Pylot Documentation

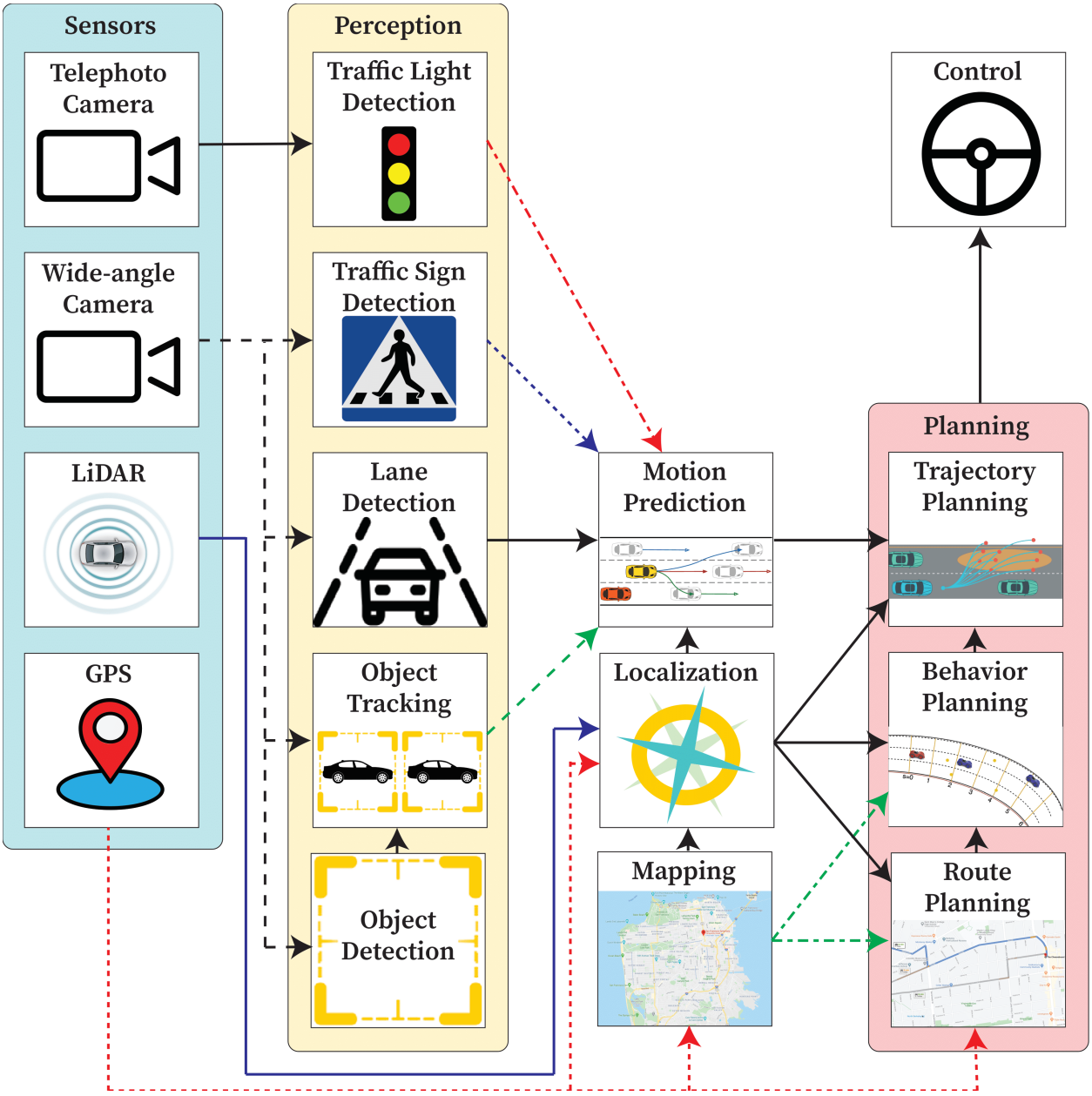
Release 0.2

The Pylot team

Sep 29, 2021

Installation guide

Pylot is an autonomous vehicle platform for developing and testing autonomous vehicle components (e.g., perception, prediction, planning) on the CARLA simulator and real-world cars.



CHAPTER 1

More Information

1.1 Slides

- Talk given at RISE Retreat 2020

CHAPTER 2

Getting Involved

- GitHub Issues: For reporting bugs and feature requests.
- Pull Requests: For submitting code contributions.

2.1 Quick start

The easiest way to run Pylot is to use our Docker image. First, please ensure you have *nvidia-docker* on your machine before you start installing Pylot. In case you do not have *nvidia-docker* please run `./scripts/install-nvidia-docker.sh`.

We provide a Docker image with both Pylot and CARLA already setup.

```
docker pull erdosproject/pylot
nvidia-docker run -itd --name pylot -p 20022:22 erdosproject/pylot /bin/bash
```

Following, start the simulator in the container:

```
nvidia-docker exec -i -t pylot /home/erdos/workspace/pylot/scripts/run_simulator.sh
```

Finally, start Pylot in the container:

```
nvidia-docker exec -i -t pylot /bin/bash
cd workspace/pylot/
python3 pylot.py --flagfile=configs/detection.conf
```

In case you desire to visualize outputs of different components (e.g., bounding boxes), you have to forward X from the container. First, add your public ssh key to the `~/.ssh/authorized_keys` in the container:

```
nvidia-docker cp ~/.ssh/id_rsa.pub pylot_new:/home/erdos/.ssh/authorized_keys
nvidia-docker exec -i -t pylot_new sudo chown erdos /home/erdos/.ssh/authorized_keys
nvidia-docker exec -i -t pylot /bin/bash
sudo service ssh start
exit
```

Finally, ssh into the container with X forwarding:

```
ssh -p 20022 -X erdos@localhost /bin/bash
cd /home/erdos/workspace/pylot/
python3 pylot.py --flagfile=configs/detection.conf --visualize_detected_obstacles
```

If everything worked ok, you should be able to see a window that visualizes detected obstacles like the one below:

2.2 Installation guide

You can install Pylot on your base system by executing the following commands:

```
git clone https://github.com/erdos-project/pylot
cd pylot
export PYLOT_HOME=`pwd`/
./install.sh
pip install -e ./
```

Next, start the simulator:

```
export CARLA_HOME=$PYLOT_HOME/dependencies/CARLA_0.9.10.1/
./scripts/run_simulator.sh
```

In a different terminal, setup the paths:

```
export CARLA_HOME=$PYLOT_HOME/dependencies/CARLA_0.9.10.1/
cd $PYLOT_HOME/scripts/
source ./set_pythonpath.sh
```

Finally, run Pylot:

```
cd $PYLOT_HOME/
python3 pylot.py --flagfile=configs/detection.conf
```

2.3 Perception

2.3.1 Depth estimation

The package provides operators for estimating depth using cameras. It currently offers a DepthEstimationOperator, which implements stereo depth estimation using the AnyNet neural network.

Execute the following command to run a depth estimation demo:

```
python3 pylot.py --flagfile=configs/depth_estimation.conf
```

Important flags

- `--depth_estimation`: Enables stereo depth estimation.
- `--depth_estimation_model_path`: File path to a trained Anytime network model.
- `--perfect_depth_estimation`: The component outputs frames with perfect depth values. This frames are obtained from the simulator.
- `--offset_left_right_cameras`: Offset distance (in meteres) between the left and right cameras used for depth estimation.
- `--visualize_depth_camera`: Enables visualization of the sensor depth camera.

More information

See the reference for more information.

2.3.2 Detection

The package provides operators and classes useful for detecting obstacles, traffic lights and lanes. It provides operators that use trained models and algorithms, as well as operators that use data from the simulator to perfectly detect obstacles, traffic lights, and lanes.

Obstacle detection

Pylot provides two options for obstacle detection:

1. An obstacle detection operator that can use any model that adheres to the Tensorflow object detection model zoo. By default, we provide three models that were trained on 1080p CARLA images (`faster-rcnn`, `ssd-mobilenet-fpn-640`, and `ssdlit-mobilenet-v2`), but models that have been trained on other data sets can be easily plugged in by changing the `--obstacle_detection_model_paths` flag.
2. An operator that can infer any of the EfficientDet models. The EfficientDet models we use are not trained on CARLA data, but on the COCO data set.

To see a demo of obstacle detection run:

```
python3 pylot.py --flagfile=configs/detection.conf
```

Traffic light detection

The traffic light detection component uses Faster RCNN weight, which have been trained on 1080p CARLA images.

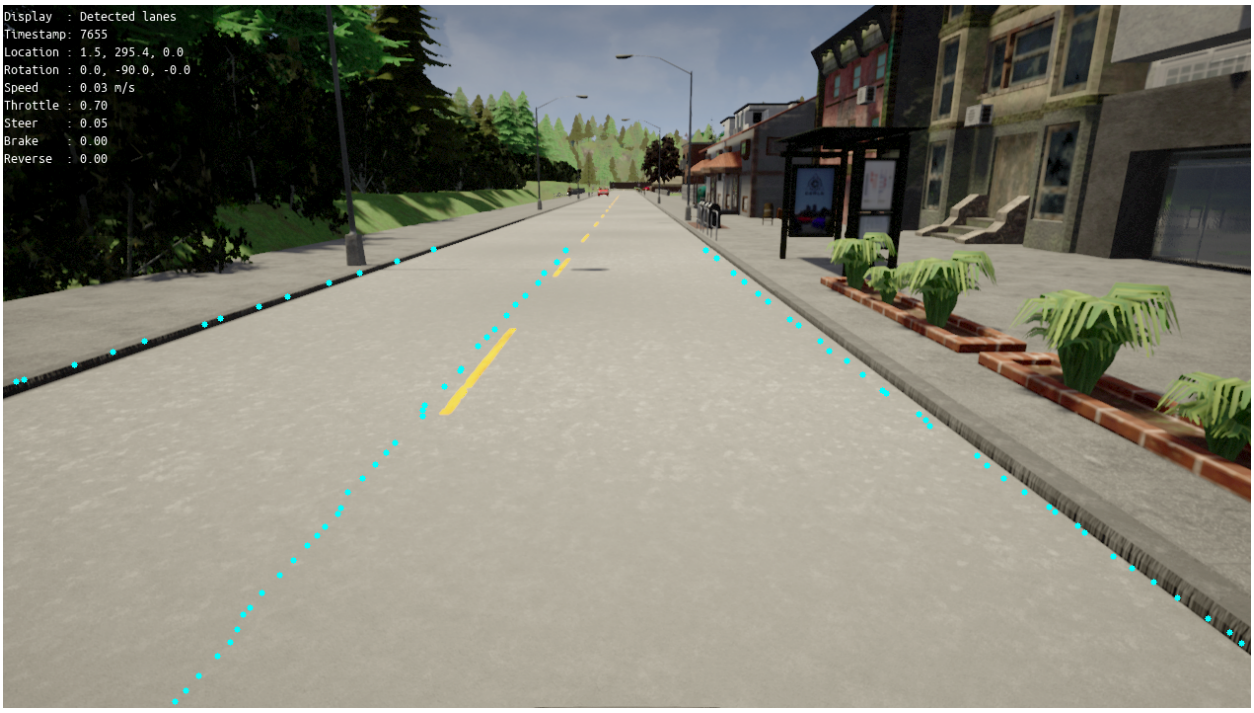
To see a demo of traffic light detection run:

```
python3 pylot.py --flagfile=configs/traffic_light.conf
```

Lane detection

To see a demo of lane detection run:

```
python3 pylot.py --flagfile=configs/lane_detection.conf
```



Warning: Our lane detection component works optimally with frames have a resolution of 1280x720.

Important flags

- `--obstacle_detection`: Enables the obstacle detection component of the stack. Depending on accuracy and runtime requirements, the component can use different obstacle detection models. Pylot currently offers three trained models: `faster-rcnn`, `ssd-mobilenet-fpn-640`, and `ssdlit-mobilenet-v2`.
- `--perfect_obstacle_detection`: Enables the component to use an obstacle detector which perfectly detects obstacles using ground information from the simulator.
- `--simulator_obstacle_detection`: The component outputs obstacle info that is obtained directly from the simulator.
- `--evaluate_obstacle_detection`: Compute and log accuracy metrics of the obstacle detection component.
- `--visualize_detected_obstacles`: Enables visualization of detected obstacles.
- `--traffic_light_detection`: Enables the traffic light detection component of the stack. This component attaches to the ego vehicle a forward facing camera with a narrow field of view, which the component uses to detect traffic lights.
- `--perfect_traffic_ligth_detection`: Enables the component to use a traffic light detector which perfectly detects traffic lights using ground information from the simulator.
- `--simulator_traffic_light_detection`: The component outputs traffic light ifo that is obtained directly from the simulator.
- `--visualize_detected_traffic_lights`: Enables visualization of detected traffic lights.
- `--lane_detection`: Enables the lane detection component, which currently implements a simple Canny edge detector.
- `--lane_detection_type`: Specifies which lane detection solution to use. Pylot current supports a standard vision implementation that uses *Canny edge*, and a neural network-based implementation that uses *Lanenet*.

- `--perfect_lane_detection`: Enables the component to perfectly detect lanes using information from the simulator.
- `--visualize_lane_detection`: Enables visualization of detected lanes.

More information

See the reference for more information.

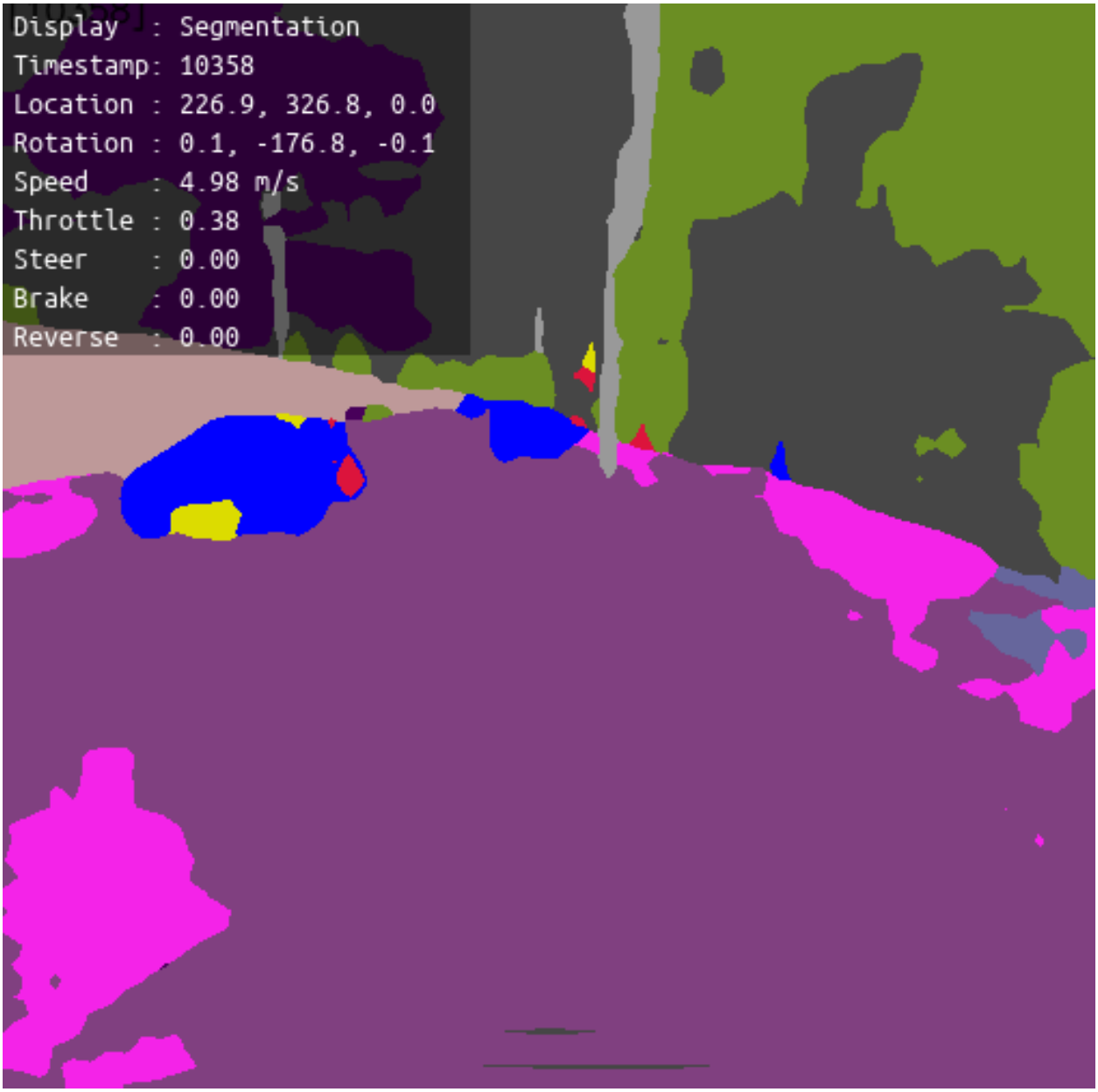
2.3.3 Segmentation

The package provides operators and classes useful for segmenting camera frames:

- SegmentedFrame is the class used to store segmented frames. It also provides utilities to transform a frame between different encodings (e.g., CARLA, Cityscapes), and to compute accuracy metrics (e.g., mIoU).
- SegmentationDRNOperator is an operator that uses the DRN neural network to segment camera frames.
- SegmenationEvalOperator implements logic to evaluate the output of a segmentation operator. It receives data on two streams and computes intersection over union (IoU) between the frames tagged with the same timestamp.

Execute the following command to run a semantic segmentation demo:

```
python3 pylot.py --flagfile=configs/segmentation.conf
```



Note: The segmentation model we used has not been trained on CARLA data, and the output of the segmentation component is not currently used by any other Pylot components.

Important flags

- `--segmentation`: Enables the segmentation component of the stack.
- `--segmentation_model_path`: File path to a trained DRN segmentation model.
- `--perfect_segmentation`: The component outputs perfectly segmented frames it receives from the simulator, instead of the results obtained using a trained model.
- `--visualize_segmentation`: Enables visualization of the output of the segmentation component.
- `--evaluate_segmentation`: Compute and log accuracy metrics of the segmentation component.

- `--segmentation_metric`: Sets the accuracy metric the SegmentationEvalOperator computes.

More information

See the reference for more information.

2.3.4 Obstacle tracking

The package provides operators and classes useful for tracking obstacles across frames:

- ObjectTrackerOperator is the operator that provides four options for tracking obstacles:
  1. `da_siam_rpn`: a high-quality DaSiamRPN network single obstacle tracker, which Pylot repurposed to track serially track multiple obstacles.
  2. `sort`: uses a simple combination of Kalman Filter and Hungarian algorithm for tracking and matching (see SORT).
  3. `deep_sort`: An extended version of SORT that integrates detection and appearance features (see Deep SORT).
- ObstacleTrajectory is used to store the trajectories of the tracked obstacles.
- MultiObjectTracker is an interfaces which must be implemented by multiple obstacles trackers.
- MultiObjectDaSiamRPNTracker is a class that implements a multiple obstacle tracker using the DASiamRPN neural network for single obstacle tracking. The class executes model inference for every single obstacle, and matches obstacles across frames using the Hungarian algorithm for bipartite graph matching.
- MultiObjectDeepSORTTracker is a wrapper class around the DeepSORT multi obstacle tracker. It executes the DeepSORT neural network on every frame.
- MultiObjectSORTTracker is wrapper class around the SORT tracker.

Execute the following command to run an obstacle tracking demo:

```
python3 pylot.py --flagfile=configs/tracking.conf
```

Important flags

- `--obstacle_tracking`: Enables the obstacle tracking component of the stack.
- `--tracker_type`: Sets which obstacle tracker the component use.
- `--perfect_obstacle_tracking`: Enables the component to perfectly track obstacles using information it receives from the simulator (only works in simulation).
- `--visualize_tracked_obstacles`: Enables visualization of tracked obstacles.
- `--tracking_num_steps`: Limit on the number of past bounding boxes to track.
- `--min_matching_iou`: Sets the minimum intersetion over union (IoU) two bounding boxes must have for the tracker matching state to consider them.

More information

See the reference for more information.

2.3.5 More information

See the reference for more information.

2.4 Prediction

The package provides operators and classes useful for predicting future trajectories of other agents. Pylot predicts future trajectories of obstacles detected and tracked by the perception component. However, if you desire to run the

prediction components using perfectly tracked obstacles, you can pass `--perfect_obstacle_tracking` when you're running in simulation.

Execute the following command to run a prediction demo:

```
python3 pylot.py --flagfile=configs/prediction.conf
```

2.4.1 Important flags

- `--prediction`: Enables the prediction component of the pipeline.
- `--prediction_type`: Sets which prediction operator to use. Pylot currently offers two prediction implementations: a simple linear predictor and R2P2.
- `--prediction_num_past_steps`: Sets the number of past obstacle locations the prediction components uses. The duration of the history used for prediction is equal to the number of past steps multiplied by the time between each step run.
- `--prediction_num_future_steps`: Sets the number of future steps to predict.
- `--evaluate_prediction`: Enables computation and logging of accuracy metrics of the prediction component.
- `--visualize_prediction`: Enables visualization of predicted obstacle trajectories.

2.4.2 More information

See the reference for more information.

2.5 Localization

The package provides two types of localizations:

1. **Extended Kalman filter**: Fuses GNSS and IMU data for decimeter level accuracy in simulation.
2. **NDT Matching**: Pylot does not implement NDT matching, but instead provides and operator that bridges between the Autoware's NDT matching implentation and the pipeline. This operator can be used on real-world vehicles on which Autoware's NDT matching is deployed.

2.5.1 Important flags

- `--localization`: Enables the localization component of the pipeline. If this flag is `False`, then the pipeline uses the perfect localization it receives from the simulator.

2.5.2 More information

See the reference for more information.

2.6 HD Map

The package provides classes useful for interacting with HD map. The map can be either instantiated from an OpenDrive file, or received from the simulator.

When a HD Map is not available, Pylot attempts to build a naive map using its own components (e.g., lane detection, LiDAR). However, this feature is still in early stages, and not good enough for driving.

2.6.1 More information

See the reference for more information.

2.7 Planning

The package provides operators and classes useful for planning the trajectory the ego vehicle must follow. The planners ensure the ego-vehicle follows the lanes, decide when the ego-vehicle must brake to avoid collision with other agents and static obstacles, ensure that the ego-vehicle respects traffic lights.

Execute the following command to run a planning demo:

```
# To run the Frenet Optimal Trajectory planner.
python3 pylot.py --flagfile=configs/frenet_optimal_trajectory_planner.conf

# To run the RRT* planner.
python3 pylot.py --flagfile=configs/rrt_star_planner.conf

# To run the Hybrid A* planner.
python3 pylot.py --flagfile=configs/hybrid_astar_planner.conf
```

2.7.1 Important flags

- `--planning_type`: Sets which planner to use. Pylot currentlys offers four alternatives: waypoint following, Frenet optimal trajectory, RRT*, and Hybrid A*.
- `--visualize_waypoints`: Enables visualization of the waypoints computed by the planning operators.
- `--draw_waypoints_on_world`: Enables drawing of waypoints directly in the simulator.
- `--draw_waypoints_on_camera_frames`: Enables drawing of waypoints on camera frames.
- `--target_speed`: Sets the desired ego-vehicle target speed.
- `--num_waypoints_ahead`: Number of future waypoints to plan for.
- `--num_waypoints_behind`: Number of past waypoints to use while planning.
- `--obstacle_distance_threshold`: Obstacles that are more than this many meters away from the ego-vehicle are not going to be considered in the planner.

Note: Each planner has further other flags, which are not described here.

2.7.2 More information

See the reference for more information.

2.8 Control

The package provides operators and classes useful for controlling the ego vehicle. These operators ensure that the vehicle closely follows a sequence of waypoints sent by the planning component.

Execute the following command to run a demo of the MPC controller:

```
python3 pylot.py --flagfile=configs/mpc_agent.conf
```

Execute the following command to run a demo using solely the PID controller:

```
python3 pylot.py --flagfile=configs/e2e.conf
```

2.8.1 Important flags

- `--control`: Sets which control algorithm to use: Pylot currently offers three alternatives:
  1. mpc: An operator that implements a model predictive controller.
  2. pid: An operator that uses a PID controller to follow the waypoints.
  3. `simulator_auto_pilot`: The simulator controls the ego-vehicle, and drives it on a predefined path. The path differs depending on the spawning position.
- `--pid_p`: Sets the p parameter of the PID controller.
- `--pid_i`: Sets the i parameter of the PID controller.
- `--pid_d`: Sets the d parameter of the PID controller.

2.8.2 More information

See the reference for more information.

2.9 Visualization

2.9.1 Important flags

The following flags can be set to enable visualization of the output and state of the different Pylot components. The visualization is done in a pygame window, and users can switch between different views by pressing `n`.

- `--visualize_rgb_camera`: Enables the visualization of the camera.
- `--visualize_depth_camera`: Enables the visualization of the depth estimation.
- `--visualize_lidar`: Enables top down visualization of the LiDAR.
- `--visualize_detected_obstacles`: Enables visualization of detected obstacles.
- `--visualize_detected_traffic_lights`: Enables visualization of detected traffic lights.
- `--visualize_detected_lanes`: Enables visualization of detected lanes.
- `--visualize_tracked_obstacles`: Enables the visualization of tracked obstacles. The visualization includes info such as: id of the obstacle, distance from ego vehicle, label.
- `--visualize_segmentation`: Enables the visualization of segmented frames.

- `--visualize_waypoints`: Enables the visualization of the waypoints output by the planning component. These waypoints can be drawn on the camera frame (pass `-draw_waypoints_on_camera_frames`), or directly in the simulator when running in simulation mode (pass `--draw_waypoints_on_world`).
- `--visualize_world`: Enables visualization of the current state of the ego-vehicle. This is the best way to visualize what the self-driving car is currently perceiving and predicting. This visualization includes the past trajectories and predicted future trajectories of other agents, detected traffic lights and lanes, and the waypoints the ego-vehicle is trying to follow.

Flags that only work when running in simulation:

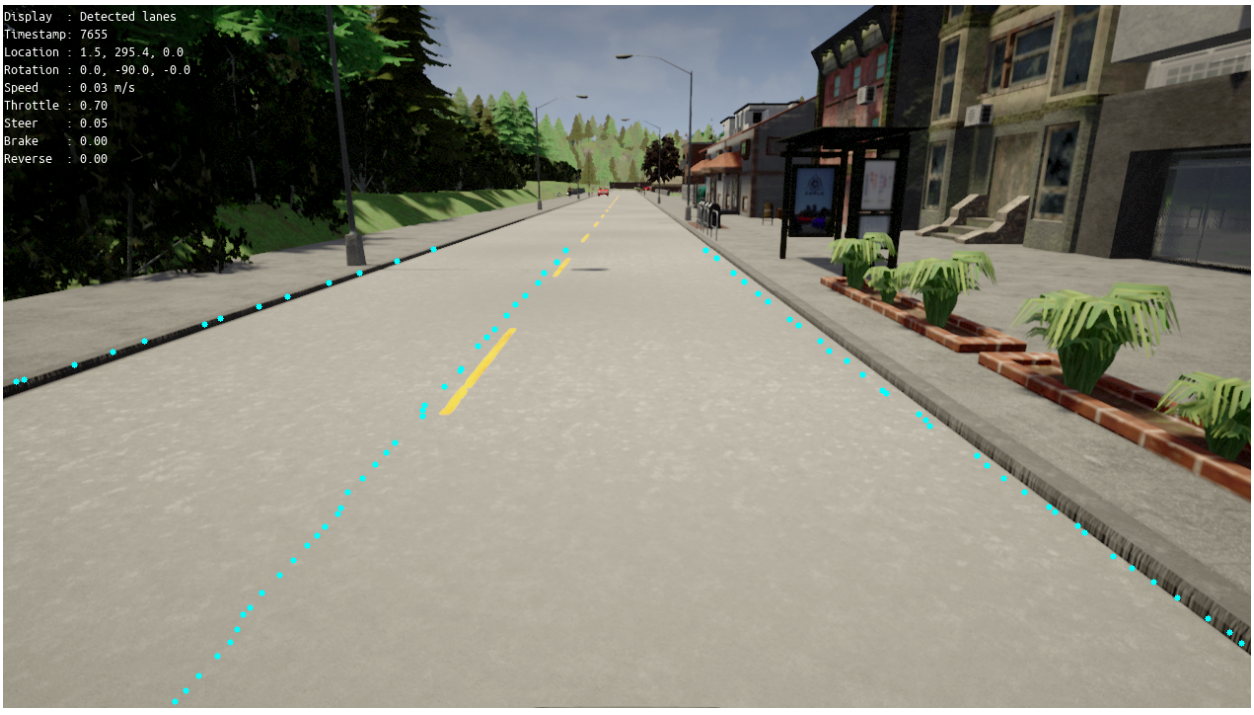
- `--visualize_imu`: Enables visualization of the IMU.
- `--visualize_pose`: Enables the visualization of the ego-vehicle pose.
- `--visualize_prediction`: Enables the visualization of obstacle predictions.

2.9.2 Examples

Obstacle detection:

Traffic light detection:

Lane detection:



Planning waypoints:

Planning world:

In this visualization the ego-vehicle is driving forward at 9.0 meters per second, the vehicle on the opposite lane is stationary, and the prediction component predicts that the predestrian will cross the street.

2.10 Simulation

2.10.1 Using the CARLA simulator

Important flags

- `--simulator_host`: Specifies the hostname where simulator is running.
- `--simulator_port`: Specifies the port on which the simulator server is listening.
- `--simulator_mode`:
- `--simulator_fps`: Specifies the frames per second the simulator must run at.
- `--simulator_town`: Specifies the simulator town to use.
- `--simulator_weather`: Sets the weather in the simulator.
- `--simulator_num_people`: Specifies the number of people agents to spawn.
- `--simulator_num_vehicles`: Specifies the number of vehicle agents to spawn.
- `--simulator_spawn_point_index`: Specifies the spawning location of the ego-vehicle.
- `--simulator_camera_frequency`: Specifies the frequency at which the cameras are publishing frames.
- `--simulator_gnss_frequency`: Specifies the frequency at which the GNSS sensor is publishing readings.
- `--simulator_imu_frequency`: Specifies the frequency at which the IMU sensor is publishing readings.
- `--simulator_lidar_frequency`: Specifies the frequency at which the LiDARs are publishing point clouds.
- `--simulator_localization_frequency`: Specifies the frequency at which pose messages are sent when using perfect localization.
- `--simulator_control_frequency`: Speicifes the frequenc at which control commands are applied.

2.10.2 Running scenarios

- `--scenario_runner`:

See the reference for more information.

2.11 Drivers

The package provides operators that implement drivers to interact with the CARLA simulator and with real-world hardware.

See the reference for more information.

2.12 Steps to deploy on a car

2.13 How to collect data

Pylot also provides a script for collecting CARLA data such as: RGB images, segmented images, obstacle 2D bounding boxes, depth frames, point clouds, traffic lights, obstacle trajectories, and data in Chauffeur format.

Run the following command to see what data you can collect, and which flags you must set:

```
python3 data_gatherer.py --help
```

Alternatively, you can inspect data_gatherer.conf for an example of a data collection setup.

2.14 Package Reference

2.14.1 Subpackages

pylot.control package

Subpackages

pylot.control.mpc package

Submodules

pylot.control.mpc.mpc_operator module

pylot.control.mpc.mpc module

pylot.control.mpc.utils module

Module contents

Submodules

pylot.control.control_eval_operator module

pylot.control.messages module

pylot.control.pid module

pylot.control.pid_control_operator module

pylot.control.utils module

Module contents

pylot.debug package

Submodules

pylot.debug.visualizer_operator module

Module contents

pylot.drivers package

Submodules

pylot.drivers.carla_camera_driver_operator module

pylot.drivers.carla_collision_sensor_operator module

pylot.drivers.carla_gnss_driver_operator module

pylot.drivers.carla_imu_driver_operator module

pylot.drivers.carla_lane_invasion_sensor_operator module

pylot.drivers.carla_lidar_driver_operator module

pylot.drivers.carla_traffic_light_invasion_sensor_operator module

pylot.drivers.drive_by_wire_operator module

pylot.drivers.grasshopper3_driver_operator module

pylot.drivers.sensor_setup module

pylot.drivers.velodyne_driver_operator module

Module contents

pylot.localization package

Submodules

pylot.localization.localization_operator module

pylot.localization.ndt_autoware_operator module

Module contents

Submodules

pylot.perception.detection.detection_decay_operator module

pylot.perception.detection.detection_operator module

pylot.perception.detection.efficientdet_operator module

pylot.perception.detection.detection_eval_operator module

pylot.perception.detection.lane module

pylot.perception.detection.lane_detection_canny_operator module

pylot.perception.detection.lanenet_detection_operator module

pylot.perception.detection.obstacle module

pylot.perception.detection.obstacle_location_finder_operator module

pylot.perception.detection.speed_limit_sign module

pylot.perception.detection.stop_sign module

pylot.perception.detection.traffic_light_det_operator module

pylot.perception.detection.traffic_light module

pylot.perception.detection.utils module

Module contents

pylot.perception.fusion package

Submodules

pylot.perception.fusion.fusion_operator module

pylot.perception.fusion.fusion_verification_operator module

Module contents

pylot.perception.segmentation package

Submodules

pylot.perception.segmentation.segmentation_decay_operator module

pylot.perception.segmentation.segmentation_drn_operator module

pylot.perception.segmentation.segmentation_eval_operator module

pylot.perception.segmentation.segmented_frame module

Module contents

pylot.perception.tracking package

Submodules

pylot.perception.tracking.da_siam_rpn_tracker module

pylot.perception.tracking.deep_sort_tracker module

pylot.perception.tracking.multi_object_tracker module

class pylot.perception.tracking.multi_object_tracker.**MultiObjectTracker**
Bases: object

reinitialize(*frame*, *obstacles*)
Reinitializes a multiple obstacle tracker.

Parameters

- **frame** – perception.camera_frame.CameraFrame to reinitialize with.
- **obstacles** – List of perception.detection.obstacle.Obstacle.

track(*frame*)
Tracks obstacles in a frame.

Parameters **frame** – perception.camera_frame.CameraFrame to track in.

pylot.perception.tracking.object_tracker_operator module

pylot.perception.tracking.obstacle_location_history_operator module

pylot.perception.tracking.obstacle_trajectory module

pylot.perception.tracking.sort_tracker module

pylot.perception.tracking.tracking_eval_operator module

Module contents

Submodules

pylot.planning.planning_operator module

pylot.planning.utils module

class pylot.planning.utils.**BehaviorPlannerState**
Bases: enum.Enum

States in which the FSM behavior planner can be in.

FOLLOW_WAYPOINTS = 0

READY = 1

KEEP_LANE = 2

PREPARE_LANE_CHANGE_LEFT = 3

LANGE_CHANGE_LEFT = 4

PREPARE_LANE_CHANGE_RIGHT = 5

LANE_CHANGE_RIGHT = 6

OVERTAKE = 7

pylot.planning.utils.**compute_person_speed_factor**(*ego_location_2d*, *person_location_2d*, *wp_vector*, *flags*, *logger*) → float

pylot.planning.utils.**compute_vehicle_speed_factor**(*ego_location_2d*, *vehicle_location_2d*, *wp_vector*, *flags*, *logger*) → float

pylot.planning.waypoints module

pylot.planning.world module

Module contents

pylot.prediction package

Submodules

pylot.prediction.linear_predictor_operator module

pylot.prediction.messages module

pylot.prediction.obstacle_prediction module

pylot.prediction.prediction_eval_operator module

pylot.prediction.utils module

Module contents

2.14.2 Submodules

2.14.3 pylot.component_creator module

2.14.4 pylot.operator_creator module

2.14.5 pylot.utils module

2.14.6 Module contents

Python Module Index

p

Index