PlaFoSim

Release 0.17.3

Julian Heinovski

Jan 11, 2024

CONTENTS

"Platoon Formation Simulator", or "PlaFoSim" for short, is an open source simulator for platoon formation, aiming for simplicity, flexibility, and scalability. PlaFoSim aims to facilitate and accelerate the research of platoon maneuvers and formation for individually driven vehicles. While the main focus of the simulator is on the assignment process, simulation of advertisements and maneuvers is implemented in a more abstract way.

PlaFoSim has been published at IEEE VNC 2021:

> Julian Heinovski, Dominik S. Buse and Falko Dressler, "Scalable Simulation of Platoon Formation Maneuvers with PlaFoSim," Proceedings of 13th IEEE Vehicular Networking Conference (VNC 2021), Poster Session, Virtual Conference, November 2021, pp. 137–138.

Note: This project is under active development.

CHAPTER

ONE

CITING

If you are working with PlaFoSim, we would appreciate a citation of our paper:

```bibtex
@inproceedings{heinovski2021scalable,
    author = {Heinovski, Julian and Buse, Dominik S. and Dressler, Falko},
    doi = {10.1109/VNC52810.2021.9644678},
    title = {{Scalable Simulation of Platoon Formation Maneuvers with PlaFoSim}},
    pages = {137--138},
    publisher = {IEEE},
    issn = {2157-9865},
    isbn = {978-1-66544-450-7},
    address = {Virtual Conference},
    booktitle = {13th IEEE Vehicular Networking Conference (VNC 2021), Poster Session},
    month = {11},
    year = {2021},
}
```

1.1 License

PlaFoSim is licensed under the terms of the GNU General Public License 3.0 or later.

```
# Copyright (c) 2020-2024 Julian Heinovski <heinovski@ccs-labs.org>
#
# SPDX-License-Identifier: GPL-3.0-or-later
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <https://www.gnu.org/licenses/>.
```

1.2 Introduction

The following documentation is based on our initial publication at IEEE VNC 2021.

1.2.1 About

PlaFoSim is a simulator for platoon formation on freeways. It is implemented as an integrated, stand-alone command-line tool based on Python 3 and only a few additional libraries. The focus of the simulation is on vehicle-to-platoon assignment algorithms. Thus, platooning mobility, wireless communication, and platooning maneuvers are implemented in a more abstract way. PlaFoSim enables fast integration of new algorithms, easy configuration via command line arguments, and fast simulation of large-scale scenarios. For easy inspection, debugging, and show-casing, PlaFoSim uses Sumo to integrate a live GUI. PlaFoSim can record statistics such as vehicle speeds and positions, fuel consumption, and emissions, as well as platoon formation data, platoon sizes, and performed maneuvers. Produced traces can later be visualized in the GUI or processed with external tools.

1.2.2 Architecture

PlaFoSim is based on the concept for microscopic simulation of multi-lane traffic proposed by Krauß. Thus, its main simulation loop consist of the following steps: (1) insert new vehicles, (2) trigger actions (e.g., running formation algorithms, recording statistics), (3) execute lane changes, (4) execute car- following models to update vehicle speeds, (5) update vehicle positions, and (6) perform collision checks among all vehicles. Afterwards, PlaFoSim updates the live GUI, records further statistics, and advances the simulation time. These steps are subject to a discrete characteristic of the simulator.

1.2.3 Scenario & Simulation Configuration

PlaFoSim focuses on simulating long freeways with multiple lanes and periodic on-/off-ramps, which are used by the vehicles to enter and leave the freeway. The road network (e.g., length, number of lanes) as well as simulation time and steps can be configured via parameters. Vehicle influx is configured in a macroscopic way via (1) a flow with constant insertion (e.g., by departure rate) or (2) a constant number of vehicles. Independent of the method, vehicles drive trips of configurable length between on-/off-ramps. The desired driving speed of a vehicle can be sampled from a normal distribution. A penetration rate chooses between human driving and platooning. To reduce effects from the initial transient period, PlaFoSim allows pre-filling the scenario according to a target density (i.e., vehicles per km and lane) before the simulation starts. Deterministic variation of stochastic behavior can be achieved through a random seed.

1.2.4 Vehicle Dynamics

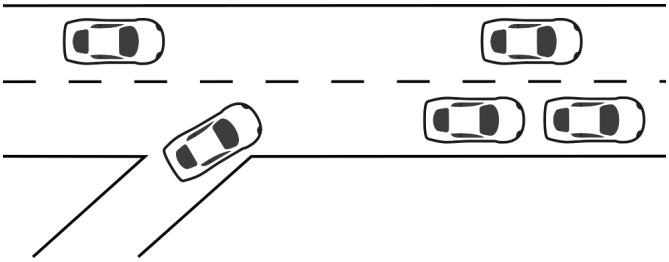
Longitudinal (speed) and lateral (lane) control of vehicles follow the models also used in Sumo and Plexe. This facilitates the validation of the results. It also enables a migration path for porting promising platoon formation strategies from PlaFoSim to these simulators with higher levels of detail. Multiple car following models are implemented for longitudinal control: A Krauss model for human-driven vehicles, Adaptive Cruise Control (ACC) similar to Plexe, and CACC for platooning (see Vehicle Dynamics and Control). CACC abstracts communication and platoon control by transferring the speed of the leader vehicle to all its followers without delay. Lateral control for lane changing is also modeled following the Sumo implementation. Vehicles aim to stay on the rightmost lane, but can perform multi-lane overtake maneuvers if needed. Platoons can also perform overtake maneuvers by executing lane-changes simultaneously.

1.2.5 Platoon Formation

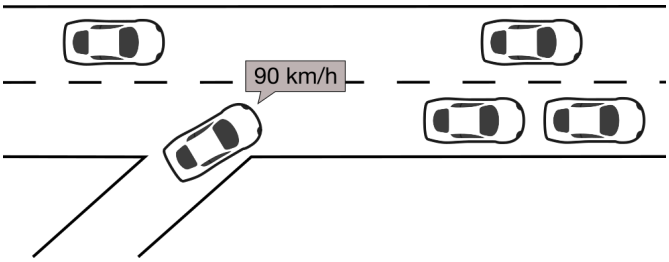
Conceptual Overview

A conceptional overview of the process of Platoon Formation that is employed in PlaFoSim:

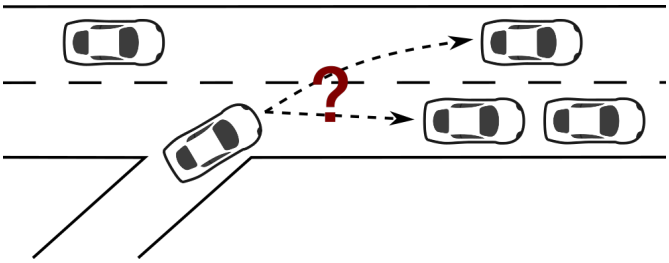
1. Scenario: A new vehicle enters the highway.



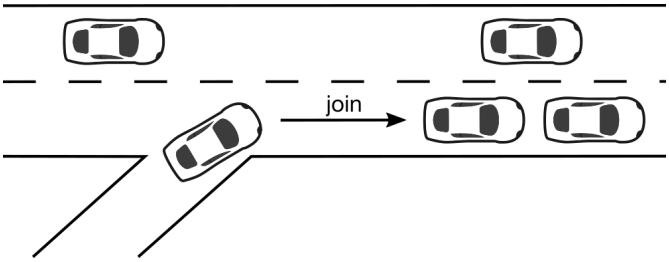
2. Advertisement: The vehicle advertises itself as interested in Platooning.



3. Assignment: A Vehicle-to-Platoon assignment is computed.



4. Maneuver: The new vehicle performs a join maneuver.



Implementation in PlaFoSim

Platoon formation is performed within actions, either by individual vehicles or centrally coordinated. Actions in general are triggered at every simulation step but the period for executing the platoon formation algorithms can be bigger (e.g., 60 s). Forming platoons typically consist of three steps: (1) data collection of available vehicles and platoons, (2) computation of vehicle-to-platoon assignments, (3) execution of maneuvers. PlaFoSim simplifies some driving maneuvers and communication details in order to boost simulation performance in extensive scenarios. Vehicles know other vehicles within a certain range whereas central coordinators have complete knowledge about the scenario. Platoon formation algorithms utilize the available knowledge to compute optimal vehicle-to-platoon assignments. Desired assignments are fulfilled by join maneuvers which teleport vehicles to their designated platoon position, considering the approximate time for approaching. New algorithms and corresponding parameters can be quickly integrated into the existing code.

1.2.6 Included Console Scripts

PlaFoSim includes the following console scripts

- *plafosim*: The actual simulator for platoon formation
- *plafosim-replay*: A tool to replay simulation traces in a GUI
- *plafosim-img2video*: A tool to create a video from continuous screenshots

1.2.7 Further Reading

- https://www.tkn.tu-berlin.de/bib/heinovski2021scalable/
- https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/
- https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

1.3 Usage

1.3.1 Installation

- Install Python (>=3.7,<=3.9)
- Install PlaFoSim from pypi: `pip install plafosim`

NOTE: The project is developed and tested only on Linux.

1.3.2 Running a Simulation

NOTE: Since PlaFoSim is a command-line interface (CLI) application, running PlaFoSim requires using a shell.

Quickstart

Run PlaFoSim with:

```
plafosim
```

You can use PlaFoSim's help to get a list of available parameters:

```
plafosim -h, --help
```

Or, run a simulation with the default configuration (`-d`):

```
plafosim -d
```

Live GUI

You can visualize the simulation via a simple live GUI based on sumo-gui, using the argument `gui`:

```
plafosim --gui
```

docs/gui.png *A screenshot of PlaFoSim's live GUI showing 2 platoons and various individual vehicles. Copyright © 2021 IEEE.*

More options for the live GUI can be found within the `GUI properties` section of the help.

NOTE: This requires installation of SUMO (>=1.6.0) and declaration of the `SUMO_HOME` variable (see documentation).

Advanced Simulation Control

You can use a variety of different parameters to customize the scenario and the simulation itself. E.g., use the parameter `vehicles` to configure the number of vehicles in the simulation:

```
plafosim --vehicles 1000
```

The available parameters are grouped into different categories:

```
- road network properties
- vehicle properties
- trip properties
- communication properties
- platoon properties
- formation properties
- infrastructure properties
- simulation properties
- GUI properties
- result recording properties
```

You can find a list of available parameters for each category in the help:

```
plafosim -h, --help
```

Examples

```
# Configure a 100km freeway with ramps at every 10km
plafosim --road-length 100 --ramp-interval 10

# Configure random (normally distributed) desired driving speed of 130km/h
plafosim --random-desired-speed true --desired-speed 36

# Configure random trips for 500 vehicles
plafosim --vehicles 500 --random-depart-position true --random-arrival-position true --
↪depart-desired true

# Pre fill the freeway with 1000 vehicles
plafosim --vehicles 1000 --pre-fill true

# Configure 50% of the vehicles with Advanced Cruise Control (ACC) and a headway time of␣
↪1.5s
plafosim --penetration 0.5 --acc-headway-time 1.5

# Enable a simple, distributed platoon formation algorithm [1] in order to form platoons␣
↪every 30s
plafosim --formation-algorithm SpeedPosition --formation-strategy distributed --
↪execution-interval 30
```

Faster Simulation

You can speed up the simulation performance by enabling Python's optimization `PYTHONOPTIMIZE`, e.g., in order to disable assertions:

```
PYTHONOPTIMIZE=1 plafosim
```

See the Python documention for more details.

1.3.3 Re-Playing a Simulation

The simulation can write a trace file including the mobility details of every simulated vehicle (default `results_vehicle_traces.csv`). You can replay the simulation in the GUI (see above) based on the trace file by using the corresponding binary:

```
plafosim-replay results_vehicle_traces.csv
```

To see all options of this script, run:

```
plafosim-replay -h, --help
```

NOTE: This requires installation of SUMO (>=1.6.0) and declaration of the `SUMO_HOME` variable.

1.3.4 Recording of Screenshots and Video from a Simulation

PlaFoSim offers functionality to automatically record a screenshot of the GUI in every simulation step with

`plafosim --gui --screenshot-file screenshot`

These screenshots can be used to create a video by using an integrated script based on ffmpeg:

`plafosim-img2video 'screenshot_*.png' video.mp4`

NOTE: This requires installation of ffmpeg.

1.4 Extending PlaFoSim

You can extend and customize PlaFoSim flexibly by modifying its source code. For this, you first need ot install it from source.

1.4.1 Installing from Source

- Install poetry: `pip install poetry`
- Clone this repository: `git clone https://github.com/heinovski/plafosim.git`
- Navigate to newly created directory of the cloned repository in the command-line
- Install PlaFoSim from source in editable mode: `poetry install`
- Run PlaFoSim in the virtual environment: `poetry run plafosim` You can also activate the virtual enviroment first with `poetry shell` and run the commands as usual (see above).

1.4.2 Adding a new Formation Algorithm

In order to add a new formation algorithm, you need to follow these steps:

- Create a new sub-class of `FormationAlgorithm` (see `formation_algorithm.py`) within the sub-directory `algorithms`. You can use the `Dummy` algorithm (see `algorithms/dummy.py`) as an example.
- Add specific arguments for your algorithm to the argument parser group within the new sub-class if necessary.

You should now be able to use your new algorithm with

```
plafosim --formation-algorithm dummy_algorithm_name
```

1.5 Contributing to the Project

In order to contribute, please follow these steps:

- Install PlaFoSim from source (see Extending PlaFoSim)
- Make desired changes and adjust the documentation if required
- Run the tests located in `tests` as well as the validation scripts located in `scripts` (see `.drone.yml` for details)
- Submit a Pull Request (PR)

1.5.1 Documenting

When making changes to the code, make sure to add or adjust corresponding documentation in form of python docstrings. Those should use the Numpy docstring format for Sphinx and follow the style guide by numpydoc to ensure consistency and compatibility. See also the style guide by pandas. You can build the documentation using Sphinx `make -C docs`.

1.5.2 Testing

When adding methods and functions, make sure to add corresponding unit tests for `py.test`. The tests are located under `tests` and can be executed with `./scripts/run-pytest.sh`. This will also generate a test coverage report.

1.5.3 Validation

To validate the behavior of PlaFoSim, it is compared to SUMO 1.6.0 by means of simulation results (e.g., vehicle traces). The corresponding scripts are located under `scripts` and executed withn CI/CD pipelines. You can have a look at `.drone.yml` for details regarding the execution.

1.5.4 Profiling

You can profile the runtime of PlaFoSim's code by using cProfile:

```
poetry run python -m cProfile -o profile.out -m plafosim.cli.plafosim
```

You can visualize the results of the profiling run by using SnakeViz:

```
snakeviz profile.out
```

1.6 plafosim

1.6.1 plafosim package

Subpackages

plafosim.algorithms package

Submodules

plafosim.algorithms.dummy module

class plafosim.algorithms.dummy.**Dummy**(*owner: object*, *dummy: int = -1*, ***kw_args*)

Bases: *FormationAlgorithm*

Dummy Platoon Formation Algorithm.

__init__(*owner: object*, *dummy: int = -1*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **dummy** (*int, optional*) – The value for the dummy parameter

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

do_formation()

Run platoon formation algorithm to search for a platooning opportunity and perform the corresponding join maneuver.

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.algorithms.dummy.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters
owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns
The specific argument group for this algorithm.

Return type
argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.algorithms.speed_position module

class plafosim.algorithms.speed_position.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters

owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns

The specific argument group for this algorithm.

Return type

argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.algorithms.speed_position.**PlatoonRole**(*value*)

Bases: *Enum*

A collection of available platoon roles.

FOLLOWER = 2

JOINER = 3

LEADER = 1

LEAVER = 4

NONE = 0

class plafosim.algorithms.speed_position.**SpeedPosition**(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Bases: *FormationAlgorithm*

Platoon Formation Algorithm based on Similarity, considering Speed and Position.

See papers

Julian Heinovski and Falko Dressler, "Where to Decide? Centralized vs. Distributed Vehicle Assignment for Platoon Formation," arXiv, cs.MA, 2310.09580, October 2023. https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

and

Julian Heinovski and Falko Dressler, "Platoon Formation: Optimized Car to Platoon Assignment Strategies and Protocols," Proceedings of 10th IEEE Vehicular Networking Conference (VNC 2018), Taipei, Taiwan, December 2018. https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/

__init__(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **alpha** (*float*) – The weighting factor alpha
- **speed_deviation_threshold** (*float*) – The threshold for speed deviation
- **position_deviation_threshold** (*int*) – The threshold for position deviation
- **formation_centralized_kind** (*str*) – TODO
- **solver_time_limit** (*int*) – The time limit in s to apply to the solver
- **record_solver_traces** (*bool*) – Whether to record continuous solver traces
- **record_infrastructure_assignments** (*bool*) – Whether to record infrastructure assignments

_do_formation_centralized()

Run centralized greedy formation approach.

This selects candidates and triggers join maneuvers.

_do_formation_distributed()

Run distributed greedy formation approach.

This selects a candidate and triggers a join maneuver.

_do_formation_optimal()

Run centralized optimal formation approach.

This selects candidates and triggers join maneuvers.

_record_infrastructure_assignments(*basename: str*)

Record infrastructure assignments.

Parameters
basename (*str*) – The basename of the result file

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

cost_speed_position(*ds: float*, *dp: float*) → float

Return the overall cost (i.e., the weighted deviation) for a candidate.

Parameters

- **ds** (*float*) – The deviation in speed
- **dp** (*int*) – The deviation in position

Returns
The weighted relative deviation

Return type
float

do_formation()

Run platoon formation algorithms to search for a platooning opportunity and perform the corresponding join maneuver.

dp(*vehicle: PlatooningVehicle*, *platoon: Platoon*) → float

Return the deviation in position from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in position

Return type
float

ds(*vehicle: PlatooningVehicle*, *platoon: Platoon*) → float

Return the deviation in speed from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in speed

Return type
float

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.algorithms.speed_position.**initialize_infrastructure_assignments**(*basename: str*)

Initialize the infrastructure assignments result file.

Parameters
basename (*str*) – The basename of the result file

plafosim.algorithms.speed_position.**initialize_solver_traces**(*basename: str*)

Initialize the solver trace file.

Parameters
basename (*str*) – The basename of the trace file

plafosim.algorithms.speed_position.**record_solver_trace**(*basename: str*, *step: float*, *iid: int*, *variables: int*, *constraints: int*, *run_time: float*, *result_status: int*, *solution_value: float*, *best_bound: float*, *solution_quality: float*)

Record one line in the solver trace file.

Parameters

- **basename** (*str*) – The basename of the trace file
- **step** (*float*) – The current simulation step
- **iid** (*int*) – The id of the infrastructure which executed the solver run
- **variables** (*int*) – The number of variables
- **constraints** (*int*) – The number of constraints
- **run_time** (*float*) – The run time of the solver
- **result_status** (*int*) – The result status of the solver
- **solution_value** (*float*) – The solution value of the solver
- **best_bound** (*float*) – The best bound of the problem
- **solution_quality** (*float*) – The quality of the solution

`plafosim.algorithms.speed_position.strtobool(val)`

Convert a string representation of truth to true (1) or false (0).

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'val' is anything else.

`plafosim.algorithms.speed_position.timer()`

perf_counter() -> float

Performance counter for benchmarking.

Module contents

class `plafosim.algorithms.Dummy(owner: object, dummy: int = -1, **kw_args)`

Bases: *FormationAlgorithm*

Dummy Platoon Formation Algorithm.

__init__(*owner: object*, *dummy: int = -1*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **dummy** (*int, optional*) – The value for the dummy parameter

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

do_formation()

Run platoon formation algorithm to search for a platooning opportunity and perform the corresponding join maneuver.

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class `plafosim.algorithms.FormationAlgorithm(owner: object)`

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters

owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns

The specific argument group for this algorithm.

Return type

argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.algorithms.**SpeedPosition**(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Bases: *FormationAlgorithm*

Platoon Formation Algorithm based on Similarity, considering Speed and Position.

See papers

Julian Heinovski and Falko Dressler, "Where to Decide? Centralized vs. Distributed Vehicle Assignment for Platoon Formation," arXiv, cs.MA, 2310.09580, October 2023. https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

and

Julian Heinovski and Falko Dressler, "Platoon Formation: Optimized Car to Platoon Assignment Strategies and Protocols," Proceedings of 10th IEEE Vehicular Networking Conference (VNC 2018), Taipei, Taiwan, December 2018. https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/

__init__(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

- **alpha** (*float*) – The weighting factor alpha
- **speed_deviation_threshold** (*float*) – The threshold for speed deviation
- **position_deviation_threshold** (*int*) – The threshold for position deviation
- **formation_centralized_kind** (*str*) – TODO
- **solver_time_limit** (*int*) – The time limit in s to apply to the solver
- **record_solver_traces** (*bool*) – Whether to record continuous solver traces
- **record_infrastructure_assignments** (*bool*) – Whether to record infrastructure assignments

_do_formation_centralized()

Run centralized greedy formation approach.

This selects candidates and triggers join maneuvers.

_do_formation_distributed()

Run distributed greedy formation approach.

This selects a candidate and triggers a join maneuver.

_do_formation_optimal()

Run centralized optimal formation approach.

This selects candidates and triggers join maneuvers.

_record_infrastructure_assignments(*basename: str*)

Record infrastructure assignments.

Parameters
basename (*str*) – The basename of the result file

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

cost_speed_position(*ds: float*, *dp: float*) → float

Return the overall cost (i.e., the weighted deviation) for a candidate.

Parameters

- **ds** (*float*) – The deviation in speed
- **dp** (*int*) – The deviation in position

Returns
The weighted relative deviation

Return type
float

do_formation()

Run platoon formation algorithms to search for a platooning opportunity and perform the corresponding join maneuver.

dp(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in position from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in position

Return type
float

ds(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in speed from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in speed

Return type
float

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.algorithms.**attribute()**

perf_counter() -> float

Performance counter for benchmarking.

plafosim.algorithms.**import_module**(*name*, *package=None*)

Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

plafosim.algorithms.**isclass**(*object*)

Return true if the object is a class.

Class objects provide these attributes:
__doc__ documentation string __module__ name of module in which this class was defined

`plafosim.algorithms.iter_modules(path=None, prefix='')`

Yields ModuleInfo for all submodules on path, or, if path is None, all top-level modules on sys.path.

'path' should be either None or a list of paths to look for modules in.

'prefix' is a string to output on the front of every module name on output.

plafosim.cli package

Submodules

plafosim.cli.img2video module

`plafosim.cli.img2video.main()`

The main entry point of PlaFoSim's img2video script.

This script uses ffmpeg.

plafosim.cli.plafosim module

class `plafosim.cli.plafosim.CustomFormatter(prog, indent_increment=2, max_help_position=24, width=None)`

Bases: `ArgumentDefaultsHelpFormatter`, `RawDescriptionHelpFormatter`, `MetavarTypeHelpFormatter`

Metaclass combining multiple formatter classes for argparse.

class `_Section(formatter, parent, heading=None)`

Bases: `object`

`__init__(formatter, parent, heading=None)`

`format_help()`

`__init__(prog, indent_increment=2, max_help_position=24, width=None)`

`_add_item(func, args)`

`_dedent()`

`_expand_help(action)`

`_fill_text(text, width, indent)`

`_format_action(action)`

`_format_action_invocation(action)`

`_format_actions_usage(actions, groups)`

`_format_args(action, default_metavar)`

`_format_text(text)`

`_format_usage(usage, actions, groups, prefix)`

_get_default_metavar_for_optional(*action*)

_get_default_metavar_for_positional(*action*)

_get_help_string(*action*)

_indent()

_iter_indented_subactions(*action*)

_join_parts(*part_strings*)

_metavar_formatter(*action*, *default_metavar*)

_split_lines(*text*, *width*)

add_argument(*action*)

add_arguments(*actions*)

add_text(*text*)

add_usage(*usage*, *actions*, *groups*, *prefix=None*)

end_section()

format_help()

start_section(*heading*)

class plafosim.cli.plafosim.**Dummy**(*owner: object*, *dummy: int = -1*, ***kw_args*)

Bases: *FormationAlgorithm*

Dummy Platoon Formation Algorithm.

__init__(*owner: object*, *dummy: int = -1*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **dummy** (*int, optional*) – The value for the dummy parameter

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

do_formation()

Run platoon formation algorithm to search for a platooning opportunity and perform the corresponding join maneuver.

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.cli.plafosim.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters

owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns

The specific argument group for this algorithm.

Return type

argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.cli.plafosim.**Simulator**(*, *road_length: int = 100000*, *number_of_lanes: int = 3*, *ramp_interval: int = 5000*, *pre_fill: bool = False*, *number_of_vehicles: int = 100*, *vehicle_density: float = -1*, *max_speed: float = 55*, *acc_headway_time: float = 1.0*, *cacc_spacing: float = 5.0*, *penetration_rate: float = 1.0*, *random_depart_position: bool = False*, *depart_all_lanes: bool = True*, *desired_speed: float = 33.0*, *random_desired_speed: bool = True*, *speed_variation: float = 0.1*, *min_desired_speed: float = 22.0*, *max_desired_speed: float = 44.0*, *random_depart_speed: bool = False*, *depart_desired: bool = False*, *depart_flow: bool = False*, *depart_method: str = 'interval'*, *depart_interval: float = 2.0*, *depart_probability: float = 1.0*, *depart_rate: int = 3600*, *random_arrival_position: bool = False*, *minimum_trip_length: int = 0*, *maximum_trip_length: int = -1000*, *communication_range: int = 500*, *distributed_platoon_knowledge: bool = True*, *distributed_maneuver_knowledge: bool = False*, *start_as_platoon: bool = False*, *reduced_air_drag: bool = True*, *maximum_teleport_distance: int = 2000*, *maximum_approach_time: int = 60*, *delay_teleports: bool = True*, *update_desired_speed: bool = True*, *formation_algorithm: str | None = None*, *formation_strategy: str = 'distributed'*, *execution_interval: int = 10*, *number_of_infrastructures: int = 0*, *step_length: float = 1.0*, *max_step: int = 3600*, *actions: bool = True*, *collisions: bool = True*, *random_seed: int = -1*, *log_level: int = 30*, *progress: bool = True*, *gui: bool = False*, *gui_delay: int = 0*, *gui_track_vehicle: int = -1*, *sumo_config: str = 'sumocfg/freeway.sumo.cfg'*, *gui_play: int = True*, *gui_start: int = 0*, *draw_ramps: bool = True*, *draw_ramp_labels: bool = False*, *draw_road_end: bool = True*, *draw_road_end_label: bool = True*, *draw_infrastructures: bool = True*, *draw_infrastructure_labels: bool = True*, *screenshot_filename: str | None = None*, *result_base_filename: str = 'results'*, *record_simulation_trace: bool = False*, *record_end_trace: bool = True*, *record_vehicle_trips: bool = False*, *record_vehicle_emissions: bool = False*, *record_vehicle_traces: bool = False*, *record_vehicle_changes: bool = False*, *record_emission_traces: bool = False*, *record_platoon_trips: bool = False*, *record_platoon_maneuvers: bool = False*, *record_platoon_formation: bool = False*, *record_platoon_traces: bool = False*, *record_vehicle_platoon_traces: bool = False*, *record_platoon_changes: bool = False*, *record_infrastructure_assignments: bool = False*, *record_vehicle_teleports: bool = False*, *record_prefilled: bool = False*, ***kwargs: dict*)

Bases: `object`

A collection of parameters and information of the simulator.

__init__(*, *road_length: int = 100000*, *number_of_lanes: int = 3*, *ramp_interval: int = 5000*, *pre_fill: bool = False*, *number_of_vehicles: int = 100*, *vehicle_density: float = -1*, *max_speed: float = 55*, *acc_headway_time: float = 1.0*, *cacc_spacing: float = 5.0*, *penetration_rate: float = 1.0*, *random_depart_position: bool = False*, *depart_all_lanes: bool = True*, *desired_speed: float = 33.0*, *random_desired_speed: bool = True*, *speed_variation: float = 0.1*, *min_desired_speed: float = 22.0*, *max_desired_speed: float = 44.0*, *random_depart_speed: bool = False*, *depart_desired: bool = False*, *depart_flow: bool = False*, *depart_method: str = 'interval'*, *depart_interval: float = 2.0*, *depart_probability: float = 1.0*, *depart_rate: int = 3600*, *random_arrival_position: bool = False*, *minimum_trip_length: int = 0*, *maximum_trip_length: int = -1000*, *communication_range: int = 500*, *distributed_platoon_knowledge: bool = True*, *distributed_maneuver_knowledge: bool = False*, *start_as_platoon: bool = False*, *reduced_air_drag: bool = True*, *maximum_teleport_distance: int = 2000*, *maximum_approach_time: int = 60*, *delay_teleports: bool = True*, *update_desired_speed: bool = True*, *formation_algorithm: str | None = None*, *formation_strategy: str = 'distributed'*, *execution_interval: int = 10*, *number_of_infrastructures: int = 0*, *step_length: float = 1.0*, *max_step: int = 3600*, *actions: bool = True*, *collisions: bool = True*, *random_seed: int = -1*, *log_level: int = 30*, *progress: bool = True*, *gui: bool = False*, *gui_delay: int = 0*, *gui_track_vehicle: int = -1*, *sumo_config: str = 'sumocfg/freeway.sumo.cfg'*, *gui_play: int = True*, *gui_start: int = 0*, *draw_ramps: bool = True*, *draw_ramp_labels: bool = False*, *draw_road_end: bool = True*, *draw_road_end_label: bool = True*, *draw_infrastructures: bool = True*, *draw_infrastructure_labels: bool = True*, *screenshot_filename: str | None = None*, *result_base_filename: str = 'results'*, *record_simulation_trace: bool = False*, *record_end_trace: bool = True*, *record_vehicle_trips: bool = False*, *record_vehicle_emissions: bool = False*, *record_vehicle_traces: bool = False*, *record_vehicle_changes: bool = False*, *record_emission_traces: bool = False*, *record_platoon_trips: bool = False*, *record_platoon_maneuvers: bool = False*, *record_platoon_formation: bool = False*, *record_platoon_traces: bool = False*, *record_vehicle_platoon_traces: bool = False*, *record_platoon_changes: bool = False*, *record_infrastructure_assignments: bool = False*, *record_vehicle_teleports: bool = False*, *record_prefilled: bool = False*, ***kwargs: dict*)

Initialize a simulator instance.

_add_vehicle(*vid: int*, *vtype:* VehicleType, *depart_position: float*, *arrival_position: float*, *desired_speed: float*, *depart_lane: int*, *depart_speed: float*, *depart_time: float*, *depart_delay: float = 0*, *communication_range: int = 500*, *pre_filled: bool = False*) → *Vehicle*

Add a vehicle to the simulation based on the given parameters.

NOTE: Make sure that you set last_vehicle_id correctly.

Parameters

- **vid** (*int*) – The id of the vehicle
- **vtype** (VehicleType) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float, optional*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int, optional*) – The maximum communication range of the vehicle

- **pre_filled** (*bool, optional*) – Whether this vehicle was pre-filled

Returns
The added vehicle

Return type
Vehicle

_call_infrastructure_actions()

Triggers actions on all infrastructures in the simulation.

_call_vehicle_actions()

Triggers actions on all vehicles in the simulation.

_finish()

Clean up the simulation.

_generate_infrastructures(*number_of_infrastructures: int*)

Generate infrastructures for the simulation.

Parameters
number_of_infrastructures (*int*) – The number of infrastructures to generate

_generate_vehicles()

Add pre-filled vehicles to the simulation.

_get_predecessor(*vehicle:* Vehicle, *lane: int = -1*) → *Vehicle*

Return the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_predecessor_rear_position(*vehicle:* Vehicle, *lane: int = -1*) → float

Return the rear position of the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_predecessor_speed(*vehicle:* Vehicle, *lane: int = -1*) → float

Return the speed of the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_successor(*vehicle:* Vehicle, *lane: int = -1*) → *Vehicle*

Return the succeeding (i.e., back) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider

- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_vehicles_df() → DataFrame

Return a pandas Dataframe from the internal data structure.

Returns
The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Return type
pandas.DataFrame

_initialize_gui()

Initialize the GUI.

_initialize_prefilled_platoon()

Initialize all pre-filled vehicles as one platoon.

_initialize_result_recording()

Create output files for all (enabled) statistics and writes the headers.

_record_lane_changes(*vdf: DataFrame*)

Record lane changes.

Parameters
vdf (*pd.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

_remove_arrived_vehicles(*arrived_vehicles: list*)

Remove arrived vehicles from the simulation.

Parameters
arrived_vehicles (*list*) – The ids of arrived vehicles

_spawn_vehicles(*vdf: DataFrame*)

Spawns vehicles within the current step.

1) Calculate how many vehicles should be spawned according to the departure method
2) Calculate properties for these vehicles (e.g., desired speed)
3) Add vehicles to spawn queue
4) Spawn as many vehicles as possible from the queue (sorted by waiting time)
5) Update queue

_statistics(*vehicles_in_simulator: int*, *vehicles_in_queue: int*, *vehicles_spawned: int*, *vehicles_arrived: int*, *runtime: float*, *average_vehicle_speed: float*, *vehicles_braking_rough: int*)

Record some period statistics.

Parameters

- **vehicles_in_simulator** (*int*) – The number of vehicles in the scenario within this step
- **vehicles_in_queue** (*int*) – The number of vehicles in the spawn queue within this step
- **vehicles_spawned** (*int*) – The number of vehicles that departed within this step
- **vehicles_arrived** (*int*) – The number of vehicles that arrived within this step
- **runtime** (*float*) – The run time of this step

- **average_vehicle_speed** (*int*) – The average driving speed amog all vehicles in the scenario within this step
- **vehicles_braking_rough** (*int*) – The number of vehicles performing rough braking within this step

_update_gui()

Update the GUI.

_vehicles_to_be_scheduled() → int

Calculate how many vehicles should be spawned according to the departure method.

Returns
int

Return type
The number of vehicles to be spawned within this time step

_write_back_vehicles_df(*vdf: DataFrame*)

Write back the vehicle updates from a given pandas dataframe to the internal data structure.

Parameters
vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

run()

Run the simulation with the specified parameters until it is stopped.

Main simulation method.

This is based on Krauss' multi lane traffic: laneChange(); adjust(); move();

stop(*msg: str*)

Stop the simulation with the given message.

Parameters
msg (*str*) – The message to show after stopping the simulation

property number_of_lanes: int

Return the number of lanes.

property road_length: int

Return the road length in m.

property step: int

Return the current simulation step.

property step_length: float

Return the length of a simulation step.

class plafosim.cli.plafosim.**SpeedPosition**(*owner: object, alpha: float = 0.5, speed_deviation_threshold: float = 0.2, position_deviation_threshold: int = 1000, formation_centralized_kind: str = 'greedy', solver_time_limit: int = 60, record_solver_traces: bool = False, record_infrastructure_assignments: bool = False, **kw_args*)

Bases: *FormationAlgorithm*

Platoon Formation Algorithm based on Similarity, considering Speed and Position.

See papers

Julian Heinovski and Falko Dressler, "Where to Decide? Centralized vs. Distributed Vehicle Assignment for Platoon Formation," arXiv, cs.MA, 2310.09580, October 2023. https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

and

Julian Heinovski and Falko Dressler, "Platoon Formation: Optimized Car to Platoon Assignment Strategies and Protocols," Proceedings of 10th IEEE Vehicular Networking Conference (VNC 2018), Taipei, Taiwan, December 2018. https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/

__init__(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **alpha** (*float*) – The weighting factor alpha
- **speed_deviation_threshold** (*float*) – The threshold for speed deviation
- **position_deviation_threshold** (*int*) – The threshold for position deviation
- **formation_centralized_kind** (*str*) – TODO
- **solver_time_limit** (*int*) – The time limit in s to apply to the solver
- **record_solver_traces** (*bool*) – Whether to record continuous solver traces
- **record_infrastructure_assignments** (*bool*) – Whether to record infrastructure assignments

_do_formation_centralized()

Run centralized greedy formation approach.

This selects candidates and triggers join maneuvers.

_do_formation_distributed()

Run distributed greedy formation approach.

This selects a candidate and triggers a join maneuver.

_do_formation_optimal()

Run centralized optimal formation approach.

This selects candidates and triggers join maneuvers.

_record_infrastructure_assignments(*basename: str*)

Record infrastructure assignments.

Parameters

basename (*str*) – The basename of the result file

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters

parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

`cost_speed_position`(*ds: float*, *dp: float*) → float

Return the overall cost (i.e., the weighted deviation) for a candidate.

Parameters

- **ds** (*float*) – The deviation in speed
- **dp** (*int*) – The deviation in position

Returns
The weighted relative deviation

Return type
float

`do_formation`()

Run platoon formation algorithms to search for a platooning opportunity and perform the corresponding join maneuver.

`dp`(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in position from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in position

Return type
float

`ds`(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in speed from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in speed

Return type
float

`finish`()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.cli.plafosim.**attribute**()

perf_counter() -> float

Performance counter for benchmarking.

plafosim.cli.plafosim.**create_simulator**(***kwargs: dict*) → *Simulator*

Create a simulator object from given keyword arguments.

Parameters
kwargs (*dict*) – The dictionary of keyword arguments to use for the creation

Returns
Simulator

Return type
The created simulator object

plafosim.cli.plafosim.**find_resource**(*path: str*) → str

Find the resouces under relpath locally or as a packaged resource.

Parameters
path (*str*) – The path to search for the ressource

Returns
str

Return type
The path of the resource

plafosim.cli.plafosim.**format_help**(*parser: ArgumentParser*, *groups=None*) → str

Format help message for argument groups.

Taken from https://stackoverflow.com/a/40730878.

plafosim.cli.plafosim.**import_module**(*name*, *package=None*)

Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

plafosim.cli.plafosim.**isclass**(*object*)

Return true if the object is a class.

Class objects provide these attributes:
__doc__ documentation string __module__ name of module in which this class was defined

plafosim.cli.plafosim.**iter_modules**(*path=None*, *prefix=''*)

Yields ModuleInfo for all submodules on path, or, if path is None, all top-level modules on sys.path.

'path' should be either None or a list of paths to look for modules in.

'prefix' is a string to output on the front of every module name on output.

plafosim.cli.plafosim.**load_snapshot**(*snapshot_filename: str*) → *Simulator*

Load a simulator object from a snapshot file.

Parameters
snapshot_filename (*str*) – The name of the file containing the snapshot

Returns
Simulator

Return type
The loaded simulator object

`plafosim.cli.plafosim.main()`

The main entry point of PlaFoSim.

`plafosim.cli.plafosim.parse_args() -> (<class 'argparse.Namespace'>, <class 'argparse._ArgumentGroup'>)`

Parse arguments given to this module.

Returns

- **args** (*argparse.Namespace*) – The namespace of parsed arguments and corresponding values.
- **g_gui** (*argparse._ArgumentGroup*) – The specific argument group for the GUI properties.

`plafosim.cli.plafosim.save_snapshot(simulator: Simulator, snapshot_filename: str)`

Store a simulator object to a snapshot file.

Parameters

- **simulator** (`Simulator`) – The simulator object to store
- **snapshot_filename** (*str*) – The name of the file for storing the snapshot

`plafosim.cli.plafosim.signal(signalnum, handler, /)`

Set the action for the given signal.

The action can be SIG_DFL, SIG_IGN, or a callable Python object. The previous action is returned. See getsignal() for possible return values.

* **IMPORTANT NOTICE** * A signal handler function is called with two arguments: the first is the signal number, the second is the interrupted stack frame.

`plafosim.cli.plafosim.strtobool(val)`

Convert a string representation of truth to true (1) or false (0).

True values are 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. Raises ValueError if 'val' is anything else.

`plafosim.cli.plafosim.timer()`

perf_counter() -> float

Performance counter for benchmarking.

plafosim.cli.trace_replay module

class `plafosim.cli.trace_replay.CustomFormatter(prog, indent_increment=2, max_help_position=24, width=None)`

Bases: `ArgumentDefaultsHelpFormatter`, `RawDescriptionHelpFormatter`, `MetavarTypeHelpFormatter`

Metaclass combining multiple formatter classes for argparse.

class `_Section(formatter, parent, heading=None)`

Bases: `object`

__init__(*formatter*, *parent*, *heading=None*)

format_help()

__init__(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

_add_item(*func*, *args*)

_dedent()

_expand_help(*action*)

_fill_text(*text*, *width*, *indent*)

_format_action(*action*)

_format_action_invocation(*action*)

_format_actions_usage(*actions*, *groups*)

_format_args(*action*, *default_metavar*)

_format_text(*text*)

_format_usage(*usage*, *actions*, *groups*, *prefix*)

_get_default_metavar_for_optional(*action*)

_get_default_metavar_for_positional(*action*)

_get_help_string(*action*)

_indent()

_iter_indented_subactions(*action*)

_join_parts(*part_strings*)

_metavar_formatter(*action*, *default_metavar*)

_split_lines(*text*, *width*)

add_argument(*action*)

add_arguments(*actions*)

add_text(*text*)

add_usage(*usage*, *actions*, *groups*, *prefix=None*)

end_section()

format_help()

start_section(*heading*)

class plafosim.cli.trace_replay.**tqdm**(*_, **__)

Bases: Comparable

Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

__init__(*iterable=None*, *desc=None*, *total=None*, *leave=True*, *file=None*, *ncols=None*, *mininterval=0.1*, *maxinterval=10.0*, *miniters=None*, *ascii=None*, *disable=False*, *unit='it'*, *unit_scale=False*, *dynamic_ncols=False*, *smoothing=0.3*, *bar_format=None*, *initial=0*, *position=None*, *postfix=None*, *unit_divisor=1000*, *write_bytes=None*, *lock_args=None*, *nrows=None*, *colour=None*, *delay=0*, *gui=False*, ***kwargs*)

Parameters

- **iterable** (*iterable, optional*) – Iterable to decorate with a progressbar. Leave blank to manually manage the updates.
- **desc** (*str, optional*) – Prefix for the progressbar.
- **total** (*int or float, optional*) – The number of expected iterations. If unspecified, len(iterable) is used if possible. If float("inf") or as a last resort, only basic progress statistics are displayed (no ETA, no progressbar). If *gui* is True and this parameter needs subsequent updating, specify an initial arbitrary large positive number, e.g. 9e9.
- **leave** (*bool, optional*) – If [default: True], keeps all traces of the progressbar upon termination of iteration. If *None*, will leave only if *position* is *0*.
- **file** (*io.TextIOWrapper* or *io.StringIO*, optional) – Specifies where to output the progress messages (default: sys.stderr). Uses *file.write(str)* and *file.flush()* methods. For encoding, see *write_bytes*.
- **ncols** (*int, optional*) – The width of the entire output message. If specified, dynamically resizes the progressbar to stay within this bound. If unspecified, attempts to use environment width. The fallback is a meter width of 10 and no limit for the counter and statistics. If 0, will not print any meter (only stats).
- **mininterval** (*float, optional*) – Minimum progress display update interval [default: 0.1] seconds.
- **maxinterval** (*float, optional*) – Maximum progress display update interval [default: 10] seconds. Automatically adjusts *miniters* to correspond to *mininterval* after long display update lag. Only works if *dynamic_miniters* or monitor thread is enabled.
- **miniters** (*int or float, optional*) – Minimum progress display update interval, in iterations. If 0 and *dynamic_miniters*, will automatically adjust to equal *mininterval* (more CPU efficient, good for tight loops). If > 0, will skip display of specified number of iterations. Tweak this and *mininterval* to get very efficient loops. If your progress is erratic with both fast and slow iterations (network, skipping items, etc) you should set miniters=1.
- **ascii** (*bool or str, optional*) – If unspecified or False, use unicode (smooth blocks) to fill the meter. The fallback is to use ASCII characters " 123456789#".
- **disable** (*bool, optional*) – Whether to disable the entire progressbar wrapper [default: False]. If set to None, disable on non-TTY.
- **unit** (*str, optional*) – String that will be used to define the unit of each iteration [default: it].
- **unit_scale** (*bool or int or float, optional*) – If 1 or True, the number of iterations will be reduced/scaled automatically and a metric prefix following the International System of Units standard will be added (kilo, mega, etc.) [default: False]. If any other non-zero number, will scale *total* and *n*.
- **dynamic_ncols** (*bool, optional*) – If set, constantly alters *ncols* and *nrows* to the environment (allowing for window resizes) [default: False].

- **smoothing** (*float, optional*) – Exponential moving average smoothing factor for speed estimates (ignored in GUI mode). Ranges from 0 (average speed) to 1 (current/instantaneous speed) [default: 0.3].
- **bar_format** (*str, optional*) – Specify a custom bar string formatting. May impact performance. [default: '{l_bar}{bar}{r_bar}'], where l_bar='{desc}: {percentage:3.0f}%|' and r_bar='| {n_fmt}/{total_fmt} [{elapsed}<{remaining}, '

  '{rate_fmt}{postfix}]'

  Possible vars: l_bar, bar, r_bar, n, n_fmt, total, total_fmt,
  percentage, elapsed, elapsed_s, ncols, nrows, desc, unit, rate, rate_fmt, rate_noinv, rate_noinv_fmt, rate_inv, rate_inv_fmt, postfix, unit_divisor, remaining, remaining_s, eta.

  Note that a trailing ": " is automatically removed after {desc} if the latter is empty.
- **initial** (*int or float, optional*) – The initial counter value. Useful when restarting a progress bar [default: 0]. If using float, consider specifying *{n:.3f}* or similar in *bar_format*, or specifying *unit_scale*.
- **position** (*int, optional*) – Specify the line offset to print this bar (starting from 0) Automatic if unspecified. Useful to manage multiple bars at once (eg, from threads).
- **postfix** (dict or *, optional) – Specify additional stats to display at the end of the bar. Calls *set_postfix(**postfix)* if possible (dict).
- **unit_divisor** (*float, optional*) – [default: 1000], ignored unless *unit_scale* is True.
- **write_bytes** (*bool, optional*) – If (default: None) and *file* is unspecified, bytes will be written in Python 2. If *True* will also write bytes. In all other cases will default to unicode.
- **lock_args** (*tuple, optional*) – Passed to *refresh* for intermediate output (initialisation, iterating, and updating).
- **nrows** (*int, optional*) – The screen height. If specified, hides nested bars outside this bound. If unspecified, attempts to use environment height. The fallback is 20.
- **colour** (*str, optional*) – Bar colour (e.g. 'green', '#00ff00').
- **delay** (*float, optional*) – Don't display until [default: 0] seconds have elapsed.
- **gui** (*bool, optional*) – WARNING: internal parameter - do not use. Use tqdm.gui.tqdm(...) instead. If set, will attempt to use matplotlib animations for a graphical output [default: False].

Returns
out

Return type
decorated iterator.

classmethod _decr_instances(*instance*)

Remove from list and reposition another unfixed bar to fill the new gap.

This means that by default (where all nested bars are unfixed), order is not maintained but screen flicker/blank space is minimised. (tqdm<=4.44.1 moved ALL subsequent unfixed bars up.)

classmethod _get_free_pos(*instance=None*)

Skips specified instance.

clear(*nolock=False*)

Clear current bar display.

close()

Cleanup and (if leave=False) close the progressbar.

display(*msg=None*, *pos=None*)

Use *self.sp* to display *msg* in the specified *pos*.

Consider overloading this function when inheriting to use e.g.: *self.some_frontend(**self.format_dict)* instead of *self.sp*.

Parameters

- **msg** (str, optional. What to display (default: *repr(self)*).) –
- **pos** (int, optional. Position to *moveto*) – (default: *abs(self.pos)*).

classmethod external_write_mode(*file=None*, *nolock=False*)

Disable tqdm within context and refresh tqdm when exits. Useful when writing to standard output stream

static format_interval(*t*)

Formats a number of seconds as a clock time, [H:]MM:SS

Parameters
t (*int*) – Number of seconds.

Returns
out – [H:]MM:SS

Return type
str

static format_meter(*n*, *total*, *elapsed*, *ncols=None*, *prefix=''*, *ascii=False*, *unit='it'*, *unit_scale=False*, *rate=None*, *bar_format=None*, *postfix=None*, *unit_divisor=1000*, *initial=0*, *colour=None*, ***extra_kwargs*)

Return a string-based progress bar given some parameters

Parameters

- **n** (*int or float*) – Number of finished iterations.
- **total** (*int or float*) – The expected total number of iterations. If meaningless (None), only basic progress statistics are displayed (no ETA).
- **elapsed** (*float*) – Number of seconds passed since start.
- **ncols** (*int, optional*) – The width of the entire output message. If specified, dynamically resizes *{bar}* to stay within this bound [default: None]. If *0*, will not print any bar (only stats). The fallback is *{bar:10}*.
- **prefix** (*str, optional*) – Prefix message (included in total width) [default: ''']. Use as {desc} in bar_format string.
- **ascii** (*bool, optional or str, optional*) – If not set, use unicode (smooth blocks) to fill the meter [default: False]. The fallback is to use ASCII characters " 123456789#".
- **unit** (*str, optional*) – The iteration unit [default: 'it'].
- **unit_scale** (*bool or int or float, optional*) – If 1 or True, the number of iterations will be printed with an appropriate SI metric prefix (k = 10^3, M = 10^6, etc.) [default: False]. If any other non-zero number, will scale *total* and *n*.

- **rate** (*float, optional*) – Manual override for iteration rate. If [default: None], uses n/elapsed.
- **bar_format** (*str, optional*) – Specify a custom bar string formatting. May impact performance. [default: '{l_bar}{bar}{r_bar}'], where l_bar='{desc}: {percentage:3.0f}%|' and r_bar='| {n_fmt}/{total_fmt} [{elapsed}<{remaining}, '

  '{rate_fmt}{postfix}]'

  Possible vars: l_bar, bar, r_bar, n, n_fmt, total, total_fmt,
  percentage, elapsed, elapsed_s, ncols, nrows, desc, unit, rate, rate_fmt, rate_noinv, rate_noinv_fmt, rate_inv, rate_inv_fmt, postfix, unit_divisor, remaining, remaining_s, eta.

  Note that a trailing ": " is automatically removed after {desc} if the latter is empty.
- **postfix** (*, optional) – Similar to *prefix*, but placed at the end (e.g. for additional stats). Note: postfix is usually a string (not a dict) for this method, and will if possible be set to postfix = ', ' + postfix. However other types are supported (#382).
- **unit_divisor** (*float, optional*) – [default: 1000], ignored unless *unit_scale* is True.
- **initial** (*int or float, optional*) – The initial counter value [default: 0].
- **colour** (*str, optional*) – Bar colour (e.g. 'green', '#00ff00').

Returns
out

Return type
Formatted meter and stats, ready to display.

static format_num(*n*)

Intelligent scientific notation (.3g).

Parameters
n (*int or float or Numeric*) – A Number.

Returns
out – Formatted number.

Return type
str

static format_sizeof(*num*, *suffix=''*, *divisor=1000*)

Formats a number (greater than unity) with SI Order of Magnitude prefixes.

Parameters

- **num** (*float*) – Number (>= 1) to format.
- **suffix** (*str, optional*) – Post-postfix [default: ''].
- **divisor** (*float, optional*) – Divisor between prefixes [default: 1000].

Returns
out – Number with Order of Magnitude SI unit postfix.

Return type
str

classmethod get_lock()

Get the global lock. Construct it if it does not exist.

moveto(*n*)

classmethod pandas(***tqdm_kwargs*)

Registers the current *tqdm* class with
pandas.core. (frame.DataFrame | series.Series | groupby.(generic.)DataFrameGroupBy | groupby.(generic.)SeriesGroupBy).progress_apply

A new instance will be create every time *progress_apply* is called, and each instance will automatically *close()* upon completion.

Parameters
tqdm_kwargs (*arguments for the tqdm instance*) –

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from tqdm import tqdm
>>> from tqdm.gui import tqdm as tqdm_gui
>>>
>>> df = pd.DataFrame(np.random.randint(0, 100, (100000, 6)))
>>> tqdm.pandas(ncols=50)  # can use tqdm_gui, optional kwargs, etc
>>> # Now you can use `progress_apply` instead of `apply`
>>> df.groupby(0).progress_apply(lambda x: x**2)
```

References

<https://stackoverflow.com/questions/18603270/ progress-indicator-during-pandas-operations-python>

refresh(*nolock=False*, *lock_args=None*)

Force refresh the display of this bar.

Parameters

- **nolock** (*bool, optional*) – If *True*, does not lock. If [default: *False*]: calls *acquire()* on internal lock.
- **lock_args** (*tuple, optional*) – Passed to internal lock's *acquire()*. If specified, will only *display()* if *acquire()* returns *True*.

reset(*total=None*)

Resets to 0 iterations for repeated use.

Consider combining with *leave=True*.

Parameters
total (*int or float, optional. Total to use for the new bar.*) –

set_description(*desc=None*, *refresh=True*)

Set/modify description of the progress bar.

Parameters

- **desc** (*str, optional*) –
- **refresh** (*bool, optional*) – Forces refresh [default: True].

set_description_str(*desc=None*, *refresh=True*)

Set/modify description without ': ' appended.

classmethod set_lock(*lock*)

Set the global lock.

set_postfix(*ordered_dict=None*, *refresh=True*, ***kwargs*)

Set/modify postfix (additional stats) with automatic formatting based on datatype.

Parameters

- **ordered_dict** (*dict or OrderedDict, optional*) –
- **refresh** (*bool, optional*) – Forces refresh [default: True].
- **kwargs** (*dict, optional*) –

set_postfix_str(*s=''*, *refresh=True*)

Postfix without dictionary expansion, similar to prefix handling.

static status_printer(*file*)

Manage the printing and in-place updating of a line of characters. Note that if the string is longer than a line, then in-place updating may not work (it will print a new line at each refresh).

unpause()

Restart tqdm timer from last print time.

update(*n=1*)

Manually update the progress bar, useful for streams such as reading files. E.g.: >>> t = tqdm(total=filesize) # Initialise >>> for current_buffer in stream: … … … t.update(len(current_buffer)) >>> t.close() The last line is highly recommended, but possibly not necessary if *t.update()* will be called in such a way that *filesize* will be exactly reached and printed.

Parameters

n (*int or float, optional*) – Increment to add to the internal counter of iterations [default: 1]. If using float, consider specifying *{n:.3f}* or similar in *bar_format*, or specifying *unit_scale*.

Returns

out – True if a *display()* was triggered.

Return type

bool or None

classmethod wrapattr(*stream*, *method*, *total=None*, *bytes=True*, ***tqdm_kwargs*)

stream : file-like object. method : str, "read" or "write". The result of *read()* and

the first argument of *write()* should have a *len()*.

```
>>> with tqdm.wrapattr(file_obj, "read", total=file_obj.size) as fobj:
...     while True:
...         chunk = fobj.read(chunk_size)
...         if not chunk:
...             break
```

classmethod write(*s*, *file=None*, *end='\n'*, *nolock=False*)

Print a message via tqdm (without overlap with bars).

property _comparable

_instances = set()

property format_dict

Public API for read-only member access.

monitor = None

monitor_interval = 10

plafosim.cli.trace_replay.**add_gui_vehicle**(*vid: int*, *position: float*, *lane: int*, *speed: float*, *color: tuple = (0, 255, 0)*, *track: bool = False*)

Add a vehicle to the GUI.

Parameters

- **vid** (*int*) – The vehicle's id
- **position** (*float*) – The vehicle's current position
- **lane** (*int*) – The vehicle's current lane
- **speed** (*float*) – The vehicle's current speed
- **color** (*tuple*) – The vehicle's current color
- **track** (*bool*) – Whether to track this vehicle within the GUI

plafosim.cli.trace_replay.**change_gui_vehicle_color**(*vid: int*, *color: tuple*)

Change the color of a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **color** (*tuple*) – The color (R, G, B) to use for the vehicle

plafosim.cli.trace_replay.**check_and_prepare_gui**()

Check and prepare GUI environment.

plafosim.cli.trace_replay.**close_gui**()

Close the GUI.

plafosim.cli.trace_replay.**find_resource**(*path: str*) → str

Find the resouces under relpath locally or as a packaged resource.

Parameters

path (*str*) – The path to search for the ressource

Returns

str

Return type

The path of the resource

plafosim.cli.trace_replay.**gui_step**(*target_step: int*, *screenshot_filename: str | None = None*)

Increases the simulation step in the GUI.

Parameters

- **target_step** (*int*) – The target simulation step
- **screenshot_filename** (*str, optional*) – The name of the screenshot file

`plafosim.cli.trace_replay.hex2rgb(c_hex: str) → tuple`

Convert a color in hex values to a color in RGB values.

Parameters
c_hex (*str*) – The color in hex values

Returns
tuple(int, int, int)

Return type
The color in RGB values

`plafosim.cli.trace_replay.main()`

The main entry point of PlaFoSim's trace replay.

`plafosim.cli.trace_replay.move_gui_vehicle(vid: int, position: float, lane: int, speed: float)`

Move a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **position** (*float*) – The vehicle's new position
- **lane** (*int*) – The vehicle's new lane
- **speed** (*float*) – The vehicle's new speed

`plafosim.cli.trace_replay.parse_args() → Namespace`

Parse arguments given to this module.

Returns
The namespace of parsed arguments and corresponding values.

Return type
argparse.Namespace

`plafosim.cli.trace_replay.prune_vehicles(keep_vids: list)`

Prunes vehicles from the GUI.

Parameters
keep_vids (*list*) – The ids of the vehicle that should be kept

`plafosim.cli.trace_replay.start_gui(config: str, step_length: float, play: bool = True)`

Start the GUI.

Parameters

- **config** (*str*) – The name of the configuration file
- **step_length** (*float*) – The length of one simulation step in s
- **play** (*bool, optional*) – Whether to start the simulation automatically

Module contents

Submodules

plafosim.emissions module

class plafosim.emissions.**EmissionClass**(*value*)

Bases: *Enum*

Emission class for combustion engines using the HBEFA3 model.

Website: https://www.hbefa.net/

PC_G_EU4 = 0

property emission_factors: dict

Return the emission factors of the emission class.

Returns
dict

Return type
The emission factors of the emission class

property is_diesel: bool

Return whether the emission class is for a diesel engine.

Returns
bool

Return type
Whether the emission class is for a diesel engine

class plafosim.emissions.**Enum**(*value*)

Bases: `object`

Generic enumeration.

Derive from this class to define new enumerations.

_member_type_

alias of `object`

_generate_next_value_(*start*, *count*, *last_values*)

Generate the next value when not given.

name: the name of the member start: the initial start value or None count: the number of existing members last_value: the last value assigned or None

classmethod _missing_(*value*)

_member_map_ = {}

_member_names_ = []

_value2member_map_ = {}

name

The name of the Enum member.

value

The value of the Enum member.

plafosim.formation_algorithm module

class plafosim.formation_algorithm.**ABC**

Bases: object

Helper class that provides a standard way to create an ABC using inheritance.

_abc_impl = <_abc_data object>

class plafosim.formation_algorithm.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters

owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns

The specific argument group for this algorithm.

Return type

argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.formation_algorithm.**abstractmethod**(*funcobj*)

A decorator indicating abstract methods.

Requires that the metaclass is ABCMeta or derived from it. A class that has a metaclass derived from ABCMeta cannot be instantiated unless all of its abstract methods are overridden. The abstract methods can be called using any of the normal 'super' call mechanisms. abstractmethod() may be used to declare abstract methods for properties and descriptors.

Usage:

class C(metaclass=ABCMeta):
@abstractmethod def my_abstract_method(self, . . .):

. . .

plafosim.gui module

`plafosim.gui.add_gui_vehicle(vid: int, position: float, lane: int, speed: float, color: tuple = (0, 255, 0), track: bool = False)`

Add a vehicle to the GUI.

Parameters

- **vid** (*int*) – The vehicle's id
- **position** (*float*) – The vehicle's current position
- **lane** (*int*) – The vehicle's current lane
- **speed** (*float*) – The vehicle's current speed
- **color** (*tuple*) – The vehicle's current color
- **track** (*bool*) – Whether to track this vehicle within the GUI

`plafosim.gui.change_gui_vehicle_color(vid: int, color: tuple)`

Change the color of a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **color** (*tuple*) – The color (R, G, B) to use for the vehicle

`plafosim.gui.check_and_prepare_gui()`

Check and prepare GUI environment.

`plafosim.gui.close_gui()`

Close the GUI.

`plafosim.gui.draw_infrastructures(infrastructures: list, labels: bool)`

Draws infrastructures in the GUI.

Parameters

- **infrastructures** (*list*) – The list of infrastructure objects to add
- **labels** (*bool*) – Whether to draw infrastructure labels

`plafosim.gui.draw_ramps(road_length: int, interval: int, labels: bool)`

Draws on-/off-ramps in the GUI.

Parameters

- **road_length** (*int*) – The length of the road in m
- **interval** (*int*) – The ramp interval in m
- **labels** (*bool*) – Whether to draw ramp labels

`plafosim.gui.draw_road_end(road_length: int, label: bool)`

Draws the end of the road in the GUI.

Parameters

- **road_length** (*int*) – The length of the road in m
- **label** (*bool*) – Whether to draw a label

`plafosim.gui.gui_step(target_step: int, screenshot_filename: str | None = None)`

Increases the simulation step in the GUI.

Parameters

- **target_step** (*int*) – The target simulation step
- **screenshot_filename** (*str, optional*) – The name of the screenshot file

`plafosim.gui.move_gui_vehicle(vid: int, position: float, lane: int, speed: float)`

Move a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **position** (*float*) – The vehicle's new position
- **lane** (*int*) – The vehicle's new lane
- **speed** (*float*) – The vehicle's new speed

`plafosim.gui.prune_vehicles(keep_vids: list)`

Prunes vehicles from the GUI.

Parameters

keep_vids (*list*) – The ids of the vehicle that should be kept

`plafosim.gui.remove_gui_vehicle(vid: int)`

Remove a vehicle from the GUI.

Parameters

vid (*int*) – The id of the vehicle to remove

`plafosim.gui.set_gui_window(road_length: int)`

Set the window of the GUI according to the road length.

Parameters

road_length (*int*) – The length of the road in m

`plafosim.gui.start_gui(config: str, step_length: float, play: bool = True)`

Start the GUI.

Parameters

- **config** (*str*) – The name of the configuration file
- **step_length** (*float*) – The length of one simulation step in s
- **play** (*bool, optional*) – Whether to start the simulation automatically

plafosim.infrastructure module

class plafosim.infrastructure.**Dummy**(*owner: object*, *dummy: int = -1*, ***kw_args*)

Bases: *FormationAlgorithm*

Dummy Platoon Formation Algorithm.

__init__(*owner: object*, *dummy: int = -1*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **dummy** (*int, optional*) – The value for the dummy parameter

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

do_formation()

Run platoon formation algorithm to search for a platooning opportunity and perform the corresponding join maneuver.

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.infrastructure.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters
owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns
The specific argument group for this algorithm.

Return type
argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.infrastructure.**Infrastructure**(*simulator: Simulator, iid: int, position: int, formation_algorithm: str, execution_interval: int, **kw_args*)

Bases: object

A class representing road side infrastructure / road side units.

The infrastructure can execute formation algorithms.

__init__(*simulator: Simulator, iid: int, position: int, formation_algorithm: str, execution_interval: int, **kw_args*)

Initialize an Infrastructure instance.

Parameters

- **simulator** (Simulator) – The global simulator object
- **iid** (*int*) – The id of the infrastructure
- **position** (*int*) – The position (x) of the infrastructure
- **formation_algorithm** (*str*) – The platoon formation (i.e., assignment calculation) algorithm to run
- **execution_interval** (*int*) – The execution interval for the formation algorithm

_action(*step: int*)

Triggers concrete actions of an infrastructure.

Parameters
step (*int*) – The current simulation step

_get_neighbors()

action(*step: int*)

Triggers actions of an infrastructure.

Parameters
step (*int*) – The current simulation step

finish()

Clean up the instance of the infrastructure.

property iid: int

Return the id of an infrastructure.

property position: int

Return the position of an infrastructure.

class plafosim.infrastructure.**PlatooningVehicle**(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Bases: *Vehicle*

A vehicle that has platooning functionality.

__init__(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Initialize a platooning vehicle instance.

Parameters

- **simulator** (*Simulator*) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (*VehicleType*) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int*) – The maximum communication range of the vehicle
- **acc_headway_time** (*float*) – The headway time for the ACC
- **cacc_spacing** (*float*) – The constant spacing for the CACC
- **formation_algorithm** (*str*) – The platoon formation algorithm to use
- **execution_interval** (*int*) – The interval for executing the formation algorithm
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a PlatooningVehicle.

Parameters
step (*float*) – The current simulation step

_calculate_emission(*a: float*, *v: float*, *f: list*, *scale: float*) → float

Calculate the emitted pollutant amount using the given speed and acceleration.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns
float

Return type
The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_get_available_platoons() → list

Return the available platoon candidates of the vehicle.

This imitates neighbor maintenance by using a neighbor table.

Returns
list(PlatooninVehicle)

Return type
The list of available platoons

_join(*platoon_id: int*, *leader_id: int*)

Lets a vehicle join a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

Parameters

- **platoon_id** (*int*) – The id of the target platoon
- **leader_id** (*int*) – The id of the leader of the target platoon

_join_teleport(*leader:* PlatooningVehicle, *last:* PlatooningVehicle, *new_position: float*)

Perform the actual teleporting of the join maneuver.

Parameters

- **leader** (PlatooningVehicle) – The leader of the target platoon
- **last** (PlatooningVehicle) – The last vehicle of the target platoon
- **new_position** (*float*) – The new position for joining the target platoon

_leave()

Lets a vehicle leave a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

_left_lane_blocked() → bool

Check whether a vehicle can move to the left lane in order to leave its current platoon.

Returns
bool

Return type
Whether the left lane is blocked

_start()

Start this Vehicle.

_statistics()

Write continuous statistics.

_teleport(*new_position: float*, *new_lane: int*, *new_speed: float*)

Teleport a vehicle to a given position.

Parameters

- **new_position** (*float*) – The new position
- **new_lane** (*int*) – The new lane
- **new_speed** (*float*) – The new speed

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

calculate_approaching_time(*target_position: float*, *target_speed: float*) → float

Calculate approximate time to approach the target position at target speed.

Parameters

- **target_position** (*float*) – The target position to approach
- **target_speed** (*float*) – The target speed after approaching

Returns
float

Return type
The calculated approaching time

finish()

Clean up the instance of the PlatooningVehicle.

This includes leaving the platoon and mostly statistics recording.

get_front_gap() → float

Return the gap to the vehicle in the front.

This imitates a measurement of the front radar sensor.

get_front_speed() → float

Return the speed to the vehicle in the front.

This imitates a measurement of the front radar sensor.

info() → str

Return information about the PlatooningVehicle.

is_in_platoon() → bool

Return whether the vehicle currently is in a platoon.

This is based on the current PlatoonRole. A joining or leaving vehicle is either not yet or still part of a platoon, thus the returned value should be true.

property acc_headway_time: float

Return the ACC headway time of the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: tuple

Return the current color of the vehicle.

property depart_lane: int

Return the departure lane of the vehicle.

property depart_position: int

Return the departure position of the vehicle.

property depart_speed: float

Return the departure speed of the vehicle.

property depart_time: float

Return the departure time of the vehicle.

property desired_gap: float

Return the desired gap of the vehicle.

This is based on the currently active car following model.

property desired_headway_time: float

Return the desired headway time of the vehicle.

This is based on the currently active car following model.

property desired_speed: float

Return the desired driving speed of the vehicle.

If the vehicle is in a platoon, it returns the desired driving speed of the entire platoon.

property distance_in_platoon: float

Return the travelled distance within platoons.

property headway_time: float

Return the human headway time of the vehicle.

This is based on the vehicle type.

property in_maneuver: bool

Return whether the vehicle is currently in a maneuver.

property lane: int

Return the current lane of the vehicle.

property length: int

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: float

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: float

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: float

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: float

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property platoon: *Platoon*

Return the platoon of the vehicle.

property platoon_role: *PlatoonRole*

Return the current platoon role of the vehicle.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property time_in_platoon: int

Return the travelled time within platoons.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: **int**

Return the id of the vehicle.

class plafosim.infrastructure.**SpeedPosition**(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Bases: *FormationAlgorithm*

Platoon Formation Algorithm based on Similarity, considering Speed and Position.

See papers

Julian Heinovski and Falko Dressler, "Where to Decide? Centralized vs. Distributed Vehicle Assignment for Platoon Formation," arXiv, cs.MA, 2310.09580, October 2023. https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

and

Julian Heinovski and Falko Dressler, "Platoon Formation: Optimized Car to Platoon Assignment Strategies and Protocols," Proceedings of 10th IEEE Vehicular Networking Conference (VNC 2018), Taipei, Taiwan, December 2018. https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/

__init__(*owner: object*, *alpha: float = 0.5*, *speed_deviation_threshold: float = 0.2*, *position_deviation_threshold: int = 1000*, *formation_centralized_kind: str = 'greedy'*, *solver_time_limit: int = 60*, *record_solver_traces: bool = False*, *record_infrastructure_assignments: bool = False*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **alpha** (*float*) – The weighting factor alpha
- **speed_deviation_threshold** (*float*) – The threshold for speed deviation
- **position_deviation_threshold** (*int*) – The threshold for position deviation
- **formation_centralized_kind** (*str*) – TODO
- **solver_time_limit** (*int*) – The time limit in s to apply to the solver
- **record_solver_traces** (*bool*) – Whether to record continuous solver traces
- **record_infrastructure_assignments** (*bool*) – Whether to record infrastructure assignments

_do_formation_centralized()

Run centralized greedy formation approach.

This selects candidates and triggers join maneuvers.

_do_formation_distributed()

Run distributed greedy formation approach.

This selects a candidate and triggers a join maneuver.

_do_formation_optimal()

Run centralized optimal formation approach.

This selects candidates and triggers join maneuvers.

_record_infrastructure_assignments(*basename: str*)

Record infrastructure assignments.

Parameters
basename (*str*) – The basename of the result file

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

cost_speed_position(*ds: float*, *dp: float*) → float

Return the overall cost (i.e., the weighted deviation) for a candidate.

Parameters

- **ds** (*float*) – The deviation in speed
- **dp** (*int*) – The deviation in position

Returns
The weighted relative deviation

Return type
float

do_formation()

Run platoon formation algorithms to search for a platooning opportunity and perform the corresponding join maneuver.

dp(*vehicle: PlatooningVehicle*, *platoon: Platoon*) → float

Return the deviation in position from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in position

Return type
float

ds(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in speed from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in speed

Return type
float

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

plafosim.infrastructure.**attribute**()

perf_counter() -> float

Performance counter for benchmarking.

plafosim.infrastructure.**import_module**(*name*, *package=None*)

Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

plafosim.infrastructure.**isclass**(*object*)

Return true if the object is a class.

Class objects provide these attributes:
__doc__ documentation string __module__ name of module in which this class was defined

plafosim.infrastructure.**iter_modules**(*path=None*, *prefix=''*)

Yields ModuleInfo for all submodules on path, or, if path is None, all top-level modules on sys.path.

'path' should be either None or a list of paths to look for modules in.

'prefix' is a string to output on the front of every module name on output.

plafosim.mobility module

class plafosim.mobility.**ACC_SPEED_DF**(*predecessor_speed*, *speed*, *acc_lambda*, *desired_gap*, *predecessor_gap*, *desired_headway_time*)

Bases: tuple

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new ACC_SPEED_DF object from a sequence or iterable

_replace(***kwds*)

Return a new ACC_SPEED_DF object replacing specified fields with new values

count(*value*, /)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

_field_defaults = {}

_fields = ('predecessor_speed', 'speed', 'acc_lambda', 'desired_gap', 'predecessor_gap', 'desired_headway_time')

_fields_defaults = {}

acc_lambda

Alias for field number 2

desired_gap

Alias for field number 3

desired_headway_time

Alias for field number 5

predecessor_gap

Alias for field number 4

predecessor_speed

Alias for field number 0

speed

Alias for field number 1

class plafosim.mobility.**CF_Model**(*value*)

Bases: *Enum*

Car Following models that vehicles can use for their mobility.

ACC = 1

CACC = 2

HUMAN = 0

class plafosim.mobility.**Enum**(*value*)

Bases: object

Generic enumeration.

Derive from this class to define new enumerations.

_member_type_

alias of object

_generate_next_value_(*start*, *count*, *last_values*)

Generate the next value when not given.

name: the name of the member start: the initial start value or None count: the number of existing members last_value: the last value assigned or None

classmethod _missing_(*value*)

_member_map_ = {}

_member_names_ = []

_value2member_map_ = {}

name

The name of the Enum member.

value

The value of the Enum member.

class plafosim.mobility.**FAKE_PREDECESSOR**(*vid*, *position*, *speed*, *length*)

Bases: tuple

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new FAKE_PREDECESSOR object from a sequence or iterable

_replace(**kwds*)

Return a new FAKE_PREDECESSOR object replacing specified fields with new values

count(*value*, /)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

_field_defaults = {}

_fields = ('vid', 'position', 'speed', 'length')

_fields_defaults = {}

length

Alias for field number 3

position

Alias for field number 1

speed

Alias for field number 2

vid

Alias for field number 0

class plafosim.mobility.**SAFE_SPEED_DF**(*predecessor_speed*, *speed*, *max_deceleration*, *predecessor_gap*, *desired_gap*, *desired_headway_time*)

Bases: tuple

_asdict()

Return a new dict which maps field names to their values.

classmethod _make(*iterable*)

Make a new SAFE_SPEED_DF object from a sequence or iterable

_replace(**kwds*)

Return a new SAFE_SPEED_DF object replacing specified fields with new values

count(*value*, /)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

_field_defaults = {}

_fields = ('predecessor_speed', 'speed', 'max_deceleration', 'predecessor_gap', 'desired_gap', 'desired_headway_time')

_fields_defaults = {}

desired_gap

Alias for field number 4

desired_headway_time

Alias for field number 5

max_deceleration

Alias for field number 2

predecessor_gap

Alias for field number 3

predecessor_speed

Alias for field number 0

speed

Alias for field number 1

plafosim.mobility.**assert_index_equal**(*one*, *two*) → bool

Ensure the indices of two Sequences/DataFrames are equal.

Parameters

- **one** (*pandas.Sequence / pandas.DataFrame*) – The first object for the comparison
- **two** (*pandas.Sequence / pandas.DataFrame*) – The second object for the comparsion

Returns

bool

Return type

Whether the two indices are equal

`plafosim.mobility.clamp_speed(new_speed: Series, vdf: DataFrame, step_length: float)` → Series

Clamp (two-way limit) a new speed value to vehicle's maximum.

`plafosim.mobility.clip_position(position: Series, vdf: DataFrame)` → Series

Return the clipped positions (i.e., by arrival position) of vehicles within a pandas Dataframe.

Parameters

- **position** (*pandas.Series*) – The series containing the positions to be clipped index: vid columns: [position, length, lane, ..]
- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Returns
The series of clipped positions index: vid

Return type
pandas.Series

`plafosim.mobility.compute_lane_changes(vdf: DataFrame, max_lane: int, step_length: float)` → DataFrame

Find desired and safe lane changes.

1) compute all speed gains
2) "apply" all speed gains
3) compute all keep rights
4) "apply" all keep rights

This is based on Krauss' multi lane traffic: laneChange() congested = (v_safe < v_thresh) and (v^0_safe < v_thresh) favorable(right->left) = (v_safe < v_max) and (not congested) favorable(left->right) = (v_safe >= v_max) and (v^0_safe >= v_max) if ((favorable(i->j) or (rand < p_change)) and safe(i->j)) then change(i->j) for vehicles on the right lane: if (v > v^0_safe) and (not congested) then v <- v^0_safe

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **max_lane** (*int*) – The largest lane id
- **step_length** (*float*) – The length of a simulation step

Returns
The Dataframe containing the vehicles as rows index: vid columns: [lane, reason]

Return type
pandas.DataFrame

`plafosim.mobility.compute_new_speeds(vdf: DataFrame, step_length: float)` → Series

Compute the new speed for all vehicles in the simulation.

Can compute "potential" new speed for different target lanes. Assume vdf already contains predecessor and successor data. Just pass the right predecessor/successor data for different lanes.

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **step_length** (*float*) – The length of a simulation step

Returns
The series of new speeds index: vid

Return type
pandas.Series

`plafosim.mobility.get_crashed_vehicles(vdf: DataFrame) → list`

Return the list of crashed vehicles' ids.

Parameters
vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Returns
list(int)

Return type
the list of vehicles that crashed

`plafosim.mobility.get_predecessors(vdf: DataFrame, predecessor_map: DataFrame, target_lane: Series) → DataFrame`

Return DataFrame of successors to the vehicles on a target lane.

`plafosim.mobility.get_successors(vdf: DataFrame, successor_map: DataFrame, target_lane: Series) → DataFrame`

Return DataFrame of successors to the vehicles on a target lane.

`plafosim.mobility.is_gap_safe(front_position: float, front_speed: float, front_max_deceleration: float, front_length: float, back_position: float, back_speed: float, back_max_acceleration: float, back_min_gap: float, step_length: float) → bool`

Return whether the gap between the front and back vehicle is safe.

Safe means: - the front vehicle can decelerate as hard as possible for one step - the back vehicle can accelerate as hard as possible for one step - the vehicle will not crash

Assumes euclidean/non-ballistic position updates.

Parameters

- **front_position** (*float*) – The position of the front vehicle in m
- **front_speed** (*float*) – The speed of the front vehicle in m/s
- **front_max_deceleration** (*float*) – The maximum deceleration of the front vehicle in m/s2
- **front_length** (*length*) – The length of the front vehicle in m
- **back_position** (*float*) – The position of the back vehicle in m
- **back_speed** (*float*) – The speed of the back vehicle in m/s
- **back_max_acceleration** (*float*) – The maximum acceleration of the back vehicle in m/s2
- **back_min_gap** (*float*) – The minimum gap of the back vehicle in m
- **step_length** (*float*) – The length of a simulation step in s

Returns
bool

Return type
Whether the gap between the two vehicles is safe

`plafosim.mobility.lane_predecessors(vdf: DataFrame, max_lane: int) → DataFrame`

Find the current (potential) predecessor for each lane and each vehicle.

This means: Which other vehicle would be the predecessor if a vehicle was on lane *column*. A predecessor id of -1 means there is no predecessor.

Preconditions: - vdf.sorted_values(['position', 'lane'], ascending=False)

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **max_lane** (*int*) – The largest lane id

`plafosim.mobility.lane_successors(vdf: DataFrame, max_lane: int) → DataFrame`

Find the current (potential) successors for each lane and each vehicle.

This means: Which other vehicle would be the successor if a vehicle was on lane *column*. A successor id of -1 means there is no successor.

Preconditions: - vdf.sorted_values(['position', 'lane'], ascending=False)

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **max_lane** (*int*) – The largest lane id

`plafosim.mobility.namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)`

Returns a new subclass of tuple with named fields.

```
>>> Point = namedtuple('Point', ['x', 'y'])
>>> Point.__doc__                   # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)             # instantiate with positional args or keywords
>>> p[0] + p[1]                     # indexable like a plain tuple
33
>>> x, y = p                        # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                       # fields also accessible by name
33
>>> d = p._asdict()                 # convert to a dictionary
>>> d['x']
11
>>> Point(**d)                      # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)               # _replace() is like str.replace() but targets␣
↪named fields
Point(x=100, y=22)
```

`plafosim.mobility.safe_speed(speed_predecessor, speed_current, gap_to_predecessor, desired_headway_time, max_deceleration, desired_gap)`

`plafosim.mobility.safe_speed_df(vdf: DataFrame) → Series`

Compute the safe speed according to the Krauss model, DataFrame variant.

See S. Krauss, Microscopic Modeling of Traffic Flow: Investigation of Collision Free Vehicle Dynamics, 1998, Equation (4.21). v_safe = v_leader + (gap - gap_desired) / tau_b + tau gap_desired = tau * v_leader tau_b = v_avg / b b = typical deceleration of drivers tau = reaction time of drivers

safety is guaranteed, if step length <= tau and gap_desired >= v_leader * step_length <=> if the true reaction time (i.e., the length of one time step) is smaller than or equal to the reaction time that each driver assumes

Parameters

vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

`plafosim.mobility.speed_acc_df(vdf: DataFrame, step_length: float) → Series`

Compute new speed for ACC vehicles, DataFrame variant.

Clamping is done outside this function.

See Eq. 6.18 of R. Rajamani, Vehicle Dynamics and Control, 2nd. Springer, 2012.

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **step_length** (*float*) – The length of a simulation step in s

`plafosim.mobility.speed_human_df(vdf: DataFrame) → Series`

Compute new speed for human vehicles, DataFrame variant.

Basically just safe speed, clamping is done outside this function.

Parameters

vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

`plafosim.mobility.update_position(vdf: DataFrame, step_length: float) → DataFrame`

Update the position of vehicles within a pandas Dataframe.

This is based on Krauss' single lane traffic: adjust position (move) x(t + step_size) = x(t) + v(t)*step_size

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **step_length** (*float*) – The length of the simulated step

Returns

The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Return type

pandas.DataFrame

plafosim.platoon module

class plafosim.platoon.**Platoon**(*platoon_id: int*, *formation: list*, *desired_speed: float*)

Bases: object

A collection of parameters for a specific platoon.

__init__(*platoon_id: int*, *formation: list*, *desired_speed: float*)

Initialize a platoon instance.

Parameters

- **platoon_id** (*int*) – The id of the platoon
- **formation** (*list*) – The list of PlatooningVehicles within the platoon
- **desired_speed** (*float*) – The platoon's desired driving speed

get_back(*vehicle: PlatooningVehicle*)

Return the PlatooningVehicle in the back.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
PlatooningVehicle

Return type
The member in the back

get_front(*vehicle: PlatooningVehicle*)

Return the PlatooningVehicle in the front.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
PlatooningVehicle

Return type
The member in the front

get_member_index(*vehicle: PlatooningVehicle*) → int

Return the index of a member within the platoon.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
int

Return type
The index of the member within the platoon

update_cf_target_speed()

Update the cf target speed for the platoon.

update_desired_speed()

Update the desired driving speed of the platoon.

This is based on the desired driving speed of all members.

update_limits()

Update mobility limits for the platoon.

update_max_acceleration()

Update the maximum acceleration of the platoon.

The maximum acceleration is based on the slowest vehicle within the platoon.

update_max_deceleration()

Update the maximum deceleration of the platoon.

The maximum deceleration is based on the slowest vehicle within the platoon.

update_max_speed()

Update the maximum speed of the platoon.

The maximum speed is based on the slowest vehicle within the platoon.

property desired_speed: float

Return the desired driving speed of the platoon.

property formation: list

Return the complete formation of the platoon.

property lane: int

Return the current lane of the platoon.

property last: *PlatooningVehicle*

Return the last PlatooningVehicle of the platoon.

property leader: *PlatooningVehicle*

Return the leading PlatoonVehicle of the platoon.

property length: float

Return the length of the platoon.

property max_acceleration: float

Return the maximum acceleration of the platoon.

The maximum acceleration is based on the slowest vehicle within the platoon.

property max_deceleration: float

Return the maximum deceleration of the platoon.

The maximum deceleration is based on the slowest vehicle within the platoon.

property max_speed: float

Return the maximum speed of the platoon.

The maximum speed is based on the slowest vehicle within the platoon.

property member_ids: list

Return the ids of all platoon members.

property platoon_id: int

Return the id of the platoon.

property position: int

Return the current position of the platoon.

property rear_position: int

Return the current rear position of the platoon.

property size: int

Return the size of the platoon.

property speed: float

Return the current driving speed of the platoon.

plafosim.platoon.**mean**(*data*)

Return the sample arithmetic mean of data.

```
>>> mean([1, 2, 3, 4, 4])
2.8
```

```
>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)
```

```
>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

If `data` is empty, StatisticsError will be raised.

plafosim.platoon_role module

class plafosim.platoon_role.**Enum**(*value*)

Bases: `object`

Generic enumeration.

Derive from this class to define new enumerations.

_member_type_

alias of `object`

_generate_next_value_(*start*, *count*, *last_values*)

Generate the next value when not given.

name: the name of the member start: the initial start value or None count: the number of existing members last_value: the last value assigned or None

classmethod _missing_(*value*)

_member_map_ = {}

_member_names_ = []

_value2member_map_ = {}

name

The name of the Enum member.

value

The value of the Enum member.

class plafosim.platoon_role.**PlatoonRole**(*value*)

Bases: *Enum*

A collection of available platoon roles.

FOLLOWER = 2

JOINER = 3

LEADER = 1

LEAVER = 4

NONE = 0

plafosim.platooning_vehicle module

class plafosim.platooning_vehicle.**CF_Model**(*value*)

Bases: *Enum*

Car Following models that vehicles can use for their mobility.

ACC = 1

CACC = 2

HUMAN = 0

class plafosim.platooning_vehicle.**Dummy**(*owner: object*, *dummy: int = -1*, ***kw_args*)

Bases: *FormationAlgorithm*

Dummy Platoon Formation Algorithm.

__init__(*owner: object*, *dummy: int = -1*, ***kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **dummy** (*int, optional*) – The value for the dummy parameter

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

do_formation()

Run platoon formation algorithm to search for a platooning opportunity and perform the corresponding join maneuver.

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.platooning_vehicle.**FormationAlgorithm**(*owner: object*)

Bases: *ABC*

Abstract base class for any type of platoon formation algorithm (i.e., assignment calculation).

Implementing sub-classes need to override the do_formation() method.

__init__(*owner: object*)

Initialize an instance of a formation algorithm.

Parameters

owner (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.

abstract add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

Returns

The specific argument group for this algorithm.

Return type

argparse._ArgumentGroup

abstract do_formation()

Abstract method for performing any type of platoon formation (i.e., assignment calculation).

This methods needs to be overridden in implementing sub-classes.

finish()

Reserved for future use.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.platooning_vehicle.**Platoon**(*platoon_id: int*, *formation: list*, *desired_speed: float*)

Bases: object

A collection of parameters for a specific platoon.

__init__(*platoon_id: int*, *formation: list*, *desired_speed: float*)

Initialize a platoon instance.

Parameters

- **platoon_id** (*int*) – The id of the platoon
- **formation** (*list*) – The list of PlatooningVehicles within the platoon
- **desired_speed** (*float*) – The platoon's desired driving speed

get_back(*vehicle:* PlatooningVehicle)

Return the PlatooningVehicle in the back.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
PlatooningVehicle

Return type
The member in the back

get_front(*vehicle:* PlatooningVehicle)

Return the PlatooningVehicle in the front.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
PlatooningVehicle

Return type
The member in the front

get_member_index(*vehicle:* PlatooningVehicle) → int

Return the index of a member within the platoon.

Parameters
vehicle (PlatooningVehicle) – The considered vehicle within the platoon.

Returns
int

Return type
The index of the member within the platoon

update_cf_target_speed()

Update the cf target speed for the platoon.

update_desired_speed()

Update the desired driving speed of the platoon.

This is based on the desired driving speed of all members.

update_limits()

Update mobility limits for the platoon.

update_max_acceleration()

Update the maximum acceleration of the platoon.

The maximum acceleration is based on the slowest vehicle within the platoon.

update_max_deceleration()

Update the maximum deceleration of the platoon.

The maximum deceleration is based on the slowest vehicle within the platoon.

update_max_speed()

Update the maximum speed of the platoon.

The maximum speed is based on the slowest vehicle within the platoon.

property desired_speed: float

Return the desired driving speed of the platoon.

property formation: list

Return the complete formation of the platoon.

property lane: int

Return the current lane of the platoon.

property last: *PlatooningVehicle*

Return the last PlatooningVehicle of the platoon.

property leader: *PlatooningVehicle*

Return the leading PlatoonVehicle of the platoon.

property length: float

Return the length of the platoon.

property max_acceleration: float

Return the maximum acceleration of the platoon.

The maximum acceleration is based on the slowest vehicle within the platoon.

property max_deceleration: float

Return the maximum deceleration of the platoon.

The maximum deceleration is based on the slowest vehicle within the platoon.

property max_speed: float

Return the maximum speed of the platoon.

The maximum speed is based on the slowest vehicle within the platoon.

property member_ids: list

Return the ids of all platoon members.

property platoon_id: int

Return the id of the platoon.

property position: int

Return the current position of the platoon.

property rear_position: int

Return the current rear position of the platoon.

property size: int

Return the size of the platoon.

property speed: float

Return the current driving speed of the platoon.

class plafosim.platooning_vehicle.**PlatoonRole**(*value*)

Bases: *Enum*

A collection of available platoon roles.

FOLLOWER = 2

JOINER = 3

LEADER = 1

LEAVER = 4

NONE = 0

class plafosim.platooning_vehicle.**PlatooningVehicle**(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Bases: *Vehicle*

A vehicle that has platooning functionality.

__init__(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Initialize a platooning vehicle instance.

Parameters

- **simulator** (*Simulator*) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (*VehicleType*) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int*) – The maximum communication range of the vehicle
- **acc_headway_time** (*float*) – The headway time for the ACC
- **cacc_spacing** (*float*) – The constant spacing for the CACC
- **formation_algorithm** (*str*) – The platoon formation algorithm to use
- **execution_interval** (*int*) – The interval for executing the formation algorithm
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a PlatooningVehicle.

Parameters
step (*float*) – The current simulation step

_calculate_emission(*a: float*, *v: float*, *f: list*, *scale: float*) → float

Calculate the emitted pollutant amount using the given speed and acceleration.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns
float

Return type
The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_get_available_platoons() → list

Return the available platoon candidates of the vehicle.

This imitates neighbor maintenance by using a neighbor table.

Returns
list(PlatooninVehicle)

Return type
The list of available platoons

_join(*platoon_id: int*, *leader_id: int*)

Lets a vehicle join a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

Parameters

- **platoon_id** (*int*) – The id of the target platoon
- **leader_id** (*int*) – The id of the leader of the target platoon

_join_teleport(*leader:* PlatooningVehicle, *last:* PlatooningVehicle, *new_position: float*)

Perform the actual teleporting of the join maneuver.

Parameters

- **leader** (PlatooningVehicle) – The leader of the target platoon
- **last** (PlatooningVehicle) – The last vehicle of the target platoon
- **new_position** (*float*) – The new position for joining the target platoon

_leave()

Lets a vehicle leave a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

_left_lane_blocked() → bool

Check whether a vehicle can move to the left lane in order to leave its current platoon.

Returns
bool

Return type
Whether the left lane is blocked

_start()

Start this Vehicle.

_statistics()

Write continuous statistics.

_teleport(*new_position: float*, *new_lane: int*, *new_speed: float*)

Teleport a vehicle to a given position.

Parameters

- **new_position** (*float*) – The new position
- **new_lane** (*int*) – The new lane
- **new_speed** (*float*) – The new speed

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

calculate_approaching_time(*target_position: float*, *target_speed: float*) → float

Calculate approximate time to approach the target position at target speed.

Parameters

- **target_position** (*float*) – The target position to approach
- **target_speed** (*float*) – The target speed after approaching

Returns
float

Return type
The calculated approaching time

finish()

Clean up the instance of the PlatooningVehicle.

This includes leaving the platoon and mostly statistics recording.

get_front_gap() → float

Return the gap to the vehicle in the front.

This imitates a measurement of the front radar sensor.

get_front_speed() → float

Return the speed to the vehicle in the front.

This imitates a measurement of the front radar sensor.

info() → str

Return information about the PlatooningVehicle.

is_in_platoon() → bool

Return whether the vehicle currently is in a platoon.

This is based on the current PlatoonRole. A joining or leaving vehicle is either not yet or still part of a platoon, thus the returned value should be true.

property acc_headway_time: float

Return the ACC headway time of the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: tuple

Return the current color of the vehicle.

property depart_lane: int

Return the departure lane of the vehicle.

property depart_position: int

Return the departure position of the vehicle.

property depart_speed: float

Return the departure speed of the vehicle.

property depart_time: float

Return the departure time of the vehicle.

property desired_gap: float

Return the desired gap of the vehicle.

This is based on the currently active car following model.

property desired_headway_time: float

Return the desired headway time of the vehicle.

This is based on the currently active car following model.

property desired_speed: float

Return the desired driving speed of the vehicle.

If the vehicle is in a platoon, it returns the desired driving speed of the entire platoon.

property distance_in_platoon: float

Return the travelled distance within platoons.

property headway_time: float

Return the human headway time of the vehicle.

This is based on the vehicle type.

property in_maneuver: bool

Return whether the vehicle is currently in a maneuver.

property lane: int

Return the current lane of the vehicle.

property length: int

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: float

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: float

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: float

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: float

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property platoon: *Platoon*

Return the platoon of the vehicle.

property platoon_role: *PlatoonRole*

Return the current platoon role of the vehicle.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property time_in_platoon: int

Return the travelled time within platoons.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: **int**

Return the id of the vehicle.

class plafosim.platooning_vehicle.**SpeedPosition**(*owner: object, alpha: float = 0.5, speed_deviation_threshold: float = 0.2, position_deviation_threshold: int = 1000, formation_centralized_kind: str = 'greedy', solver_time_limit: int = 60, record_solver_traces: bool = False, record_infrastructure_assignments: bool = False, **kw_args*)

Bases: *FormationAlgorithm*

Platoon Formation Algorithm based on Similarity, considering Speed and Position.

See papers

Julian Heinovski and Falko Dressler, "Where to Decide? Centralized vs. Distributed Vehicle Assignment for Platoon Formation," arXiv, cs.MA, 2310.09580, October 2023. https://www.tkn.tu-berlin.de/bib/heinovski2023where-preprint/

and

Julian Heinovski and Falko Dressler, "Platoon Formation: Optimized Car to Platoon Assignment Strategies and Protocols," Proceedings of 10th IEEE Vehicular Networking Conference (VNC 2018), Taipei, Taiwan, December 2018. https://www.tkn.tu-berlin.de/bib/heinovski2018platoon/

__init__(*owner: object, alpha: float = 0.5, speed_deviation_threshold: float = 0.2, position_deviation_threshold: int = 1000, formation_centralized_kind: str = 'greedy', solver_time_limit: int = 60, record_solver_traces: bool = False, record_infrastructure_assignments: bool = False, **kw_args*)

Initialize an instance of this formation algorithm to be used in a vehicle or an infrastructure.

Parameters

- **owner** (*object*) – The owning object that is execution this algorithm. This can be either a PlatooningVehicle or an Infrastructure.
- **alpha** (*float*) – The weighting factor alpha
- **speed_deviation_threshold** (*float*) – The threshold for speed deviation
- **position_deviation_threshold** (*int*) – The threshold for position deviation
- **formation_centralized_kind** (*str*) – TODO
- **solver_time_limit** (*int*) – The time limit in s to apply to the solver
- **record_solver_traces** (*bool*) – Whether to record continuous solver traces
- **record_infrastructure_assignments** (*bool*) – Whether to record infrastructure assignments

_do_formation_centralized()

Run centralized greedy formation approach.

This selects candidates and triggers join maneuvers.

_do_formation_distributed()

Run distributed greedy formation approach.

This selects a candidate and triggers a join maneuver.

_do_formation_optimal()

Run centralized optimal formation approach.

This selects candidates and triggers join maneuvers.

_record_infrastructure_assignments(*basename: str*)

Record infrastructure assignments.

Parameters
basename (*str*) – The basename of the result file

classmethod add_parser_argument_group(*parser: ArgumentParser*) → _ArgumentGroup

Create and return specific argument group for this algorithm to use in global argument parser.

Parameters
parser (*argparse.ArgumentParser*) – The global argument parser

Returns
The specific argument group for this algorithm

Return type
argparse._ArgumentGroup

cost_speed_position(*ds: float*, *dp: float*) → float

Return the overall cost (i.e., the weighted deviation) for a candidate.

Parameters

- **ds** (*float*) – The deviation in speed
- **dp** (*int*) – The deviation in position

Returns
The weighted relative deviation

Return type
float

do_formation()

Run platoon formation algorithms to search for a platooning opportunity and perform the corresponding join maneuver.

dp(*vehicle: PlatooningVehicle*, *platoon: Platoon*) → float

Return the deviation in position from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in position

Return type
float

ds(*vehicle:* PlatooningVehicle, *platoon:* Platoon) → float

Return the deviation in speed from a given platoon.

NOTE: In the original version of the paper, the deviation calculated here was not normalized.

Parameters

- **vehicle** (PlatooningVehicle) – The vehicle for which the deviation is calculated
- **platoon** (Platoon) – The platoon to which the deviation is calculated

Returns
The relative deviation in speed

Return type
float

finish()

Clean up the instance of the formation algorithm.

This includes mostly statistic recording.

_abc_impl = <_abc_data object>

property name

Print the name of the formation algorithm.

class plafosim.platooning_vehicle.**Vehicle**(*simulator:* Simulator, *vid: int*, *vehicle_type:* VehicleType, *depart_position: int*, *arrival_position: int*, *desired_speed: float*, *depart_lane: int*, *depart_speed: float*, *depart_time: float*, *depart_delay: float*, *communication_range: int*, *pre_filled: bool = False*)

Bases: object

A collection of state information for a vehicle in the simulation.

A vehicle can really be anything that can move and can be defined by a vehicle type. It does not necessarily be driven by a computer (i.e., autonomous). However, by default it does have V2X functionality.

__init__(*simulator:* Simulator, *vid: int*, *vehicle_type:* VehicleType, *depart_position: int*, *arrival_position: int*, *desired_speed: float*, *depart_lane: int*, *depart_speed: float*, *depart_time: float*, *depart_delay: float*, *communication_range: int*, *pre_filled: bool = False*)

Initialize a vehicle instance.

Parameters

- **simulator** (Simulator) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (VehicleType) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip

- **communication_range** (*int*) – The maximum communication range of the vehicle
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a vehicle.

Parameters
step (*float*) – The current simulation step

_calculate_emission(*a: float*, *v: float*, *f: list*, *scale: float*) → float

Calculate the actual emission of the vehicle.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns
float

Return type
The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_start()

Start this Vehicle.

_statistics()

Write continuous statistics for the vehicle.

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

finish()

Clean up the instance of the vehicle.

This includes mostly statistic recording.

info() → str

Return information about the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: **tuple**

Return the current color of the vehicle.

property depart_lane: **int**

Return the departure lane of the vehicle.

property depart_position: **int**

Return the departure position of the vehicle.

property depart_speed: **float**

Return the departure speed of the vehicle.

property depart_time: **float**

Return the departure time of the vehicle.

property desired_gap: **float**

Return the desired gap to the vehicle in front of the vehicle.

This is based on the desired headway time and the current driving speed.

property desired_headway_time: **float**

Return the desired headway time of the vehicle.

property desired_speed: **float**

Return the desired driving speed of the vehicle.

property headway_time: **float**

Return the human headway time of the vehicle.

This is based on the vehicle type.

property lane: **int**

Return the current lane of the vehicle.

property length: **int**

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: **float**

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: **float**

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: **float**

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: **float**

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: int

Return the id of the vehicle.

class plafosim.platooning_vehicle.**VehicleType**(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Bases: object

A collection of parameters for a concrete vehicle type.

__init__(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Initialize a specific vehicle type.

Parameters

- **name** (*str*) – The name of the vehicle type
- **length** (*int*) – The length of the vehicle type
- **max_speed** (*float*) – The maximum speed of the vehicle type
- **max_acceleration** (*float*) – The maximum acceleration of the vehicle type
- **max_deceleration** (*float*) – The maximum deceleration of the vehicle type
- **min_gap** (*float*) – The minimum safety gap to the vehicle in front of the vehicle type
- **headway_time** (*float*) – The human headway time of the vehicle type
- **emission_class** (EmissionClass) – The emission class of the vehicle type

property emission_class: *EmissionClass*

Return the emission class of a vehicle type.

property emission_factors: dict

Return the emission factors of a vehicle type.

property headway_time: float

Return the desired human headway time of a vehicle type.

property length: int

Return the length of a vehicle type.

property max_acceleration: float

Return the maximum acceleration of a vehicle type.

property max_deceleration: float

Return the maximum deceleration of a vehicle type.

property max_speed: float

Return the maximum speed of a vehicle type.

property min_gap: float

Return the minimum gap of a vehicle type.

property name: str

Return the name of a vehicle type.

plafosim.platooning_vehicle.**attribute**()

perf_counter() -> float

Performance counter for benchmarking.

plafosim.platooning_vehicle.**change_gui_vehicle_color**(*vid: int*, *color: tuple*)

Change the color of a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **color** (*tuple*) – The color (R, G, B) to use for the vehicle

plafosim.platooning_vehicle.**import_module**(*name*, *package=None*)

Import a module.

The 'package' argument is required when performing a relative import. It specifies the package to use as the anchor point from which to resolve the relative import to an absolute import.

plafosim.platooning_vehicle.**is_gap_safe**(*front_position: float*, *front_speed: float*, *front_max_deceleration: float*, *front_length: float*, *back_position: float*, *back_speed: float*, *back_max_acceleration: float*, *back_min_gap: float*, *step_length: float*) → bool

Return whether the gap between the front and back vehicle is safe.

Safe means: - the front vehicle can decelerate as hard as possible for one step - the back vehicle can accelerate as hard as possible for one step - the vehicle will not crash

Assumes euclidean/non-ballistic position updates.

Parameters

- **front_position** (*float*) – The position of the front vehicle in m
- **front_speed** (*float*) – The speed of the front vehicle in m/s
- **front_max_deceleration** (*float*) – The maximum deceleration of the front vehicle in m/s2
- **front_length** (*length*) – The length of the front vehicle in m
- **back_position** (*float*) – The position of the back vehicle in m
- **back_speed** (*float*) – The speed of the back vehicle in m/s

- **back_max_acceleration** (*float*) – The maximum acceleration of the back vehicle in m/s2
- **back_min_gap** (*float*) – The minimum gap of the back vehicle in m
- **step_length** (*float*) – The length of a simulation step in s

Returns
bool

Return type
Whether the gap between the two vehicles is safe

`plafosim.platooning_vehicle.isclass(object)`

Return true if the object is a class.

Class objects provide these attributes:
__doc__ documentation string __module__ name of module in which this class was defined

`plafosim.platooning_vehicle.iter_modules(path=None, prefix='')`

Yields ModuleInfo for all submodules on path, or, if path is None, all top-level modules on sys.path.

'path' should be either None or a list of paths to look for modules in.

'prefix' is a string to output on the front of every module name on output.

`plafosim.platooning_vehicle.record_platoon_formation(basename: str, vehicle: PlatooningVehicle, candidates_found_avg: float, candidates_found_individual_avg: float, candidates_found_platoon_avg: float, candidates_filtered_avg: float)`

`plafosim.platooning_vehicle.record_platoon_trace(basename: str, step: float, vehicle: PlatooningVehicle)`

`plafosim.platooning_vehicle.record_platoon_trip(basename: str, vehicle: PlatooningVehicle, platoon_time_ratio: float, platoon_distance_ratio: float, time_until_first_platoon: float, distance_until_first_platoon: float)`

`plafosim.platooning_vehicle.record_vehicle_change(basename: str, step: float, vid: int, position: float, speed: float, source_lane: int, target_lane: int, reason: str)`

`plafosim.platooning_vehicle.record_vehicle_platoon_maneuvers(basename: str, vehicle: PlatooningVehicle)`

`plafosim.platooning_vehicle.record_vehicle_platoon_trace(basename: str, step: float, vehicle: PlatooningVehicle)`

`plafosim.platooning_vehicle.record_vehicle_teleport(basename: str, step: float, vid: int, old_position: float, old_lane: int, old_speed: float, new_position: float, new_lane: int, new_speed: float)`

`plafosim.platooning_vehicle.round_to_next_base(value: float, base: float) → float`

Round a value to the next base value.

Parameters

- **value** (*float*) – The value to round

- **base** (*float*) – The base value to round to

Returns
float

Return type
The rounded value

plafosim.simulator module

class plafosim.simulator.**CF_Model**(*value*)

Bases: *Enum*

Car Following models that vehicles can use for their mobility.

ACC = 1

CACC = 2

HUMAN = 0

class plafosim.simulator.**EmissionClass**(*value*)

Bases: *Enum*

Emission class for combustion engines using the HBEFA3 model.

Website: https://www.hbefa.net/

PC_G_EU4 = 0

property emission_factors: dict

Return the emission factors of the emission class.

Returns
dict

Return type
The emission factors of the emission class

property is_diesel: bool

Return whether the emission class is for a diesel engine.

Returns
bool

Return type
Whether the emission class is for a diesel engine

class plafosim.simulator.**Infrastructure**(*simulator: Simulator, iid: int, position: int, formation_algorithm: str, execution_interval: int, **kw_args*)

Bases: object

A class representing road side infrastructure / road side units.

The infrastructure can execute formation algorithms.

__init__(*simulator: Simulator, iid: int, position: int, formation_algorithm: str, execution_interval: int, **kw_args*)

Initialize an Infrastructure instance.

Parameters

- **simulator** (Simulator) – The global simulator object
- **iid** (*int*) – The id of the infrastructure
- **position** (*int*) – The position (x) of the infrastructure
- **formation_algorithm** (*str*) – The platoon formation (i.e., assignment calculation) algorithm to run
- **execution_interval** (*int*) – The execution interval for the formation algorithm

_action(*step: int*)

Triggers concrete actions of an infrastructure.

Parameters
step (*int*) – The current simulation step

_get_neighbors()

action(*step: int*)

Triggers actions of an infrastructure.

Parameters
step (*int*) – The current simulation step

finish()

Clean up the instance of the infrastructure.

property iid: int

Return the id of an infrastructure.

property position: int

Return the position of an infrastructure.

class plafosim.simulator.**PlatoonRole**(*value*)

Bases: *Enum*

A collection of available platoon roles.

FOLLOWER = 2

JOINER = 3

LEADER = 1

LEAVER = 4

NONE = 0

class plafosim.simulator.**PlatooningVehicle**(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Bases: *Vehicle*

A vehicle that has platooning functionality.

__init__(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float = 0, communication_range: int = 1000, acc_headway_time: float = 1.0, cacc_spacing: float = 5.0, formation_algorithm: str = None, execution_interval: int = 1, pre_filled: bool = False, **kw_args*)

Initialize a platooning vehicle instance.

Parameters

- **simulator** (*Simulator*) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (*VehicleType*) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int*) – The maximum communication range of the vehicle
- **acc_headway_time** (*float*) – The headway time for the ACC
- **cacc_spacing** (*float*) – The constant spacing for the CACC
- **formation_algorithm** (*str*) – The platoon formation algorithm to use
- **execution_interval** (*int*) – The interval for executing the formation algorithm
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a PlatooningVehicle.

Parameters

step (*float*) – The current simulation step

_calculate_emission(*a: float, v: float, f: list, scale: float*) → float

Calculate the emitted pollutant amount using the given speed and acceleration.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns

float

Return type

The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_get_available_platoons() → list

Return the available platoon candidates of the vehicle.

This imitates neighbor maintenance by using a neighbor table.

Returns
list(PlatooninVehicle)

Return type
The list of available platoons

_join(*platoon_id: int*, *leader_id: int*)

Lets a vehicle join a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

Parameters

- **platoon_id** (*int*) – The id of the target platoon
- **leader_id** (*int*) – The id of the leader of the target platoon

_join_teleport(*leader:* PlatooningVehicle, *last:* PlatooningVehicle, *new_position: float*)

Perform the actual teleporting of the join maneuver.

Parameters

- **leader** (PlatooningVehicle) – The leader of the target platoon
- **last** (PlatooningVehicle) – The last vehicle of the target platoon
- **new_position** (*float*) – The new position for joining the target platoon

_leave()

Lets a vehicle leave a platoon.

Communication and fine-grained maneuver control is out-of-scope and thus omitted.

_left_lane_blocked() → bool

Check whether a vehicle can move to the left lane in order to leave its current platoon.

Returns
bool

Return type
Whether the left lane is blocked

_start()

Start this Vehicle.

_statistics()

Write continuous statistics.

_teleport(*new_position: float*, *new_lane: int*, *new_speed: float*)

Teleport a vehicle to a given position.

Parameters

- **new_position** (*float*) – The new position
- **new_lane** (*int*) – The new lane
- **new_speed** (*float*) – The new speed

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

calculate_approaching_time(*target_position: float*, *target_speed: float*) → float

Calculate approximate time to approach the target position at target speed.

Parameters

- **target_position** (*float*) – The target position to approach
- **target_speed** (*float*) – The target speed after approaching

Returns
float

Return type
The calculated approaching time

finish()

Clean up the instance of the PlatooningVehicle.

This includes leaving the platoon and mostly statistics recording.

get_front_gap() → float

Return the gap to the vehicle in the front.

This imitates a measurement of the front radar sensor.

get_front_speed() → float

Return the speed to the vehicle in the front.

This imitates a measurement of the front radar sensor.

info() → str

Return information about the PlatooningVehicle.

is_in_platoon() → bool

Return whether the vehicle currently is in a platoon.

This is based on the current PlatoonRole. A joining or leaving vehicle is either not yet or still part of a platoon, thus the returned value should be true.

property acc_headway_time: float

Return the ACC headway time of the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: tuple

Return the current color of the vehicle.

property depart_lane: int

Return the departure lane of the vehicle.

property depart_position: int

Return the departure position of the vehicle.

property depart_speed: float

Return the departure speed of the vehicle.

property depart_time: float

Return the departure time of the vehicle.

property desired_gap: float

Return the desired gap of the vehicle.

This is based on the currently active car following model.

property desired_headway_time: float

Return the desired headway time of the vehicle.

This is based on the currently active car following model.

property desired_speed: float

Return the desired driving speed of the vehicle.

If the vehicle is in a platoon, it returns the desired driving speed of the entire platoon.

property distance_in_platoon: float

Return the travelled distance within platoons.

property headway_time: float

Return the human headway time of the vehicle.

This is based on the vehicle type.

property in_maneuver: bool

Return whether the vehicle is currently in a maneuver.

property lane: int

Return the current lane of the vehicle.

property length: int

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: float

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: float

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: float

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: float

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property platoon: *Platoon*

Return the platoon of the vehicle.

property platoon_role: *PlatoonRole*

Return the current platoon role of the vehicle.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property time_in_platoon: int

Return the travelled time within platoons.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: int

Return the id of the vehicle.

class plafosim.simulator.**Simulator**(*, *road_length: int = 100000*, *number_of_lanes: int = 3*, *ramp_interval: int = 5000*, *pre_fill: bool = False*, *number_of_vehicles: int = 100*, *vehicle_density: float = -1*, *max_speed: float = 55*, *acc_headway_time: float = 1.0*, *cacc_spacing: float = 5.0*, *penetration_rate: float = 1.0*, *random_depart_position: bool = False*, *depart_all_lanes: bool = True*, *desired_speed: float = 33.0*, *random_desired_speed: bool = True*, *speed_variation: float = 0.1*, *min_desired_speed: float = 22.0*, *max_desired_speed: float = 44.0*, *random_depart_speed: bool = False*, *depart_desired: bool = False*, *depart_flow: bool = False*, *depart_method: str = 'interval'*, *depart_interval: float = 2.0*, *depart_probability: float = 1.0*, *depart_rate: int = 3600*, *random_arrival_position: bool = False*, *minimum_trip_length: int = 0*, *maximum_trip_length: int = -1000*, *communication_range: int = 500*, *distributed_platoon_knowledge: bool = True*, *distributed_maneuver_knowledge: bool = False*, *start_as_platoon: bool = False*, *reduced_air_drag: bool = True*, *maximum_teleport_distance: int = 2000*, *maximum_approach_time: int = 60*, *delay_teleports: bool = True*, *update_desired_speed: bool = True*, *formation_algorithm: str | None = None*, *formation_strategy: str = 'distributed'*, *execution_interval: int = 10*, *number_of_infrastructures: int = 0*, *step_length: float = 1.0*, *max_step: int = 3600*, *actions: bool = True*, *collisions: bool = True*, *random_seed: int = -1*, *log_level: int = 30*, *progress: bool = True*, *gui: bool = False*, *gui_delay: int = 0*, *gui_track_vehicle: int = -1*, *sumo_config: str = 'sumocfg/freeway.sumo.cfg'*, *gui_play: int = True*, *gui_start: int = 0*, *draw_ramps: bool = True*, *draw_ramp_labels: bool = False*, *draw_road_end: bool = True*, *draw_road_end_label: bool = True*, *draw_infrastructures: bool = True*, *draw_infrastructure_labels: bool = True*, *screenshot_filename: str | None = None*, *result_base_filename: str = 'results'*, *record_simulation_trace: bool = False*, *record_end_trace: bool = True*, *record_vehicle_trips: bool = False*, *record_vehicle_emissions: bool = False*, *record_vehicle_traces: bool = False*, *record_vehicle_changes: bool = False*, *record_emission_traces: bool = False*, *record_platoon_trips: bool = False*, *record_platoon_maneuvers: bool = False*, *record_platoon_formation: bool = False*, *record_platoon_traces: bool = False*, *record_vehicle_platoon_traces: bool = False*, *record_platoon_changes: bool = False*, *record_infrastructure_assignments: bool = False*, *record_vehicle_teleports: bool = False*, *record_prefilled: bool = False*, ***kwargs: dict*)

Bases: `object`

A collection of parameters and information of the simulator.

__init__(*, *road_length: int = 100000*, *number_of_lanes: int = 3*, *ramp_interval: int = 5000*, *pre_fill: bool = False*, *number_of_vehicles: int = 100*, *vehicle_density: float = -1*, *max_speed: float = 55*, *acc_headway_time: float = 1.0*, *cacc_spacing: float = 5.0*, *penetration_rate: float = 1.0*, *random_depart_position: bool = False*, *depart_all_lanes: bool = True*, *desired_speed: float = 33.0*, *random_desired_speed: bool = True*, *speed_variation: float = 0.1*, *min_desired_speed: float = 22.0*, *max_desired_speed: float = 44.0*, *random_depart_speed: bool = False*, *depart_desired: bool = False*, *depart_flow: bool = False*, *depart_method: str = 'interval'*, *depart_interval: float = 2.0*, *depart_probability: float = 1.0*, *depart_rate: int = 3600*, *random_arrival_position: bool = False*, *minimum_trip_length: int = 0*, *maximum_trip_length: int = -1000*, *communication_range: int = 500*, *distributed_platoon_knowledge: bool = True*, *distributed_maneuver_knowledge: bool = False*, *start_as_platoon: bool = False*, *reduced_air_drag: bool = True*, *maximum_teleport_distance: int = 2000*, *maximum_approach_time: int = 60*, *delay_teleports: bool = True*, *update_desired_speed: bool = True*, *formation_algorithm: str | None = None*, *formation_strategy: str = 'distributed'*, *execution_interval: int = 10*, *number_of_infrastructures: int = 0*, *step_length: float = 1.0*, *max_step: int = 3600*, *actions: bool = True*, *collisions: bool = True*, *random_seed: int = -1*, *log_level: int = 30*, *progress: bool = True*, *gui: bool = False*, *gui_delay: int = 0*, *gui_track_vehicle: int = -1*, *sumo_config: str = 'sumocfg/freeway.sumo.cfg'*, *gui_play: int = True*, *gui_start: int = 0*, *draw_ramps: bool = True*, *draw_ramp_labels: bool = False*, *draw_road_end: bool = True*, *draw_road_end_label: bool = True*, *draw_infrastructures: bool = True*, *draw_infrastructure_labels: bool = True*, *screenshot_filename: str | None = None*, *result_base_filename: str = 'results'*, *record_simulation_trace: bool = False*, *record_end_trace: bool = True*, *record_vehicle_trips: bool = False*, *record_vehicle_emissions: bool = False*, *record_vehicle_traces: bool = False*, *record_vehicle_changes: bool = False*, *record_emission_traces: bool = False*, *record_platoon_trips: bool = False*, *record_platoon_maneuvers: bool = False*, *record_platoon_formation: bool = False*, *record_platoon_traces: bool = False*, *record_vehicle_platoon_traces: bool = False*, *record_platoon_changes: bool = False*, *record_infrastructure_assignments: bool = False*, *record_vehicle_teleports: bool = False*, *record_prefilled: bool = False*, ***kwargs: dict*)

Initialize a simulator instance.

_add_vehicle(*vid: int*, *vtype:* VehicleType, *depart_position: float*, *arrival_position: float*, *desired_speed: float*, *depart_lane: int*, *depart_speed: float*, *depart_time: float*, *depart_delay: float = 0*, *communication_range: int = 500*, *pre_filled: bool = False*) → *Vehicle*

Add a vehicle to the simulation based on the given parameters.

NOTE: Make sure that you set last_vehicle_id correctly.

Parameters

- **vid** (*int*) – The id of the vehicle
- **vtype** (VehicleType) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float, optional*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int, optional*) – The maximum communication range of the vehicle

- **pre_filled** (*bool, optional*) – Whether this vehicle was pre-filled

Returns
The added vehicle

Return type
Vehicle

_call_infrastructure_actions()

Triggers actions on all infrastructures in the simulation.

_call_vehicle_actions()

Triggers actions on all vehicles in the simulation.

_finish()

Clean up the simulation.

_generate_infrastructures(*number_of_infrastructures: int*)

Generate infrastructures for the simulation.

Parameters
number_of_infrastructures (*int*) – The number of infrastructures to generate

_generate_vehicles()

Add pre-filled vehicles to the simulation.

_get_predecessor(*vehicle:* Vehicle, *lane: int = -1*) → *Vehicle*

Return the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_predecessor_rear_position(*vehicle:* Vehicle, *lane: int = -1*) → float

Return the rear position of the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_predecessor_speed(*vehicle:* Vehicle, *lane: int = -1*) → float

Return the speed of the preceding (i.e., front) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider
- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_successor(*vehicle:* Vehicle, *lane: int = -1*) → *Vehicle*

Return the succeeding (i.e., back) vehicle for a given vehicle on a given lane.

Parameters

- **vehicle** (Vehicle) – The vehicle to consider

- **lane** (*int, optional*) – The lane to consider. A lane of -1 indicates the vehicle's current lane.

_get_vehicles_df() → DataFrame

Return a pandas Dataframe from the internal data structure.

Returns
The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Return type
pandas.DataFrame

_initialize_gui()

Initialize the GUI.

_initialize_prefilled_platoon()

Initialize all pre-filled vehicles as one platoon.

_initialize_result_recording()

Create output files for all (enabled) statistics and writes the headers.

_record_lane_changes(*vdf: DataFrame*)

Record lane changes.

Parameters
vdf (*pd.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

_remove_arrived_vehicles(*arrived_vehicles: list*)

Remove arrived vehicles from the simulation.

Parameters
arrived_vehicles (*list*) – The ids of arrived vehicles

_spawn_vehicles(*vdf: DataFrame*)

Spawns vehicles within the current step.

1) Calculate how many vehicles should be spawned according to the departure method
2) Calculate properties for these vehicles (e.g., desired speed)
3) Add vehicles to spawn queue
4) Spawn as many vehicles as possible from the queue (sorted by waiting time)
5) Update queue

_statistics(*vehicles_in_simulator: int*, *vehicles_in_queue: int*, *vehicles_spawned: int*, *vehicles_arrived: int*, *runtime: float*, *average_vehicle_speed: float*, *vehicles_braking_rough: int*)

Record some period statistics.

Parameters

- **vehicles_in_simulator** (*int*) – The number of vehicles in the scenario within this step
- **vehicles_in_queue** (*int*) – The number of vehicles in the spawn queue within this step
- **vehicles_spawned** (*int*) – The number of vehicles that departed within this step
- **vehicles_arrived** (*int*) – The number of vehicles that arrived within this step
- **runtime** (*float*) – The run time of this step

- **average_vehicle_speed** (*int*) – The average driving speed amog all vehicles in the scenario within this step
- **vehicles_braking_rough** (*int*) – The number of vehicles performing rough braking within this step

_update_gui()

Update the GUI.

_vehicles_to_be_scheduled() → int

Calculate how many vehicles should be spawned according to the departure method.

Returns
int

Return type
The number of vehicles to be spawned within this time step

_write_back_vehicles_df(*vdf: DataFrame*)

Write back the vehicle updates from a given pandas dataframe to the internal data structure.

Parameters
vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

run()

Run the simulation with the specified parameters until it is stopped.

Main simulation method.

This is based on Krauss' multi lane traffic: laneChange(); adjust(); move();

stop(*msg: str*)

Stop the simulation with the given message.

Parameters
msg (*str*) – The message to show after stopping the simulation

property number_of_lanes: int

Return the number of lanes.

property road_length: int

Return the road length in m.

property step: int

Return the current simulation step.

property step_length: float

Return the length of a simulation step.

class plafosim.simulator.**Vehicle**(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float, communication_range: int, pre_filled: bool = False*)

Bases: object

A collection of state information for a vehicle in the simulation.

A vehicle can really be anything that can move and can be defined by a vehicle type. It does not necessarily be driven by a computer (i.e., autonomous). However, by default it does have V2X functionality.

__init__(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float, communication_range: int, pre_filled: bool = False*)

Initialize a vehicle instance.

Parameters

- **simulator** (Simulator) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (VehicleType) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int*) – The maximum communication range of the vehicle
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a vehicle.

Parameters

step (*float*) – The current simulation step

_calculate_emission(*a: float, v: float, f: list, scale: float*) → float

Calculate the actual emission of the vehicle.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns

float

Return type

The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_start()

Start this Vehicle.

_statistics()

Write continuous statistics for the vehicle.

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

finish()

Clean up the instance of the vehicle.

This includes mostly statistic recording.

info() → str

Return information about the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: tuple

Return the current color of the vehicle.

property depart_lane: int

Return the departure lane of the vehicle.

property depart_position: int

Return the departure position of the vehicle.

property depart_speed: float

Return the departure speed of the vehicle.

property depart_time: float

Return the departure time of the vehicle.

property desired_gap: float

Return the desired gap to the vehicle in front of the vehicle.

This is based on the desired headway time and the current driving speed.

property desired_headway_time: float

Return the desired headway time of the vehicle.

property desired_speed: float

Return the desired driving speed of the vehicle.

property headway_time: float

Return the human headway time of the vehicle.

This is based on the vehicle type.

property lane: int

Return the current lane of the vehicle.

property length: int

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: float

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: float

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: float

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: float

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: int

Return the id of the vehicle.

class plafosim.simulator.**VehicleType**(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Bases: object

A collection of parameters for a concrete vehicle type.

__init__(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Initialize a specific vehicle type.

Parameters

- **name** (*str*) – The name of the vehicle type
- **length** (*int*) – The length of the vehicle type
- **max_speed** (*float*) – The maximum speed of the vehicle type
- **max_acceleration** (*float*) – The maximum acceleration of the vehicle type
- **max_deceleration** (*float*) – The maximum deceleration of the vehicle type
- **min_gap** (*float*) – The minimum safety gap to the vehicle in front of the vehicle type
- **headway_time** (*float*) – The human headway time of the vehicle type
- **emission_class** (EmissionClass) – The emission class of the vehicle type

property emission_class: *EmissionClass*

Return the emission class of a vehicle type.

property emission_factors: dict

Return the emission factors of a vehicle type.

property headway_time: float

Return the desired human headway time of a vehicle type.

property length: int

Return the length of a vehicle type.

property max_acceleration: float

Return the maximum acceleration of a vehicle type.

property max_deceleration: float

Return the maximum deceleration of a vehicle type.

property max_speed: float

Return the maximum speed of a vehicle type.

property min_gap: float

Return the minimum gap of a vehicle type.

property name: str

Return the name of a vehicle type.

class plafosim.simulator.**tqdm**(*_, **__)

Bases: Comparable

Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a dynamically updating progressbar every time a value is requested.

__init__(*iterable=None*, *desc=None*, *total=None*, *leave=True*, *file=None*, *ncols=None*, *mininterval=0.1*, *maxinterval=10.0*, *miniters=None*, *ascii=None*, *disable=False*, *unit='it'*, *unit_scale=False*, *dynamic_ncols=False*, *smoothing=0.3*, *bar_format=None*, *initial=0*, *position=None*, *postfix=None*, *unit_divisor=1000*, *write_bytes=None*, *lock_args=None*, *nrows=None*, *colour=None*, *delay=0*, *gui=False*, ***kwargs*)

Parameters

- **iterable** (*iterable, optional*) – Iterable to decorate with a progressbar. Leave blank to manually manage the updates.
- **desc** (*str, optional*) – Prefix for the progressbar.

- **total** (*int or float, optional*) – The number of expected iterations. If unspecified, len(iterable) is used if possible. If float("inf") or as a last resort, only basic progress statistics are displayed (no ETA, no progressbar). If *gui* is True and this parameter needs subsequent updating, specify an initial arbitrary large positive number, e.g. 9e9.
- **leave** (*bool, optional*) – If [default: True], keeps all traces of the progressbar upon termination of iteration. If *None*, will leave only if *position* is *0*.
- **file** (*io.TextIOWrapper* or *io.StringIO*, optional) – Specifies where to output the progress messages (default: sys.stderr). Uses *file.write(str)* and *file.flush()* methods. For encoding, see *write_bytes*.
- **ncols** (*int, optional*) – The width of the entire output message. If specified, dynamically resizes the progressbar to stay within this bound. If unspecified, attempts to use environment width. The fallback is a meter width of 10 and no limit for the counter and statistics. If 0, will not print any meter (only stats).
- **mininterval** (*float, optional*) – Minimum progress display update interval [default: 0.1] seconds.
- **maxinterval** (*float, optional*) – Maximum progress display update interval [default: 10] seconds. Automatically adjusts *miniters* to correspond to *mininterval* after long display update lag. Only works if *dynamic_miniters* or monitor thread is enabled.
- **miniters** (*int or float, optional*) – Minimum progress display update interval, in iterations. If 0 and *dynamic_miniters*, will automatically adjust to equal *mininterval* (more CPU efficient, good for tight loops). If > 0, will skip display of specified number of iterations. Tweak this and *mininterval* to get very efficient loops. If your progress is erratic with both fast and slow iterations (network, skipping items, etc) you should set miniters=1.
- **ascii** (*bool or str, optional*) – If unspecified or False, use unicode (smooth blocks) to fill the meter. The fallback is to use ASCII characters " 123456789#".
- **disable** (*bool, optional*) – Whether to disable the entire progressbar wrapper [default: False]. If set to None, disable on non-TTY.
- **unit** (*str, optional*) – String that will be used to define the unit of each iteration [default: it].
- **unit_scale** (*bool or int or float, optional*) – If 1 or True, the number of iterations will be reduced/scaled automatically and a metric prefix following the International System of Units standard will be added (kilo, mega, etc.) [default: False]. If any other non-zero number, will scale *total* and *n*.
- **dynamic_ncols** (*bool, optional*) – If set, constantly alters *ncols* and *nrows* to the environment (allowing for window resizes) [default: False].
- **smoothing** (*float, optional*) – Exponential moving average smoothing factor for speed estimates (ignored in GUI mode). Ranges from 0 (average speed) to 1 (current/instantaneous speed) [default: 0.3].
- **bar_format** (*str, optional*) – Specify a custom bar string formatting. May impact performance. [default: '{l_bar}{bar}{r_bar}'], where l_bar='{desc}: {percentage:3.0f}%|' and r_bar='| {n_fmt}/{total_fmt} [{elapsed}<{remaining}, '

  '{rate_fmt}{postfix}]'

  Possible vars: l_bar, bar, r_bar, n, n_fmt, total, total_fmt,
  percentage, elapsed, elapsed_s, ncols, nrows, desc, unit, rate, rate_fmt, rate_noinv, rate_noinv_fmt, rate_inv, rate_inv_fmt, postfix, unit_divisor, remaining, remaining_s, eta.

Note that a trailing ": " is automatically removed after {desc} if the latter is empty.

- **initial** (*int or float, optional*) – The initial counter value. Useful when restarting a progress bar [default: 0]. If using float, consider specifying *{n:.3f}* or similar in *bar_format*, or specifying *unit_scale*.
- **position** (*int, optional*) – Specify the line offset to print this bar (starting from 0) Automatic if unspecified. Useful to manage multiple bars at once (eg, from threads).
- **postfix** (dict or *, optional) – Specify additional stats to display at the end of the bar. Calls *set_postfix(**postfix)* if possible (dict).
- **unit_divisor** (*float, optional*) – [default: 1000], ignored unless *unit_scale* is True.
- **write_bytes** (*bool, optional*) – If (default: None) and *file* is unspecified, bytes will be written in Python 2. If *True* will also write bytes. In all other cases will default to unicode.
- **lock_args** (*tuple, optional*) – Passed to *refresh* for intermediate output (initialisation, iterating, and updating).
- **nrows** (*int, optional*) – The screen height. If specified, hides nested bars outside this bound. If unspecified, attempts to use environment height. The fallback is 20.
- **colour** (*str, optional*) – Bar colour (e.g. 'green', '#00ff00').
- **delay** (*float, optional*) – Don't display until [default: 0] seconds have elapsed.
- **gui** (*bool, optional*) – WARNING: internal parameter - do not use. Use tqdm.gui.tqdm(...) instead. If set, will attempt to use matplotlib animations for a graphical output [default: False].

Returns
out

Return type
decorated iterator.

classmethod _decr_instances(*instance*)

Remove from list and reposition another unfixed bar to fill the new gap.

This means that by default (where all nested bars are unfixed), order is not maintained but screen flicker/blank space is minimised. (tqdm<=4.44.1 moved ALL subsequent unfixed bars up.)

classmethod _get_free_pos(*instance=None*)

Skips specified instance.

clear(*nolock=False*)

Clear current bar display.

close()

Cleanup and (if leave=False) close the progressbar.

display(*msg=None*, *pos=None*)

Use *self.sp* to display *msg* in the specified *pos*.

Consider overloading this function when inheriting to use e.g.: *self.some_frontend(**self.format_dict)* instead of *self.sp*.

Parameters

- **msg** (str, optional. What to display (default: *repr(self)*).) –
- **pos** (int, optional. Position to *moveto*) – (default: *abs(self.pos)*).

classmethod external_write_mode(*file=None*, *nolock=False*)

Disable tqdm within context and refresh tqdm when exits. Useful when writing to standard output stream

static format_interval(*t*)

Formats a number of seconds as a clock time, [H:]MM:SS

Parameters
t (*int*) – Number of seconds.

Returns
out – [H:]MM:SS

Return type
str

static format_meter(*n*, *total*, *elapsed*, *ncols=None*, *prefix=''*, *ascii=False*, *unit='it'*, *unit_scale=False*, *rate=None*, *bar_format=None*, *postfix=None*, *unit_divisor=1000*, *initial=0*, *colour=None*, ***extra_kwargs*)

Return a string-based progress bar given some parameters

Parameters

- **n** (*int or float*) – Number of finished iterations.
- **total** (*int or float*) – The expected total number of iterations. If meaningless (None), only basic progress statistics are displayed (no ETA).
- **elapsed** (*float*) – Number of seconds passed since start.
- **ncols** (*int, optional*) – The width of the entire output message. If specified, dynamically resizes *{bar}* to stay within this bound [default: None]. If *0*, will not print any bar (only stats). The fallback is *{bar:10}*.
- **prefix** (*str, optional*) – Prefix message (included in total width) [default: ''']. Use as {desc} in bar_format string.
- **ascii** (*bool, optional or str, optional*) – If not set, use unicode (smooth blocks) to fill the meter [default: False]. The fallback is to use ASCII characters " 123456789#".
- **unit** (*str, optional*) – The iteration unit [default: 'it'].
- **unit_scale** (*bool or int or float, optional*) – If 1 or True, the number of iterations will be printed with an appropriate SI metric prefix (k = 10^3, M = 10^6, etc.) [default: False]. If any other non-zero number, will scale *total* and *n*.
- **rate** (*float, optional*) – Manual override for iteration rate. If [default: None], uses n/elapsed.
- **bar_format** (*str, optional*) – Specify a custom bar string formatting. May impact performance. [default: '{l_bar}{bar}{r_bar}'], where l_bar='{desc}: {percentage:3.0f}%|' and r_bar='| {n_fmt}/{total_fmt} [{elapsed}<{remaining}, '

  '{rate_fmt}{postfix}]'

  Possible vars: l_bar, bar, r_bar, n, n_fmt, total, total_fmt,
  percentage, elapsed, elapsed_s, ncols, nrows, desc, unit, rate, rate_fmt, rate_noinv, rate_noinv_fmt, rate_inv, rate_inv_fmt, postfix, unit_divisor, remaining, remaining_s, eta.

  Note that a trailing ": " is automatically removed after {desc} if the latter is empty.

- **postfix** (*, optional) – Similar to *prefix*, but placed at the end (e.g. for additional stats). Note: postfix is usually a string (not a dict) for this method, and will if possible be set to postfix = ', ' + postfix. However other types are supported (#382).
- **unit_divisor** (*float, optional*) – [default: 1000], ignored unless *unit_scale* is True.
- **initial** (*int or float, optional*) – The initial counter value [default: 0].
- **colour** (*str, optional*) – Bar colour (e.g. 'green', '#00ff00').

Returns
out

Return type
Formatted meter and stats, ready to display.

static format_num(*n*)

Intelligent scientific notation (.3g).

Parameters
n (*int or float or Numeric*) – A Number.

Returns
out – Formatted number.

Return type
str

static format_sizeof(*num*, *suffix=''*, *divisor=1000*)

Formats a number (greater than unity) with SI Order of Magnitude prefixes.

Parameters

- **num** (*float*) – Number (>= 1) to format.
- **suffix** (*str, optional*) – Post-postfix [default: ''].
- **divisor** (*float, optional*) – Divisor between prefixes [default: 1000].

Returns
out – Number with Order of Magnitude SI unit postfix.

Return type
str

classmethod get_lock()

Get the global lock. Construct it if it does not exist.

moveto(*n*)

classmethod pandas(***tqdm_kwargs*)

Registers the current *tqdm* class with
pandas.core. (frame.DataFrame | series.Series | groupby.(generic.)DataFrameGroupBy | groupby.(generic.)SeriesGroupBy).progress_apply

A new instance will be create every time *progress_apply* is called, and each instance will automatically *close()* upon completion.

Parameters
tqdm_kwargs (*arguments for the tqdm instance*) –

Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> from tqdm import tqdm
>>> from tqdm.gui import tqdm as tqdm_gui
>>>
>>> df = pd.DataFrame(np.random.randint(0, 100, (100000, 6)))
>>> tqdm.pandas(ncols=50)  # can use tqdm_gui, optional kwargs, etc
>>> # Now you can use `progress_apply` instead of `apply`
>>> df.groupby(0).progress_apply(lambda x: x**2)
```

References

<https://stackoverflow.com/questions/18603270/ progress-indicator-during-pandas-operations-python>

refresh(*nolock=False*, *lock_args=None*)

Force refresh the display of this bar.

Parameters

- **nolock** (*bool, optional*) – If *True*, does not lock. If [default: *False*]: calls *acquire()* on internal lock.
- **lock_args** (*tuple, optional*) – Passed to internal lock's *acquire()*. If specified, will only *display()* if *acquire()* returns *True*.

reset(*total=None*)

Resets to 0 iterations for repeated use.

Consider combining with *leave=True*.

Parameters

total (*int or float, optional. Total to use for the new bar.*) –

set_description(*desc=None*, *refresh=True*)

Set/modify description of the progress bar.

Parameters

- **desc** (*str, optional*) –
- **refresh** (*bool, optional*) – Forces refresh [default: True].

set_description_str(*desc=None*, *refresh=True*)

Set/modify description without ': ' appended.

classmethod set_lock(*lock*)

Set the global lock.

set_postfix(*ordered_dict=None*, *refresh=True*, ***kwargs*)

Set/modify postfix (additional stats) with automatic formatting based on datatype.

Parameters

- **ordered_dict** (*dict or OrderedDict, optional*) –
- **refresh** (*bool, optional*) – Forces refresh [default: True].
- **kwargs** (*dict, optional*) –

set_postfix_str(*s=''*, *refresh=True*)

Postfix without dictionary expansion, similar to prefix handling.

static status_printer(*file*)

Manage the printing and in-place updating of a line of characters. Note that if the string is longer than a line, then in-place updating may not work (it will print a new line at each refresh).

unpause()

Restart tqdm timer from last print time.

update(*n=1*)

Manually update the progress bar, useful for streams such as reading files. E.g.: >>> t = tqdm(total=filesize) # Initialise >>> for current_buffer in stream: … … … t.update(len(current_buffer)) >>> t.close() The last line is highly recommended, but possibly not necessary if *t.update()* will be called in such a way that *filesize* will be exactly reached and printed.

Parameters

n (*int or float, optional*) – Increment to add to the internal counter of iterations [default: 1]. If using float, consider specifying *{n:.3f}* or similar in *bar_format*, or specifying *unit_scale*.

Returns

out – True if a *display()* was triggered.

Return type

bool or None

classmethod wrapattr(*stream*, *method*, *total=None*, *bytes=True*, ***tqdm_kwargs*)

stream : file-like object. method : str, "read" or "write". The result of *read()* and

the first argument of *write()* should have a *len()*.

```
>>> with tqdm.wrapattr(file_obj, "read", total=file_obj.size) as fobj:
...     while True:
...         chunk = fobj.read(chunk_size)
...         if not chunk:
...             break
```

classmethod write(*s*, *file=None*, *end='\n'*, *nolock=False*)

Print a message via tqdm (without overlap with bars).

property _comparable

_instances = set()

property format_dict

Public API for read-only member access.

monitor = None

monitor_interval = 10

plafosim.simulator.**add_gui_vehicle**(*vid: int*, *position: float*, *lane: int*, *speed: float*, *color: tuple = (0, 255, 0)*, *track: bool = False*)

Add a vehicle to the GUI.

Parameters

- **vid** (*int*) – The vehicle's id

- **position** (*float*) – The vehicle's current position
- **lane** (*int*) – The vehicle's current lane
- **speed** (*float*) – The vehicle's current speed
- **color** (*tuple*) – The vehicle's current color
- **track** (*bool*) – Whether to track this vehicle within the GUI

`plafosim.simulator.`**`assert_index_equal`**`(`*one*, *two*`)` → bool

Ensure the indices of two Sequences/DataFrames are equal.

Parameters

- **one** (*pandas.Sequence / pandas.DataFrame*) – The first object for the comparison
- **two** (*pandas.Sequence / pandas.DataFrame*) – The second object for the comparsion

Returns
bool

Return type
Whether the two indices are equal

`plafosim.simulator.`**`check_and_prepare_gui`**`()`

Check and prepare GUI environment.

`plafosim.simulator.`**`check_collisions`**`(`*vdf: DataFrame*`)` → bool

Do collision checks for all vehicles in the simulation.

Parameters
vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Returns
Whether there are collisions between vehicles

Return type
bool

`plafosim.simulator.`**`close_gui`**`()`

Close the GUI.

`plafosim.simulator.`**`compute_lane_changes`**`(`*vdf: DataFrame*, *max_lane: int*, *step_length: float*`)` → DataFrame

Find desired and safe lane changes.

1) compute all speed gains
2) "apply" all speed gains
3) compute all keep rights
4) "apply" all keep rights

This is based on Krauss' multi lane traffic: laneChange() congested = (v_safe < v_thresh) and (v^0_safe < v_thresh) favorable(right->left) = (v_safe < v_max) and (not congested) favorable(left->right) = (v_safe >= v_max) and (v^0_safe >= v_max) if ((favorable(i->j) or (rand < p_change)) and safe(i->j)) then change(i->j) for vehicles on the right lane: if (v > v^0_safe) and (not congested) then v <- v^0_safe

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

- **max_lane** (*int*) – The largest lane id
- **step_length** (*float*) – The length of a simulation step

Returns
The Dataframe containing the vehicles as rows index: vid columns: [lane, reason]

Return type
pandas.DataFrame

`plafosim.simulator.`**`compute_new_speeds`**(*vdf: DataFrame*, *step_length: float*) → Series

Compute the new speed for all vehicles in the simulation.

Can compute "potential" new speed for different target lanes. Assume vdf already contains predecessor and successor data. Just pass the right predecessor/successor data for different lanes.

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **step_length** (*float*) – The length of a simulation step

Returns
The series of new speeds index: vid

Return type
pandas.Series

`plafosim.simulator.`**`compute_vehicle_spawns`**(*vehicles: list*, *vdf: DataFrame*, *ramp_positions: list*, *number_of_lanes: int*, *current_step: float*, *rng: Random*, *random_depart_position: bool*, *random_arrival_position: bool*, *depart_all_lanes: bool*, *step_length: float = 1.0*)

Spawn as many vehicles as possible from the queue (sorted by waiting time).

Assumption: list of vehicles is already sorted ascending by departure priority (e.g., waiting time) Assumption: ramp positions is sorted in ascending manner

Parameters

- **vehicles** (*list(dict)*) – The list of vehicles to add
- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **ramp_positions** (*list(int)*) – The list of available on-ramp positions in m
- **number_of_lanes** (*int*) – The number of available lanes
- **current_step** (*float*) – The current simulation step
- **rng** (*random.Random*) – The random number generator to use
- **random_depart_position** (*bool*) – Whether the vehicles sould depart at a random position
- **random_arrival_position** (*bool*) – Whether the vehicles should arrive at a random position
- **depart_all_lanes** (*bool*) – Whether to use all availale lanes for depature
- **step_length** (*float, optioal*) – The length of a simulation step

`plafosim.simulator.draw_infrastructures(infrastructures: list, labels: bool)`

Draws infrastructures in the GUI.

Parameters

- **infrastructures** (*list*) – The list of infrastructure objects to add
- **labels** (*bool*) – Whether to draw infrastructure labels

`plafosim.simulator.draw_ramps(road_length: int, interval: int, labels: bool)`

Draws on-/off-ramps in the GUI.

Parameters

- **road_length** (*int*) – The length of the road in m
- **interval** (*int*) – The ramp interval in m
- **labels** (*bool*) – Whether to draw ramp labels

`plafosim.simulator.draw_road_end(road_length: int, label: bool)`

Draws the end of the road in the GUI.

Parameters

- **road_length** (*int*) – The length of the road in m
- **label** (*bool*) – Whether to draw a label

`plafosim.simulator.get_arrival_position(depart_position: int, road_length: int, ramp_interval: int, min_trip_length: int, max_trip_length: int, rng: Random, random_arrival_position: bool = False, pre_fill: bool = False) → int`

Return a (random) arrival position for a given departure position.

This considers the ramp interval, road length, and minimum trip length.

Parameters

- **depart_position** (*int*) – The departure position to consider
- **road_length** (*int*) – The length of the entire road
- **ramp_interval** (*int*) – The distance between two on-/off-ramps
- **min_trip_length** (*int*) – The minimum trip length
- **max_trip_length** (*int*) – The maximum trip length
- **rng** (*random.Random*) – The random number generator to use
- **random_arrival_position** (*bool*) – Whether to use random arrival positions
- **pre_fill** (*bool, optional*) – Whether the trip is for a pre-filled vehicle

Returns

The arrival position in m

Return type

int

`plafosim.simulator.get_crashed_vehicles(vdf: DataFrame) → list`

Return the list of crashed vehicles' ids.

Parameters
vdf (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Returns
list(int)

Return type
the list of vehicles that crashed

`plafosim.simulator.`**`get_depart_speed`**(*desired_speed: float*, *rng: Random*, *depart_desired: bool = False*, *random_depart_speed: bool = False*) → float

Return a (random) departure speed.

Parameters

- **desired_speed** (*float*) – The desired speed to consider
- **rng** (*random.Random*) – The random number generator to use
- **depart_desired** (*bool, optional*) – Whether to depart with the desired speed
- **random_depart_speed** (*bool, optional*) – Whether to choose a random value from a range

Returns
The value for the departure speed

Return type
float

`plafosim.simulator.`**`get_desired_speed`**(*desired_speed: float*, *rng: Random*, *speed_variation: float*, *min_desired_speed: float*, *max_desired_speed: float*, *random_desired_speed: bool = False*) → float

Return a (random) desired driving speed.

Parameters

- **desired_speed** (*float*) – The value to be used as is or as the mean for sampling from a normal distribution
- **rng** (*random.Random*) – The random number generator to use
- **speed_variation** (*float*) – The value to be used as variation for sampling from a normal distribution
- **min_desired_speed** (*float*) – The minimum allowed value for the desired driving speed
- **max_desired_speed** (*float*) – The maximum allowed value for the desired driving speed
- **random_desired_speed** (*bool, optional*) – Whether to choose a random value from a normal distribution

Returns
The value for the desired driving speed

Return type
float

`plafosim.simulator.`**`get_predecessors`**(*vdf: DataFrame*, *predecessor_map: DataFrame*, *target_lane: Series*) → DataFrame

Return DataFrame of successors to the vehicles on a target lane.

plafosim.simulator.**gui_step**(*target_step: int*, *screenshot_filename: str | None = None*)

Increases the simulation step in the GUI.

Parameters

- **target_step** (*int*) – The target simulation step
- **screenshot_filename** (*str, optional*) – The name of the screenshot file

plafosim.simulator.**has_collision**(*position1: float*, *rear_position1: float*, *lane1: int*, *position2: float*, *rear_position2: float*, *lane2: int*) → bool

Check for a collision between two vehicles.

Parameters

- **position1** (*float*) – The current position of vehicle 1
- **rear_position1** (*float*) – The current rear position of vehicle 1
- **lane1** (*int*) – The current lane of vehicle 1
- **position2** (*float*) – The current position of vehicle 2
- **rear_position2** (*float*) – The current rear position of vehicle 2
- **lane2** (*int*) – The current lane of vehicle 2

Returns
bool

Return type
Whether there is a collision between two vehicles

plafosim.simulator.**initialize_emission_traces**(*basename: str*)

plafosim.simulator.**initialize_platoon_changes**(*basename: str*)

plafosim.simulator.**initialize_platoon_formation**(*basename: str*)

plafosim.simulator.**initialize_platoon_maneuvers**(*basename: str*)

plafosim.simulator.**initialize_platoon_traces**(*basename: str*)

plafosim.simulator.**initialize_platoon_trips**(*basename: str*)

plafosim.simulator.**initialize_simulation_trace**(*basename: str*)

plafosim.simulator.**initialize_vehicle_changes**(*basename: str*)

plafosim.simulator.**initialize_vehicle_emissions**(*basename: str*)

plafosim.simulator.**initialize_vehicle_platoon_changes**(*basename: str*)

plafosim.simulator.**initialize_vehicle_platoon_traces**(*basename: str*)

plafosim.simulator.**initialize_vehicle_teleports**(*basename: str*)

plafosim.simulator.**initialize_vehicle_traces**(*basename: str*)

plafosim.simulator.**initialize_vehicle_trips**(*basename: str*)

`plafosim.simulator.is_gap_safe(front_position: float, front_speed: float, front_max_deceleration: float, front_length: float, back_position: float, back_speed: float, back_max_acceleration: float, back_min_gap: float, step_length: float) → bool`

Return whether the gap between the front and back vehicle is safe.

Safe means: - the front vehicle can decelerate as hard as possible for one step - the back vehicle can accelerate as hard as possible for one step - the vehicle will not crash

Assumes euclidean/non-ballistic position updates.

Parameters

- **front_position** (*float*) – The position of the front vehicle in m
- **front_speed** (*float*) – The speed of the front vehicle in m/s
- **front_max_deceleration** (*float*) – The maximum deceleration of the front vehicle in m/s2
- **front_length** (*length*) – The length of the front vehicle in m
- **back_position** (*float*) – The position of the back vehicle in m
- **back_speed** (*float*) – The speed of the back vehicle in m/s
- **back_max_acceleration** (*float*) – The maximum acceleration of the back vehicle in m/s2
- **back_min_gap** (*float*) – The minimum gap of the back vehicle in m
- **step_length** (*float*) – The length of a simulation step in s

Returns
bool

Return type
Whether the gap between the two vehicles is safe

`plafosim.simulator.is_insert_safe(depart_position: float, depart_speed: float, vtype: VehicleType, other_vehicle: Vehicle, step_length: float) → bool`

Checks if a vehicle can be inserted safely at a given position.

Parameters

- **depart_position** (*float*) – The planned depature position of the vehicle
- **depart_speed** (*float*) – The planned departure speed of the vehicle
- **vtype** (*VehicleType*) – The vehicle type of the vehicle
- **other_vehicle** (*Vehicle*) – The other vehicle to check
- **step_length** (*float*) – The step length

Returns
bool

Return type
Whether the insertion of a vehicle is safe

`plafosim.simulator.lane_predecessors(vdf: DataFrame, max_lane: int) → DataFrame`

Find the current (potential) predecessor for each lane and each vehicle.

This means: Which other vehicle would be the predecessor if a vehicle was on lane *column*. A predecessor id of -1 means there is no predecessor.

Preconditions: - vdf.sorted_values(['position', 'lane'], ascending=False)

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **max_lane** (*int*) – The largest lane id

`plafosim.simulator.move_gui_vehicle(vid: int, position: float, lane: int, speed: float)`

Move a vehicle in the GUI.

Parameters

- **vid** (*int*) – The id of the vehicle to change
- **position** (*float*) – The vehicle's new position
- **lane** (*int*) – The vehicle's new lane
- **speed** (*float*) – The vehicle's new speed

`plafosim.simulator.prune_vehicles(keep_vids: list)`

Prunes vehicles from the GUI.

Parameters

keep_vids (*list*) – The ids of the vehicle that should be kept

`plafosim.simulator.record_general_data_begin(basename: str, simulator: Simulator)`

`plafosim.simulator.record_general_data_end(basename: str, simulator: Simulator)`

`plafosim.simulator.record_platoon_change(basename: str, step: float, leader: PlatooningVehicle, source_lane: int, target_lane: int, reason: str)`

`plafosim.simulator.record_simulation_trace(basename: str, step: float, vehicles_in_simulator: int, vehicles_in_queue: int, vehicles_spawned: int, vehicles_arrived: int, runtime: float, average_vehicle_speed: float, vehicles_braking_rough: int)`

`plafosim.simulator.record_vehicle_change(basename: str, step: float, vid: int, position: float, speed: float, source_lane: int, target_lane: int, reason: str)`

`plafosim.simulator.record_vehicle_platoon_change(basename: str, step: float, member: PlatooningVehicle, source_lane: int, target_lane: int, reason: str)`

`plafosim.simulator.record_vehicle_trace(basename: str, step: int, vehicle: Vehicle)`

`plafosim.simulator.remove_gui_vehicle(vid: int)`

Remove a vehicle from the GUI.

Parameters

vid (*int*) – The id of the vehicle to remove

`plafosim.simulator.report_rough_braking(vdf: DataFrame, new_speed: Series, step_length: float, rough_factor: float = 0.5) → int`

Report about vehicle preforming rough braking maneuvers.

Rough braking meaning more than rough_factor times max_deceleration.

Return number of rough braking vehicles

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **new_speed** (*pandas.Series*) – The Series containing the new speeds of the vehicles index: vid
- **step_length** (*float*) – The length of one simulation step in s
- **rough_factor** (*float, optional*) – The factor to apply to the maximum deceleration for setting a rough braking threshold

Returns
The number of vehicles performing rough braking

Return type
int

`plafosim.simulator.set_gui_window(road_length: int)`

Set the window of the GUI according to the road length.

Parameters
road_length (*int*) – The length of the road in m

`plafosim.simulator.start_gui(config: str, step_length: float, play: bool = True)`

Start the GUI.

Parameters

- **config** (*str*) – The name of the configuration file
- **step_length** (*float*) – The length of one simulation step in s
- **play** (*bool, optional*) – Whether to start the simulation automatically

`plafosim.simulator.timer()`

perf_counter() -> float

Performance counter for benchmarking.

`plafosim.simulator.update_position(vdf: DataFrame, step_length: float)` → DataFrame

Update the position of vehicles within a pandas Dataframe.

This is based on Krauss' single lane traffic: adjust position (move) x(t + step_size) = x(t) + v(t)*step_size

Parameters

- **vdf** (*pandas.DataFrame*) – The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]
- **step_length** (*float*) – The length of the simulated step

Returns
The Dataframe containing the vehicles as rows index: vid columns: [position, length, lane, ..]

Return type
pandas.DataFrame

plafosim.spawning module

`plafosim.spawning.get_arrival_position(depart_position: int, road_length: int, ramp_interval: int, min_trip_length: int, max_trip_length: int, rng: Random, random_arrival_position: bool = False, pre_fill: bool = False) → int`

Return a (random) arrival position for a given departure position.

This considers the ramp interval, road length, and minimum trip length.

Parameters

- **depart_position** (*int*) – The departure position to consider
- **road_length** (*int*) – The length of the entire road
- **ramp_interval** (*int*) – The distance between two on-/off-ramps
- **min_trip_length** (*int*) – The minimum trip length
- **max_trip_length** (*int*) – The maximum trip length
- **rng** (*random.Random*) – The random number generator to use
- **random_arrival_position** (*bool*) – Whether to use random arrival positions
- **pre_fill** (*bool, optional*) – Whether the trip is for a pre-filled vehicle

Returns
The arrival position in m

Return type
int

`plafosim.spawning.get_depart_speed(desired_speed: float, rng: Random, depart_desired: bool = False, random_depart_speed: bool = False) → float`

Return a (random) departure speed.

Parameters

- **desired_speed** (*float*) – The desired speed to consider
- **rng** (*random.Random*) – The random number generator to use
- **depart_desired** (*bool, optional*) – Whether to depart with the desired speed
- **random_depart_speed** (*bool, optional*) – Whether to choose a random value from a range

Returns
The value for the departure speed

Return type
float

`plafosim.spawning.get_desired_speed(desired_speed: float, rng: Random, speed_variation: float, min_desired_speed: float, max_desired_speed: float, random_desired_speed: bool = False) → float`

Return a (random) desired driving speed.

Parameters

- **desired_speed** (*float*) – The value to be used as is or as the mean for sampling from a normal distribution

- **rng** (*random.Random*) – The random number generator to use
- **speed_variation** (*float*) – The value to be used as variation for sampling from a normal distribution
- **min_desired_speed** (*float*) – The minimum allowed value for the desired driving speed
- **max_desired_speed** (*float*) – The maximum allowed value for the desired driving speed
- **random_desired_speed** (*bool, optional*) – Whether to choose a random value from a normal distribution

Returns
The value for the desired driving speed

Return type
float

plafosim.statistics module

plafosim.statistics.**initialize_emission_traces**(*basename: str*)

plafosim.statistics.**initialize_platoon_changes**(*basename: str*)

plafosim.statistics.**initialize_platoon_formation**(*basename: str*)

plafosim.statistics.**initialize_platoon_maneuvers**(*basename: str*)

plafosim.statistics.**initialize_platoon_traces**(*basename: str*)

plafosim.statistics.**initialize_platoon_trips**(*basename: str*)

plafosim.statistics.**initialize_simulation_trace**(*basename: str*)

plafosim.statistics.**initialize_vehicle_changes**(*basename: str*)

plafosim.statistics.**initialize_vehicle_emissions**(*basename: str*)

plafosim.statistics.**initialize_vehicle_platoon_changes**(*basename: str*)

plafosim.statistics.**initialize_vehicle_platoon_traces**(*basename: str*)

plafosim.statistics.**initialize_vehicle_teleports**(*basename: str*)

plafosim.statistics.**initialize_vehicle_traces**(*basename: str*)

plafosim.statistics.**initialize_vehicle_trips**(*basename: str*)

plafosim.statistics.**record_emission_trace_prefix**(*basename: str*, *step: float*, *vid: int*)

plafosim.statistics.**record_emission_trace_suffix**(*basename: str*)

plafosim.statistics.**record_emission_trace_value**(*basename: str*, *value: float*)

plafosim.statistics.**record_general_data_begin**(*basename: str*, *simulator:* Simulator)

plafosim.statistics.**record_general_data_end**(*basename: str*, *simulator:* Simulator)

plafosim.statistics.**record_platoon_change**(*basename: str*, *step: float*, *leader:* PlatooningVehicle, *source_lane: int*, *target_lane: int*, *reason: str*)

plafosim.statistics.**record_platoon_formation**(*basename: str*, *vehicle:* PlatooningVehicle, *candidates_found_avg: float*, *candidates_found_individual_avg: float*, *candidates_found_platoon_avg: float*, *candidates_filtered_avg: float*)

plafosim.statistics.**record_platoon_trace**(*basename: str*, *step: float*, *vehicle:* PlatooningVehicle)

plafosim.statistics.**record_platoon_trip**(*basename: str*, *vehicle:* PlatooningVehicle, *platoon_time_ratio: float*, *platoon_distance_ratio: float*, *time_until_first_platoon: float*, *distance_until_first_platoon: float*)

plafosim.statistics.**record_simulation_trace**(*basename: str*, *step: float*, *vehicles_in_simulator: int*, *vehicles_in_queue: int*, *vehicles_spawned: int*, *vehicles_arrived: int*, *runtime: float*, *average_vehicle_speed: float*, *vehicles_braking_rough: int*)

plafosim.statistics.**record_vehicle_change**(*basename: str*, *step: float*, *vid: int*, *position: float*, *speed: float*, *source_lane: int*, *target_lane: int*, *reason: str*)

plafosim.statistics.**record_vehicle_emission**(*basename: str*, *vehicle:* Vehicle)

plafosim.statistics.**record_vehicle_platoon_change**(*basename: str*, *step: float*, *member:* PlatooningVehicle, *source_lane: int*, *target_lane: int*, *reason: str*)

plafosim.statistics.**record_vehicle_platoon_maneuvers**(*basename: str*, *vehicle:* PlatooningVehicle)

plafosim.statistics.**record_vehicle_platoon_trace**(*basename: str*, *step: float*, *vehicle:* PlatooningVehicle)

plafosim.statistics.**record_vehicle_teleport**(*basename: str*, *step: float*, *vid: int*, *old_position: float*, *old_lane: int*, *old_speed: float*, *new_position: float*, *new_lane: int*, *new_speed: float*)

plafosim.statistics.**record_vehicle_trace**(*basename: str*, *step: int*, *vehicle:* Vehicle)

plafosim.statistics.**record_vehicle_trip**(*basename: str*, *vehicle:* Vehicle, *depart_delay: int*, *time_loss: int*, *expected_travel_time: float*, *travel_time_ratio: float*, *average_driving_speed: float*, *average_deviation_desired_speed: float*)

plafosim.statistics.**rgb2hex**(*c_rgb: tuple*) → str

Convert a color in RGB values to a color in hex values.

Parameters

c_rgb (*tuple(int, int, int)*) – The color in RGB values

Returns

str

Return type

The color in hex values

plafosim.util module

class plafosim.util.**FakeLog**

Bases: object

A fake logger, hide LOG behind this and be happy.

debug(**arg*, ***kwd*)

error(**arg*, ***kwd*)

fatal(**arg*, ***kwd*)

info(**arg*, ***kwd*)

trace(**arg*, ***kwd*)

warn(**arg*, ***kwd*)

warning(**arg*, ***kwd*)

plafosim.util.**acceleration2speed**(*acceleration: float*, *time_interval: float = 1.0*) → float

Convert an acceleration to a driving speed for a given time interval.

Parameters

- **acceleration** (*float*) – The acceleration to be converted
- **time_interval** (*float, optional*) – The time to consider

Returns
float

Return type
The converted value

plafosim.util.**add_logging_level**(*level_name: str*, *level_num: int*, *method_name: str | None = None*)

Comprehensively adds a new logging level to the *logging* module and the currently configured logging class.

level_name becomes an attribute of the *logging* module with the value *level_num*. *method_name* becomes a convenience method for both *logging* itself and the class returned by *logging.getLoggerClass()* (usually just *logging.Logger*). If *method_name* is not specified, *level_name.lower()* is used.

To avoid accidental clobberings of existing attributes, this method will raise an *AttributeError* if the level name is already an attribute of the *logging* module or if the method name is already present.

Taken from https://stackoverflow.com/a/35804945.

Parameters

- **level_name** (*str*) – The name of the level to add
- **level_num** (*int*) – The number of the level to add
- **method_name** (*str*) – The name of the method for the level to add

plafosim.util.**assert_index_equal**(*one*, *two*) → bool

Ensure the indices of two Sequences/DataFrames are equal.

Parameters

- **one** (*pandas.Sequence / pandas.DataFrame*) – The first object for the comparison
- **two** (*pandas.Sequence / pandas.DataFrame*) – The second object for the comparsion

Returns
bool

Return type
Whether the two indices are equal

`plafosim.util.distance2speed(distance: float, time_interval: float = 1.0)` → float

Convert a driven distance to a driving speed for a given time interval.

Parameters

- **distance** (*float*) – The distance to be converted
- **time_interval** (*float, optional*) – The time to consider

Returns
float

Return type
The converted value

`plafosim.util.find_resource(path: str)` → str

Find the resouces under relpath locally or as a packaged resource.

Parameters
path (*str*) – The path to search for the ressource

Returns
str

Return type
The path of the resource

`plafosim.util.hex2rgb(c_hex: str)` → tuple

Convert a color in hex values to a color in RGB values.

Parameters
c_hex (*str*) – The color in hex values

Returns
tuple(int, int, int)

Return type
The color in RGB values

`plafosim.util.rgb2hex(c_rgb: tuple)` → str

Convert a color in RGB values to a color in hex values.

Parameters
c_rgb (*tuple(int, int, int)*) – The color in RGB values

Returns
str

Return type
The color in hex values

`plafosim.util.round_to_next_base(value: float, base: float)` → float

Round a value to the next base value.

Parameters

- **value** (*float*) – The value to round

- **base** (*float*) – The base value to round to

Returns
float

Return type
The rounded value

plafosim.util.**speed2acceleration**(*speed_from: float*, *speed_to: float*, *time_interval: float = 1.0*) → float

Convert a speed range to an acceleration within a given time interval.

Parameters

- **speed_from** (*float*) – The initial speed
- **speed_to** (*float*) – The target speed
- **time_interval** (*float, optional*) – The time to consider

Returns
float

Return type
The converted value

plafosim.util.**speed2distance**(*speed: float*, *time_interval: float = 1.0*) → float

Convert a driving speed to a distance driven within a given time interval.

Parameters

- **speed** (*float*) – The speed to be converted
- **time_interval** (*float, optional*) – The time to consider

Returns
float

Return type
The converted value

plafosim.vehicle module

plafosim.vehicle.**ArrayLike**

alias of `float`

class plafosim.vehicle.**CF_Model**(*value*)

Bases: *Enum*

Car Following models that vehicles can use for their mobility.

ACC = 1

CACC = 2

HUMAN = 0

class plafosim.vehicle.**Vehicle**(*simulator: Simulator*, *vid: int*, *vehicle_type: VehicleType*, *depart_position: int*, *arrival_position: int*, *desired_speed: float*, *depart_lane: int*, *depart_speed: float*, *depart_time: float*, *depart_delay: float*, *communication_range: int*, *pre_filled: bool = False*)

Bases: `object`

A collection of state information for a vehicle in the simulation.

A vehicle can really be anything that can move and can be defined by a vehicle type. It does not necessarily be driven by a computer (i.e., autonomous). However, by default it does have V2X functionality.

__init__(*simulator: Simulator, vid: int, vehicle_type: VehicleType, depart_position: int, arrival_position: int, desired_speed: float, depart_lane: int, depart_speed: float, depart_time: float, depart_delay: float, communication_range: int, pre_filled: bool = False*)

Initialize a vehicle instance.

Parameters

- **simulator** (*Simulator*) – The global simulator object
- **vid** (*int*) – The id of the vehicle
- **vehicle_type** (*VehicleType*) – The vehicle type of the vehicle
- **depart_position** (*int*) – The departure position of the vehicle
- **arrival_position** (*int*) – The arrival position of the vehicle
- **desired_speed** (*float*) – The desired driving speed of the vehicle
- **depart_lane** (*int*) – The departure lane of the vehicle
- **depart_speed** (*float*) – The departure speed of the vehicle
- **depart_time** (*float*) – The actual departure time of the vehicle
- **depart_delay** (*float*) – The time the vehicle had to wait before starting its trip
- **communication_range** (*int*) – The maximum communication range of the vehicle
- **pre_filled** (*bool*) – Whether this vehicle was pre-filled

_action(*step: float*)

Triggers specific actions of a vehicle.

Parameters

step (*float*) – The current simulation step

_calculate_emission(*a: float, v: float, f: list, scale: float*) → float

Calculate the actual emission of the vehicle.

Parameters

- **a** (*float*) – The current acceleration
- **v** (*float*) – The current speed
- **f** (*list*) – The emission factors to use for current emission variable to be calculated
- **scale** (*float*) – The scale to normalize the calculated value

Returns

float

Return type

The calculcated emission in ml/mg per s

_calculate_emissions()

Calculate the emitted pollutant amount using the given speed and acceleration based on the HBEFA3 model.

As the functions are defining emissions in g/hour, the function's result is normed by 3.6 (seconds in an hour/1000) yielding mg/s. For fuel ml/s is returned. Negative acceleration results directly in zero emission.

The amount emitted by the given emission class when moving with the given velocity and acceleration [mg/s or ml/s]

_start()

Start this Vehicle.

_statistics()

Write continuous statistics for the vehicle.

action(*step: int*)

Triggers actions of a vehicle.

Parameters
step (*int*) – The current simulation step

finish()

Clean up the instance of the vehicle.

This includes mostly statistic recording.

info() → str

Return information about the vehicle.

property arrival_position: int

Return the arrival position of the vehicle.

property blocked_front: bool

Return whether the vehicle is currently blocked by a slow vehicle in the front.

property cf_model: *CF_Model*

Return the currently activated car following model of the vehicle.

property color: tuple

Return the current color of the vehicle.

property depart_lane: int

Return the departure lane of the vehicle.

property depart_position: int

Return the departure position of the vehicle.

property depart_speed: float

Return the departure speed of the vehicle.

property depart_time: float

Return the departure time of the vehicle.

property desired_gap: float

Return the desired gap to the vehicle in front of the vehicle.

This is based on the desired headway time and the current driving speed.

property desired_headway_time: float

Return the desired headway time of the vehicle.

property desired_speed: float

Return the desired driving speed of the vehicle.

property headway_time: float

Return the human headway time of the vehicle.

This is based on the vehicle type.

property lane: int

Return the current lane of the vehicle.

property length: int

Return the length of the vehicle.

This is based on the vehicle type.

property max_acceleration: float

Return the maximum acceleration of the vehicle.

This is based on the vehicle type.

property max_deceleration: float

Return the maximum deceleration of the vehicle.

This is based on the vehicle type.

property max_speed: float

Return the maximum speed of the vehicle.

This is based on the vehicle type.

property min_gap: float

Return the minimum safety gap to the vehicle in front of the vehicle.

This is based on the vehicle type.

property position: float

Return the current position of the vehicle.

property rear_position: int

Return the current rear position of the vehicle.

property speed: float

Return the current driving speed of the vehicle.

property travel_distance: float

Return the current traveled distance of the vehicle.

property travel_time: float

Return the current traveled time of the vehicle.

property vehicle_type: *VehicleType*

Return the VehicleType of the vehicle.

property vid: int

Return the id of the vehicle.

class plafosim.vehicle.**VehicleType**(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Bases: `object`

A collection of parameters for a concrete vehicle type.

__init__(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Initialize a specific vehicle type.

Parameters

- **name** (*str*) – The name of the vehicle type
- **length** (*int*) – The length of the vehicle type
- **max_speed** (*float*) – The maximum speed of the vehicle type
- **max_acceleration** (*float*) – The maximum acceleration of the vehicle type
- **max_deceleration** (*float*) – The maximum deceleration of the vehicle type
- **min_gap** (*float*) – The minimum safety gap to the vehicle in front of the vehicle type
- **headway_time** (*float*) – The human headway time of the vehicle type
- **emission_class** (`EmissionClass`) – The emission class of the vehicle type

property emission_class: *EmissionClass*

Return the emission class of a vehicle type.

property emission_factors: dict

Return the emission factors of a vehicle type.

property headway_time: float

Return the desired human headway time of a vehicle type.

property length: int

Return the length of a vehicle type.

property max_acceleration: float

Return the maximum acceleration of a vehicle type.

property max_deceleration: float

Return the maximum deceleration of a vehicle type.

property max_speed: float

Return the maximum speed of a vehicle type.

property min_gap: float

Return the minimum gap of a vehicle type.

property name: str

Return the name of a vehicle type.

`plafosim.vehicle.`**record_emission_trace_prefix**(*basename: str*, *step: float*, *vid: int*)

`plafosim.vehicle.`**record_emission_trace_suffix**(*basename: str*)

`plafosim.vehicle.`**record_emission_trace_value**(*basename: str*, *value: float*)

`plafosim.vehicle.`**record_vehicle_emission**(*basename: str*, *vehicle:* Vehicle)

plafosim.vehicle.**record_vehicle_trace**(*basename: str*, *step: int*, *vehicle:* Vehicle)

plafosim.vehicle.**record_vehicle_trip**(*basename: str*, *vehicle:* Vehicle, *depart_delay: int*, *time_loss: int*, *expected_travel_time: float*, *travel_time_ratio: float*, *average_driving_speed: float*, *average_deviation_desired_speed: float*)

plafosim.vehicle.**speed2distance**(*speed: float*, *time_interval: float = 1.0*) → float

Convert a driving speed to a distance driven within a given time interval.

Parameters

- **speed** (*float*) – The speed to be converted
- **time_interval** (*float, optional*) – The time to consider

Returns
float

Return type
The converted value

plafosim.vehicle_type module

class plafosim.vehicle_type.**EmissionClass**(*value*)

Bases: *Enum*

Emission class for combustion engines using the HBEFA3 model.

Website: https://www.hbefa.net/

PC_G_EU4 = 0

property emission_factors: dict

Return the emission factors of the emission class.

Returns
dict

Return type
The emission factors of the emission class

property is_diesel: bool

Return whether the emission class is for a diesel engine.

Returns
bool

Return type
Whether the emission class is for a diesel engine

class plafosim.vehicle_type.**VehicleType**(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Bases: object

A collection of parameters for a concrete vehicle type.

__init__(*name: str*, *length: int*, *max_speed: float*, *max_acceleration: float*, *max_deceleration: float*, *min_gap: float*, *headway_time: float*, *emission_class: str*)

Initialize a specific vehicle type.

Parameters

- **name** (*str*) – The name of the vehicle type
- **length** (*int*) – The length of the vehicle type
- **max_speed** (*float*) – The maximum speed of the vehicle type
- **max_acceleration** (*float*) – The maximum acceleration of the vehicle type
- **max_deceleration** (*float*) – The maximum deceleration of the vehicle type
- **min_gap** (*float*) – The minimum safety gap to the vehicle in front of the vehicle type
- **headway_time** (*float*) – The human headway time of the vehicle type
- **emission_class** (EmissionClass) – The emission class of the vehicle type

property emission_class: *EmissionClass*

Return the emission class of a vehicle type.

property emission_factors: dict

Return the emission factors of a vehicle type.

property headway_time: float

Return the desired human headway time of a vehicle type.

property length: int

Return the length of a vehicle type.

property max_acceleration: float

Return the maximum acceleration of a vehicle type.

property max_deceleration: float

Return the maximum deceleration of a vehicle type.

property max_speed: float

Return the maximum speed of a vehicle type.

property min_gap: float

Return the minimum gap of a vehicle type.

property name: str

Return the name of a vehicle type.

Module contents

class plafosim.**CustomFormatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

Bases: ArgumentDefaultsHelpFormatter, RawDescriptionHelpFormatter, MetavarTypeHelpFormatter

Metaclass combining multiple formatter classes for argparse.

class _Section(*formatter*, *parent*, *heading=None*)

Bases: object

__init__(*formatter*, *parent*, *heading=None*)

format_help()

__init__(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)

_add_item(*func*, *args*)

_dedent()

_expand_help(*action*)

_fill_text(*text*, *width*, *indent*)

_format_action(*action*)

_format_action_invocation(*action*)

_format_actions_usage(*actions*, *groups*)

_format_args(*action*, *default_metavar*)

_format_text(*text*)

_format_usage(*usage*, *actions*, *groups*, *prefix*)

_get_default_metavar_for_optional(*action*)

_get_default_metavar_for_positional(*action*)

_get_help_string(*action*)

_indent()

_iter_indented_subactions(*action*)

_join_parts(*part_strings*)

_metavar_formatter(*action*, *default_metavar*)

_split_lines(*text*, *width*)

add_argument(*action*)

add_arguments(*actions*)

add_text(*text*)

add_usage(*usage*, *actions*, *groups*, *prefix=None*)

end_section()

format_help()

start_section(*heading*)

`plafosim.add_logging_level(level_name: str, level_num: int, method_name: str | None = None)`

Comprehensively adds a new logging level to the *logging* module and the currently configured logging class.

level_name becomes an attribute of the *logging* module with the value *level_num*. *method_name* becomes a convenience method for both *logging* itself and the class returned by *logging.getLoggerClass()* (usually just *logging.Logger*). If *method_name* is not specified, *level_name.lower()* is used.

To avoid accidental clobberings of existing attributes, this method will raise an *AttributeError* if the level name is already an attribute of the *logging* module or if the method name is already present.

Taken from https://stackoverflow.com/a/35804945.

Parameters

- **level_name** (*str*) – The name of the level to add
- **level_num** (*int*) – The number of the level to add
- **method_name** (*str*) – The name of the method for the level to add

1.7 Changelog

1.7.1 v0.17.3 - 2024-01-11

- NEW FEATURES
  - Added documentation with readthedocs
- UPDATES
  - Applied sane default values for algorithm parameters
  - Improved dynamic loading of algorithms
  - Updated docstrings
  - Updated help messages in CLI
  - Updated log statements
  - Updated README

1.7.2 v0.17.2 - 2023-11-09

- BREAKING CHANGES
  - Rework SpeedPosition algorithm
- NEW FEATURES
  - Added possibility to configure knowledge about maneuver status
  - Added possibility to configure knowledge about platoon status
  - Added pydocstyle for development
  - Extended found candidate metrics
- UPDATES
  - Added PlaFoSim's logo
  - Applied some feedback from pylint

  - Updated docstrings
  - Updated flake8 config
  - Updated inline comments
  - Updated method/function signatures
  - Updated poetry.lock file
  - Updated README

1.7.3 v0.17.1 - 2023-10-27

- BREAKING CHANGES
  - Changed default communication range
  - Changed default execution interval
- FIXES
  - Fixed invalid value for headway time
  - Fixed missing choices for CLI arguments
  - Fixed links in README
- NEW FEATURES
  - GUI: Set zoom dependent on road length
  - Made algorithm loading fully dynamic
- UPDATES
  - Updated docstrings
  - Updated README

1.7.4 v0.17.0 - 2023-10-25

- BREAKING CHANGES
  - Changed default deceleration values
  - Changed default desired speed values
  - Changed default vehicle length
  - Reworked Krauss' model
  - Reworked vehicle generation
  - Reworked vehicle insertion
- FIXES
  - Fixed duplicated platoon id upon leaving
  - Fixed leave in the middle maneuver
  - Fixed time loss computation
  - Fixed updating cf target speed upon leaving
- NEW FEATURES

  - Added dedicated CLI argument for vehicle platoon trace
  - Added possibility to record continuous screenshots
  - Added possibility to record vehicle teleports
  - Added simple script to create video from screenshots
  - Added support for spawning on multiple lanes
  - Added support for various step lengths
  - Added timeloss to vehicle trace
  - Added trace recording after collision
- UPDATES
  - Applied some formatting with black
  - Extended rolling average statistics
  - Extended simulation statistics
  - Extended vehicle platoon trace
  - Improved documentation
  - Improved trace replay when tracking vehicle
  - Tightened constraint for recording platoon trace
  - Updated AUTHORS
  - Updated code comments
  - Updated README

1.7.5 v0.16.2 - 2023-10-06

- BREAKING CHANGES
  - Replace log-level with verbosity & quiet arguments
  - Remove obsolete mobility test
- NEW FEATURES
  - Add warning for high effective depart rate
- UPDATES
  - Add keywords to pyproject.toml
  - Update README
  - Extend warning for lane number support in GUI
  - Extend rough braking report with deceleration

1.7.6 v0.16.1 - 2023-07-13

- BREAKING CHANGES
  - Remove deprecated message-related code
  - Remove deprecated neighbortable-related code
- FIXES
  - Hotfix for installation of dev dependencies
- NEW FEATURES
  - Add dedicated help messages for argument groups
  - Add epilog with example commands
  - Add exit to signal handler
- UPDATES
  - Extend README
  - Update year in copyright headers

1.7.7 v0.16.0 - 2023-04-28

- BREAKING CHANGES
  - Replace log-level with verbosity & quiet arguments
- FIXES
  - Fix calculation of random arrival position
  - Fix deprecated argument
  - Fix missing function (hotfix)
  - Fix script compatibility with macOS
- NEW FEATURES
  - Add parsing of algorithm class from global identifiers
  - Add SPDX-License-Identifier
  - Add more performance tests
- UPDATES
  - Improve performance
  - Remove duplicated code in CLI scripts
  - Split integration tests into multiple steps
  - Update asserts
  - Update CI runs
  - Update citation information in README
  - Update copyright
  - Update docstrings

  - Update documentation in README

1.7.8 v0.15.4 - 2022-10-20

- Added author list
- Added simple signal handler
- Added tox for testing
- Changed logging stream to stdout
- Updated dependencies and pyproject.toml
- Updated README

1.7.9 v0.15.3 - 2022-07-07

- Extend pyproject.toml
- Extend simulation trace with spawn/arrive numbers
- Fixed bug in vehicle spawning
- Fixed wrong vehicle color recording in trace file
- Updated internal GUI API
- Updated README

1.7.10 v0.15.2 - 2022-06-10

- Added upper limit for python version
- Changed replay script print usage without arguments
- Fixed example command in README
- Removed ipython dependency
- Updated CI configuration
- Updated general.out file recording
- Updated README
- Updated SUMO related parts in code and README

1.7.11 v0.15.1 - 2022-05-10

- Improved description and extending instructions README
- Improved usage print in CLI
- Improved CI pipeline

1.7.12 v0.15.0 - 2022-05-03

- BREAKING CHANGES
  - CLI entry point now prints usage without any arguments
  - Name of formation algorithm now depends on class name
  - CI now uses Python 3.7 for all tests
- NEW FEATURES
  - Added argument for default configuration
  - Added dedicated build pipelines for other python versions in CI
  - Added dummy formation algorithm
- UPDATES
  - Updated CI pipelines
  - Updated README
  - Updated year in copyright

1.7.13 v0.14.5 - 2022-04-07

- Added dummy headway time for CACC
- Fixed calculation of approach time
- Fixed calculation of timeLoss
- Fixed maneuver state
- Fixed maximal depart position
- Fixed off-by-one error in vehicle id calculation
- Fixed statistic recording for pre-filled vehicles
- Fixed typo in README
- Fixed typos in comments
- Improved log statements
- Improved parameter recording
- Improved recording of simulation trace
- Improved vehicle spawning

1.7.14 v0.14.4 - 2022-02-03

- Added check for supported SUMO version
- Added recording of depart delay to vehicle
- Cleaned up code
- Fixed bug in trace player
- Fixed missing choices in CLI script
- Updated changelog
- Updated documentation
- Updated GUI
- Updated README
- Updated statistics recording

1.7.15 v0.14.3 - 2021-12-06

- Fixed vehicle tracking
- Formatted citation & description
- Made effective depart rate check less conservative
- Removed duplicated warning
- Updated some log statements

1.7.16 v0.14.2 - 2021-11-29

- Added vehicle color to trace file
- Fixed changelog URL in packaging configuration
- Fixed formation kind check
- Fixed usage of asserts
- Updated log levels in GUI
- Updated log statements in trace replay script

1.7.17 v0.14.1 - 2021-11-23

- Added pre-filled status to vehicles
- Added random offset for pre-filled vehicles' execution interval
- Fixed bug in GUI
- Updated CHANGELOG
- Updated README

1.7.18 v0.14.0 - 2021-11-22

- Added packaging with poetry
- Added dry-run flag
- Fixed bug in comparison script
- Updated GUI module
- Updated README
- Updated trace replay script

1.7.19 v0.13.2 - 2021-11-17

- Fixed typo in README
- Updated citation information

1.7.20 v0.13.1 - 2021-11-12

- Added dedicated directory for algorithms
- Fixed ignoring failed CI steps
- Moved CF_Model to mobility module
- Removed obsolete argument
- Updated README

1.7.21 v0.13.0 - 2021-10-20

- Added citation information of accepted poster publication
- Added flag to start GUI in paused mode
- Added integration tests for spawning
- Added new spawn procedure
- Added various checks for input parameters
- Added various tests
- Implemented vectorized lane changes
- Updated correctness tests
- Updated generation of pre-filled platoon
- Updated Platoon Data

1.7.22 v0.12.0 - 2021-09-08

- Added argument for connecting to the GUI later
- Added argument for showing the progress bar
- Added arguments for drawing labels within the GUI
- Added arguments for drawing objects within the GUI
- Added bumpversion
- Added dedicated CI step for validation data upload
- Added dedicated module for writing statistics
- Added dedicated trace files for platoon and member changes
- Added emission class
- Added integration tests
- Added pipeline step for uploading the validation plots to the wiki
- Added snapshot feature
- Cleaned up the code
- Cleaned up trace player
- Extracted GUI code to dedicated module
- Fixed doc image path
- Fixed emission calculation
- Fixed lange change check.
- Fixed leave maneuver
- Fixed references to formulas from literature
- Fixed safe speed calculation
- Renamed CF Model CC to Human
- Updated argument passing
- Updated documentation
- Updated predecessor/successor calculations
- Updated validation comparison
- Updated vehicle spawning
- Vectorized CF Models

1.7.23 v0.11.4 - 2021-07-21

- Added bootstrapping to calculate confidence interval in comparison
- Added explicit depart speed 0
- Added KS test for desired speed in comparison
- Added profile runs for all cf models to CI
- Fixed lane change duration
- Fixed simulation end upon no vehicles
- Fixed typos
- Renamed depart interval argument
- Updated check of maximum vehicle number
- Updated comparison script
- Updated implicit default value for random seed
- Updated validation scripts

1.7.24 v0.11.3 - 2021-07-19

- Added check for invalid depart rate
- Added check for useful depart probability
- Fixed calculation of depart position
- Fixed join at the end of the trip
- Fixed trace player
- Removed depart method fixed
- Updated default value of vehicle density parameter
- Updated string representation of platoon

1.7.25 v0.11.2 - 2021-07-02

- Added maximum trip length
- Added recording of continuous simulation trace
- Added safety check for insertions

1.7.26 v0.11.1 - 2021-06-24

- Updated result files for platoon related data

1.7.27 v0.11.0 - 2021-06-21

- Added additional constraints to optimization problem
- Added distinction between desired speed and CC target speed
- Added deprecation warning to communication code
- Added development advice to README
- Added more maneuver abort reasons
- Added solver time limit
- Added statists for solution quality
- Added teleport delay during maneuver
- Fixed argument choices
- Fixed centralized version of speed position algorithm
- Fixed desired headway time
- Fixed metric for successful assignments
- Fixed recording of periodic simulator statistics
- Fixed typos
- Updated CACC model to use direct speed from the leader
- Updated CACC validation to not update platoon's desired speed
- Updated calculation of speed and position deviation
- Updated CI triggers
- Updated CLI arguments
- Updated comments
- Updated condition for front join
- Updated docstring
- Updated formatting
- Updated optimization problem
- Updated parameter variables in simulator
- Updated platoon role in leave maneuver
- Updated properties and variables
- Updated run time calculation
- Updated vehicle tripinfo
- Updated validation scripts
- Updated variable access to gain speed

1.7.28 v0.10.0 - 2021-06-02

- Added comparison of cf models with single vehicle
- Added comparison to Plexe for CACC
- Added emissions to comparison
- Added pandas for predecessor calculation
- Added predecessor_id to new_speed method
- Added switch for reduced air drag
- Fixed infinite nesting of properties
- Fixed typos
- Fixed wrong execution trigger for formation algorithm
- Updated call to SUMO when using GUI
- Updated CI pipelines and comparisons
- Updated default values for formation thresholds
- Updated depart time of platoon
- Updated docstrings
- Updated logging

1.7.29 v0.9.6 - 2021-05-25

- Fixed assert in result recording
- Fixed spawning of static platoon

1.7.30 v0.9.5 - 2021-05-17

- Added argument for sumo GUI config
- Added average candidate metric
- Added checking the maximum speed
- Added maximum approach time
- Added metric for formation iterations
- Fixed issues due to floating precision
- Fixed release names in changelog
- Updated argument help
- Updated CACC calculation

1.7.31 v0.9.4 - 2021-05-10

- Added flag for updating a platoon's desired speed after a maneuver
- Fixed a lot of typos
- Fixed vehicle moving while join maneuver
- Removed all rounding to fix metrics
- Updated default parameters
- Updated documentation
- Updated maximum teleport distance

1.7.32 v0.9.3 - 2021-04-28

- Updated copyright headers
- Updated dependencies
- Updated documentation

1.7.33 v0.9.2 - 2021-04-20

- Added requirements.txt
- Added version argument
- Updated comments
- Updated README

1.7.34 v0.9.1 - 2021-03-19

- Added drawing ramps and road end
- Added maximum teleport distance
- Fixed candidate metrics

1.7.35 v0.9.0 - 2021-03-04

- Added all leave cases (simplified)
- Added formation statistics
- Added make space before a teleporting during a join
- Added maneuver statistics
- Added profile run to CI
- Added statistics for optimal solver
- Added statistics for simulator
- Added switch for actions
- Added util module

- Added vectorized collision checks with pandas
- Added vectorized position updates with pandas
- Fixed additional CACC calculation
- Fixed calculations of metrics
- Fixed depart position for pre-filled vehicles
- Fixed generation of random depart speed
- Fixed leaving in the middle of a platoon
- Fixed obsolete depart_time check
- Fixed platoon time metrics
- Fixed randomness when using prefill and GUI
- Fixed record statistics if actions disabled
- Improved code and project structure
- Removed duplicated CACC execution in between steps
- Updated cf model
- Updated comparison script
- Updated error messages
- Updated generation of trips
- Updated insert collision checks
- Updated join maneuver
- Updated log messages
- Updated new speed calculations
- Updated parameters for simulator
- Updated position correction after teleport
- Updated result recording
- Updated warnings

1.7.36 v0.8.0 - 2020-12-03

- Added communication range between vehicles
- Added desired headway time
- Added emission model
- Added execution intervals for formation algorithms
- Added minimum trip length
- Added more sanity checks for parameter
- Added new ecdfplot in comparison
- Added ortools solver for optimal assignments
- Added switch for disabling result recording for pre-filled vehicles

- Fixed CF Models
- Fixed collisions due to buggy lane change
- Fixed execution paths in scripts
- Fixed position correction after teleport
- Fixed start as platoon
- Small improvements
- Tuned performance of the simulator
- Updated adjustment of the platoon's desired speed to the avg of all members
- Updated CC cf model and comparison
- Updated depart parameters
- Updated exits due to sanity checks
- Updated logging
- Updated random depart/arrival position generation
- Updated simulation with GUI
- Updated the comparison script to sumo
- Updated the join maneuver
- Updated the leave maneuver
- Updated the minimum gap parameter
- Updated units in cli script

1.7.37 v0.7.0 - 2020-10-15

- Added a more complex join
- Added dedicated modules
- Added default values to all simulator arguments
- Added drawing of infrastructures in the GUI
- Added formation test to CI
- Added missing type hints
- Added (more) configurable result recording
- Added (more) statistics
- Added more tests
- Added simple (speed & position) centralized formation algorithm
- Added type hints
- Added updating platoon followers upon new speed of leader
- Cleaned up imports
- Fixed desired speed for vehicles in a platoon
- Fixed random state when using GUI

- Fixed random state when using pre-fill
- Fixed simulator exit code is ignored in CI
- Fixed teleport in join
- Let formation work with platoons only
- Updated advanced simulation control
- Updated CI definitions
- Updated coloring of vehicles
- Updated copyright headers
- Updated default values for arguments
- Updated join maneuver (adjusting of interfering vehicles)
- Updated logging to use format strings
- Updated methods using only ids to use vehicle instances
- Updated project structure
- Updated testing of Simulator
- Updated trace playing script

1.7.38 v0.6.0 - 2020-09-22

- Added an abstract base class for formation algorithms
- Added functionality to track a vehicle in the GUI
- Added proper logging
- Added simple join (at back) and leave at front) maneuver
- Added simple (speed & position) distributed formation algorithm
- Added support for infrastructure
- Moved formation logic to composition
- Split up vehicle.py
- Updated platoon class
- Updated traffic generation

1.7.39 v0.5.0 - 2020-09-04

- Added ACC car-following model
- Added CACC car-following model
- Added comparison to Sumo for ACC
- Added random seed
- Added step log
- Improved collision check
- Reworked lane change safety check

- Updated comparison script

1.7.40 v0.4.0 - 2020-08-10

- Added comparison step to CI
- Added execution step to CI
- Added script to compare the simulator to Sumo
- Added simple lane change model
- Updated car-following model

1.7.41 v0.3.0 - 2020-08-03

- Added blocked warning
- Added platoon class
- Added testing framework
- Improved the result recording
- Updated the GUI
- Updated trip generation
- Update the step log

1.7.42 v0.2.0 - 2020-07-06

- Added functionality to removed finished vehicles
- Added neighbor table stub
- Added simple live GUI by using Sumo
- Added simple vehicle trace player
- Reworked internal data structure for vehicles
- Updated car-following model
- Updated messaging
- Updated result recording

1.7.43 v0.1.0 - 2020-06-04

- Began a new project
- Added a VehicleType class
- Added CLI script with arguments
- Added Krauss' car-following model
- Added parameters for vehicles, trips, and the simulator
- Added PlatooningVehicle class

- Added PlatoonRole enum
- Added simple communication functionality between vehicles
- Added simple platoon advertising functionality
- Added simple result recording
- Added Simulator class
- Added Vehicle class

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

U

V

W