maliput

Woven Planet

Dec 08, 2022

CONTENTS

Road networks are at the core of autonomous driving. Agents (i.e., cars) that plan routes along roads need to be able to query the network, from basic geometric questions to more advanced concepts like physical constraints and dynamic rules. For example, an agent might want to find maximum-clearance, safe routes for different vehicle types and loadings. The agent might further want to take account of time-based rules such as traffic lights and other interactions with infrastructure or non-vehicle entities in the road network.

Maliput addresses this need by providing a library for representing road networks and answering queries about them. These networks can be expressed in many different formats, with OpenDRIVE and Lanelet2 being two popular examples. Whatever the format, Maliput provides a consistent API for interacting with the underlying data structure. Maliput can be used to implement an individual agent, or to support centralized planning and prediction for traffic management and fleet coordination. The standardized API decouples the agent behavior development from the road format, and simplifies the agent developer experience.

CHAPTER

ONE

MALIPUT VISION

The Maliput project has been actively developed by Ekumen, Open Robotics, and TRI for the last 4 years. It's actively used by the TRI/Woven team and has been developed using modern software engineering best practices including continuous integration, coverage testing, and clear developer rules.

We expect to see adoption of Maliput, for example in programmatic simulated world generation. Projects such as CARLA are generating simulated testing worlds based on road networks. Higher-fidelity road networks can be fed into the generator to create comprehensive simulated worlds which match real world environments, with controlled variations for better coverage.

Maliput has been guided by the needs of the TRI/Woven autonomous driving team and automotive use cases in general, but it also has wider applications. Maliput's standard interface can be leveraged in almost every case where a collection of mobile robots are interacting in the same space and need to be deconflicted. We foresee users for Maliput in warehouses, port drayage, and airport aprons. Any system managing this sort of multiple robot interaction needs the kind of capability provided by Maliput. Existing fleet systems have leveraged some of these capabilities but in ad hoc and duplicative ways.

1.1 Contents

1.1.1 Maliput Overview

Table of Contents

Summary

A C++ runtime API describing a Road Network model for use in agent and traffic simulations. It guarantees a continuous description of the road geometry and supports dynamic environments with varying rules states. There are currently several implementations of *maliput*, the most complex one is based on *OpenDRIVE* specification.

Features

- G1 Contiguous road geometry description with tolerance control.
- Lane-FRAME and Inertial-FRAME support.
- Customizable traffic rules.
- Handles dynamic rule environments.
- Supports Traffic Lights.
- Convenience functions to query the Road Network and its Rules.
- Available *maliput* backend based on *OpenDRIVE* specification.
- Plugin architecture to extend Road Network implementation.
- C++ 17 compatible API.
- Python bindings.
- Support for ROS2 Foxy.
- BSD 3-Clause License.

TLDR: Jump to *Why Maliput?* section to learn what it provides and compare it against other map specifications. Disclaimer: this is not another map specification.

Maliput components

Road Network

maliput is a runtime API that describes the road volume and connectivity graph. The road model is accessed via an abstract C++ API.

maliput is agnostic of the data source for a road network. Concrete implementations for different data sources will expose the same abstract interface. Some networks may be completely synthetic (i.e. they are built from the road surface mathematical function), others will be created from measurements of real-life roads (i.e. sampled surfaces).

Road Geometry model

In the lexicon of *maliput* and its C++ API, the road volume manifold is called a *RoadGeometry*. A *RoadGeometry* is partitioned into *Segments*, which correspond to stretches of asphalt (and the space above and/or below them). Each *Segment* is a group of one or more adjacent *Lanes*. A *Lane* corresponds to a lane of travel on a road, and defines a specific parameterization of the parent *Segment*'s volume from a local Lane-Frame into the Inertial-frame. *Lanes* are connected at *BranchPoints*, and the graph of *Lanes* and *BranchPoints* describes the topology of a *RoadGeometry*. *Segments* which map to intersecting volumes of the Inertial-frame (e.g., road intersections) are grouped together into *Junctions*. The semantic direction of the lane isn't defined by the geometry but by traffic rules. Almost all lane properties will be defined by rules because of their convenient state dynamics.

To summarize, there are two complementary object graphs in *maliput*. The container hierarchy (*Junctions* contain *Segments*, which contain *Lanes*) groups together different parts of the entire road surface. Solving the routing graph (*Lanes* are joined end-to-end via *BranchPoints*) allows to derive routes in the road network.

A *RoadGeometry* may also model laterally adjacent paths to the road, such as sidewalks. If there is no G1 continuity between the road and its adjacent paths, the two must be separated by *Segment* boundaries. It does not violate *maliput*'s continuity constraint because *maliput* has no notion of laterally-adjacent *Segments*.

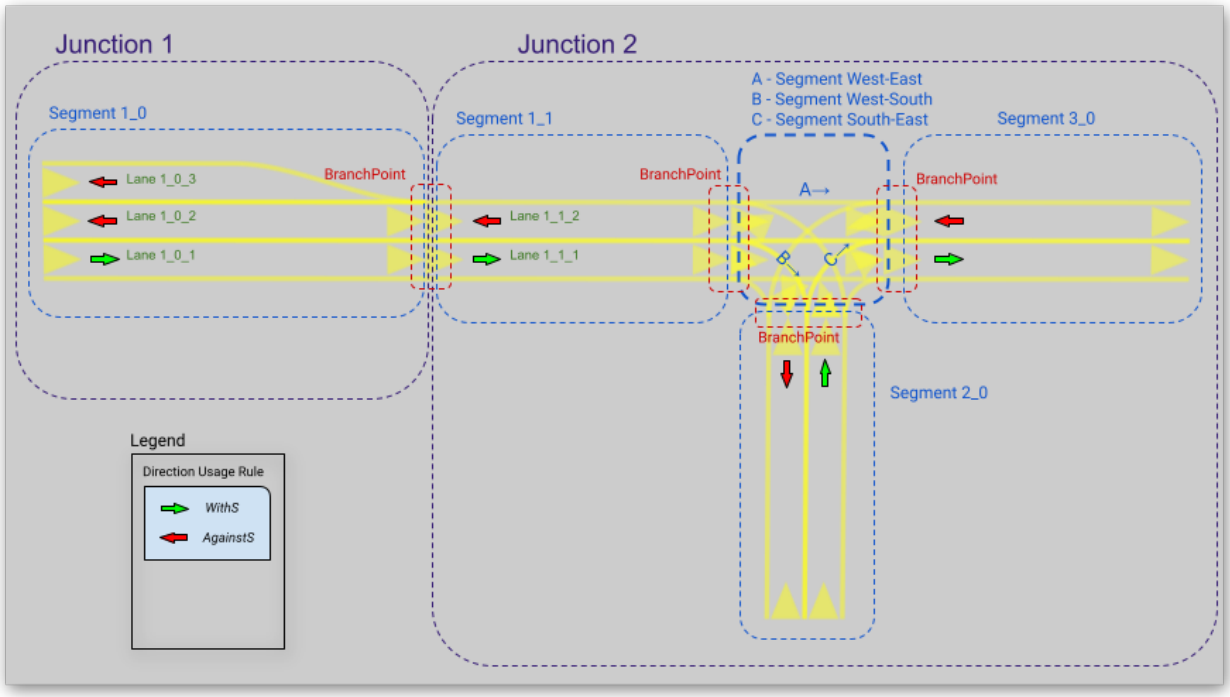
Frames

- The **Inertial-Frame** is any right-handed 3D inertial Cartesian coordinate system, with orthonormal basis (x, ŷ,) and positions expressed as triples (x,y,z).
- The **Lane-frame** is a right-handed orthonormal curvilinear coordinate system, with positions expressed as coordinates (s,r,h). Each Lane in a `RoadGeometry` defines its own embedding into the Inertial space, and thus each Lane has its own Lane-Frame.

G1 Contiguity

G1 contiguity is controlled via a linear tolerance (measured in meters) and an angular tolerance (measured in radians). Tolerances are checked at the BranchPoints by *maliput* API and it is required that the implementation respects them for all points the *RoadGeometry* volume.

Road Network Example



TODO: Mention about queries provided to traverse the graph.

Intersections

maliput provides a register of *Intersections* called *IntersectionBook* which holds all the *Intersections* in the map. Each *Intersection* aggregates related entities by zone and applied rules and their states.

Traffic Rules

Rules

In *maliput* the rules have the following properties:

- *zone*: the lane route where the rule applies.
- *type*: user defined rule types: speed-limit rule, right-of-way rule, direction usage rule, vehicle usage rule, etc.
- *states*: Each rule could be static (ie. it has one state) or dynamic (it has multiple states). The API supports having states that are either a discrete valued (which are named by string labels) or define a contiguous range of a quantity (a.k.a. *DiscreteValueRule* and *RangeValueRule*). Each state has the following properties:
  - *severity*: A non-negative quantity that specifies the level of enforcement.
  - *related rules*: Holds groups of rules that are related to the one being described.
  - *related unique ids*: Holds groups of related uniques ids typically used for traffic lights' bulb groups.

RoadRulebook

A *RoadRulebook* contains the rules for a road network. It allows to query them based on their ID and route.

Filling the book

The *RoadRulebook* can be filled with rules by two different ways:

- Manually (or by procedural code) by using the *ManualRoadRulebook* API.
- Automatically by loading a YAML file where all the rules were previously described.

RuleRegistry

The *RuleRegistry* works as a register of rule types to validate the rule type consistency. A properly created and filled *RoadRulebook* must contain rules whose type exists in the *RuleRegistry*.

The *RuleRegistry* can be filled with rules by two different ways:

- Manually, by using the *RuleRegistry* API.
- Automatically, by loading a YAML file where all the rule types were previously described.

Traffic Lights

maliput has support for traffic lights in the *RoadNetwork*.

- A **TrafficLight** models the signaling device that are typically located at road intersections. It is composed by one or more groups of light bulbs called *BulbGroup*. For each *TrafficLight* an unique id and a pose in the Inertial-frame is defined.
- A **BulbGroup** models a group of light bulbs within a traffic light. Pose is relative to the traffic light that holds it.
- A **Bulb** models a light bulb within a *BulbGroup*. The pose is relative to the *BulbGroup* it belongs. Each *Bulb* has a collection of possible states (e.g: On, Off, Blinking).

Consequently, it is possible to define pretty complex traffic lights arrays.

Dynamic Rules

maliput supports dynamic rule states. Having more than one possible state per rule and bulbs allows to define complex relations between them for a given region. *maliput* offers a set of convenient classes to ease the general state transition management,

Phases

In a typical road intersection, we may identify at least two *maliput* entities whose states may change.

- The *Bulbs*' state in *TrafficLights*.
- The rule state of dynamic rules. For instance, a *DiscreteValueRule* whose type is *Right-Of-Way*.

To couple the *Bulb* and *Rule* states, *maliput* introduces the *Phases*. A *Phase* aggregates rule states and bulb states. *PhaseRings* manage the transition cycle between *Phases*.

TODO: Here there should be a link to more information about phases. Probably to an example as it is the best way to understand phases, phase ring and phase providers.

Maliput Design and Architecture

maliput package is in essence a C++ runtime API with most of the classes being purely virtual.

Along the API, other namespaces/libraries are provided by *maliput*:

- **api**: Defines the *maliput* API.
- **base**: Base implementations of rules and traffic-lights related API.
- **geometry_base**: Base implementations of geometry-related API.
- **common**: Contains classes used by other namespaces and packages.(i.g: Logger, errors, etc)
- **math**: Math library providing support for vector, matrix, quaternion, and roll, pitch and yaw representations.
- **plugin**: *maliput* provides a plugin architecture for easily customize certain systems implementations.
- **routing**: Provides methods to obtain routes in the *RoadNetwork* graph.
- **test_utilities**: Contains convenience helpers for testing the *RoadNetwork*.
- **utilities**: Provides useful methods and classes related to mesh generation and concurrent task solvers.
- **utility**: Contains file-handling related methods.

Implementing Maliput backend

As we mentioned before *maliput* defines an API that forces the backends to meet its requirements.

When implementing a *maliput* backend, the following needs to be taken into account.

1 - Implement classes related to the road geometry model:

- *maliput::api::RoadGeometry*: It is partially implemented at *geometry_base*, however the fundamental geometric methods that define the immersion of Lane-Frame into Inertial-Frame is specific to each backend.
  - *maliput::api::Lane*

2 - Populate the *RoadNetwork*:

- Add *Lanes* to *Segments*.
- Add *Segments Junctions*.
- Add *Junctions* to the *RoadGeometry*.
- Populate RoadNetwork related entities: Many of them have a builder at *maliput* to easily create them.
  - RuleRegistry

  - RoadRulebook.
  - IntersectionBook.
  - TrafficLightBook.
  - PhaseRingBook.
  - PhaseProvider
  - DiscreteValueRuleStateProvider
  - RangeValueRuleStateProvider

Maliput Plugin Architecture

maliput provides an architecture that allows users to customize certain systems implementations in an easy and effective way. *maliput*'s clients may opt to use the plugin architecture to load at runtime specific backends. That simplifies and unifies the linkage process and reduces the number of compile time dependencies.

For further information refer to Maliput Plugin Architecture page.

Maliput backends

Available concrete implementations of the abstract API:

- maliput_dragway : *maliput_dragway is an implementation of `maliput*'s API that allows users to instantiate a multi-Lane dragway. All lanes in the dragway are straight, parallel, and in the same segment. The ends of each lane are connected together via a "magical loop" that results in vehicles traveling on the Dragway's lanes instantaneously teleporting from one end of the lane to the opposite end of the lane. The number of lanes and their lengths, widths, and shoulder widths are all user customizable.
- maliput_multilane: *maliput_multilane* is an implementation of *maliput*'s API that allows users to instantiate a *RoadNetowork* with the following relevant characteristics:
  - Multiple Lanes are allowed per Segment.
  - Constant width Lanes.
  - Segments with lateral asphalt extensions, a.k.a. shoulders.
  - Line and Arc base geometries, composed with cubic elevation and superelevation polynomials.
  - Semantic Builder API.
  - YAML based map description.
  - Adjustable linear tolerance.
  - The number of lanes and their lengths, widths, and shoulder widths are all user specifiable.
- maliput_malidrive : *maliput_malidrive* is an implementation of *maliput*'s API that allows users to instantiate a *RoadNetwork* based on the *OpenDRIVE* specification which allows defining complex *RoadGeometry* as the standard guarantees.
  - OpenDRIVE based map description.
  - Multiple Lanes per Segment.
  - Line and Arc base geometries, composition is allowed.
  - Elevation profile defined by piecewise-defined cubic polynomials
  - Lateral profile defined by piecewise-defined cubic polynomials * Supports superelevation description.

  - Varying lane width.
  - Adjustable linear tolerance.

- maliput_osm : *maliput_osm* is an implementation of *maliput*'s API that allows users to instantiate a *RoadNetwork* based on the *Lanelet2-OSM* specification.
  - Lanelet2 based map description.
  - Multiple Lanes per Segment.
  - Geometries defined by discrete points.

Maliput Sparse

maliput_sparse *maliput_sparse* is a convenient package that provides several helpers for creating a maliput backend that is expected to be built on top of waypoints without any analytical model of the surface.

The mathematical model is solved under the hood so the user doesn't have to dive into complex geometric calculations.

maliput_osm is built on top of the *maliput_sparse* package.

Maliput Vizualizer

maliput_viz is a *maliput*'s visualization tool. It allows users to visualize *maliput*'s *RoadNetwork* in a 3D scene.

Maliput Python interface

Python bindings are provided by maliput_py package. Only the API is covered.

Dependencies

maliput and its related packages have focused on being lightweight and keeping a low number of external dependencies.

The dependencies are listed in the *package.xml* file of each repository.

Why Maliput?

As it was mentioned along the document, *maliput* proposes an API to query a *RoadNetwork* model, guaranteeing, among other things, a continuous description of the road (under certain user-defined tolerance) and handling dynamic environments where traffic rules and traffic lights may change according other conditions (e.g.: time events).

maliput does not focus on a specific format, e.g. *lanelet2* or *OpenDRIVE*. It's a *maliput* backend the one that will convert / parse / load a specific data bundle described in terms of a specification into a *maliput* implementation that could be used seamlessly by simulated agents.

TODO: Should this section be located at the top of the document?

Comparison with other libraries

Even though there aren't many open-source map handling frameworks out there, it is worth noting some differences with *lanelet2* library to understand the advantages that *maliput* provides.

- Road surface definition

  maliput guarantees G1 contiguity on the *Road Network* surface under certain user-defined tolerance. The description of the surface can be as versatile as it is required by downstream packages. In particular, *maliput_malidrive* package, which is a *maliput* backend, is based on the *OpenDRIVE* specification. This *OpenDRIVE* specification provides vast control over the physical characteristics that a road may have (e.g.: elevation, banking, crossfall, OpenCRG integration) which endures obtaining a more realistic road surface model. *lanelet2* is based on an custom *OSM* (or derived schema) description format in which the lanes are defined by using two polylines to indicate both left and right boundaries. The lane surface is inferred from the polygons that those two polylines define. The standard only guarantees G0 contiguity by definition and the implementation doesn't provide tolerance control. Road's characteristics like elevation and banking profiles could be achieved by using the same points used to define lanes. However, information like crossfall of the road isn't supported.

- Traffic rules descriptions.

  In *maliput* traffic rules can be loaded via YAML file and they are independent of the underlying map format that is being used in the *maliput* backend. The rules are meant to apply to a zone in particular including one or more consecutive lanes (routes), consequently obtaining the rules that apply to a particular lane range is rather trivial. In *lanelet2* the rules are extended by creating *Regulatory Elements* and adding them into the OSM description file. Computing where each rule starts or ends isn't straightforward in comparison with *maliput*. Additional geometry calculations are required to obtain the rule range because there is no Lane-Frame in *lanelet2*.

- Rules dynamic states

  maliput supports environments with dynamic rules, that is, rules that change their states based on different conditions (e.g: time). Several entities are provided to gracefully handle these situations. *lanelet2* has no builtin support for dynamic rules. Road designer can extend the specification with custom behaviors though.

- Intersection's helpers

  In *maliput*, the intersections of the *RoadNetwork* are identified to easily manage the state of the rules that apply to a particular intersection (e.g: Right-Of-Way rules depending on traffic light's bulb states.). On the contrary, identifying crossing roads and the rules that apply to the intersection could be rather challenging in *lanelet2*.

1.1.2 Gallery

Table of Contents

Collection of demos that shows `maliput` features via `maliput-malidrive` implementation:

Road Geometry

At the moment, *maliput-malidrive* supports *Lines* and *Arcs* as geometries to describe a road. Each road can be piecewise-defined using an unlimited number of arcs and lines.

Geometries

- **Line**

Used map: SingleLane.xodr

- **Arc**

Used map: ArcLane.xodr

- **Combination of arcs and lines**

Used map: SShapeRoad.xodr

Several lanes per road

- Each road supports as many lanes as needed.

Used map: LShapeRoadVariableLanes.xodr

Lanes with varying width

Used map: LineVariableWidth.xodr

Lanes with varying offset

Used map: LineVariableOffset.xodr

Roads with elevation profile

Elevation of a Road could be piecewise-defined by using a unlimited number of third-grade polynomials.

Used map: ParkingGarageRamp.xodr

Roads with lateral profile

- Superelevation:

Superelevation of a Road could be piecewise-defined by using a unlimited number of third-grade polynomials.

Used map: SShapeSuperelevatedRoad.xodr

Intersections

- Connections/intersections between roads are supported.

Used map: TShapeRoad.xodr

Integration demos

The *delphyne_demos* package provides several demos where *delphyne* agents are placed on *maliput* Road Networks.

Agent traversing a city

Used map: Town07.xodr

Dynamic environments: Traffic Lights and Rules

Agents being aware of state rules changing according to traffic lights.

Used map: LoopRoadPedestrianCrosswalk.xodr

1.1.3 Installation

Table of Contents

Introduction

This page aims to provide a quick overview of the installation process of maliput repositories. The maliput ecosystem is compound by several packages, both core packages and utility packages.

Packages

Table 1: Maliput packages

Packages	Ros Index
maliput	ros-foxy-maliput
maliput_py	ros-foxy-maliput-py
maliput_object	ros-foxy-maliput-object
maliput_object_py	ros-foxy-maliput-object-py
maliput_dragway	ros-foxy-maliput-dragway
maliput_drake	ros-foxy-maliput-drake
maliput_multilane	ros-foxy-maliput-multilane
maliput_malidrive	ros-foxy-maliput-malidrive
maliput_sparse	ros-foxy-maliput-sparse
maliput_osm	ros-foxy-maliput-osm
maliput_viz	ros-foxy-maliput-viz
maliput_integration	ros-foxy-maliput-integration
drake_vendor	
delphyne	
delphyne_gui	
delphyne_demos	

Supported platforms

- Ubuntu Focal Fossa 20.04 LTS.

Binary Installation on Ubuntu

Prerequisites

Add ROS2 repository to your source list (see ROS2 Foxy setup process):

```
sudo apt update && sudo apt install curl gnupg2 lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key  -o /usr/
↪share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
↪keyring.gpg] http://packages.ros.org/ros2/ubuntu $(source /etc/os-release && echo
↪$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

```
sudo apt install python3-rosdep python3-colcon-common-extensions
```

Install binaries

Install the binaries of the packages:

```
sudo apt install ros-foxy-<package_name>
```

Use *maliput-full* package to install all the maliput packages:

```
sudo apt install ros-foxy-maliput-full
```

Or simple indicate the packages to install. For example to install *maliput_malidrive* package:

```
sudo apt install ros-foxy-maliput-malidrive
```

This depends on *maliput* ` and *maliput_drake* packages so they are expected to be installed too.

Source Installation on Ubuntu

See *Developer Setup*.

1.1.4 Getting Started

Table of Contents

This page will help you out getting started with the library.

Warning: Please go to the *Installation* page before you start.

As explained in the *Maliput Overview* page, maliput package is an API which implementation is provided by a backend. At the moment there are three different implementations: maliput_dragway, maliput_multilane, and maliput_malidrive.

The maliput_malidrive implementation is the one that provides more feature support, so it is the recommended choice.

The Basics

Loading a RoadNetwork

The maliput::api::RoadNetwork is the main entity from a hierarchical point of view point. It aggregates everything pertaining to *Maliput*. Once a road network is loaded, you can access its elements:

- maliput::api::RoadGeometry: Provides several geometric queries making focus on the road surface and semantic of lanes.
- maliput::api::IntersectionBook: Provides information about the intersections in the road network.
- maliput::api::rules::TrafficLightBook: Related to traffic lights in the road network.
- maliput::api::rules::RuleRegistry: Related to types of rules that are registered for the road network.
- maliput::api::rules::RangeValueRule: Related to rules indicating continuous ranges for their possible states (e.g. allowed speed).
- maliput::api::rules::DiscreteValueRule: Related to rules indicating a discrete value for each rule's state (e.g. right of way).
- maliput::api::rules::PhaseRingBook: Provides information about the phases that participate, for example, in the intersections.
- maliput::api::rules::PhaseProvider: Provides the current phase for each phase ring.
- maliput::api::rules::DiscreteValueRuleStateProvider and maliput::api::rules::RangeValueRuleStateProvider: Provide the current state for a given rule.

There are two ways of loading a RoadNetwork: - Using the maliput backend's entry point for loading a RoadNetwork. - Using the Maliput Plugin Architecture where the maliput backends/implementations are loaded in runtime.

Let's focus on the first way of loading a road network.

Load a maliput_malidrive RoadNetwork

1. If you are using CMake, link to maliput's libraries:

```
find_package(maliput)
find_package(maliput_malidrive)
# ...
# ...
target_link_libraries(<your_target>
  maliput::api
  maliput_malidrive::loader
)
```

2. Relies on the maliput_malidrive's loader for loading the maliput::api::RoadNetwork:

```
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", "<path_to_xodr_file>");
auto road_network = malidrive::loader::Load
↪<malidrive::builder::RoadNetworkBuilder>(road_network_configuration);
```

There are several parameters that can be passed to the *maliput_malidrive* loader. In this case, *opendrive_file* parameters is suggested as the *maliput_malidrive* relies on the OpenDRIVE standard for describing road networks. You can check all the *maliput_malidrive*'s parameters at Road Network Configuration Builder keys

maliput_malidrive package provides several XODR files as resources and they available at */opt/ros/<ROS_DISTRO>/share/maliput_malidrive/resources/odr*, for this case we could replace then <path_to_xodr_file> by */opt/ros/<ROS_DISTRO>/share/maliput_malidrive/resources/odr/TShapeRoad.xodr*

Note: maliput_malidrive package adds an environment variable called *MALIPUT_MALIDRIVE_RESOURCE_ROOT* that points to resources's root folder.

Querying the RoadGeometry

- maliput::api::RoadGeometry::ById: Obtains lane, segment, junction and branch point information via maliput::api::RoadGeometry::IdIndex.

```
const maliput::api::RoadGeometry* road_geometry = road_network->road_geometry();
const maliput::api::Lane* lane = road_geometry->ById.GetLane(maliput::api::LaneId{"1_0_1
↪"});
```

- maliput::api::RoadGeometry::ToRoadPosition: Convert a inertial position to a road position.

```
const maliput::api::RoadGeometry* road_geometry = road_network->road_geometry();
maliput::api::RoadPositionResult road_position_result = road_geometry->
↪ToRoadPosition(maliput::api::InertialPosition{10.0, 0.0, 0.0});;
const maliput::api::Lane* lane = road_poisition_result.road_position.lane;
```

- maliput::api::Lane::ToInertialPosition: Obtains a inertial position from a road position.

```
const maliput::api::RoadGeometry* road_geometry = road_network->road_geometry();
maliput::api::InertialPosition inertial_position = lane->
↪ToInertialPosition(maliput::api::LanePosition{0.0, 0.0, 0.0});
```

For a complete maliput api reference please visit: maliput::api

Loading a RoadNetwork via Maliput Plugin Architecture

1. If you are using CMake, link to *maliput* library:

```
find_package(maliput)
# ...
target_link_libraries(<your_target>
  maliput::api
  maliput::plugin
)
```

   We link against *maliput::api* and *maliput::plugin* for using the plugin interface. Note that we aren't linking against any maliput backend(*maliput_malidrive* in this case).

2. Use *maliput::plugin*'s convenient method for loading a maliput::api::RoadNetwork instance.

```
// ...
#include <maliput/api/road_network.h>
#include <maliput/plugin/create_road_network.h>

const std::string road_network_loader_id = "maliput_malidrive";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", "<path_to_xodr_file>");
// Use maliput plugin interface for loading a road network
std::unique_ptr<maliput::api::RoadNetwork> road_network =␣
↪maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_network_
↪configuration);
```

   The maliput's implementation, *maliput_malidrive* in this case, is loaded in runtime. Therefore, no need to link to *maliput_malidrive* library.

   See Maliput Plugin Architecture for further information.

Maliput Python Interface

maliput_py package provides bindings to the maliput library. See Maliput Python Interface for general information about the maliput python interface

Load a maliput_malidrive RoadNetwork

As the intention is to use the python interface, it is expected that maliput_py and maliput_malidrive packages are installed.

Note: Check *Installation* for setting up the ROS2 repositories and installing the packages via binaries.

Once the dependencies are installed we can load a road network using the python interface. For doing so we are relying on the maliput_py package for the corresponding maliput bindings and the maliput_malidrive package as a maliput implementation.

```
import maliput.api
import maliput.plugin

import os

configuration = {"opendrive_file" : os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/
↪resources/odr/TShapeRoad.xodr"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
print(road_network.road_geometry().id())
```

Advanced

Traffic Lights

maliput models traffic lights via maliput::api::rules::TrafficLight. It contains one or more groups of light bulbs with varying colors and shapes. Note that traffic lights are physical manifestations of underlying right-of-way rules.

- maliput::api::rules::TrafficLight: A **TrafficLight** models the signaling device that are typically located at road intersections. It is composed by one or more groups of light bulbs called *BulbGroup*. For each *TrafficLight* an unique id and a pose in the Inertial-frame is defined.
- maliput::api::rules::BulbGroup: A **BulbGroup** models a group of light bulbs within a traffic light. Pose is relative to the traffic light that holds it.
- maliput::api::rules::Bulb: A **Bulb** models a light bulb within a *BulbGroup*. The pose is relative to the *BulbGroup* it belongs. Each *Bulb* has a collection of possible states (e.g: On, Off, Blinking).

maliput::api::rules::TrafficLightBook is an interface that allows getting the traffic lights according their ids.

Loading a TrafficLightBook

maliput provides a base implementation of the maliput::api::rules::TrafficLightBook, which can be used for adding maliput::api::rules::TrafficLight s to the book. However, the most convenient way of populating this book is to load it via YAML file by using the maliput::LoadTrafficLightBookFromFile method.

As example, we will use the maliput_malidrive backend, which fully supports maliput's API.

Listing 1: C++

```
// ...
#include <maliput/api/road_network.h>
```

(continues on next page)

(continued from previous page)

```cpp
#include <maliput/plugin/create_road_network.h>

const std::string road_network_loader_id = "maliput_malidrive";
const std::string resources_path = std::string(std::getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT")) + "/resources/odr";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", resources_path + "/LoopRoadPedestrianCrosswalk.xodr");
road_network_configuration.emplace("traffic_light_book", resources_path + "/LoopRoadPedestrianCrosswalk.yaml");
auto road_network = maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_network_configuration);
```

Listing 2: Python

```python
import maliput.api
import maliput.plugin

import os

resources_path = os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/resources/odr";
configuration = {"opendrive_file" : resources_path + "/LoopRoadPedestrianCrosswalk.xodr",
                 "traffic_light_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
```

While the LoopRoadPedestrianCrosswalk.xodr file contains the road network description using the OpenDRIVE format specification, the LoopRoadPedestrianCrosswalk.yaml describes other aspects of the road network using the YAML format specification. For the moment, we focus on the *TrafficLights* section using the YAML format specification.

After loading the road network we can get the *TrafficLightBook* from the *RoadNetwork*, and obtain any required information:

Listing 3: C++

```cpp
// ...
#include <maliput/api/lane_data.h>
#include <maliput/api/rules/traffic_lights.h>
#include <maliput/api/rules/traffic_light_book.h>

// ...
const maliput::api::rules::TrafficLightBook* book = road_network->traffic_light_book();
const maliput::api::rules::TrafficLight::Id traffic_light_id{"WestFacingSouth"};
const maliput::api::InertialPosition inertial_position = book->GetTrafficLight(traffic_light_id)->position_road_network();
```

Listing 4: Python

```python
# ...
traffic_light_book = road_network.traffic_light_book()
traffic_light_id = maliput.api.rules.TrafficLight.Id("WestFacingSouth")
inertial_position = traffic_light_book.GetTrafficLight(traffic_light_id).position_road_network()
```

(continues on next page)

(continued from previous page)

```
print(inertial_position.xyz())
```

Rules

maliput provides an API for rule support. The rules are used to model all kind of traffic rules that could be applied to a road network.

The base interface for rules is maliput::api::rules::Rule. Each rule has:

- *id*: a unique identifier for the rule
- *type id*: a unique identifier for the type of the rule
- *zone*: a zone that the rule is applied to.

For each rule can be defined as many as states as needed. Each state is defined by:

- *severity*: a severity for the state.
- *related rules*: a group of rules that are related to the state.
- *related unique ids*: a group of unique ids related to the state, typically used for the TrafficLights that are affected by the state.
- *value*: a value for the state.

Depending on the nature of the values of the rule's states, two kinds of rules are defined:

- maliput::api::rules::DiscreteValueRule: a rule which states contain discrete values (e.g: Go and Stop for a right-of-way rule.)
- maliput::api::rules::RangeValueRule: a rule which states contain a range of values (e.g: Speed limit for a speed limit rule.)

Rule Registry

maliput provides a registry of rules for registering a type of rule and the states they possible have.

maliput::api::rules::RuleRegistry provides a registry of the various rule types, and enables semantic validation when building rule instances.

Loading a Rule Registry

maliput provides a way to load the rule registry via a YAML file by using the maliput::LoadRuleRegistryFromFile method.

As example, we will use the maliput_malidrive backend.

Listing 5: C++

```
// ...
#include <maliput/api/lane_data.h>
#include <maliput/api/road_network.h>
#include <maliput/api/rules/traffic_lights.h>
#include <maliput/api/rules/traffic_light_book.h>
#include <maliput/plugin/create_road_network.h>
```

(continues on next page)

(continued from previous page)

```

const std::string road_network_loader_id = "maliput_malidrive";
const std::string resources_path = std::string(std::getenv("MALIPUT_MALIDRIVE_RESOURCE_
↪ROOT")) + "/resources/odr";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", resources_path + "/
↪LoopRoadPedestrianCrosswalk.xodr");
road_network_configuration.emplace("traffic_light_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("rule_registry", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
auto road_network = maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_
↪network_configuration);
```

Listing 6: Python

```
import maliput.api
import maliput.plugin

import os

resources_path = os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/resources/odr";
configuration = {"opendrive_file" : resources_path + "/LoopRoadPedestrianCrosswalk.xodr",
                 "traffic_light_book" : resources_path + "/LoopRoadPedestrianCrosswalk.
↪yaml",
                 "rule_registry" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
```

In this example, LoopRoadPedestrianCrosswalk.yaml contains a *RuleRegistry* section where the rules types are defined. These rules are going to be used later on by the *RoadRulebook* to validate the rule types.

After loading the road network, the *RuleRegistry* is accessible from the *RoadNetwork*.

```
// ...
const maliput::api::rules::RuleRegistry* rule_registry = road_network->rule_registry();
// Obtains all the DiscreteValueRules from the registry.
auto discrete_types = rule_registry->DiscreteValueRuleTypes();
// Obtains all the RangeValueRules from the registry.
auto range_types = rule_registry->RangeValueRuleTypes();
```

Listing 7: Python

```python
# ...
rule_registry = road_network.rule_registry()
print(len(rule_registry.DiscreteValueRuleTypes()))
print(len(rule_registry.RangeValueRuleTypes()))
```

RoadRulebook

The maliput::api::rules::RoadRulebook is an interface for querying the rules in given road network. This book is expected to gathered all the available rules. It provides an API for obtaining all the rules; obtaining the rules by id; or even obtaining the rules that apply to zone in particular.

maliput provides a base implementation for loading the *RoadRulebook* with the rules. However, the most convenient way of populating this book is to load it via YAML file by using the maliput::LoadRoadRuleBookFromFile method.

As example, we will use the maliput_malidrive backend.

Listing 8: C++

```cpp
// ...
#include <maliput/api/lane_data.h>
#include <maliput/api/road_network.h>
#include <maliput/api/rules/traffic_lights.h>
#include <maliput/api/rules/traffic_light_book.h>
#include <maliput/api/rules/road_rulebook.h>
#include <maliput/plugin/create_road_network.h>

const std::string road_network_loader_id = "maliput_malidrive";
const std::string resources_path = std::string(std::getenv("MALIPUT_MALIDRIVE_RESOURCE_
↪ROOT")) + "/resources/odr";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", resources_path + "/
↪LoopRoadPedestrianCrosswalk.xodr");
road_network_configuration.emplace("traffic_light_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("rule_registry", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("road_rule_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
auto road_network = maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_
↪network_configuration);
```

Listing 9: Python

```python
import maliput.api
import maliput.plugin

import os

resources_path = os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/resources/odr";
configuration = {"opendrive_file" : resources_path + "/LoopRoadPedestrianCrosswalk.xodr",
```

(continues on next page)

(continued from previous page)

```
                    "traffic_light_book" : resources_path + "/LoopRoadPedestrianCrosswalk.
↪yaml",
                    "rule_registry" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                    "road_rule_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml
↪"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
```

In this example, LoopRoadPedestrianCrosswalk.yaml contains a *RoadRulebook* section where the rules types are defined.

After loading the road network, the *RoadRulebook* is accessible from the *RoadNetwork*.

Listing 10: C++

```
// ...
const maliput::api::rules::RoadRulebook* rulebook = road_network->rulebook();
// Obtains all the rules from the book.
auto rules = rulebook->Rules().size();
int number_of_discrete_rules = rules.discrete_value_rules.size();
// Obtains a discrete value rule by id.
maliput::api::rules::Rule::Id rule_id{"Right-Of-Way Rule Type/WestToEastSouth"};
auto discrete_rule = rulebook->GetDiscreteValueRule(rule_id);
```

Listing 11: Python

```
# ...
rulebook = road_network.rulebook()
rules = rulebook.Rules()
print(len(rules.discrete_value_rules))
rule_id = maliput.api.rules.Rule.Id("Right-Of-Way Rule Type/WestToEastSouth")
discrete_rule = rulebook.GetDiscreteValueRule(rule_id)
```

Rule State Providers

As it was mentioned, *maliput*'s rule API lets the user to add rules that may contain as many states as needed. The current state of a rule may depend on certain condition. For instance, a rule state may vary on a time basis, as right-of-way rules in a intersection according to the traffic lights.

***maliput* defines two interfaces for getting the current state of a rule depending of the nature of the rules:**

- maliput::api::rules::DiscreteValueRuleStateProvider.
- maliput::api::rules::RangeValueRuleStateProvider.

DiscreteValueRuleStateProvider

Listing 12: C++

```cpp
// ...
const maliput::api::rules::DiscreteValueRuleStateProvider* state_provider = road_network->discrete_value_rule_state_provider();
maliput::api::rules::Rule::Id rule_id{"Right-Of-Way Rule Type/WestToEastSouth"};
auto state_result = state_provider->GetState(rule_id);
auto discrete_value = state_result->state;
std::cout << discrete_value.value << std::endl;
```

Listing 13: Python

```python
# ...
state_provider = road_network.discrete_value_rule_state_provider()
rule_id = maliput.api.rules.Rule.Id("Right-Of-Way Rule Type/WestToEastSouth")
state_result = state_provider.GetState(rule_id)
discrete_value = state_result.state
print(discrete_value.value)
```

RangeValueRuleStateProvider

Listing 14: C++

```cpp
// ...
const maliput::api::rules::RangeValueRuleStateProvider* state_provider = road_network->range_value_rule_state_provider();
maliput::api::rules::Rule::Id rule_id{"Speed-Limit Rule Type/1_1_-1_1"};
auto state_result = state_provider->GetState(rule_id);
auto range_value = state_result->state;
std::cout << range_value.min << std::endl;
std::cout << range_value.max << std::endl;
```

Listing 15: Python

```python
# ...
state_provider = road_network.range_value_rule_state_provider()
rule_id = maliput.api.rules.Rule.Id("Speed-Limit Rule Type/1_1_-1_1")
state_result = state_provider.GetState(rule_id)
range_value = state_result.state
print("Rule: {} --> Current State: min={}, max={}, units={}".format(rule_id, range_value.
↪min, range_value.max, range_value.description))
```

Phases

Maliput models the sequencing of rule states and traffic lights' bulbs as a ring of *maliput::api::rules::Phase `_s. Each `Phase* holds a dictionary of rule IDs to rule states (*DiscreteValues*) and related bulb IDs (*UniqueBulbIds*) to the bulb state (*BulbState*).

The maliput::api::rules::PhaseRing acts as a container of all the related Phases in a sequence. A designer might query them by the maliput::api::rules::Phase::Id or the next Phases, but no strict order should be expected. Instead, maliput::api::rules::PhaseProvider offers an interface to obtain the current and next *Phase::Id `s for a `PhaseRing*. Custom time based or event driven behaviors could be implemented for this interface. Similarly to the rules, there are convenient "manual" implementations to exercise the interfaces in integration examples.

PhaseRingBook

The PhaseRingBook is expected to contain all the *PhaseRing `s in the road network. It provides an interface for obtaining the PhaseRings in the road network and some convenient queries to retrieve the PhaseRing that governs a specific `Rule::Id*

maliput provides a base implementation for loading the maliput::api::rules::PhaseRingBook with the rules. However, the most convenient way of populating this book is to load it via YAML file by using the maliput::LoadPhaseRingBookFromFile method.

As example, we will use the maliput_malidrive backend.

Listing 16: C++

```cpp
// ...
#include <maliput/api/lane_data.h>
#include <maliput/api/road_network.h>
#include <maliput/api/rules/phase_ring_book.h>
#include <maliput/api/rules/traffic_lights.h>
#include <maliput/api/rules/traffic_light_book.h>
#include <maliput/api/rules/road_rulebook.h>
#include <maliput/plugin/create_road_network.h>

const std::string road_network_loader_id = "maliput_malidrive";
const std::string resources_path = std::string(std::getenv("MALIPUT_MALIDRIVE_RESOURCE_
↪ROOT")) + "/resources/odr";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", resources_path + "/
↪LoopRoadPedestrianCrosswalk.xodr");
road_network_configuration.emplace("traffic_light_book", resources_path + "/
```

(continues on next page)

(continued from previous page)

```
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("rule_registry", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("road_rule_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("phase_ring_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
auto road_network = maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_
↪network_configuration);
```

Listing 17: Python

```
import maliput.api
import maliput.plugin

import os

resources_path = os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/resources/odr";
configuration = {"opendrive_file" : resources_path + "/LoopRoadPedestrianCrosswalk.xodr",
                 "traffic_light_book" : resources_path + "/LoopRoadPedestrianCrosswalk.
↪yaml",
                 "rule_registry" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                 "road_rule_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml
↪",
                 "phase_ring_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml
↪"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
```

In this example, LoopRoadPedestrianCrosswalk.yaml contains a *PhaseRings* section where all the phase rings are defined.

After loading the road network, the *PhaseRingBook* is accessible from the *RoadNetwork*.

Listing 18: C++

```
// ...
const maliput::api::rules::PhaseRingBook* phase_ring_book = road_network->phase_ring_
↪book();
// Obtains all the phase rings from the book.
auto phase_rings = phase_ring_book->GetPhaseRings();
const int number_of_phase_rings = phase_rings.size();
// Obtains a phase ring containing the specified rule.
const maliput::api::rules::Rule::Id rule_id{"Right-Of-Way Rule Type/WestToEastSouth"};
auto phase_ring = phase_ring_book->FindPhaseRing(rule_id);
// Obtains a phase of that phase ring.
const maliput::api::rules::Phase::Id phase_id{"AllGoPhase"};
auto phase = phase_ring->GetPhase(phase_id);
// Obtains all the discrete value rule states from the phase.
auto discrete_value_rule_states = phase->discrete_value_rule_states();
// Obtains all the bulb states from the phase.
auto bulb_states = phase->bulb_states();
```

Listing 19: Python

```python
# ...
phase_ring_book = road_network.phase_ring_book()
# Obtains all the phase rings from the book.
phase_rings = phase_ring_book.GetPhaseRings()
number_of_phase_rings = len(phase_rings)
# Obtains a phase ring containing the specified rule.
rule_id = maliput.api.rules.Rule.Id("Right-Of-Way Rule Type/WestToEastSouth")
phase_ring = phase_ring_book.FindPhaseRing(rule_id)
# Obtains a phase of that phase ring.
phase_id = maliput.api.rules.Phase.Id("AllGoPhase")
phase = phase_ring.GetPhase(phase_id)
# Obtains all the discrete value rule states from the phase.
discrete_value_rule_states = phase.discrete_value_rule_states()
# Obtains all the bulb states from the phase.
bulb_states = phase.bulb_states()

print(len(discrete_value_rule_states))
for key in discrete_value_rule_states.keys():
  print(key)
  print(discrete_value_rule_states[key].value)

print(len(bulb_states))
for key in bulb_states.keys():
  print(key)
  print(bulb_states[key])
```

PhaseProvider

In a dynamic environment, phases in a phase ring are expected to change over a certain condition, such as traffic light changing its state in a time basis. maliput introduces a maliput::api::rules::PhaseProvider interface to allow the user to obtain the current phase.

Listing 20: C++

```cpp
// ...
const maliput::api::rules::PhaseProvider* phase_provider = road_network->phase_
↪provider();
maliput::api::rules::PhaseRing::Id phase_ring_id{"PedestrianCrosswalkIntersectionSouth"};
auto current_phase = phase_provider->GetPhase(phase_ring_id);
std::cout << current_phase.state << std::endl;
```

Listing 21: Python

```python
# ...
phase_provider = road_network.phase_provider()
phase_ring_id = maliput.api.rules.PhaseRing.Id("PedestrianCrosswalkIntersectionSouth")
current_phase = phase_provider.GetPhase(phase_ring_id);
print(current_phase.state)
```

maliput_integration package provides an example where a dynamic environment is simulated using the *PhaseProvider*

interface. For trying out the example please visit maliput_dynamic_environment tutorial example. The source code is located at maliput_dynamic_environment.cc

Intersection

An abstract convenience class that aggregates information pertaining to an intersection. Its primary purpose is to serve as a single source of this information and to remove the need for users to query numerous disparate data structures and state providers.

See maliput::api::Intersection's API for more details.

IntersectionBook

The maliput::api::IntersectionBook is an interface for querying for the intersection in a given road network. This book is expected to gather all the available maliput::api::Intersection s. The API allows you to find intersections by maliput::api::Intersection, maliput::api::rules::TrafficLight or *maliput::api::rules::DiscretValueRule* ID and even by inertial position.

maliput provides a base implementation for loading the maliput::api::Intersection s. However, the most convenient way of populating this book is to load it via YAML file by using the maliput::LoadIntersectionBookFromFile method.

As example, we will use the maliput_malidrive backend.

Listing 22: C++

```cpp
// ...
#include <maliput/api/intersection_book.h>
#include <maliput/api/lane_data.h>
#include <maliput/api/road_network.h>
#include <maliput/api/rules/phase_ring_book.h>
#include <maliput/api/rules/traffic_lights.h>
#include <maliput/api/rules/traffic_light_book.h>
#include <maliput/api/rules/road_rulebook.h>
#include <maliput/plugin/create_road_network.h>

const std::string road_network_loader_id = "maliput_malidrive";
const std::string resources_path = std::string(std::getenv("MALIPUT_MALIDRIVE_RESOURCE_
↪ROOT")) + "/resources/odr";
std::map<std::string, std::string> road_network_configuration;
road_network_configuration.emplace("opendrive_file", resources_path + "/
↪LoopRoadPedestrianCrosswalk.xodr");
road_network_configuration.emplace("traffic_light_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("rule_registry", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("road_rule_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("phase_ring_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
road_network_configuration.emplace("intersection_book", resources_path + "/
↪LoopRoadPedestrianCrosswalk.yaml");
auto road_network = maliput::plugin::CreateRoadNetwork(road_network_loader_id, road_
↪network_configuration);
```

Listing 23: Python

```python
import maliput.api
import maliput.plugin

import os

resources_path = os.getenv("MALIPUT_MALIDRIVE_RESOURCE_ROOT") + "/resources/odr";
configuration = {"opendrive_file" : resources_path + "/LoopRoadPedestrianCrosswalk.xodr",
                 "traffic_light_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                 "rule_registry" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                 "road_rule_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                 "phase_ring_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml",
                 "intersection_book" : resources_path + "/LoopRoadPedestrianCrosswalk.yaml"}
road_network = maliput.plugin.create_road_network("maliput_malidrive", configuration)
```

In this example, LoopRoadPedestrianCrosswalk.yaml contains a *Intersections* section where all the intersections are defined.

After loading the road network, the *IntersectionBook* is accessible from the *RoadNetwork*.

Listing 24: C++

```cpp
// ...
const maliput::api::IntersectionBook* intersection_book = road_network->intersection_book();
// Obtains all intersections from the book.
auto intersections = intersection_book->GetIntersections();
const auto number_of_intersections = intersections.size();

// Obtains a intersection containing the specified traffic light.
const maliput::api::rules::TrafficLight::Id traffic_light_id{"WestFacingSouth"};
maliput::api::Intersection* traffic_light_intersection = intersection_book->FindIntersection(traffic_light_id);

// Obtain a intersection containing the specified discrete value rule.
const maliput::api::rules::DiscreteValueRule::Id discrete_value_rule_id{"Right-Of-Way␣Rule Type/WestToEastSouth"};
maliput::api::Intersection* discrete_rule_intersection = intersection_book->FindIntersection(discrete_value_rule_id);

// Obtains the rule states of the intersection.
auto discrete_value_rule_states = discrete_rule_intersection->DiscreteValueRuleStates();
// Obtains the bulb states of the intersection.
auto bulb_states = discrete_rule_intersection->bulb_states();
```

Listing 25: Python

```python
# ...
```

(continues on next page)

(continued from previous page)

```
intersection_book = road_network.intersection_book()
# Obtains all intersections from the book.
intersections = intersection_book.GetIntersections()
number_of_intersections = len(intersections)

# Obtains a intersection containing the specified traffic light.
traffic_light_id = maliput.api.rules.TrafficLight.Id("WestFacingSouth")
traffic_light_intersection = intersection_book.FindIntersection(traffic_light_id)

# Obtain a intersection containing the specified discrete value rule.
discrete_value_rule_id = maliput.api.rules.DiscreteValueRule.Id("Right-Of-Way Rule Type/
↪WestToEastSouth")
discrete_rule_intersection = intersection_book.FindIntersection(discrete_value_rule_id)

# Obtains the rule states of the intersection.
discrete_value_rule_states = discrete_rule_intersection.DiscreteValueRuleStates()
# Obtains the bulb states of the intersection.
bulb_states = discrete_rule_intersection.bulb_states()
```

Further readings

Maliput Design contains addition information about the API in case you are interested in the details.

1.1.5 Tutorials

Table of Contents

Maliput Malidrive

- Maliput Malidrive Tutorials

Maliput Integration

- Maliput Integration Tutorials

1.1.6 API Documentation

Table of Contents

Maliput

- Maliput C++ namespace

Maliput Malidrive

- Maliput_malidrive C++ namespace

Dragway

- Dragway C++ namespace

Maliput Object

- Maliput Object C++ namespace

Maliput Object Py

- Maliput Object Py Python Interface TODO

Maliput Py

- Maliput Python Interface

Maliput Sparse

- Maliput Sparse C++ namespace

Maliput Osm

- Maliput Osm C++ namespace

Multilane

- Multilane C++ namespace

Integration

- Integration C++ namespace

Delphyne

- Delphyne C++/Python namespace

Delphyne Gui

- Delphyne Gui doc

Delphyne Demos

- Delphyne Demos doc

1.1.7 Design

Table of Contents

- *Multilane*

Maliput

- Maliput Design
- Maliput Plugin Architecture

Maliput Py

- Maliput Python Interface

Maliput Malidrive

- RoadCurve design

Multilane

- Multilane Design

1.1.8 Roadmap

The entire `maliput` project has a cross package roadmap. Not all of the topics affect them all but they are important enough to list them here. Specific discussions will be opened in tickets to work on design documents and labor breakdown. Ordering in the following list does not mean priority.

- Infrastructure
  - ROS2 targeted branches and semantic versioning.
- Tooling and integration
  - Migrate to ignition fortress.
  - Update `drake_vendor` dependency and refactor `delphyne`.
  - Build a visualizer to evaluate the connectivity graph and routing results.
- `maliput`
  - Spacial queries * Provide spacial queries convenient methods to query the underlying graph.
  - Speed up maliput backend creation * Provide a common interface for road network creation. * Add support for road network creation from waypoints.
  - Routing API
    - Design and implementation.
    - Integration examples.
  - Scenario modeling
    - Furniture API / Lane markings API
      - Design and implementation.
      - Integration examples.

        * Surface data
            - Integrate with OpenCRG.
            - Define and implement and API to run surface queries beyond the current `Lane` support.
    - Complex integration examples
        * Build smarter agents that take into account rules and phasing.

1.1.9 Changelog

All notable changes to the maliput packages will be documented in this file.

Table of Contents

maliput

1.0.7 (2022-09-14)

- **maliput::api::Lane::ToLanePosition behavior has changed to return a position within the lane boundaries. (#521)**
    - maliput::api::Lane::ToSegmentPosition was added to return a position within the lane segment boundaries, replacing the previous behavior of *ToLanePosition*.
- Fixes FindRoadPosition behavior for correctly returning road position that are only located within look-up radius. #496

1.0.6 (2022-08-16)

- KDTree support was added to the maliput::math namespace. (#520, #515)
- Brings in BoundingRegion support from maliput_object package. (#518, #519)

1.0.5 (2022-07-26)

- Provides convenient method for loading a RoadNetwork using the RoadNetworkPlugin architecture. (#512)

1.0.4 (2022-06-13)

1.0.3 (2022-06-08)

1.0.2 (2022-06-06)

1.0.1 (2022-06-02)

- First official release

For a complete list of changes, see the maliput's changelog

maliput_dragway

0.1.3 (2022-09-14)

- Matches *maliput::api::Lane::ToLanePosition* change of behavior and implements new *maliput::api::Lane::ToSegmentPosition* method. (#76)

0.1.2 (2022-07-01)

0.1.1 (2022-06-21)

0.1.0 (2022-06-16)

- First official release

For a complete list of changes, see the maliput_dragway's changelog

maliput_integration

0.1.3 (2022-09-15)

- Adds maliput::api::Lane::ToSegmentPosition query to the maliput_query app. (#123)

0.1.2 (2022-08-16)

0.1.1 (2022-07-29)

0.1.0 (2022-06-21)

- First official release

For a complete list of changes, see the maliput_integration's changelog

maliput_malidrive

0.1.3 (2022-09-14)

- Matches *maliput::api::Lane::ToLanePosition* change of behavior and implements new *maliput::api::Lane::ToSegmentPosition* method. (#227)

0.1.2 (2022-07-01)

0.1.1 (2022-06-16)

0.1.0 (2022-06-13)

- First official release

For a complete list of changes, see the maliput_malidrive's changelog

maliput_multilane

0.1.3 (2022-09-14)

- Matches *maliput::api::Lane::ToLanePosition* change of behavior and implements new *maliput::api::Lane::ToSegmentPosition* method. (#95)

0.1.2 (2022-07-29)

0.1.1 (2022-07-01)

0.1.0 (2022-06-16)

- First official release

For a complete list of changes, see the maliput_multilane's changelog

maliput_object

0.1.2 (2022-08-16)

- Moves *BoundingRegion*, *BoundingBox* and *OverlappingType* to maliput::math (#44)

0.1.1 (2022-06-21)

0.1.0 (2022-06-10)

- First official release

For a complete list of changes, see the maliput_object's changelog

maliput_object_py

0.1.2 (2022-08-23)

- Pairs with BoundingRegion being moved to maliput. (#8)

0.1.1 (2022-06-21)

0.1.0 (2022-06-21)

- First official release

For a complete list of changes, see the maliput_object_py's changelog

maliput_py

0.1.3 (2022-09-14)

- Adds binding for maliput::api::ToSegmentPosition method. (#70)

0.1.2 (2022-07-28)

- Use maliput's method for creating road network via plugin api. (#68)
- Adds TrafficLightBook bindings. (#65)
- Fixes IntersectionBook's bug. (#69)

0.1.1 (2022-06-16)

0.1.0 (2022-06-16)

For a complete list of changes, see the maliput_py's changelog

1.1.10 Contributing

Contributing to Maliput

The following is a set of guidelines for contributing to Maliput and its components, which are hosted in the Maliput Organization on GitHub. These are mostly guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.

Table of Contents

Code of Conduct

This project and everyone participating in it is governed by the Maliput *ROS Community Code of Conduct*. By participating, you are expected to uphold this code.

How to Contribute

Reporting Bugs

Before Submitting a Bug Report

1. Determine the repository which should receive the problem.
2. Search the repository's issues to see if the same or similar problem has been opened. If it has and the issue is still open, then add a comment to the existing issue. Otherwise, create a new issue.

How to Submit a Good Bug Report

Create an issue on the repository that is related to your bug, explain the problem, and include additional details to help maintainers reproduce the problem. Refer to the Short, Self Contained, Correct (Compilable), Example Guide as well as the following tips:

- **Use a clear and descriptive title** for the issue to identify the problem.
- **Describe the exact steps which reproduce the problem** in as many details as possible. When listing steps, don't just say what you did, but explain how you did it.
- **Provide specific examples to demonstrate the steps.** Include links to files or projects, or copy/pasteable snippets, which you use in those examples.
- **Describe the behavior you observed after following the steps** and point out what exactly is the problem with that behavior.
- **Explain which behavior you expected to see instead and why**.
- **Include screenshots and animated GIFs** which show you following the described steps and clearly demonstrate the problem.
- **If the problem wasn't triggered by a specific action**, describe what you were doing before the problem happened and share more information using the guidelines below.

Provide more context by answering these questions:

- **Did the problem start happening recently** (e.g. after updating to a new version) or was this always a problem?
- If the problem started happening recently, **can you reproduce the problem in an older version?** What's the most recent version in which the problem doesn't happen?
- **Can you reliably reproduce the issue?** If not, provide details about how often the problem happens and under which conditions it normally happens.

Include details about your configuration and environment:

- **Which version of Maliput are you using?**?
- **What's the name and version of the OS you're using**?
- **Are you running Maliput using the provided docker container?** See *Developer Setup*.
- **Are you running Maliput in a virtual machine?** If so, which VM software are you using and which operating systems and versions are used for the host and the guest?

Suggesting Enhancements

This section guides you through submitting an enhancement suggestion, including completely new features and minor improvements to existing functionality. Following these guidelines helps maintainers and the community understand your suggestion and find related suggestions.

Before creating enhancement suggestions, please check *Before Submitting a Bug Report* as you might find out that you don't need to create one. When you are creating an enhancement suggestion, please include as many details as possible. When filling in the issue form for an enhancement suggestion, include the steps that you imagine you would take if the feature you're requesting existed.

Before Submitting An Enhancement Suggestion

- **Check if you're using the latest software version**. A more recent version may contain your desired feature.
- **Determine which repository the enhancement should be suggested in**
- **Perform a** cursory search to see if the enhancement has already been suggested. If it has, add a comment to the existing issue instead of opening a new one.

How Do I Submit A (Good) Enhancement Suggestion

Enhancement suggestions are tracked as GitHub issues . After you've determined which repository your enhancement suggestion is related to, create an issue on that repository and provide the following information:

- **Use a clear and descriptive title** for the issue to identify the suggestion.
- **Provide a step-by-step description of the suggested enhancement** in as many details as possible.
- **Provide specific examples to demonstrate the steps**. Include copy/pasteable snippets which you use in those examples, as Markdown code blocks.
- **Describe the current behavior** and **explain which behavior you expected to see instead** and why.
- **Include screenshots and animated GIFs** which show you following the described steps and clearly demonstrate the problem.
- **Explain why this enhancement would be useful** to most users and isn't something that can or should be implemented as a separate application.
- **Specify which version of Maliput you're using.**
- **Specify the name and version of the OS you're using.**

Contributing Code

We follow a development process designed to reduce errors, encourage collaboration, and make high quality code. Review the following to get acquainted with this development process.

1. **Read the** *Reporting Bugs* **and** *Suggesting Enhancements* **sections first.**
2. **Read the** *Developer Guidelines* **page first.**
3. **Fork the Maliput package** you want to contribute to. This will create your own personal copy of the package. All of your development should take place in your fork. - An important thing to do is create a remote pointing to the upstream remote repository . This way, you can always check for modifications on the original repository and **always** keep your fork repository up to date.
4. **Work out of a new branch**, one that is not a release / main branch. This is a good habit to get in, and will make your life easier.
5. **Write your code.** To remember: - Look at the existing code and try to maintain the existing style and pattern as much as possible - **Always** keep your branch updated with the original repository

Process

All Maliput team members actively:

- **Watch** all maliput repositories to receive email notifications of new issues / pull requests
- Provide **feedback** to issues as soon as possible
- **Review** pull requests as soon as possible
  - Team members can review pull requests already under review or approved
  - Team members can provide some feedback without doing a full review

Pull requests can be merged when:

- They have at least 1 approval from a member of the core team
- There are no unresolved comments
- CI is passing
- Developer Certificate of Origin (DCO) check is passing
  - DCO is a declaration of ownership, basically saying that you created the contribution and that it is suitable to be covered under an open source license (not proprietary).
  - All you have to do is end your commit message with Signed-off-by: Your Full Name <your.name@email.com>
  - If your user.name and user.email configurations are set up in git, then you can simply run git commit -s to have your signature automatically appended.

Merging strategy:

- For internal contributions, give the original author some time to hit the merge button themselves / check directly with them if it's ok to merge.
- Default to "squash and merge" * Review the pull request title and reword if necessary since this will be part of the commit message. * Make sure the commit message concisely captures the core ideas of the pull request and contains all authors' signatures.

1.1.11 ROS Community Code of Conduct

Preamble / Attribution

The ROS Code of Conduct draws heavily from the work of other open source software communities and tries to synthesize their efforts to address the specific needs of the ROS community. In particular, we draw heavily from the following prior work.

- Python Software Foundation Code of Conduct
- The RUST language Code of Conduct
- Contributor Covenant.
- Frame Shift Consulting Code of Conduct Book

It is worth noting that this is a living document to be used to protect community members. It should not be interpreted as a hard and fast legal guide for community behavior. Instead, it should be interpreted as broad outline of acceptable and unacceptable behavior, how to report unacceptable behavior, and how it will be dealt with.

Scope of the Code of Conduct

This guide applies to all people, events, web properties, communication media, and source code repositories administered by Open Robotics and the ROS 2 Technical Steering Committee. This includes but is not limited to:

- ROS source code repositories
- ROS Discourse
- ROS Wiki
- ROS Documentation
- ROS Answers
- ROSCon
- ROS.org
- All ROS 2 TSC working groups
- All Open Robotics administered e-mail lists
- Comments made on event video hosting services
- Comments made on the official event or ROS hashtags
- If you administer a ROS affiliated organization outside of the organizations listed above and would like to adopt this code of conduct for your own project please contact us at conduct@openrobotics.org. We will require the contact information for at least one administrator for your project.

Outside organizations that have adopted this code of conduct include the following organizations.

- None at this time.

Code of Conduct

The ROS community is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences great successes and continued growth. When you're working with members of the community, this Code of Conduct will help steer your interactions and keep ROS a positive, successful, and growing community.

Our Community

Members of the ROS community are open, considerate, and respectful. Behaviors that reinforce these values contribute to a positive environment, and include:

- *Being open*. Members of the community are open to collaboration, whether it's on REPs, patches, problems, or otherwise.
- *Focusing on what is best for the community*. We're respectful of the processes set forth in the community, and we work within them.
- *Acknowledging time and effort*. We're respectful of the volunteer efforts that permeate the ROS community. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community.
- *Being respectful of differing viewpoints and experiences*. We're receptive to constructive comments and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts.

- *Showing empathy towards other community members.* We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.
- *Being considerate.* Members of the community are considerate of their peers – other ROS users.
- *Being respectful.* We're respectful of others, their positions, their skills, their commitments, and their efforts.
- *Gracefully accepting constructive criticism.* When we disagree, we are courteous in raising our issues.
- *Using welcoming and inclusive language.* We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.

Our Standards

Every member of our community has the right to have their identity respected. The ROS community is dedicated to providing a positive experience for everyone, regardless of age, gender identity and expression, sexual orientation, disability, physical appearance, body size, ethnicity, nationality, race, or religion (or lack thereof), education, or socio-economic status.

Inappropriate Behavior

Examples of unacceptable behavior by participants include:

- Harassment of any participants in any form.
- Deliberate intimidation, stalking, or following.
- Logging or taking screenshots of online activity for harassment purposes.
- Publishing others' private information, such as a physical or electronic address, without explicit permission.
- Violent threats or language directed against another person.
- Incitement of violence or harassment towards any individual, including encouraging a person to commit suicide or to engage in self-harm.
- Creating additional online accounts in order to harass another person or circumvent a ban.
- Sexual language and imagery in online communities or in any conference venue, including talks.
- Insults, put downs, or jokes that are based upon stereotypes, that are exclusionary, or that hold others up for ridicule.
- Excessive swearing.
- Unwelcome sexual attention or advances.
- Unwelcome physical contact, including simulated physical contact (eg, textual descriptions like "hug" or "back-rub") without consent or after a request to stop.
- Pattern of inappropriate social contact, such as requesting/assuming inappropriate levels of intimacy with others.
- Sustained disruption of online community discussions, in-person presentations, or other in-person events.
- Continued one-on-one communication after requests to cease.
- Other conduct that is inappropriate for a professional audience including people of many different backgrounds.

Community members asked to stop any inappropriate behavior are expected to comply immediately.

Weapons Policy

No weapons are allowed at ROS physical events. Weapons include but are not limited to explosives (including fireworks), guns, and large knives such as those used for hunting or display, as well as any other item used for the purpose of causing injury or harm to others. Anyone seen in possession of one of these items will be asked to leave immediately, and will only be allowed to return without the weapon.

Enforcement Responsibilities

The Code of Conduct Team is responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

The Code of Conduct Team has the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

Current Code of Conduct Team

The Code of Conduct Team consists of three volunteers from the ROS community. Optimally, these members are located at multiple locations across the globe to provide for the timely response to conduct questions and violations, and provide native language support as best we can. The Conduct Team members serve a two year term with replacements nominated by the ROS 2 Technical Steering Committee. The Conduct Team works to adjudicate conduct violations using a consensus model. For most situations, that is those that don't require an immediate response, the Conduct Team will issue reports and enforcement actions representing the consensus of the team.

The current code of conduct team consists of:

- Person A + Contact Info
- Person B + Contact Info
- Person C + Contact Info

The entire team can be contacted using conduct@openrobotics.org. The team can arrange for other means of communications after the initial contact. We recommend you use the conduct@openrobotics.org address unless you wish to report a Conduct Team member, or you feel uncomfortable communicating with a certain team member.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the Conduct Team responsible for enforcement at conduct@openrobotics.org. All complaints will be reviewed and responded to within 48 hours. The Conduct Team is obligated to respect the privacy and security of the reporter of any incident.

Enforcement Guidelines

The Conduct Team will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

Correction

Example Behavior: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

Warning

Example Behavior: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

Temporary Ban

Example Behavior: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

Permanent Ban

Example Behavior: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

How To Report A Conduct Violation

If you believe someone is in physical danger, including from themselves, the most important thing is to get that person help. Please contact the appropriate crisis number, non-emergency number, or police number as appropriate. If you are at a ROS event, you can consult with a volunteer or staff member to help find an appropriate number.

If you believe someone has violated the ROS Code of Conduct, we encourage you to report it. If you are unsure whether the incident is a violation, or whether the space where it happened is covered by the Code of Conduct, we encourage you to still report it. We are fine with receiving reports where we decide to take no action for the sake of creating a safer space.

The ROS related forums and events listed above should have a designated moderator or Code of Conduct point of contact. Larger gatherings, like conferences, may have several people to contact. Specific information should be available for each listed gathering, online or off.

Report Template

When you make a report via email or phone, please provide as much information as possible to help us make a fair and accurate decision about the appropriate response. The following template should serve as a guide for reporting.

- Your contact info (so we can get in touch with you if we need to follow up)
- Date and time of the incident
- Any links, screen shots, videos, or other media that may help
- Location of the incident
- Whether the incident is ongoing
- Description of the incident
- Identifying information of the reported person
- Additional circumstances surrounding the incident
- Other people involved in or witnesses to the incident and their contact information or description

Confidentiality

All reports will be kept confidential. When we discuss incidents with people who are reported, we will anonymize details as much as we can to protect reporter privacy.

However, some incidents happen in one-on-one interactions, and even if the details are anonymized, the reported person may be able to guess who made the report. If you have concerns about retaliation or your personal safety, please note those in your report. In some cases, we can compile several anonymized reports into a pattern of behavior, and take action on that pattern.

In some cases we may determine that a public statement will need to be made. If that's the case, the identities of all victims and reporters will remain confidential unless those individuals instruct us otherwise.

Report Handling Procedure

When you make a report to the Conduct Team, they will gather information about the incident. If the incident is ongoing and needs to be immediately addressed, any Conduct Team member may take appropriate action to ensure the safety of everyone involved. If the situation requires it, this may take the form of a referral to an appropriate agency, including the local police. The Conduct Team is not equipped to handle emergency situations.

If the incident is less urgent, the report will be discussed by the Conduct Team to determine an appropriate response. You should receive a response from the Conduct Team within 48 hours confirming the receipt of the report, and potentially asking for follow up information. Within one week of an incident report, a member of the Conduct Team will follow up with the person who made the report. The follow up may include:

- An acknowledgment that the Code of Conduct responders discussed the situation
- Whether or not the report was determined to be a violation of the Code of Conduct
- What actions (if any) were taken to correcting the reporter behavior

1.1.12 Developer Guidelines

Table of Contents

Workspace

Package description

The following packages are provided:

- **Core packages:**
  - maliput
  - maliput_py
  - maliput_object
  - maliput_object_py
  - maliput_sparse
  - maliput_viz
  - ament_cmake_doxygen
  - maliput_documentation
- **Backend implementations:**

    - maliput_dragway
    - maliput_malidrive
    - maliput_multilane
    - maliput_osm
- **Integration packages:**
    - maliput_integration
    - maliput_integration_tests
- **Mathematical support:**
    - maliput_drake
- **Utility packages:**
    - drake_vendor
- **Delphyne:**
    - delphyne
    - delphyne_gui
    - delphyne_demos

Each repository may contain one or more packages. When a repository contains two or more packages, it is expected that all packages' versions within the same repository get updated together. Git will tag them all simultaneously.

Continuous integration

Each repository in the workspace contains a few jobs that run under the following circumstances:

- On a push event: the last commit of branch compiles and runs the package tests with `gcc` and `ld`.
- On a pull request event:
    - **`gcc` build and tests will be executed:**
      For this workflow, action-ros-ci is used, in an attempt to be as standard as possible with ros packages. This action supports having interdependent pull request, see here. This may be useful when your PR depends on PRs/MRs/branches from other repos for it to work or be properly tested.
    - `clang` build, sanitizers and static analyzer will be triggered using specific labels in the pull request:
        * `do-clang-test` executes `clang` build and test, `asan`, `ubsan` and `tsan` (when enabled) build and test.
        * `do-static-analyzer-test` executes `scan-build` in the project.
- Scheduled job: there are two types of scheduled jobs. One that runs every night and executes a full build and test of the entire workspace with `gcc`. Another weekly event runs also the sanitizers and static analyzer. Every night, a tarball with a full build is generated as a way to have snapshots of the workspace binaries.

See *Compiler support* for more details.

Coding standards

Languages

C++ programming

The workspace uses C++ 17 and each package should compile with `gcc` and `clang` (see *Compiler support* for more details).

The workspace follows Drake style guide which is a derivative of Google style guide.

Exceptions:

- Line length: is modified to 120 columns to better fit the modern and wider screens.

Python programming

Code is formatted following PEP8.

CMake programming

CMake is the underlying build system tool and language, it is treated as a first class citizen like C++ and Python. For that reason, the following general conventions must be followed on top of ROS2 CMake conventions.

Libraries

- All libraries should be `SHARED` libraries. Consider using `set(BUILD_SHARED_LIBS true)` in your top level CMakeLists.txt file.
- Libraries should not include in their target name the project name unless they are the main library in the package. We don't expect to have as target names `maliput_foo` for the `foo` functionality.
- Use namespaces the following way: `project_name\:\:library_name` as follows:

```
add_library(maliput::foo ALIAS foo)
```

- Use _ instead of - in compound names.
- Include in the binary name the project name:

```
set_target_properties(foo
  PROPERTIES
    OUTPUT_NAME maliput_foo
)
```

- General `install()` commands are expected as follows:

```
install(
  TARGETS foo
  EXPORT ${PROJECT_NAME}-targets
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)
```

- Use `ament_export_libraries(my_custom_library)`.
- Consider using the generation expressions for target_include_directories within the project:

```
target_include_directories(foo
  PUBLIC
    $<BUILD_INTERFACE:${PROJECT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)
```

- Header file only libraries should be created as interfaces and header files must be placed in the include directory at the top level of the package. Make sure to install those header files later on. The target name is superfluous because those files will be discoverable by the consuming target if paths are properly set. However, the decision is to add another layer of security at the target level.
- When using the maliput plugin architecture system, if shared library and executable are compiled using *ubsan `(undefined behavior sanitizer) the property `ENABLE_EXPORTS* should be enabled on the executable target in order to instruct the linker to add all symbols to the dynamic symbol table. For further information see next reference link.

```
set_target_properties(foo
  PROPERTIES
    ENABLE_EXPORTS ON
)
```

Executables

- Use _ instead of - in compound names.
- `install()` commands are expected as follows:

```
install(foo
  EXPORT ${PROJECT_NAME}-targets
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin
)
```

Resources

- Define a project resources path and install resources following your structure within `share/project_name/resources` folder in the install space.

Testing

- 100% coverage of the public API of any entity must be unit-tested.
- Complex pieces of code that are not exposed should be considered to be re-engineered in favor of increased coverage.
- Integration test between modules can be done when appropriate.
- Consider using `maliput_integration_tests` for complex integration tests.
- `gtest` and `gmock` via ament_cmake packages are the default testing frameworks for C++.
- `python3-pytest` via ament_cmake packages is the default testing frameworks for Python.

Linting

`ament_clang_format` alone cannot be used because we have a custom format. So packages hold a tools folder at the root level in which a script called `reformat_code.sh` calls the previous tool with the custom package.

For Python code, make sure to use ament cmake flake8. To do so, you should follow the instructions here and use one of the `.flake8` files in your package root directory to tell the linter which are the tests you want to perform. In particular, we edit it so it has the following extras:

```
# Set the maximum length that any line (with some exceptions) may be.
max-line-length = 100
# Set the maximum allowed McCabe complexity value for a block of code.
max-complexity = 10
# Toggle whether pycodestyle should enforce matching the indentation of the opening␣
↪bracket's line.
# incluences E131 and E133
hang-closing = True
# Specify a list of codes to ignore.
ignore =
    E133,
    E226,
# Specify the list of error codes you wish Flake8 to report.
select =
  E,
  W,
  F,
  C
```

Supported OSs and environments

The workspace is only maintained on Ubuntu 18.04 and ROS2 Dashing.

Compiler support

The workspace is built with Ubuntu's default `gcc` (version 7.5) and `ld` (version 2.30) and `clang` and `llvm` tools (version 8).

- Address sanitizer
- Undefined behavior sanitizer.
- Thread sanitizer.
- Static analyzer (scan-build): it runs with `clang`.

Building the documentation

maliput_documentation package is in charge of concentrating the documentation of the entire maliput ecosystem.

The page is built upon Sphinx framework, while the docstring's code is converted to *html* by Doxygen.

The documentation is finally served via Read the Docs.

In order to build the documentation, the cmake flag *-DBUILD_DOCS=On* should be added:

```
colcon build --packages-up-to maliput_documentation --cmake-args "-DBUILD_DOCS=On"
```

1.1.13 Developer Setup

Table of Contents

Introduction

This section explains how to build a workspace for `maliput` development. You'll learn how to work with `maliput` and with `delphyne` repositories in a multi-repository workspace.

Workspaces

Supported platforms

- Docker containerized workspaces (**recommended**): A docker image is provided in order to show the steps needed to set up the environment in a containerized workspace. When setting up docker, do *not* add yourself to the "docker" group since that represents a security risk (it is equivalent to password-less sudo). As a workaround, the instructions below use `sudo` for building the image and running the container.
- Non-containerized workspaces: Ubuntu Focal Fossa 20.04 LTS only.

Prerequisites

- To get all the necessary tools, clone `maliput_infrastructure` locally.

```
git clone git@github.com:maliput/maliput_infrastructure.git
```

- To pull private repositories, the current user default SSH keys will be used (and thus assumed as both necessary and sufficient for the purpose).
- Containerized workspaces require having docker engine installed in host machine. Also, you can use `nvidia-docker2`. Follow their instructions if you want to install it.

Workspace creation

The following assumes that you want to create a workspace containing all of Maliput's repositories, with a focus on Maliput and Malidrive development. As such, it uses a workspace directory named `maliput_ws` and pulls sources from foxy/maliput.repos.

The instructions also cover how to optionally add `delphyne` repositories into your workspace. These repositories include a simulator and visualizer useful when working with Maliput road networks.

The instructions below first cover how to create a non-containerized workspace, followed by a containerized workspace. **Note that using a containerized workspace is recommended**.

Option 1: Create a non-containerized workspace

The following creates a non-containerized workspace. Follow it if you are willing to install Maliput and its dependencies directly in your base operating system. Bear in mind that using a non-containerized workspace makes reproducing and troubleshooting issues harder for others. If in doubt, install your workspace within a container by following the instructions in the subsequent "Option 2" section.

Create the workspace folder

```
mkdir maliput_ws
```

Note: These instructions assume `maliput_ws` is at the same level as `maliput_infrastructure` and `repos_index`.

Copy .repos file

copy `maliput_infrastructure/repos_index/foxy/maliput.repos` into `maliput_ws`. It will be used to add the Maliput-related repositories to the workspace.

```
cp maliput_infrastructure/repos_index/foxy/maliput.repos maliput_ws/
```

Note: You can optionally add Delphyne-related repositories to your workspace:

```
cp maliput_infrastructure/repos_index/foxy/delphyne.repos maliput_ws/
```

Install dependencies

```
sudo ./maliput_infrastructure/tools/install_dependencies.sh
```

Update all the repositories in your workspace

Bring all the repositories listed in `maliput.repos` file:

```
cd maliput_ws
mkdir src
vcs import src < maliput.repos  # clone and/or checkout
# Optionally, run:
# vcs import src < delphyne.repos
vcs pull src  # fetch and merge (usually fast-forward)
```

This will clone repositories and/or checkout branches, tags or commits as necessary, followed by fetching and (likely) fast-forward merging to get branches up to date with their upstream counterpart. No merging takes place when a repository is at a given tag or commit. Note that you can continue to bring other repositories into your workspace by repeating the `import` and `pull` operation using additional `.repos` files.

Install all packages' dependencies

First update the `ROS_DISTRO` environment variable with your `ros2` version, e.g.:

```
export ROS_DISTRO=foxy
```

Install dependencies via `rosdep`

```
rosdep update
rosdep install -i -y --rosdistro $ROS_DISTRO --skip-keys "pybind11" --from-paths src
```

Warning: Package dependencies are installed system wide. `rosdep` does not provide any support to remove the dependencies it brings. In this regard, disposable containerized workspaces help keep development environments clean (as system wide installations within a container are limited to that container).

Install drake

Installing drake is **only** necessary when working with `delphyne.repos`, otherwise it will fail because `drake_vendor` is not in the workspace.

```
sudo ./src/drake_vendor/drake_installer
```

Source ROS environment

```
source /opt/ros/$ROS_DISTRO/setup.bash
```

Option 2: Create a containerized workspace

The following creates a containerized workspace. Maliput and its dependencies remain in the container and do not impact your operating system. Likewise, packages installed on your operating system do not impact the container. The uniformity of the container's environment makes it easier for other developers to reproduce and resolve problems you may encounter.

Configuring a containerized workspace is similar to that of a non-containerized workspace. When the steps are identical, links to the non-containerized setup instructions are used. Machinery is provided to `build` and `run` a docker image and container for Maliput workspace development:

Build the docker image

```
./maliput_infrastructure/docker/build.sh
```

If you are using nvidia-docker2 add the `--nvidia` option.

```
./maliput_infrastructure/docker/build.sh --nvidia
```

Note: `build.sh --help` for more options:

1. `-i --image_name` Name of the image to be built (default maliput_ws_ubuntu_focal).
2. `-w --workspace_name` Name of the workspace folder (default maliput_ws).

Create the workspace folder

Copy .repos file

Run the container

```
./maliput_infrastructure/docker/run.sh
```

If you are using nvidia-docker2 add the `--nvidia` option.

```
./maliput_infrastructure/docker/run.sh --nvidia
```

Note: `run.sh --help` for more options:

1. `-i --image_name` Name of the image to be run (default maliput_ws_ubuntu_focal).
2. `-c --container_name` Name of the container (default maliput_ws_focal).
3. `-w --workspace` Relative or absolute path to the workspace you want to bind (default to location of maliput_infrastructure folder).

Install dependencies

During docker build stage a script is copied into the container at `/home/$USER/`.

```
sudo ./../install_dependencies.sh
```

Update all the repositories in your workspace

Install all packages' dependencies

Install drake

Source ROS environment

Staging changes in your container

Once you finish your setup and tried the workspace, you might want to stage it. You can achieve that by `exit`-ing the container and accepting to commit the changes.

```
user@a3b6a70d7b7d:~/maliput_ws$ exit
exit
access control enabled, only authorized clients can connect
Do you want to overwrite the image called 'maliput_ws_ubuntu' with the current changes?␣
↪[y/n]: y
Overwriting docker image...
[sudo] password for user:
sha256:9fdf391051f702f6b3fcd9c7ab258e5e014361bf18918b86155db3acda355147
```

Check your workspace

Workspace state as a whole encompasses both current local repositories' state plus the state of the filesystem that hosts it. However, if a workspace is containerized and not customized, repositories alone carry the source code and state the list of system dependencies necessary to build and execute. And we can easily inspect repositories.

1. To check repositories' status, run:

```
vcs status src
```

1. To see changes in the repositories' working tree, run:

```
vcs diff src
```

1. To see if (most of) our versioned packages' dependencies have been met, run:

```
rosdep check --rosdistro $ROS_DISTRO --skip-keys "pybind11" --from-paths src
```

Note: not all workspace prerequisites are handled using `rosdep` meaning `rosdep check` may fall short. For example, pure binary dependencies like `drake`'s binary tarball is not handled by `rosdep`. Another example is `apt` source lists.

In any given case, one can always resort to the specific tool used for repository versioning (e.g. `git`) if `vcs` isn't enough or to the specific package managers (e.g. `apt` or `pip`) if `rosdep` isn't enough.

Build your workspace

Build

Change the directory to `maliput_ws`:

```
cd ~/maliput_ws
```

To build all packages:

```
colcon build
```

To build some packages, use `--packages-up-to`. For example, to build `maliput` and `maliput_malidrive`:

```
colcon build --packages-up-to maliput maliput_malidrive
```

To build some packages and only those packages (i.e. without their dependencies), use `--packages-select`:

```
colcon build --packages-select maliput maliput_malidrive
```

Note that if dependencies cannot be met, regardless of whether it's because they are not installed or not built, the build will fail. Thus, this flag is usually helpful only to quickly rebuild a package after building it along with its dependencies.

Note: If you are building `drake` from source as well, make sure `--cmake-args -DWITH_PYTHON_VERSION=3` is passed to `colcon`. Otherwise, python packages and scripts in `delphyne` and `delphyne_gui` packages won't find `pydrake`.

Note: To build with debug symbols, and given that we use CMake packages only, just make sure that `CMAKE_BUILD_TYPE=Debug`. You can force it by passing `--cmake-args -DCMAKE_BUILD_TYPE=Debug` to `colcon`.

Note: If you want to build with `clang-8`, run the following:

```
LDFLAGS="-fuse-ld=lld-8" CC=clang-8 CXX=clang++-8 colcon build --packages-up-to maliput␣
↪maliput_malidrive
```

Source the workspace

```
source install/setup.bash
```

Note: If `delphyne` is available, run `delphyne-gazoo` and `delphyne-mali` to see if everything is working.

Note: See colcon build documentation for further reference on `build` support.

Test your workspace

In a built workspace, run:

```
colcon test --event-handlers=console_direct+ --return-code-on-test-failure --packages-
↪skip pybind11
```

Note: See colcon test documentation for further reference on `test` support.

Build your workspace using Static Analyzer

To verify your code, run the Clang Static Analyzer. A useful script called `run_scan_build` is located in `.github` in every repository.

The script will forward arguments to `colcon build` so you can use Colcon's CLI machinery to choose which packages to evaluate.

To run `scan-build` on all packages in the workspace:

```
./src/maliput/.github/run_scan_build
```

To run scan-build up to `maliput_malidrive`:

```
./src/maliput/.github/run_scan_build --packages-up-to maliput_malidrive
```

Build doxygen documentation

Build the workspace. In particular, we are interested in compiling `dsim_docs_bundler`.

```
cd ~/maliput_ws
colcon build --packages-up-to dsim_docs_bundler
```

Open the documentation with your favorite browser. If Google Chrome is available, you can run:

```
google-chrome install/dsim-docs-bundler/share/dsim-docs-bundler/doc/dsim-docs/html/index.
↪html
```

Delete your workspace

Containerized workspace could be deleted simply deleting the docker image:

```
docker rmi maliput_ws_ubuntu_focal
```

Consider replacing `maliput_ws_ubuntu_focal` by your image name when using a custom one.

Contributing

Usual workflow

Ours is similar to ROS2's development workflow, and thus many of their tools and practices apply equally.

Workspaces are managed via vcs , a tool that helps in dealing with sources distributed across multiple repositories, not necessarily versioned with the same tool (support for `git`, `hg`, `svn` and `bazaar` is readily available). `vcs` uses `.repos` files for a listing of version pinned sources.

Dependency management is taken care of by rosdep, a tool that can crawl `package.xml` files and resolve dependencies into a call to the appropriate package manager for the current platform by means of a public database known as rosdistro.

To build and test packages, colcon abstracts away the details of the specific build system and testing tools in use and arbitrates these operations to take place in topological order. Operations will be run in parallel by default.

Note: In all three cases above, the tools delegate the actual work to the right tool for each package and focus instead on bridging the gap between them. Thus, for instance, `colcon` builds interdependent CMake packages by running `cmake` and `make` in the right order and setting up the environment for the artifacts to be available. Same applies for `vcs` and `rosdep`.

Note: These tools do not strive to act like a proxy for every configuration setting or command line option that underlying tools they delegate work to may have. Thus, it may be necessary to configure the underlying tool in addition to the configuration for these tools to attain a desired behavior. For instance, limiting `colcon` parallelism with the `--parallel-workers` switch has no impact on `make` parallelization settings if this tool is being used.

Using binary underlays

In ROS 2 workspace parlance, an overlay workspace is a workspace that builds on top of another, previously built workspace i.e. the underlay workspace. A binary underlay is thus the install space of a pre-built workspace, that packages in downstream workspaces can use to meet their dependencies. As a result, the amount of code that needs to be compiled when building downstream workspaces gets reduced, enabling faster builds. You may refer to colcon documentation and tutorials for further details.

Several binary underlays are available for download and installation:

- `dsim-desktop-YYYYMMDD-focal-tar.gz`

  Built nightly, targeting Ubuntu Focal 20.04 LTS. Contains all known packages in all our repositories as of the specified date (DD/MM/YYYY). To be found at `s3://driving-sim/projects/maliput/packages/nightlies/`.

- `dsim-desktop-latest-focal.tar.gz`

  Built nightly, targeting Ubuntu Focal 20.04 LTS. Contains the most recent versions of all packages known in all our repositories. To be found at `s3://driving-sim/projects/maliput/packages/nightlies/`.

In the following, it is assumed that you want to use a full `dsim-desktop` underlay for working on a downstream package of your own. As such, it suggests the installation of a `dsim-desktop` binary underlay, that brings all known packages in all our repositories. You should choose an underlay that is appropriate for your intended purpose.

Download the binary underlay tarball of choice from dsim's S3 bucket

```
aws s3 cp s3://driving-sim/projects/maliput/packages/nightlies/dsim-desktop-latest-focal.
↪tar.gz \
    /path/to/workspace/dsim-desktop-latest-focal.tar.gz
```

It is assumed that you have the right AWS credentials configured in your system. See AWS CLI user guide to configuration for further reference.

Extract binary underlay tarball

```
sudo mkdir -p /opt/dsim-desktop
sudo tar -zxvf dsim-desktop-latest-focal.tar.gz -C /opt/dsim-desktop --strip 1
```

Install prerequisites

```
echo "deb http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | \
    sudo tee --append /etc/apt/sources.list.d/ros2-latest.list

sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys␣
↪C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

sudo apt update
sudo apt install -y python3-rosdep
sudo rosdep init
```

Install all underlay packages' dependencies

For foxy

```
export ROS_DISTRO=foxy
rosdep update
rosdep install -i -y --rosdistro $ROS_DISTRO --skip-keys "pybind11" --from-paths /opt/
↪dsim-desktop/*
```

Install drake

```
cd /opt/dsim-desktop
./drake_vendor/bin/drake_installer -f drake_vendor/share/VERSION.TXT
```

From then on, before building the workspace, you must source the underlay as follows:

```
source /opt/dsim-desktop/setup.bash
```

Note: Having an underlay around does not make it a requirement for all workspace builds, but only for those that rely on that underlay to get their dependencies met.

How to use CI

CI jobs build and test relevant packages for each repository on every PR. Being a multi-repository project, patches that are not limited to a single repository must be separately PR'd but built and tested together. To that end, make sure that all PR'd branches that are part of the same patch have the same name e.g. `my_github_user/my_patch_name`.

Warning: Fork based development is currently not supported. All PRs must come from origin and not a fork.

Troubleshooting

Issue Forensics

When reproducing issues, either related to the codebase or to the infrastructure that supports it, recreating the environment in which these issues arose is crucial.

1.1.14 Releases

Table of Contents

Versioning

`maliput` packages will adhere to semantic versioning and will follow the ROS2 Versioning guidelines .

IMPORTANT: `maliput` is still under actively development for so semantic versioning is not yet strictly followed for the *1.X.Y* versions.

Branches and tags for releases

The following branches and tags schemes will be used:

- Use `main` as the mainline development branch. The tip of that branch will be the latest development state. It is not safe. Downstream projects are encouraged to avoid using it unless there is a business need to do so.
- Each repository will have branches with the following pattern: `release/major.minor.x`, e.g. `release/1.2.x`. Each patch release `(x)` will contain one or more additional bug fix commits relative to the previous patch release `(x - 1)`.

Releases

Named major releases

Significant releases of Maliput packages will be named. The names will be chosen based on famous roads and will be alphabetically sorted. Significant releases will be created on demand.

Major release output

Every new major release will provide:

- Updated `maliput_rolling.repos` file when appropriate (new major release, update to latest minor release or patch release).
- New `maliput_<name>.repos` file.
- Binaries distributed via ROS Build Farm.

How to release?

There are different steps to follow based on the type of release you want to create.

Make a new major `maliput` workspace release

- Choose a name that is next in the alphabet relative to the previous major release.
- Prepare the workspace by pinning all dependencies and downstream packages to their target branches or tags.
- Build and test the workspace with all packages pointing to their pinned versions.
- Update `maliput_rolling.repos` file in repos_index under the appropriate ROS2 distro folder.
- Create a new `maliput_<name>.repos` file in repos_index under the appropriate ROS2 distro folder.
- Rely on bloom tools for the releasing process.

Create a new package major release

- Prepare the workspace by pinning all dependencies and downstream packages to their target branches or tags.
- Prepare the release branch:
  - Update the `CHANGELOG.rst` and `package.xml` files via a PR targeting `main`.
  - From `main` branch, create a new branch called `release/major.minor.x`. `x` is not a placeholder, it is the literal **x** because this branch will contain all the potential future patch releases in the series of `major.minor.0`, `major.minor.1` and so on (e.g. *1.3.0*, *1.3.1*, *1.3.2*, etc.). A tag will be used to name the specific commit in the branch.
  - Run **all** tests. If you encounter any problem, send PRs to fix them targeting `main` branch. Merge those commits into `release/major.minor.x`.
- Push the branch.
- Make a tag with the appropriate version number: `release/major.minor.0`.
- Push the tag.
- Create a PR to repos_index and update `maliput_rolling.repos` to indicate the branch name `release/major.minor.x` as the latest package version.
- Rely on bloom tools for the releasing process.

Create a new package minor release

- Prepare the workspace by pinning all dependencies and downstream packages to their target branches or tags.
- Prepare the release branch:
  - From the tip of `release/major.[minor - 1].x`, create a new branch called `release/major.minor.x`.
  - Push the branch `release/major.minor.x`.
  - Cherry-pick commits as needed from `main` and include them into `release/major.minor.x` via PRs. Alternatively, create feature branches whose PRs target `release/major.minor.x`.
  - Update the `CHANGELOG.rst` and `package.xml` via a PR targeting `release/major.minor.x`.
  - Run **all** tests. If you encounter any problem, send PRs to fix them targeting `release/major.minor.x` branch.
- Make a tag with the appropriate version number: `release/major.minor.0`.
- Push the tag.
- When the `major` and `minor` version numbers are the greatest: create a PR to repos_index and update `maliput_rolling.repos` to indicate the branch name `release/major.minor.x` as the latest package version.
- Consider updating the affected named `maliput` workspace releases.
- Rely on bloom tools for the releasing process.

Create a new package patch release

- Prepare the workspace by pinning all dependencies and downstream packages to their target branches or tags.
- Prepare the release branch:
  - Cherry-pick commits as needed from `main` and include them into `release/major.minor.x` via PRs. Alternatively, create feature branches whose PRs target `release/major.minor.x`.
  - Update the `CHANGELOG.rst` and `package.xml` via a PR targeting `release/major.minor.x`.
  - Run **all** tests. If you encounter any problem, send PRs to fix them targeting `release/major.minor.x` branch.
- Make a tag with the appropriate version number: `release/major.minor.patch`.
- Push the tag.
- Consider updating the affected named `maliput` workspace release.
- Rely on bloom tools for the releasing process.