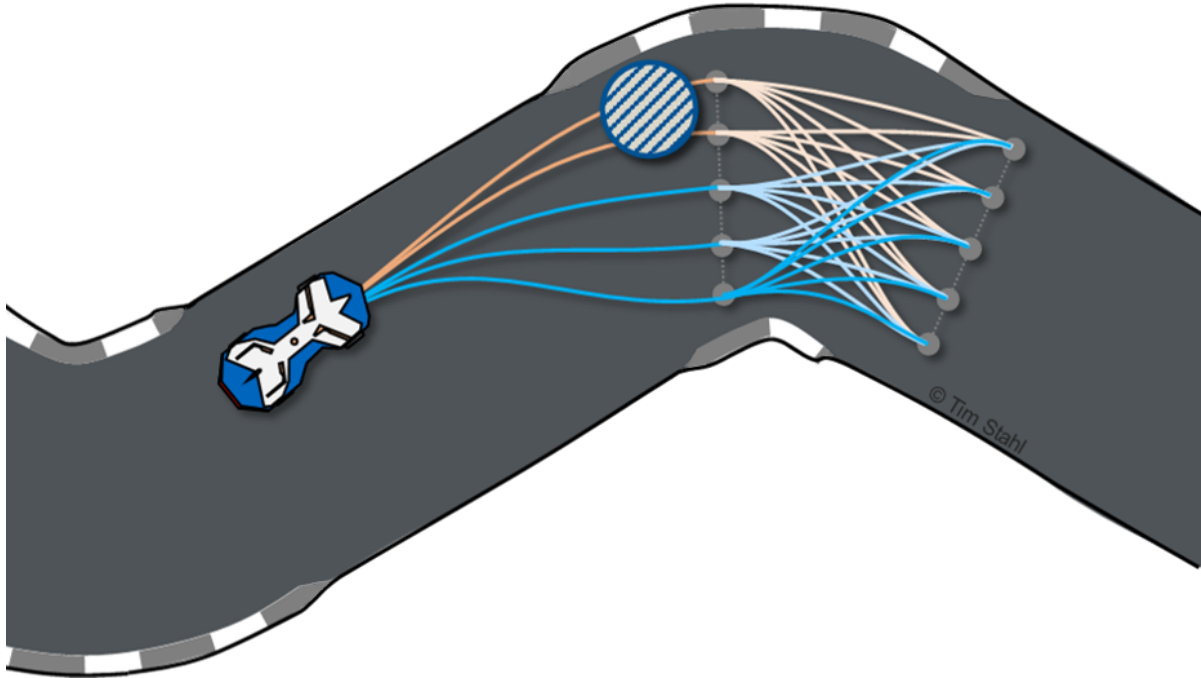

Graph-Based Local Trajectory Planner

Release 0.0.2

Jul 16, 2021

1	Contents	3
1.1	Repository Overview	3
1.2	Installation	5
1.3	Launching the Planner	5
1.4	Inputs to the Trajectory Planner	6
1.5	Configuration	7
1.6	Using the Planner Package	8
1.7	Log Visualization	12
1.8	Development Tools	14
1.9	graph_ltpl	15
2	Contributions	19
3	Contact Information	21



The graph-based local trajectory planner is python-based and comes with open interfaces as well as debug, visualization and development tools. The local planner is designed in a way to return an action set (e.g. keep straight, pass left, pass right), where each action is the globally cost optimal solution for that task. If any of the action primitives is not feasible, it is not returned in the set. That way, one can either select available actions based on a priority list (e.g. try to pass if possible) or use an own dedicated behaviour planner.

The planner was used on a real race vehicle during the Roborace Season Alpha and achieved speeds above 200kph. A video of the performance at the Montebianco track can be found [here](#).

Warning: This software is provided *as-is* and has not been subject to a certified safety validation. Autonomous Driving is a highly complex and dangerous task. In case you plan to use this software on a vehicle, it is by all means required that you assess the overall safety of your project as a whole. By no means is this software a replacement for a valid safety-concept. See the license for more details.

1.1 Repository Overview

1.1.1 Folder structure

The repository is composed of different components. Each of them is contained in a sub-folder.

Folder	Description
docs	This folder holds documentation files (including this page).
graph_tpl	This folder holds the algorithmic part of the local trajectory planner. The contained folders and files are explained in Using the Planner Package and graph_tpl .
inputs	This folder holds input files to the trajectory planner, among them are: global racelines (including map information) and vehicle dynamics config.
params	This folder contains parameterization files for the overall planner.

In the root folder is the *main_min_example.py* and *main_std_example.py*-file located. These python scripts allow an illustrative execution of the planner.

1.1.2 Framework

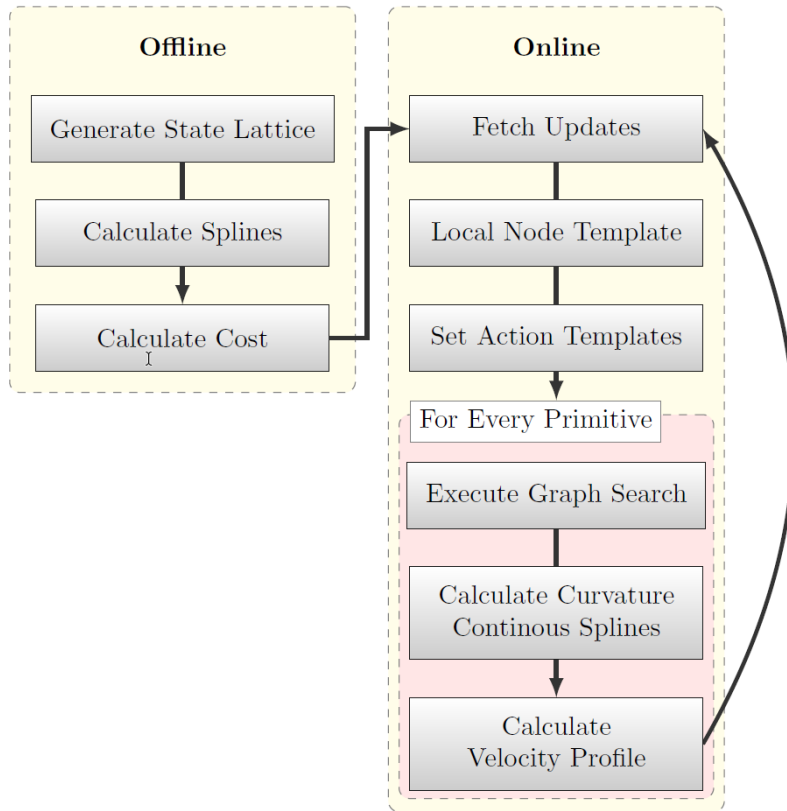


Figure 1: Basic execution flow of the planner¹.

In the following, the basic execution steps within the trajectory planner are outlined. The planner is divided into an online and an offline part (Figure 1), where the goal is to calculate as much information beforehand as possible in order to keep the online computation effort low.

The *offline* part generates a graph consisting of nodes - a lattice spanning the whole track - and vertices - cost-assigned splines connecting adjacent nodes. In that favor, first the track and map information is imported. Then, a state lattice is generated by sampling nodes in equally spaced (for curves and straights separately parameterizable) layers along the whole track. The lateral displacement of the nodes is chosen rather small compared to the longitudinal displacement. Afterwards, cubic splines are calculated for each pair of nodes in adjacent layers. Once completed, the graph is pruned, i.e. loose ends or splines violating the vehicles dynamic constraints are removed. Finally, a cost value is calculated for each remaining spline in the graph. By doing so, edges with higher curvatures or larger displacement to the race line receive a bigger cost penalty. Depending on the track size, this offline process may take some minutes to complete (progress shown in the command line).

The *online* part first fetches latest updates (pose estimate, object list, ...) and then extracts an online node template from the offline graph. Thereby only nodes belonging to the layers from the current position up to a specified planning horizon are considered in the next planning step. Furthermore, all edges intersecting any static or dynamic object in the scene are removed. Within the local node template a graph subset (action template) is generated for each desired action primitive (e.g. 'pass left', 'pass right', 'keep straight'). For example, for a 'pass left' template, all nodes to the right of a target vehicle are removed. That way, only edges passing the vehicle on the left remain in the graph. For each action primitive, a graph search (e.g. Dijkstra) is triggered in the corresponding action template. Afterwards, the resulting path is refined by calculating a curvature continuous spline passing all returned nodes. A forward-backward planner

¹ T. Stahl, A. Wischnewski, J. Betz, and M. Lienkamp, "Multilayer Graph-Based Trajectory Planning for Race Vehicles in Dynamic Scenarios," in 2019 IEEE Intelligent Transportation Systems Conference (ITSC), Oct. 2019, pp. 3149–3154. ([view pre-print](#))

or SQP-optimizer then generates a matching velocity profile for each of the C2-continuous splines (maximizing the feasible velocity based on the configured vehicle dynamics). This process is executed in an repetitive manner.

1.2 Installation

This is a brief tutorial how to prepare your workspace to execute the source code of the local planner.

All the code was tested with **Python 3.7**, newer versions should also work. The code was tested for compatibility with **Linux (Ubuntu)** and **Windows** operating systems.

Use the provided ‘requirements.txt’ in the root directory of the repository, in order to install all required modules.

```
pip3 install -r /path/to/requirements.txt
```

Note: If you want to use the planner as is, without adding code in the source-code, you can install the graph-ltpl package, as stated below:

```
pip3 install graph-ltpl
```

1.3 Launching the Planner

In order to use the planner, integrate the graph_ltpl package in your code. In order to get started, we provide two example scripts. These scripts demonstrate the code integration and allow to test the planners features.

1.3.1 Minimal Example

A minimal example of the local trajectory planner is parametrized in the ‘main_min_example.py’ script in the root directory. Within this example, the planner is executed without any interfaces (e.g. object list) and no configured logging. By default a live-visualization is shown (note: the live-visualization slows down the execution drastically). Launch the code with the following command:

```
python3 main_min_example.py
```

Note: When a certain track configuration is executed the first time, a offline graph is generated first. This will take some time (progress bar is displayed). On completion, a plot of the generated graph with all its edges is displayed (in case the visualization is enabled). After closing the figure with the offline graph, a live visualization is launched and the vehicle starts driving.

1.3.2 Standard Example

A more comprehensive example of the local trajectory planner is given in the ‘main_std_example.py’ script (also in the root directory). Within this example, the planner is executed with basic interfaces:

- Dummy object list - another vehicle driving with reduced speed along the racing-line (NOTE: here executed without any prediction of the object-vehicle’s motion)

- Dummy blocked zone - a small region of the track is blocked for the ego vehicle (useful to block certain regions, e.g. pit lane or dirty track)
- Logging to file - the environment and planned trajectories of every time-stamp are logged to a file, which can be visualized with the interactive log-viewer afterwards
- Prioritized action selection - instead of a behavior planner, in this example any 'pass' action is executed, when available

By default a live-visualization is shown (note: the live-visualization slows down the execution drastically). Launch the code with the following command:

```
python3 main_std_example.py
```

Note: When a certain track configuration is executed the first time, an offline graph is generated first. This will take some time (progress bar is displayed). On completion, a plot of the generated graph with all its edges is displayed (in case the visualization is enabled). After closing the figure with the offline graph, a live visualization is launched and the vehicle starts driving.

1.3.3 Further Steps

A description for usage of the planner class in your project is given in [Using the Planner Package](#). Furthermore, the parameterization, development tools and the log visualizer is tackled in that Chapter.

1.4 Inputs to the Trajectory Planner

All relevant input files can be found in the 'input'-folder, located in the root-directory. All the files / folders are explained in the following.

1.4.1 traj_ltpl_cl

This folder holds a set of global race lines with corresponding map information. The files hold comments in the first two lines, a semicolon separated header in the third line and corresponding data in the following lines. The file must hold the following columns with the headers listed:

- x_ref_m - x coordinates of the reference line (e.g. center line of the track)
- y_ref_m - y coordinates of the reference line (e.g. center line of the track)
- width_right_m - width of the track measured from the corresponding point on the reference line to the right (along the normal vector)
- width_left_m - width of the track measured from the corresponding point on the reference line to the left (along the normal vector)
- x_normvec_m - x coordinate of the normalized normal vector based on the corresponding point of the reference line
- y_normvec_m - y coordinate of the normalized normal vector based on the corresponding point of the reference line
- alpha_m - location of the point building the race line on the normal vector of a point belonging to the corresponding reference line

- `s_racetraj_m` - travelled distance along the race line (starting at 0.0 m for the first point)
- `psi_racetraj_rad` - heading of the race line point (north = 0.0)
- `kappa_racetraj_radpm` - curvature along the race line
- `vx_racetraj_mps` - globally optimal velocity profile along the race line
- `ax_racetraj_mps2` - acceleration profile matching the velocity profile

It should be noted, that all trajectory files should start with '`traj_ltpl_cl`' in order to work with the provided sample files. That way, the last part can be used to specify the track to be driven in the '`driving_task.ini`'.

Further race lines and track information can be generated with an global race trajectory optimization algorithm. The open-source version of the global planner providing matching files for this local planner is hosted on [GitHub](#). (In order to generate the appropriate files, read the Readme and enable the "LTPL Trajectory" output.

1.4.2 veh_dyn_info

The dynamical behavior of the vehicle for the initially generated velocity profile can be adjusted with the files in the '`params/veh_dyn_info`'-folder. The '`ax_max_machines.csv`'-file describes the acceleration resources of the motor at certain velocity thresholds (values in between are interpolated linearly).

Note: The '`ax_max_machines.csv`' can be imported with a function of the 'trajectory-planning-helpers' package.

```
import trajectory_planning_helpers as tph

ax_max = tph.import_veh_dyn_info.\
    import_veh_dyn_info(ax_max_machines_import_path="/path/to/ax_max_machines.csv") [1]
```

The imported matrix can then be provided to the '`calc_vel_profile()`' function of the `Graph_LTPL` class.

1.5 Configuration

All relevant configuration files can be found in the '`params`'-folder, located in the root-directory. All the files are explained in the following.

1.5.1 driving_task.ini

The driving task file holds information about the task to be executed. Currently, it solely holds the track to be loaded and driven. Further information, like a maximum velocity may be added by the user.

1.5.2 ltpl_config_offline.ini

The offline configuration file holds parameters primarily specifying the offline generation of the graph. Thereby, the discretization of the lattice as well as offline cost weighting can be adjusted. All parameters are described in detail in the config file.

1.5.3 ltpl_config_online.ini

The online configuration file holds parameters specifying the online planning of with the graph. As with the offline config file, all parameters are described in detail in the config file.

An important parameter in the config file is ‘vp_type’, which specifies the executed velocity planner. Currently two variants are integrated into the planner. ‘fb’ is a forward-backward planner, which is enabled by default and fully integrated into the stack. ‘sqp’ is an optimization based approach and must be installed separately (including parameter and input files).

Note: Details on the SQP velocity planner (incl. the parameter files), as well as required packages can be found in the corresponding repository (https://github.com/TUMFTM/velocity_optimization).

1.6 Using the Planner Package

This page describes the basic steps when using the planner (along with the two exemplary launch files). Furthermore, the tools coming with the local trajectory planner are introduced. Here, the focus is on the basic implementation and usage, a detailed documentation of all the functions, parameters and return values is in the *graph_ltpl*.

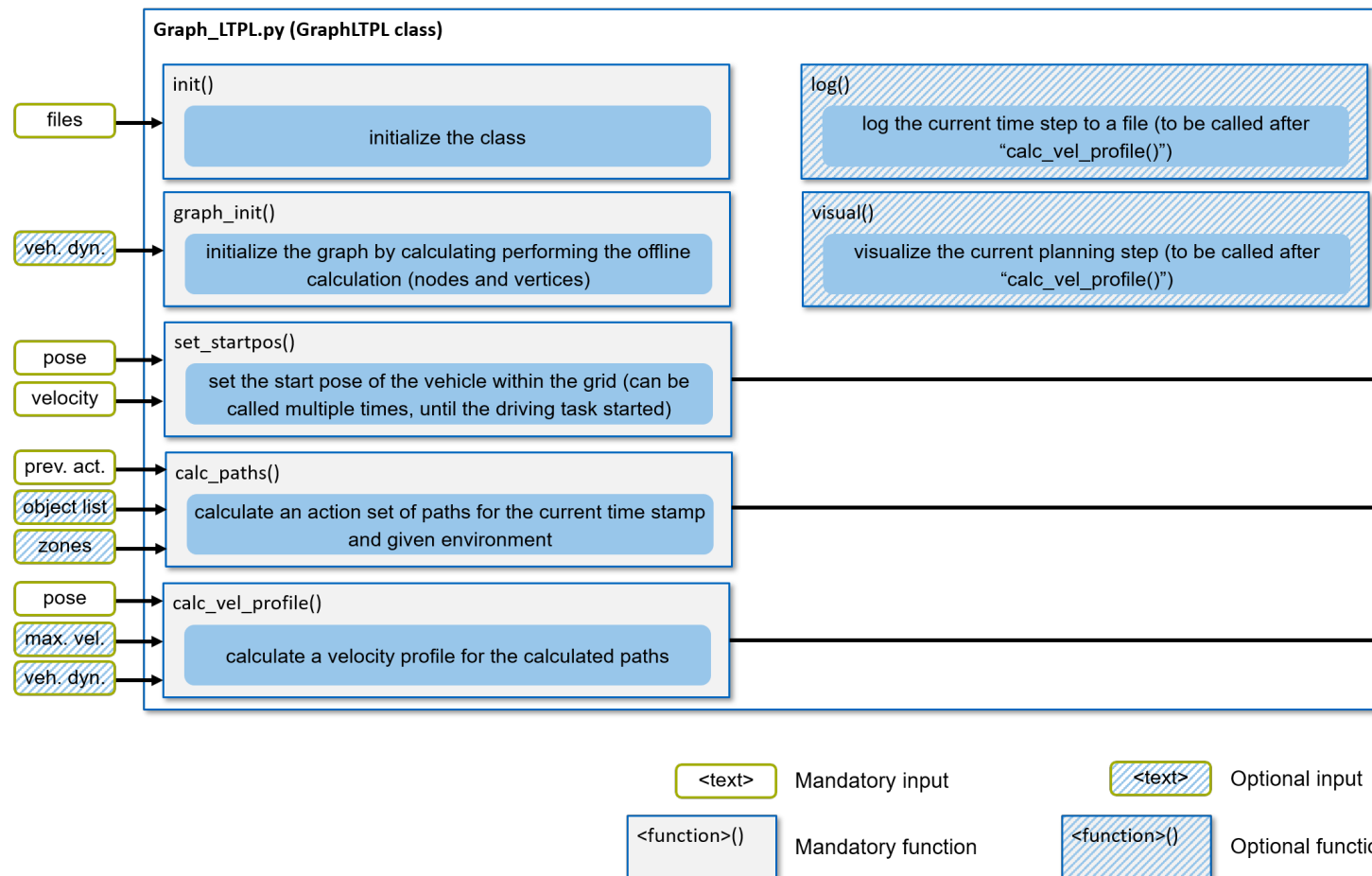


Figure 1: Simplified visualization of the GraphLTPL class with its functions and inputs / outputs.

Figure 1 shows the core functions of the GraphLTPL class. The visualization highlights mandatory functions and

inputs with a solid infill, while optional functions and inputs are hatched. The key steps are shown in the following snippet:

```
import graph_ltpl

# instantiate class and initialize graph
ltpl_obj = graph_ltpl.Graph_LTPL.Graph_LTPL()
ltpl_obj.graph_init()
ltpl_obj.set_startpos()

while True:
    # <-- get previously chosen trajectory set -->
    # <-- update objectlist -->

    <path data> = ltpl_obj.calc_paths()

    # <-- update dynamic parameters of ego vehicle -->

    <trajectory data> = ltpl_obj.calc_trajectories()

    # logging - optional
    ltpl_obj.log()

    # live visualization - optional
    ltpl_obj.visual()
```

In the following, the individual steps are outlined in more detail. First the mandatory initialization steps are addressed, followed by mandatory iterative steps and optional functions.

1.6.1 Initialization

Before working with the class, it has to be imported to your python script, this is done by the following command:

```
import graph_ltpl
```

Next, the class itself has to be initialized.

```
ltpl_obj = graph_ltpl.Graph_LTPL.Graph_LTPL(path_dict=path_dict,
                                             visual_mode=True,
                                             log_to_file=False)
```

The class initialization takes two boolean flags for the use of the ‘visual_mode’ (initialization of the live-visualization) and the ‘log_to_file’ function (preparation of the log-files). If any of these is set to ‘False’, the corresponding function call (‘log()’ or ‘visual()’) will have no effect. Furthermore, the class initialization takes a path dictionary (‘path_dict’). The path dict can be set up as follows:

```
path_dict = {'globtraj_input_path': "path/to/globtraj.csv",
             'graph_store_path': "path/to/stored_graph.pckl",
             'ltpl_offline_param_path': "path/to/ltpl_config_offline.ini",
             'ltpl_online_param_path': "path/to/ltpl_config_online.ini",
             'log_path': "path/to/logging_folder/",          # only if 'log_to_
↪ file=True'
             'graph_log_id': "unique_id123"                  # only if 'log_to_
↪ file=True'
             'graph_log_path': "path/to/logging_folder/"      # only if 'log_to_
↪ file=True'
             }
```

An exemplary and flexible setup of this path dict is shown in the ‘main_min_example.py’ (without logging) and ‘main_std_example.py’ (with logging) in the root of the repository.

Once the class is initialized, one can trigger the initialization of the graph. If the graph has not been calculated in a previous software execution, this process may take some time (several minutes). To trigger the initialization, use the following command:

```
ltpl_obj.graph_init()
```

As shown in Figure 1, this step allows to optionally provide some vehicle dynamics parameters. The detailed parameter description can be found in the *graph_ltpl*.

Before starting the autonomous driving part, the start pose of the vehicle is initialized with the following command:

```
ltpl_obj.set_startpos(pos_est=pos_est,  
                     heading_est=heading_est)
```

Note: The ‘set_startpos()’ function returns an optional boolean flag, whether the initialization succeeded (i.e. vehicle on track and pointing in correct direction). On a real vehicle, the software is often started in the pit, driven to the grid by an human which then exits the vehicle. Therefore, it is possible to add this function to the iterative part and set the startpos iteratively, until the returned flag indicates a successful initialization.

1.6.2 Mandatory iterative steps

Within the iterative steps, the planner first plans a spatial path for different action primitives and then plans a matching velocity profile. The corresponding steps are outlined in the following.

In order to calculate the paths, the following function call must be triggered:

```
ltpl_obj.calc_paths(prev_action_id=sel_action,  
                   object_list=obj_list,  
                   blocked_zones=zone_example)
```

By doing so, the specifier (string) of the previously chosen action primitive has to be provided. This is the case, since the planner must find a smooth transition from the passed path to the new generated / planned one (imagine two trajectories - one going left, one right - in the previous time step, the planner must now know, which of the two was chosen in order to continue from the slightly left turned or right turned pose).

Furthermore, in this step it is possible to provide an object list with all vehicles to be considered in the next planning step. The object list is a list of dicts, where each dict describes an object / vehicle and must at least host the following keys:

```
obj1 = {'id': 123,                # integer id of the object  
        'type': "physical",       # type 'physical' (only class implemented so far)  
        'X': 123.0,              # x coordinate  
        'Y': 0.123,              # y coordinate  
        'theta': 0.02,           # orientation (north = 0.0)  
        'v': 50.2,               # velocity along theta  
        'length': 3.2,           # length of the object  
        'width': 2.5             # width of the object  
}
```

```
{'X': 127, 'Y': 82, 'theta': 0.0, 'type': 'physical', 'form': 'rectangle',  
 'id': 1, 'length': 5.0, 'width': 2.5, 'v': 0.0}
```

```
obj_list = [obj1, obj2]
```

Note: In this published version, only a short constant velocity (CV) prediction (200ms) of other vehicles is implemented. In order to enable safe maneuvers, your own prediction of other vehicles must be provided. Since the planner is split into a sequence of spatial and temporal planning, the prediction must be translated into the spatial domain only (i.e. the nodes / edges blocked in the current iteration are blocked for all temporal steps).

The prediction itself can be provided in two ways. One option is to provide the prediction via the object list. In this favor, the object dict should host the key 'prediction', hosting a numpy array, where each line represents a position to be blocked in the graph (with the dimensions / radius of the object). For a more sophisticated / individual concept, we recommend the integration in the 'data_objects/ObjetListInterface.py' class.

In addition, it is possible to provide a blocked zones (regions on the track to be avoided by the ego-vehicle). An exemplary small zone is given in the 'main_std_example.py'.

The planned paths are stored inside the class but can also be retrieved from the return value of the function call (e.g. for visualization or decision making based on the path).

Finally, a the velocity profile is planned for the calculated paths.

```
ltpl_obj.calc_vel_profile(pos_est=pos_est,
                        vel_est=vel_est)
```

In order to plan the velocity profile, at least the position and velocity estimate of the current position must be provided. It should be noted, that the position is only used to project the position onto the last planned path (instead of iteratively starting the plan at the actual position of the vehicle, which would cause feedback loops interfering with the controller). The velocity estimate is required in order to calculate a set point velocity, when following a lead vehicle.

Furthermore it is possible to provide further (optional) vehicle dynamics parameters as well as an maximum velocity (e.g. currently set by a race control). The function returns at least one trajectory for each feasible action primitive.

Note: The trajectory set is provided in the following format:

```
{"straight": <list of trajectories>,
 "left":    <list of trajectories>,
 "right":   <list of trajectories>}
```

Each list of trajectories might be an empty set or one (default) to multiple trajectories. Each trajectory in this list is an numpy array with the columns [s, x, y, heading, curvature, vx, ax].

Furthermore, a unique ID for each action primitive is returned. The ID is an uint32 number and generated by the following scheme:

- Each set of trajectories obtains a base number incrementally growing in decade steps [10, 20, 30, ...]
- Each individual trajectory obtains an added type specifier with the following convention ['straight': 0, 'follow': 1, 'left': 2, 'right': 3, <type_error>: 9]

Further details about the parameters and return values can be found in the code documentation ([graph_ltpl](#)).

1.6.3 Optional iterative steps

Besides the mandatory steps, two optional functions may be called at the end of the iteration (after the 'calc_vel_profile()' function).

The ‘log()’ function can be called when the corresponding flag was set and all required paths were provided on class initialization. The function does not require any parameters and does not return any values. The function writes the (internally stored) information of the current planning step to a file.

The ‘visual()’ function can be called when the corresponding flag was set during initialization. The function does not require any parameters and does not return any values. The function updates the live-visualization by refreshing all changed information (e.g. planned path, obstacle positions).

Note: Rendering the live-visualization is computationally expensive and slows down the trajectory planner by a factor of 2-3. It is also possible to trigger the ‘visual()’ function only every n-th iteration.

1.6.4 Sub-packages of the graph_ltpl

- *data_objects*: This python module holds larger data structures (i.e. classes) providing the interface for basic manipulation procedures.
- *helper_funcs*: This python module contains various helper functions used in several other functions when calculating the local trajectories.
- *imp_global_traj*: This python modules handles the import and pre-processing of a global trajectory for the local planner.
- *offline_graph*: This python module holds all relevant functions for the offline generation of the graph.
- *online_graph*: This python module holds all relevant functions for the online execution of the graph.
- *testing_tools*: This folder hosts simple scripts for standalone simulations on the local machine.
- *visualization*: This python module contains functions used for visualization

The individual classes and functions are specified in the *graph_ltpl*.

1.7 Log Visualization

The stored logs generated during execution of the planner (if the logging functionality was activated) can be viewed with an log-visualization tool.

1.7.1 Launch of the Tool

The corresponding script is located in the folder ‘graph_ltpl/visualization/src’ and executed with the following command:

```
python3 visualize_graph_log.py
```

When the log files are stored in a folder ‘log’ in the root of the repository (as configured in the ‘main_std_example.py’) the script will automatically load the most recent log. In order to visualize a specific log, use the following command:

```
python3 visualize_graph_log.py path/to/log/xx_xx_xx_data.csv
```

Note: Each log consists of online information (‘*_data.csv’, ‘*_msg.csv’) and offline information that is stored in a pickle of the graph object (only stored when changed due to parameter or track changes). In order to view the logs,

always both - the online files and the graph pickle - must be present. In order to allow the tool to find the corresponding files, always stick to the folder structure demonstrated in the 'main_std_example.py'.

1.7.2 Usage of the Tool

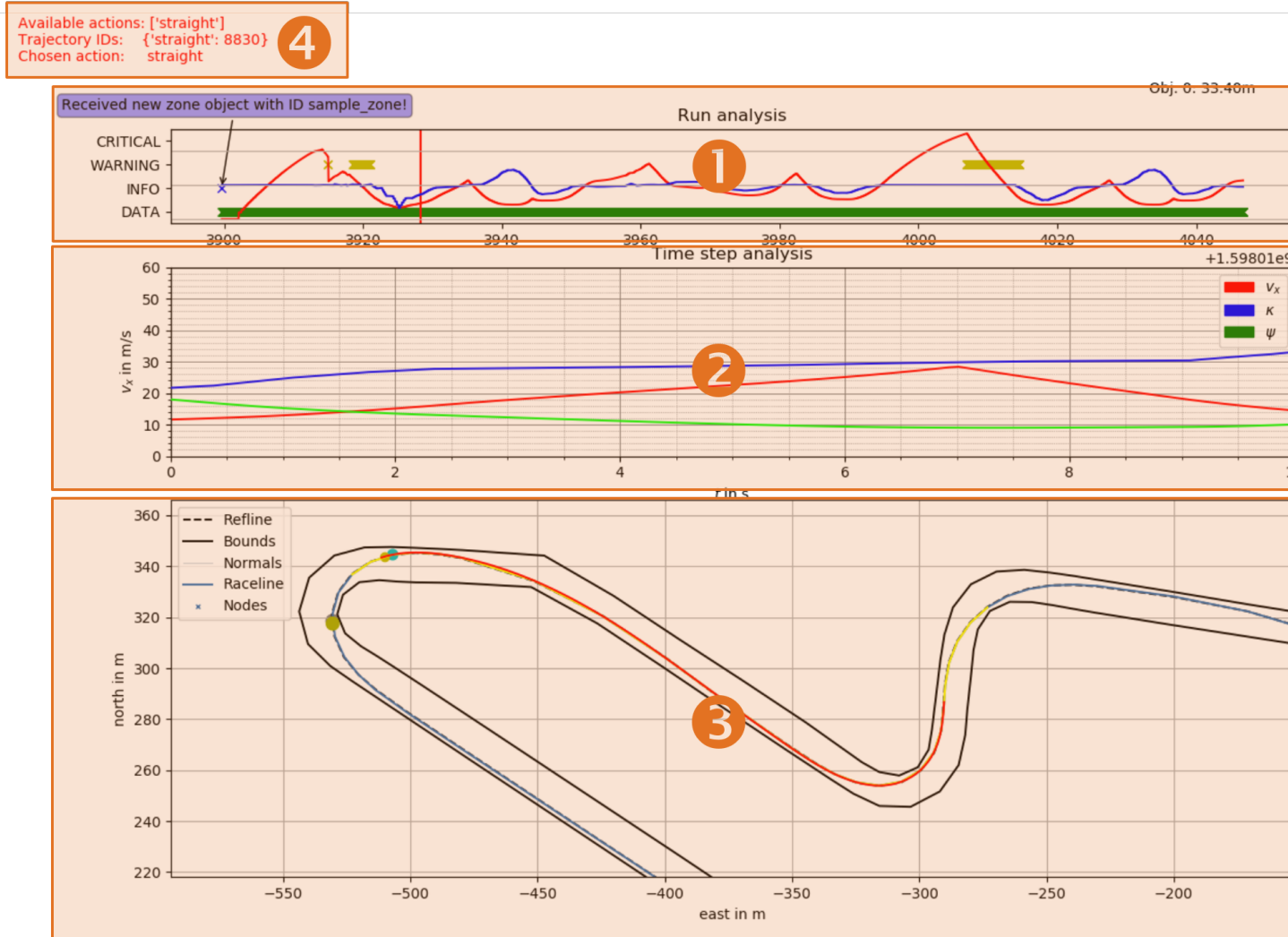


Figure 1: Key elements of the log viewer.

The tool consists (Figure 1) of three major plot windows (1 - 3) and additional information in the top left corner (4). The key functionality of each of the parts is explained in the following.

- 1. The run overview segment.** The velocity (red) and curvature (blue) course during the whole run are displayed here. That way it is easy to spot curves or certain straights. Events are highlighted with crosses on the corresponding time line. That way, each time a new data point was received, a green cross is drawn (green bar in the bottom). In a similar manner, information (blue), warnings (yellow) and critical events (red) are highlighted. When hovering with the mouse above one of the crosses, a description is shown (see blue bubble on the top left of (1)). Furthermore, when dragging them mouse over the run analyses, the information displayed in all of the following segments is updated accordingly (e.g. vehicle pose and trajectory at the selected time instance,

Animation 1).

2. **The trajectory time step analysis.** In this plot, one can see the course of velocity (red), heading (green) and curvature (blue) for all trajectories planned at the selected time stamp. The left most value corresponds to the position of the vehicle, values to the right show the course in the temporal future.
3. **The track overview.** In this plot, the overall track is visualized, including the all objects and the ego-vehicle with its planned trajectories at the selected time stamp (1). When hovering with the mouse over this plot, the nodes in the graph with adjacent edges (and their cost values) are highlighted (Animation 2).
4. **Additional information.** Information about the generated trajectory at the selected time stamp (1) is shown here. This includes the available action sets, corresponding trajectory IDs and the action chosen by the behavior planner or user.

The process of skidding through certain time stamps and the corresponding update of the other plots is shown in Animation 1.

Animation 1: Time selection and corresponding plot update.

The process of highlighting certain nodes and adjacent edges in the graph is shown in Animation 2. Thereby, the cost of each edge is displayed. This feature is helpful to understand chosen decisions by the planner in the past.

Animation 2: Node and edge highlight in the track plot.

When clicking on elements in the legend of the track plot, it is possible to toggle the visibility of the respective entity. That way it is possible to only display the information needed for a less cluttered view.

Animation 3: Toggling visibility of elements in the track plot.

1.8 Development Tools

In order to ease trajectory planning development, we provide some useful tools in the ‘graph_ltpl/testing_tools’ folder. The intent of these tools is to test the planner without the need to always launch the whole software stack (e.g. a dedicated perception and controller module). In that way, these scripts provide a rudimentary / simplified functionality of an object list (‘objectlist_dummy.py’) and a controller that ideally tracks the planned trajectory (‘vdc_dummy.py’).

1.8.1 objectlist_dummy.py

The dummy object list serves the purpose of providing a (simple) object list. By default, a single object vehicle is driving with reduced speed along the race line.

The object list dummy can be executed in two ways:

- Retrieve the object list via function call. The class keeps track of the passed time between the function calls and moves the object vehicle accordingly. (This variant is implemented in the ‘main_std_example.py’ script)
- Retrieve the object list via ZMQ communication. The object list dummy is launched in a separate thread and publishes the object list via ZMQ in a specified frequency. The list can then be received in the planner with a matching ZMQ receiver. This variant corresponds more to the situation in the vehicle.

1.8.2 vdc_dummy.py

The vehicle dynamics controller (VDC) dummy emulates an ideal controller that tracks the planned trajectory perfectly. The script can be called iteratively and requests the last planned trajectory, last position estimate as well as the passed time since the last call. The script then travels the specified iteration time along the trajectory and returns the resulting position and velocity defined by the planned trajectory.

Note: Details on the usage / parameterization can be found in the *graph_ltpl* ('testing_tools' package). Furthermore, an exemplary usage of both of the tools is implemented in the 'main_std_example.py' script.

Note: Further details about the actual implementation and the purpose of individual functions can be found in the *graph_ltpl*.

1.9 graph_ltpl

1.9.1 graph_ltpl package

Subpackages

`graph_ltpl.data_objects` package

Submodules

`graph_ltpl.data_objects.GraphBase` module

`graph_ltpl.data_objects.ObjectListInterface` module

Module contents

`graph_ltpl.helper_funcs` package

Subpackages

`graph_ltpl.helper_funcs.src` package

Submodules

`graph_ltpl.helper_funcs.src.Logging` module

`graph_ltpl.helper_funcs.src.calc_brake_emergency` module

`graph_ltpl.helper_funcs.src.calc_vel_profile_follow` module

`graph_ltpl.helper_funcs.src.closest_path_index` module

graph_itpl.helper_funcs.src.get_s_coord module

Module contents

Module contents

graph_itpl.imp_global_traj package

Subpackages

graph_itpl.imp_global_traj.src package

Submodules

graph_itpl.imp_global_traj.src.import_globtraj_csv module

graph_itpl.imp_global_traj.src.variable_step_size module

Module contents

Module contents

graph_itpl.offline_graph package

Subpackages

graph_itpl.offline_graph.src package

Submodules

graph_itpl.offline_graph.src.gen_edges module

graph_itpl.offline_graph.src.gen_node_skeleton module

graph_itpl.offline_graph.src.gen_offline_cost module

graph_itpl.offline_graph.src.main_offline_callback module

graph_itpl.offline_graph.src.prune_graph module

Module contents

Module contents

graph_itpl.online_graph package

Subpackages

graph_itpl.online_graph.src package

Submodules

graph_itpl.online_graph.src.OnlineTrajectoryHandler module

graph_itpl.online_graph.src.VpForwardBackward module

graph_itpl.online_graph.src.VpSQP module

graph_itpl.online_graph.src.check_inside_bounds module

graph_itpl.online_graph.src.gen_local_node_template module

graph_itpl.online_graph.src.get_intersec_edges module

graph_itpl.online_graph.src.get_zone_nodes module

graph_itpl.online_graph.src.main_online_path_gen module

Module contents

Module contents

graph_itpl.testing_tools package

Subpackages

graph_itpl.testing_tools.src package

Submodules

graph_itpl.testing_tools.src.objectlist_dummy module

graph_itpl.testing_tools.src.vdc_dummy module

Module contents

Module contents

graph_itpl.visualization package

Subpackages

graph_itpl.visualization.src package

Submodules

graph_itpl.visualization.src.PlotHandler module

graph_itpl.visualization.src.visualize_graph_log module

Module contents

Module contents

Submodules

graph_itpl.Graph_LTPL module

Module contents

[1] T. Stahl, A. Wischnewski, J. Betz, and M. Lienkamp, “Multilayer Graph-Based Trajectory Planning for Race Vehicles in Dynamic Scenarios,” in 2019 IEEE Intelligent Transportation Systems Conference (ITSC), Oct. 2019, pp. 3149–3154. ([view pre-print](#))

If you find our work useful in your research, please consider citing:

```
@inproceedings{stahl2019,  
  title = {Multilayer Graph-Based Trajectory Planning for Race Vehicles in Dynamic_  
↪Scenarios},  
  booktitle = {2019 IEEE Intelligent Transportation Systems Conference (ITSC)},  
  author = {Stahl, Tim and Wischnewski, Alexander and Betz, Johannes and Lienkamp,_  
↪Markus},  
  year = {2019},  
  pages = {3149--3154}  
}
```


CHAPTER 3

Contact Information

Email tim.stahl@tum.de