_env

env = gym.make('your-env-v0')
obs = env.reset()
obs, reward, done, info = env.step(env.action_space.sample())
env.render()
```

API

```python
class highway_env.envs.common.abstract.AbstractEnv(config: Optional[dict] = None)
   A generic environment for various tasks involving a vehicle driving on a road.
   The environment contains a road populated with vehicles, and a controlled ego-vehicle that can change lane and speed. The action space is fixed, but the observation space and reward function must be defined in the environment implementations.

   PERCEPTION_DISTANCE = 200.0
   The maximum distance of any vehicle present in the observation [m]

   property vehicle: Vehicle
      First (default) controlled vehicle.

classmethod default_config() -> dict
   Default environment configuration.
   Can be overloaded in environment implementations, or by calling configure().:return: a configuration dict
```
**seed**(seed: Optional[int] = None) → List[int]

**Deprecated**

function that sets the seed for the environment’s random number generator(s).

Use `env.reset(seed=seed)` as the new API for setting the seed of the environment.

**Note:**

Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren’t accidental correlations between multiple generators.

**Args:**

- **seed** (Optional int): The seed value for the random number generator

**Returns:**

- **seeds** (List[int]): Returns the list of seeds used in this environment’s random number generators. The first value in the list should be the “main” seed, or the value which a reproducer should pass to ‘seed’. Often, the main seed equals the provided ‘seed’, but this won’t be true if `seed=None`, for example.

**define_spaces** () → None

Set the types and spaces of observation and action from config.

**_reward**(action: Union[int, ndarray]) → float

Return the reward associated with performing a given action and ending up in the current state.

**Parameters**

- **action** – the last action performed

**Returns**

- the reward

**_is_terminal** () → bool

Check whether the current state is a terminal state

:returns: is the state terminal

**_info**(obs: ndarray, action: Union[int, ndarray]) → dict

Return a dictionary of additional information

**Parameters**

- **obs** – current observation
- **action** – current action

**Returns**

- info dict

**_cost**(action: Union[int, ndarray]) → float

A constraint metric, for budgeted MDP.

If a constraint is defined, it must be used with an alternate reward that doesn’t contain it as a penalty.

**Parameters**

- **action** – the last action performed

**Returns**

- the constraint signal, the alternate (constraint-free) reward

**reset** () → ndarray

Reset the environment to it’s initial configuration

**Returns**

- the observation of the reset state
`_reset()` → None
Reset the scene: roads and vehicles.
This method must be overloaded by the environments.

`step(action: Union[int, ndarray])` → Tuple[ndarray, float, bool, dict]
Perform an action and step the environment dynamics.

The action is executed by the ego-vehicle, and all other vehicles on the road performs their default behaviour for several simulation timesteps until the next decision making step.

**Parameters**
- `action` – the action performed by the ego-vehicle

**Returns**
- a tuple (observation, reward, terminal, info)

`_simulate()` → None
Perform several steps of simulation with constant action.

`render(mode: str = 'human')` → Optional[ndarray]
Render the environment.

Create a viewer if none exists, and use it to render an image. :param mode: the rendering mode

`close()` → None
Close the environment.
Will close the environment viewer if it exists.

`_automatic_rendering()` → None
Automatically render the intermediate frames while an action is still ongoing.

This allows to render the whole video and not only single steps corresponding to agent decision-making.
If a RecordVideo wrapper has been set, use it to capture intermediate frames.

`simplify()` → AbstractEnv
Return a simplified copy of the environment where distant vehicles have been removed from the road.
This is meant to lower the policy computational load while preserving the optimal actions set.

**Returns**
- a simplified environment state

`change_vehicles(vehicle_class_path: str)` → AbstractEnv
Change the type of all vehicles on the road

**Parameters**
- `vehicle_class_path` – The path of the class of behavior for other vehicles Example: “highway_env.vehicle.behavior.IDMVehicle”

**Returns**
- a new environment with modified behavior model for other vehicles

class highway_env.envs.common.abstract.MultiAgentWrapper(env: Env)

`step(action)`
Steps through the environment with action.
2.4 Frequently Asked Questions

This is a list of Frequently Asked Questions about highway-env. Feel free to suggest new entries!

I try to train an agent using the Kinematics Observation and an MLP model, but the resulting policy is not optimal. Why?

I also tend to get reasonable but sub-optimal policies using this observation-model pair. In [LM19], we argued that a possible reason is that the MLP output depends on the order of vehicles in the observation. Indeed, if the agent revisits a given scene but observes vehicles described in a different order, it will see it as a novel state and will not be able to reuse past information. Thus, the agent struggles to make use of its observation.

This can be addressed in two ways:

- Change the model, to use a permutation-invariant architecture which will not be sensitive to the vehicles order, such as e.g. [QSMG17] or [LM19].

  This example is implemented here (DQN) or here (SB3’s PPO).

- Change the observation. For example, the Grayscale Image does not depend on an ordering. In this case, a CNN model is more suitable than an MLP model.

  This example is implemented here (SB3’s DQN).

My videos are too fast / have a low framerate.

This is because in openai/gym, a single video frame is generated at each call of `env.step(action)`. However, in highway-env, the policy typically runs at a low-level frequency (e.g. 1 Hz) so that a long action (e.g. change lane) actually corresponds to several (typically, 15) simulation frames. In order to also render these intermediate simulation frames, the following should be done:

```python
import gym
import highway_env

# Wrap the env by a RecordVideo wrapper
env = gym.make("highway-v0")
env = RecordVideo(env, video_folder="run",
                  episode_trigger=lambda e: True)  # record all episodes

# Provide the video recorder to the wrapped environment
# so it can send it intermediate simulation frames.
env.unwrapped.set_record_video_wrapper(env)

# Record a video as usual
obs = env.reset()
done = False:
    while not done:
        action = env.action_space.sample()
        obs, reward, done, info = env.step(action)
        env.render()
env.close()
```
2.5 Bibliography


PYTHON MODULE INDEX

h
highway_env.envs.common.abstract, 58
highway_env.envs.common.action, 27
highway_env.envs.common.graphics, 49
highway_env.envs.common.observation, 23
highway_env.road.graphics, 50
highway_env.road.lane, 30
highway_env.road.regulation, 37
highway_env.road.road, 36
highway_env.vehicle.behavior, 44
highway_env.vehicle.controller, 40
highway_env.vehicle.graphics, 53
highway_env.vehicle.kinematics, 38
Symbols

_automatic_rendering() (highway_env.envs.common.abstract.AbstractEnv method), 60
_cost() (highway_env.envs.common.abstract.AbstractEnv method), 59
_info() (highway_env.envs.common.abstract.AbstractEnv method), 59
_is_terminal() (highway_env.envs.common.abstract.AbstractEnv method), 59
_reset() (highway_env.envs.common.abstract.AbstractEnv method), 59
_reward() (highway_env.envs.common.abstract.AbstractEnv method), 59
_simulate() (highway_env.envs.common.abstract.AbstractEnv method), 60

A

AbstractEnv (class in highway_env.envs.common.abstract), 58
AbstractLane (class in highway_env.road.lane), 30
ACC_MAX (highway_env.vehicle.behavior.IDMVehicle attribute), 44
acceleration() (highway_env.vehicle.behavior.IDMVehicle method), 45
acceleration() (highway_env.vehicle.behavior.LinearVehicle method), 47
ACCELERATION_RANGE (highway_env.envs.common.action.ContinuousAction attribute), 28
act() (highway_env.envs.common.action.ActionType method), 27
act() (highway_env.envs.common.action.ContinuousAction method), 28
act() (highway_env.envs.common.action.DiscreteAction method), 28
act() (highway_env.envs.common.action.DiscreteMetaAction method), 29
act() (highway_env.envs.common.action.MultiAgentAction method), 29
act() (highway_env.road.road.Road method), 36
act() (highway_env.vehicle.behavior.IDMVehicle method), 45
act() (highway_env.vehicle.behavior.LinearVehicle method), 46
act() (highway_env.vehicle.controller.ControlledVehicle method), 41
act() (highway_env.vehicle.controller.MDPVehicle method), 42
act() (highway_env.vehicle.kinematics.Vehicle method), 39
ACTIONS_ALL (highway_env.envs.common.action.DiscreteMetaAction attribute), 29
ACTIONS_LAT (highway_env.envs.common.action.DiscreteMetaAction attribute), 29
ACTIONS_LONGI (highway_env.envs.common.action.DiscreteMetaAction attribute), 29
ActionType (class in highway_env.envs.common.action), 27
AggressiveVehicle (class in highway_env.vehicle.behavior), 47

B

blit_rotate() (highway_env.road.graphics.RoadObjectGraphics static method), 53

C

change_lane_policy() (highway_env.vehicle.behavior.IDMVehicle method), 46
change_vehicles() (highway_env.envs.common.abstract.AbstractEnv method), 60
CircularLane (class in highway_env.road.lane), 33
close() (highway_env.envs.common.abstract.AbstractEnv method), 60
close() (highway_env.envs.common.graphics.EnvViewer method), 50
collect_data() (highway_env.vehicle.behavior.LinearVehicle method), 47
COMFORT_ACC_MAX (highway_env.vehicle.behavior.IDMVehicle attribute), 44
COMFORT_ACC_MIN (highway_env.vehicle.behavior.IDMVehicle attribute), 44
compute_reward() (highway_env.envs.parking_env.ParkingEnv method), 11
continuous_curve() (highway_env.road.graphics.LaneGraphics class method), 51
continuous_line() (highway_env.road.graphics.LaneGraphics class method), 51
ContinuousAction (class in highway_env.envs.common.action), 27
controlled_vehicle (highway_env.envs.common.action.ActionType property), 27
ControlledVehicle (class in highway_env.vehicle.controller), 40
create_from() (highway_env.vehicle.behavior.IDMVehicle method), 45
create_from() (highway_env.vehicle.controller.ControlledVehicle class method), 41
create_from() (highway_env.vehicle.kinematics.Vehicle method), 39
create_random() (highway_env.vehicle.kinematics.Vehicle method), 38

D
default_config() (highway_env.envs.common.abstract.AbstractEnv class method), 58
default_config() (highway_env.envs.racetrack_env.RacetrackEnv class method), 14
default_config() (highway_env.envs.roundabout_env.RoundaboutEnv class method), 9, 10
DEFINITIAL_SPEEDS (highway_env.vehicle.kinematics.Vehicle attribute), 38
DefensiveVehicle (class in highway_env.vehicle.behavior), 48
define_spaces() (highway_env.envs.common.abstract.AbstractEnv method), 59
define_spaces() (highway_env.envs.parking_env.ParkingEnv method), 11
DELTA (highway_env.vehicle.behavior.IDMVehicle attribute), 45
DELTA_RANGE (highway_env.vehicle.behavior.IDMVehicle attribute), 45
desired_gap() (highway_env.vehicle.behavior.IDMVehicle method), 45
DiscreteAction (class in highway_env.envs.common.action), 28
DiscreteMetaAction (class in highway_env.envs.common.action), 28
display() (highway_env.envs.common.graphics.EnvViewer method), 50
display() (highway_env.road.graphics.LaneGraphics class method), 51
display() (highway_env.road.graphics.RoadGraphics static method), 52
display() (highway_env.road.graphics.RoadObjectGraphics class method), 53
display_road_objects() (highway_env.road.graphics.RoadGraphics static method), 52
display_traffic() (highway_env.road.graphics.RoadGraphics static method), 52
distance() (highway_env.road.lane.AbstractLane method), 31
DISTANCE_WANTED (highway_env.vehicle.behavior.IDMVehicle attribute), 44
distance_with_heading() (highway_env.road.lane.AbstractLane method), 31
draw_stripes() (highway_env.road.graphics.LaneGraphics class method), 52
enforce_road_rules() (highway_env.road.regulation.RegulatedRoad method), 37
EnvViewer (class in highway_env.envs.common.graphics), 49
ExitObservation (class in highway_env.envs.common.observation), 25

fill_road_layer_by_cell() (highway_env.envs.common.observation.OccupancyGridObservation method), 25
fill_road_layer_by_lanes() (highway_env.envs.common.observation.OccupancyGridObservation method), 25
follow_road() (highway_env.vehicle.controller.ControlledVehicle method), 41
from_config() (highway_env.road.lane.AbstractLane class method), 31
from_config() (highway_env.road.lane.CircularLane class method), 34
from_config() (highway_env.road.lane.PolyLaneFixedWidth class method), 35
from_config() (highway_env.road.lane.SineLane class method), 33
from_config() (highway_env.road.lane.StraightLane class method), 32

get_available_actions() (highway_env.envs.common.action.ActionType method), 27
get_available_actions() (highway_env.envs.common.action.DiscreteMetaAction method), 29
get_available_actions() (highway_env.envs.common.action.MultiAgentAction method), 29
get_image() (highway_env.envs.common.graphics.EnvViewer method), 50
get_routes_at_intersection() (highway_env.vehicle.controller.ControlledVehicle method), 41
GrayscaleObservation (class in highway_env.envs.common.observation), 23

handle_event() (highway_env.road.graphics.WorldSurface method), 51
handle_events() (highway_env.envs.common.graphics.EnvViewer method), 50
heading_at() (highway_env.road.lane.CircularLane method), 30
heading_at() (highway_env.road.lane.PolyLaneFixedWidth method), 35
heading_at() (highway_env.road.lane.SineLane method), 33
heading_at() (highway_env.road.lane.StraightLane method), 32
highway_env.envs.common.abstract module, 58
highway_env.envs.common.action module, 26, 27
highway_env.envs.common.graphics module, 49
highway_env.envs.common.observation module, 23
highway_env.road.graphics module, 50
highway_env.road.lane module, 30
highway_env.road.regulation module, 37
highway_env.road.road module, 36
highway_env.vehicle.behavior module, 44
highway_env.vehicle.controller module, 40
highway_env.vehicle.graphics module, 53
highway_env.vehicle.kinematics module, 38
HighwayEnv (class in highway_env.envs.highway_env), 8
HISTORY_SIZE (highway_env.vehicle.kinematics.Vehicle attribute), 38

IDMVehicle (class in highway_env.vehicle.behavior), 44
index_to_speed() (highway_env.vehicle.controller.MDPVehicle method), 42
IntersectionEnv (class in highway_env.envs.intersection_env), 13
is_reachable_from() (highway_env.road.lane.AbstractLane method), 31
is_visible() (highway_env.road.graphics.WorldSurface method), 50
highway-env Documentation

K
KinematicObservation (class in highway_env.envs.common.observation), 24
KinematicsGoalObservation (class in highway_env.envs.common.observation), 25

L
LaneGraphics (class in highway_env.road.graphics), 51
LENGTH (highway_env.vehicle.kinematics.Vehicle attribute), 38
LinearVehicle (class in highway_env.vehicle.behavior), 46
LineType (class in highway_env.road.lane), 32
local_angle() (highway_env.road.lane.AbstractLane method), 31
local_coordinates() (highway_env.road.lane.AbstractLane method), 30
local_coordinates() (highway_env.road.lane.CircularLane method), 34
local_coordinates() (highway_env.road.lane.PolyLaneFixedWidth method), 34
local_coordinates() (highway_env.road.lane.SineLane method), 33
local_coordinates() (highway_env.road.lane.StraightLane method), 32

M
MAX_SPEED (highway_env.vehicle.kinematics.Vehicle attribute), 38
MDPVehicle (class in highway_env.vehicle.controller), 42
MergeEnv (class in highway_env.envs.merge_env), 9
metaclass__ (highway_env.road.lane.AbstractLane attribute), 30
MIN_SPEED (highway_env.vehicle.kinematics.Vehicle attribute), 38
mobil() (highway_env.vehicle.behavior.IDMVehicle method), 46
module
highway_env.envs.common.abstract, 58
highway_env.envs.common.action, 26, 27
highway_env.envs.common.graphics, 49
highway_env.envs.common.observation, 23
highway_env.road.graphics, 50
highway_env.road.lane, 30
highway_env.road.regulation, 37
highway_env.road.road, 36
highway_env.vehicle.behavior, 44
highway_env.vehicle.controller, 40
highway_env.vehicle.graphics, 53

N
neighbour_vehicles() (highway_env.road.road.Road method), 36
normalize() (highway_env.envs.common.observation.OccupancyGridObservation method), 25
normalize_obs() (highway_env.envs.common.observation.KinematicObservation method), 24

O
observe() (highway_env.envs.common.observation.ExitObservation method), 25
observe() (highway_env.envs.common.observation.GrayscaleObservation method), 24
observe() (highway_env.envs.common.observation.KinematicObservation method), 24
observe() (highway_env.envs.common.observation.KinematicsGoalObservation method), 25
observe() (highway_env.envs.common.observation.OccupancyGridObservation method), 25
OccupancyGridObservation (class in highway_env.envs.common.observation), 24
on_lane() (highway_env.road.lane.AbstractLane method), 31

P
ParkingEnv (class in highway_env.envs.parking_env), 11
PERCEPTION_DISTANCE (highway_env.envs.common.abstract.AbstractEnv attribute), 58
pix() (highway_env.road.graphics.WorldSurface method), 50
plan_route_to() (highway_env.vehicle.controller.ControlledVehicle method), 41
PolyLane (class in highway_env.road.lane), 35
PolyLaneFixedWidth (class in highway_env.road.lane), 34
pos2pix() (highway_env.road.graphics.WorldSurface method), 50
pos_to_index() (highway_env.envs.common.observation.OccupancyGridObservation method), 25
Index
STRIPE_SPACING (highway_env.road.graphics.LaneGraphics attribute), 51

STRIPE_WIDTH (highway_env.road.graphics.LaneGraphics attribute), 51

stripped_line() (highway_env.road.graphics.LaneGraphics method), 51

T

target_speed (highway_env.vehicle.behavior.AggressiveVehicle attribute), 48
target_speed (highway_env.vehicle.behavior.DefensiveVehicle attribute), 48
target_speed (highway_env.vehicle.behavior.IDMVehicle attribute), 46
target_speed (highway_env.vehicle.behavior.LinearVehicle attribute), 47
target_speed (highway_env.vehicle.controller.ControlledVehicle attribute), 40
target_speed (highway_env.vehicle.controller.MDPVehicle attribute), 42

TIME_WANTED (highway_env.vehicle.behavior.IDMVehicle attribute), 45
TIME_WANTED (highway_env.vehicle.behavior.LinearVehicle attribute), 40
to_config() (highway_env.road.lane.AbstractLane method), 31
to_config() (highway_env.road.lane.CircularLane method), 34
to_config() (highway_env.road.lane.PolyLane method), 35
to_config() (highway_env.road.lane.PolyLaneFixedWidth method), 35
to_config() (highway_env.road.lane.SineLane method), 33
to_config() (highway_env.road.lane.StraightLane method), 32

W

W

Vehicle (class in highway_env.vehicle.kinematics), 38

vec2pix() (highway_env.road.graphics.WorldSurface method), 50

vehicle (highway_env.envs.common.abstract.AbstractEnv property), 58

vehicle_class (highway_env.envs.common.action.ActionType property), 27

vehicle_class (highway_env.envs.common.action.ContinuousAction property), 28

vehicle_class (highway_env.envs.common.action.DiscreteMetaAction property), 29

vehicle_class (highway_env.envs.common.action.MultiAgentAction property), 29

WorldSurface (class in highway_env.road.graphics), 50

width_at() (highway_env.road.lane.AbstractLane method), 31

width_at() (highway_env.road.lane.CircularLane method), 34

width_at() (highway_env.road.lane.PolyLane method), 35

width_at() (highway_env.road.lane.PolyLaneFixedWidth method), 35

window_position() (highway_env.envs.common.graphics.EnvViewer method), 50