FLUIDS Documentation

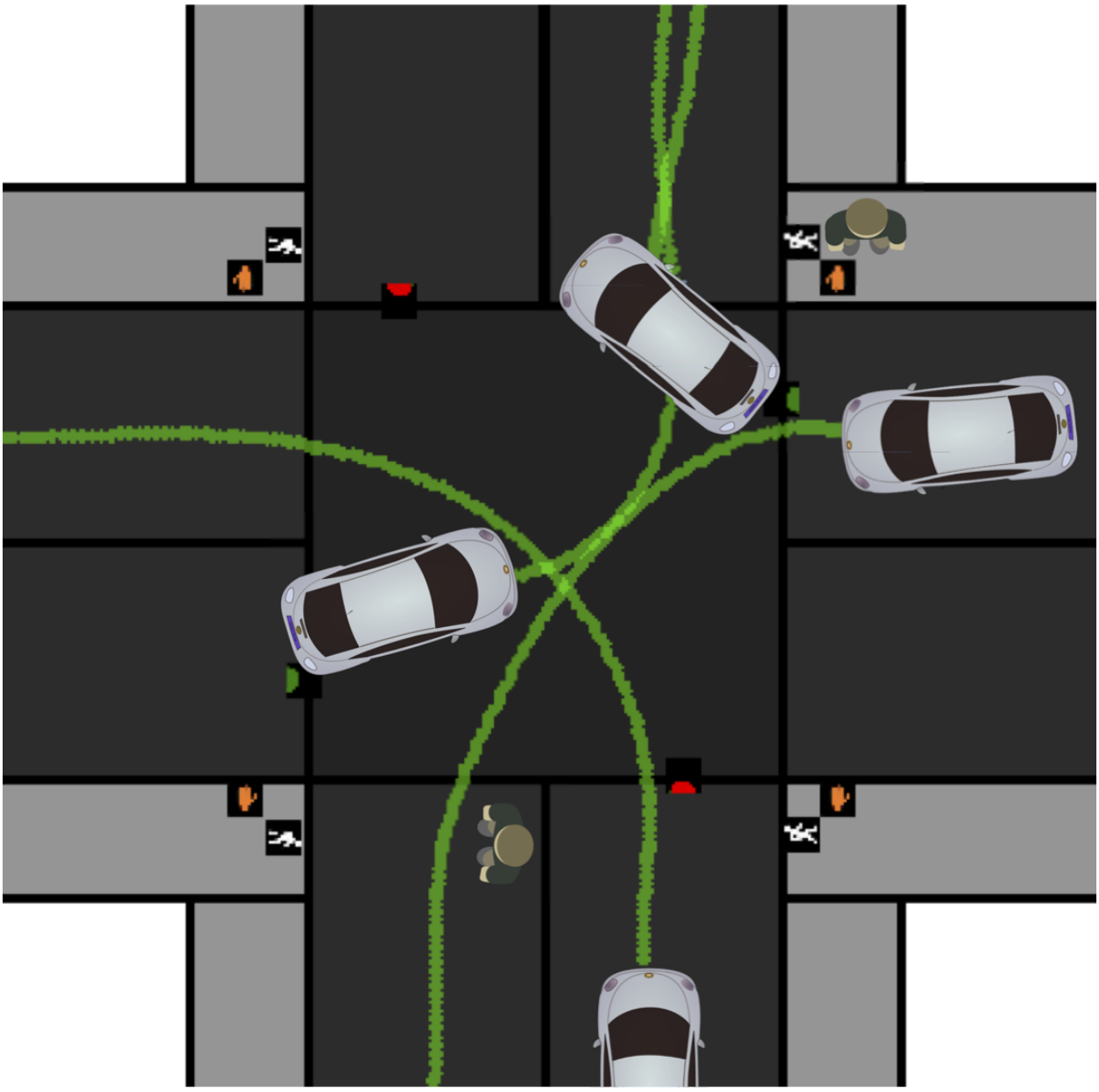
Release 1.0

Andrew Cui, Jerry Zhao, Michael Laskey, Berkeley AUTOLAB

Jul 05, 2018

Contents:

FLUIDS - a First-Order Local Urban Intersection Driving Simulator. `Read our paper`



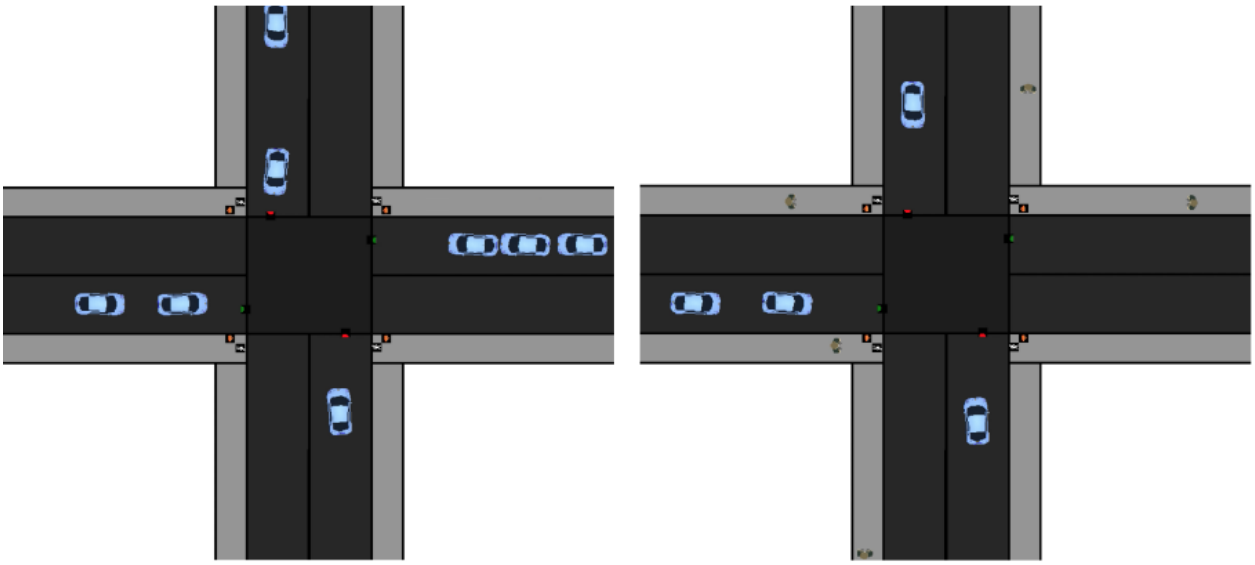
CHAPTER 1

Why FLUIDS?

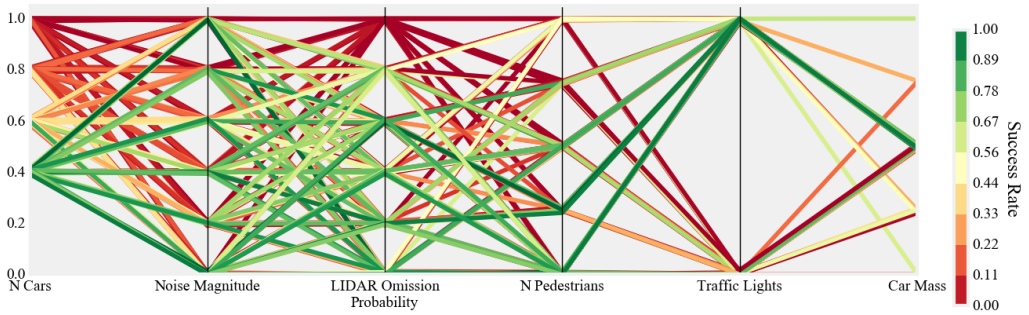
To study and compare Reinforcement and Imitation Learning algorithms, the most commonly used benchmarks are OpenAI Gym, Mujoco, and ATARI games. However, these benchmarks generally fail to capture real-life challenges to learning algorithms, including multi-agent interactions, noisy sensors, and generalization. FLUIDS aims to fill this gap by providing a fast, birds-eye simulation of cars and pedestrians in an urban driving setting. Below, we highlight several notable capabilities of FLUIDS.

1.1 Tests Generalization

FLUIDS is designed to test how well agents generalize to new environments that are both in and out of sample from the initial state distribution. The initial state distribution can be specified to be a range of cars on the road starting at various lane positions. The randomness stems from the number of cars currently on the road and the location of each car.

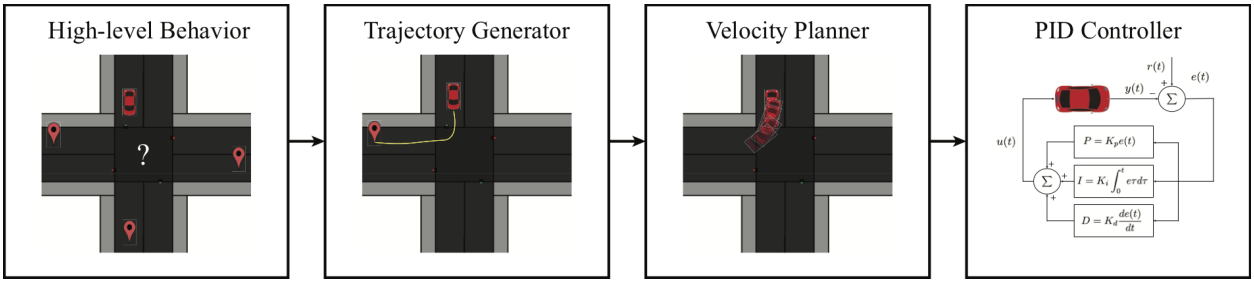


To test how robust a policy is to out of sample distributions, FLUIDS allows for perturbations such as enabling pedestrians, varying traffic light timing and changing the amount of noise in the sensor readings. FLUIDS can convey how robust a policy is via the generation of a coordinate axis plot as shown below, which demonstrates which helps shows what disturbances significantly affect policy performance.



1.2 Hierarchal Learning

Self-driving cars commonly use a hierarchical planning stack to accomplish tasks, thus they make for a great benchmark to test recent advances in learning hierarchal structure. As shown below each background agent in has a planning stack broken into four levels; behavior logic, nominal trajectory generation, velocity planning and low-level PID. The details of how these components can be composed in a planning stack can be found in the FLUIDS white paper.



When a tested agent acts in the simulator it can operate at the trajectory, velcoity or steering level of control. Thus the whole hierarchy can be learned or only sub-components. To specify which level of control to apply, the action space in the JSON config file can be changed to the designated level.

```
"agents":{
     ...
     "action_space":"steering", #other options:"trajectory", "velocity"
     ...}
 },
```

1.3 Multi-Agent Planning

Another advantage of FLUIDS is that the number of supervisors that can be controlled by an agent and that controlled by the simulator is variable. Experiments such as coordinating a fleet of self-driving cars traversing an intersection or having a single self-driving car pass an intersection can all be supported by simply changing the JSON config file.

```
"agents":{
     "controlled_cars":1,
     "background_cars":3,
     ...}
 },
```

1.4 Built-in Supervisors

In order to collect consist training data for Imitation Learning experiments and a baseline for performance. FLUIDS provides access to an array of supervisors, which can perform the driving tasks by having access to the global state of the world. The algorithms for the supervisors are the same planning stack used for the background agents.

FLUIDS supports a supervisor for each level of the hierarchy. Thus, supervision can be used to potentially help discover the hierarchy or to only learn sub-components. FLUIDS also allows for human supervision via the use of a keyboard interface.

```
#Hierarchy Supervisors

#Provides Examples at the TrajectoryLevel of the Planner
sup_traj = TrajectorySupervisor(agent_num=0)

#Provides Examples at the Veloicty Level of the Planner
sup_vel = VelocitySupervisor(agent_num=1)

#Provides Examples at the Steering Level of the Planner
sup_steer = SteeringSupervisor(agent_num=2)

#Human Supervisor

#Uses keyboard to get signal

sup_human = KeyBoardAgent
```

See the Examples section for a tutorial on how to use the supervisor agents.

CHAPTER 2

Installation

2.1 Install FLUIDS

```
pip3 install gym-urbandriving
```

2.2 Install FLUIDS from Source

These commands install gym-urbandriving and its requirements in the current Python environment.

```
git clone https://github.com/BerkeleyAutomation/Urban_Driving_Simulator.git
cd Urban_Driving_Simulator
pip3 install -e .
```

2.3 Optional OMPL Install

Additional trajectory generation features are available if OMPL (Open Motion Planning Library) is installed. The following is the installation instructino for Mac users.

Install Macports https://www.macports.org/install.php (Note macports is very heavily tied to xcode to guarantee this to work you will need to have xcode installed) With macports installed perform the following three lines of code

```
sudo port sync
sudo port clean castxml
sudo port install ompl +app
```

Macport will download its own version of python2.7 that everything will work off of. To link the command line python type the following:

```
sudo port select python2 python2.7
```

CHAPTER 3

Examples

This tutorial will walk you though many of the features of FLUIDs to run an intersection simulation with multiple user controlled cars, background cars, pedestrians, and traffic lights. A link ot the final code is below:

`Download`

3.1 Configuring the Environment

The environment and agent configuration in FLUIDS is controlled by JSON configuration files.

```
"environment":{
    "state":"four_way_intersection",
    "visualize": true,
    "visualize_lidar": true,
    "max_time": 100
},
```

The "state" flag specifies the layout of the roads, terrain, and sidewalks in the scene by pointing the simulator to a scene description JSON. We currently package only the four way intersection.

The "visualize" and "visualize_lidar" flags enable the graphical display, which is optional.

The "max_time" field specifies the maximum number of ticks the simulation will run before resetting. This is useful for performing many roll-outs back-to-back.

```
"agents":{
   "controlled_cars":1,
   "background_cars":2,
   "action_space":"steering",
   "state_space":"Q-LIDAR",
   "state_space_config":{
      "goal_position":false,
      "noise":0,
      "omission_prob":0
```

(continues on next page)

(continued from previous page)

```
    },
    "bg_state_space_config":{
        "noise":0,
        "omission_prob":0
    },
    "use_traffic_lights":true,
    "number_of_pedestrians":0,
    "agent_mappings":{
        "Car":"PlanningPursuitAgent",
        "TrafficLight":"TrafficLightAgent",
        "CrosswalkLight":"CrosswalkLightAgent",
        "Pedestrian":"PedestrianAgent"
    }
 },
```

There's a lot here, but most of it is self-explanatory. Here, we specify 1 controlled car with user-defined controls, and 3 background cars controlled by our supervisor. The "action_space" of our controlled car will be the steering control. This can be configured to multiple levels of the self-driving hierarchy. The "state_space" and "state_space_config" fields configure the state representation available to the user agent. Here we use our "quasi-lidar" representation.

We create the state with traffic lights and pedestrians. The "agent_mappings" field marks what types of agents are controlling every type of background object.

3.2 Running the Environment

The basic evaluation loop is very simple. We initialize the environment with the config file. In the simulation loop, we repeatedly step forward through the environment, receive observations, and provide new actions for all controlled cars in the scene.

```
import gym
import gym_urbandriving as uds
from gym_urbandriving.actions import SteeringAction
import numpy as np
import json

config = json.load(open('configs/default_config.json'))
env = uds.UrbanDrivingEnv(config_data=config)

env._reset()
env._render()
obs = env.get_initial_observations()
action = SteeringAction(0, 0)

while(True):
    obs, reward, done, info_dict = env._step([action])
    env._render()
    if done:
        print("done")
        env._reset()
        obs = env.get_initial_observations()
```

Here we step forward through the simulation until either there is a collision, or the max time is reached. We provide a SteeringAction because the environment was configured such that user cars received SteeringActions. The actions are provided in an array to support multiple controlled vehicles.

Now we connect agents to the controlled cars. For this test, we use keyboard agents.

```
from gym_urbandriving.agents import KeyboardAgent
agent = KeyboardAgent()
while(True):
    action = agent.eval_policy(obs[0])
    obs, reward, done, info_dict = env._step([action])
    env._render()
    if done:
        print("done")
        env._reset()
        obs = env.get_initial_observations()
```

Notice that the observations returned are an array, one for each controlled car. The observation is specified in the config file. For this example, Q-LIDAR observations are used. Q-LIDAR represents a set of observations similar to what a self-driving car might receive from camera and LIDAR sensors.

3.2.1 Using Neural Background Agents

In default mode the simulator is running a predictive planner to automatically adjust the velocity of the car to avoid collisions with others. This planner can be computationally expensive though, so an alternative is to used a neural network based implementation that was trained to approximate the predictive planner. In order to active the approximate planner and increase speed, adjust the "agent_mappings" flag in the configuration.

```
config['agents']['agent_mappings']['Car'] = 'NeuralPursuitAgent'
```

3.2.2 Using a Steering Supervisor

Instead of using a Keyboard agent, FLUIDS is packaged with supervisor agents at several levels of the controls hierarchy for a self-driving car. First we replace the KeyboardAgent with a SteeringSupervisor. Since the steering supervisor expects access to the full state, we specify this in the config file.

```
config['agents']['state_space'] = 'raw'
```

```
from gym_urbandriving.agents import SteeringSupervisor
agent = SteeringSupervisor()
while(True):
    action = agent.eval_policy(obs[0])
    obs, reward, done, info_dict = env._step([action])
    env._render()
```

3.2.3 Using a Velocity Supervisor

While the steering supervisor provides full steering and acceleration controls to the car, FLUIDS also supports controlling the car at different levels in the planning stack. For example, we can control the target velocity that the car operates at using the Velocity Supervisor.

```
config['agents']['state_space'] = 'raw'
config['agents']['action_space'] = 'velocity'
```

```
from gym_urbandriving.agents import VelocitySupervisor
agent = VelocitySupervisor()
while(True):
    action = agent.eval_policy(obs[0])
    obs, reward, done, info_dict = env._step([action])
    env._render()
```

3.2.4 Using Pedestrians

FLUIDS also supports the simulation of pedesterian agents. Uncomment the following line in the provided file to add background pedesterians in the scene. Adjust the flag in the configuration, which is loaded as a Python dictionary.

```
config['agents']["number_of_pedestrians"]:4
```

CHAPTER 4

Environments

4.1 The UrbanDrivingEnvironment Class

class gym_urbandriving.**UrbanDrivingEnv**(*config_data={}*, *init_state=None*, *reward_fn=<function default_reward_function>*, *randomize=False*)

This class controls the evolution of a world state. While the `PositionState` represents the layout of objects in the scene, the *UrbanDrivingEnv* controls the evolution of the scene, and manages background actors.

Note: This class is used both to represent the true, global state of the world, and as a search agent's internal view of the world.

Parameters

- **config_data** (*Dict*) – JSON config file is loaded into a dictonary specifing prameters
- **init_state** (PositionState) – The starting state for this environment (Only needed if now JSON config file is used)
- **reward_fn** – A function which takes in one parameter, the `PositionState`,
- **randomize** (*bool*) – Sets whether `env._reset()` returns the environment to the initial state, or if it returns it to a random state generated by the state's `randomize()` function

_render(*mode='human'*, *close=False*, *waypoints=[]*, *traffic_trajectories=[]*, *transparent_surface=None*)

If the renderer was specifed at construction, renders the current state of the world

Parameters

- **mode** (*str*) – For OpenAI Gym compatibility
- **waypoints** – Extra points you would like to render over top of the the scene, for debugging

_reset(*new_state=None*)
Resets the environment to its initial state, or a new state

Parameters **new_state** (PositionState) – If specified, the environment will reset to this state

_step(*action*, *background_simplified=False*, *supervisor=False*)
The step function accepts a control for the 0th agent in the scene. Then, it queries all the background agents to determine their actions. Then, it updates the scene and returns.

Parameters

- **action** – An action for the agentnum object in the scene.
- **agentnum** (*int*) – The index for the object which the action is applied for.

Returns

- *PositionState* – State of the world after this step;
- *float* – Reward calculated by `self.reward_fn()`,
- *bool* – Whether we have timed out, or there is a collision)
- *dict*

CHAPTER 5

Agents

5.1 Background

Used internally to control agents in the scene such as traffic lights, cross walks, pedestrians and background cars. These are not intended to be instantiated outside the UDS environment

class gym_urbandriving.agents.background.planning_pursuit_agent.**PlanningPursuitAgent**(*agent_n*

Background agent which implements the full plannning stack given known behavioral logic. The planner first generates a nominal trajecotry, then at each timestep plans its velocity to avoid collisons.

agent_num

int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*state*, *simplified=False*)

Returns action based on current state

Parameters **state** (PositionState) – State of the world, unused

Returns

Return type tuple with floats (steering,acceleration)

class gym_urbandriving.agents.background.pursuit_agent.**PursuitAgent**(*agent_num=0*)

eval_policy(*state*, *type_of_agent='background_cars'*)

Returns action based next state in trajectory.

Parameters **state** (PositionState) – State of the world, unused

Returns

Return type *SteeringAction*

5.2 Hierarchical

Used to build control at different levels of the planning software stack. They are used internally in FLUIDS to allow the user to apply a different type of actions to the environment.

class gym_urbandriving.agents.hierarchical.velocity_action_agent.**VelocityActionAgent**(*agent_n*
Hierarichal agent which implements the full plannning stack except the velocity component The planner first generates a nominal trajecotry, then at each timestep recives a target velocity to track with PID controller.

agent_num
int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*action*, *state*, *simplified=False*)
Returns action based next state in trajectory.

Parameters

- **state** (PositionState) – State of the world, unused
- **action** (VelocityAction *or None*) – Target velocity for car to travel at

Returns

Return type tuple with floats (steering,acceleration)

class gym_urbandriving.agents.hierarchical.steering_action_agent.**SteeringActionAgent**(*agent_n*
Hierarichal agent which does not include any planning stack and only requires specifiying the steering agent.

agent_num
int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*action*, *state*, *simplified=False*)
Returns action based next state in trajectory.

Parameters

- **state** (PositionState) – State of the world, unused
- **action** (SteeringAction) –

Returns

Return type tuple with floats (steering,acceleration)

5.3 Supervisors

Classes use to control different cars in the scene. Logically they are similar to background cars and have access to the internal state of the world to make a decision. However, they can be used to collect data and provide ground truth of what the supervisor would have done in a single state.

class gym_urbandriving.agents.supervisor.velocity_supervisor.**VelocitySupervisor**(*agent_num=0*)
Superivsor agent which implements the planning stack to obtain velocity level supervision of which the car should follow.

agent_num
int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*state*, *simplified=False*)
Returns action based next state in trajectory.

Parameters

- **state** (PositionState) – State of the world, unused
- **simplified** (*bool*) – specifies whether or not to use a simplified greedy model for look ahead planning

Returns

Return type float specifying target velocity

class gym_urbandriving.agents.supervisor.steering_supervisor.**SteeringSupervisor**(*agent_num=0*)

Superivsor agent which implements the planning stack to obtain steering level supervision of which the car should follow.

agent_num
int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*state*)
Returns action based on current world state

Parameters

- **state** (PositionState) – State of the world, unused
- **action** (*float*) – Target velocity for car to travel at

Returns

Return type *SteeringAction*

5.4 Tele-Op

Used for a human supervisor to operate the vehicle. Currently, the use of keyboard commands is supported.

class gym_urbandriving.agents.tele_op.keyboard_agent.**KeyboardAgent**(*agent_num=0*)

Agent which interprets user keyboard inputs

agent_num
int – Index of this agent in the world. Used to access its object in state.dynamic_objects

eval_policy(*state*)
Returns action based on keyboard input

Parameters **state** (PositionState) – State of the world, unused

Returns

Return type numpy array with elements (steering,acceleration)

5.5 Pedestrian

FLUIDS also packages a basic pedestrian controller for controling the movement of background pedestrians. Pedestrians display simple behavior, walking forwards unless blocked by a crosswalk signal.

class gym_urbandriving.agents.supervisor.pedestrian_supervisor.**PedestrianAgent**(*agent_num=0*)

Supervisor Agent for controlling pedestrians

agent_num
int – Index of this agent in the world

eval_policy(*state*)
Returns action based on state of world

Parameters **state** (PositionState) – State of the world

Returns

Return type Turning angle, acceleration pair

CHAPTER 6

Intersections

6.1 Base Intersections

class gym_urbandriving.state.**PositionState**(*data*, *car_model='kinematic'*)

Abstract class representing the objects in a scene

collides_any(*agentnum*, *type_of_agent='background_cars'*)

Returns if the agentnum object in the scene is colliding with any other object

Parameters **agentnum** (*int*) – The index of the object to query

Returns True if this object is colliding

Return type bool

create_agents()

Creates agents for objects in the scene

get_collisions()

Get list of all collisions in this state

Returns

- *list* – List of tuples, where each tuple contains a pair of coliding object indices. Dynamic_collisions contains collisions between cars and other cars.
- *list* – The corresponding list for collisions between dynamic objects and static objects

min_dist_to_coll(*agentnum*, *type_of_agent='background_cars'*)

Returns the minimum distance between the object with id agentnum and a collideable object.

Parameters **agentnum** (*int*) – The index of the object to query

Returns Distance to nearest collideable object

Return type float

randomize()

Randomly generates car and pedestrian positions

CHAPTER 7

Actions

7.1 Velocity Action

class gym_urbandriving.actions.velocity_action.**VelocityAction**(*velocity=0.0*)

This class is a wrapper for the velocity control in the hierarchy it represents the target velocity for the car to drive at

get_value()

Gets the value of the current velocity

Returns

Return type float

sample()

Samples a random control in this class using the OpenAI Box class

Returns

Return type velocity in numpy array shape (1,)

7.2 Steering Action

class gym_urbandriving.actions.steering_action.**SteeringAction**(*steering=0.0*, *acceleration=0.0*)

This class is a wrapper for the action space at the lowest level of the heirarchy it represents the steering and acceleration applied directly to the car.

get_value()

Gets the numpy array of the class

Returns

Return type steering and acceleration in numpy array shape (2,)

sample()
Samples a random control in this class using the OpenAI Box class

Returns

Return type steering and acceleration in numpy array shape (2,)

CHAPTER 8

Observations

Observations are generated for each controlled car in the scene. The type of observation can be specified in the config file as 'raw', 'Q-LIDAR', or 'bitmap'. The step function in the environment returns an observation for each controlled car in the scene as a list.

8.1 Raw

A copy of the the raw environment, giving agents full access to the scene, and all other objects in the scene. Using this observation type should be avoided, as duplicating the environment incurs a significant performance penalty.

8.2 Q-LIDAR

A representation based on features a autonomus vehicle might extract from LIDAR sensors, which is the relative distance to collideable objects in the scene. This is a numpy array of distances produced by a Featurizer. The density and range of the Q-LIDAR beams can be configured in the featurizer.

class gym_urbandriving.utils.featurizer.**Featurizer**(*config_data={}*, *beam_distance=300*, *n_arcs=9*)

Object to convert a state observation into a Q-LIDAR observation.

beam_distance
int – How far each "LIDAR" beam will project into the scene

n_arcs
How many "LIDAR" beams to project around the car

featurize(*current_state*, *controlled_key*, *type_of_agent='controlled_cars'*)
Returns a Numpy array of a Q-LIDAR representation of the state

Parameters

- **current_state** (PositionState) – State of the world
- **controlled_key** – Key for controlled car in the state to generate a feature for

Returns

Return type Numpy array. For each ray projected into the scene, adds distance to collision, angle to collision, and velocity of intersected object

8.3 Bitmap

Returns a Numpy image array as generated by the visualizer, for vision-based control agents. Image is a top-down view of the intersection.

CHAPTER 9

Objects

9.1 Cars

class gym_urbandriving.assets.**Car**(*x*, *y*, *xdim=80*, *ydim=40*, *angle=0.0*, *vel=0.0*, *max_vel=5*, *mass=100.0*, *dynamics_model='kinematic'*, *destination=None*, *trajectory=None*)

Represents a point-model car.

Parameters

- **x** (*float*) – Starting x coordinate of car's center
- **y** (*float*) – Starting y coordinate of car's center
- **angle** (*float*) – Starting angle of car in world space
- **vel** (*float*) – Starting velocity of car

vel
float – Forwards velocity of car

max_vel
float – Maximum allowable velocity of this carst

xdim
float – Length of car

ydim
float – Width of car

can_collide(*other*)
Specifies whether this object can collide with another object

Parameters **other** – Object to test collision against

Returns True if this object can collide with other

Return type bool

step(*action*)
Updates this object given this action input

Parameters **action** – The action to take

9.2 Pedestrians

class gym_urbandriving.assets.**Pedestrian**(*x*, *y*, *radius=12*, *angle=0.0*, *vel=0.0*, *acc=0.0*, *max_vel=2.0*, *mass=100.0*, *dynamics_model='point'*)
Represents a pedestrian as a circle

Parameters

- **x** (*float*) – Center x coordinate
- **y** (*float*) – Center y coordinate
- **radius** (*float*) – Size of the pedestrian
- **angle** (*float*) – Initial orientation, in degrees
- **vel** (*float*) – Initial velocity
- **max_vel** (*float*) – Maximum velocity
- **mass** (*float*) – Mass of pedestrian

step(*action*, *info_dict=None*)
Updates the pedestrian for one timestep.

Parameters

- **action** (*1x2 array*) – Steering / acceleration action.
- **info_dict** (*dict*) – Contains information about the environment.

9.3 Terrain

class gym_urbandriving.assets.**Terrain**(*x*, *y*, *xdim=0*, *ydim=0*, *points=[]*, *radius=0*, *excludes=[]*)
Represents a square of impassable terrain

Parameters **points** (*list*) – List of X-Y tuples in ccw order describing vertices of the polygon

9.4 Lanes

class gym_urbandriving.assets.**Lane**(*x=0*, *y=0*, *xdim=0*, *ydim=0*, *angle=0.0*, *angle_deg=0*, *points=[]*, *curvature=0*, *inner_r=0*, *outer_r=0*)
Represents a lane of road. Lanes have directionality, so cars should drive in the right direction. Default construction creates a rectangular block.

Parameters

- **x** (*float*) – Upper left x coordinate of the lane block
- **y** (*float*) – Upper left y coordinate of the lane block

- **xdim** (*float*) – Width of the lane block
- **ydim** (*float*) – Height of the lane block
- **; float** (*angle*) – In degrees, the rotation of the lane block. The correct direction of travel along this lane.
- **points** (*list*) – List of XY coordinates specifying edge points of a polygon. If specified, lane will be constructed as a polygon.
- **curvature** (*list*) – If specified, generates a curved road segment with this arc angle, centered at x, y, and with inner and outer radii
- **outer_r** (*inner_r,*) – Use with curvature argument to generated curved road segment.

generate_car(*car_model='kinematic'*)

Creates a car on this lane ready to drive into the intersection

Parameters **car_type** (*"kinematic" or "point" or "reeds_shepp"*) – Specifies dynamics model for the car

Returns Generated Car object

Return type *Car*

9.5 Sidewalks

class gym_urbandriving.assets.**Sidewalk**(*x*, *y*, *xdim*, *ydim*, *angle=0.0*, *angle_deg=0*, *points=[]*)

Represents a block of sidewalk. Passable for pedestrians, not for cars

Parameters

- **x** (*float*) – Upper left x coordinate of the sidewalk block
- **y** (*float*) – Upper left y coordinate of the sidewalk block
- **xdim** (*float*) – Width of the sidewalk block
- **ydim** (*float*) – Height of the sidewalk block
- **points** (*list*) – If specified, constructs sidewalk as polygon

generate_man(*man_type=<class 'gym_urbandriving.assets.pedestrian.Pedestrian'>*)

Generates a man on the sidewalk

Returns Generated Pedestrian object

Return type *Pedestrian*

9.6 Streets

class gym_urbandriving.assets.**Street**(*x*, *y*, *xdim*, *ydim*, *angle=0*, *points=[]*)

Represents a block of street. Passable for cars and pedestrians. Does not have directionality associated with it, so use this for the middle of an intersection

Parameters

- **x** (*float*) – Upper left x coordinate of the street block
- **y** (*float*) – Upper left y coordinate of the street block

- **xdim** (*float*) – Width of the street block
- **ydim** (*float*) – Height of the street block
- **points** – If specified, constructs this shape as a polygon

Index

Symbols

A

B

C

E

F

G

K

L

M

N

P

R

S

T

U

V

X

Y