noCollide

Nico Päller, Manuel Wilke

Feb 14, 2021

CONTENTS:

The noCollide is a driver warning system to avoid collisions with objects. It can be implemented using LiDAR Sensors and a Raspberry Pi or the CARLA Simulator.

CHAPTER

ONE

NOCOLLIDE

The NoCollide class is the brain of the system. Here all the sensor information is collected and used to calculate the Time-to-Collision

class lib.nocollide.**TtcTimes**(*too_late: float*, *brake: float*, *warning: float*)

A struct like class to enhance readability when storing Time-to-Collision values. `too_late < brake < warning` must be True

Parameters

- **too_late** (*float*) – The time after which an accident is unavoidable
- **brake** (*float*) – The time after which the driver must be braking to avoid an accident
- **warning** (*float*) – The time after which the driver should be warned

class lib.nocollide.**NoCollide**(*driver:* lib.driver.Driver, *sensors:* lib.sensor.SensorGroup, *ttc_times: Optional[*lib.nocollide.TtcTimes*] = None*)

The class that calculates the acc and regulates the warning.

Parameters

- **driver** (Driver) – The configuration of the CAN-Bus to initialise
- **sensors** (*SensorGroup*) – The sensor group, on which the calculation should be done
- **ttc_times** (Union[*TtcTimes*, None]) – The times which define when to warn based on the TTC (Time-to-collision). If None, the default TTC-Times will be used: `TtcTimes(too_late=0.6, brake=1.6, warning=3.0)`

calc()

The method that calculates the Time-to-Collision. Takes the Value of the sensor, that measures the closest object and calculates the relative velocity. The TTC results in dividing the distance by the relative velocity :return:

run(*block: bool = True*)

The method to start the calculation. Can be run blocking (in an endless loop) or not (1 calulcation only)

Parameters **block** (*bool*) – wether the method should block or not

warn(*ttc: float*)

The method to warn the user based on the Time-to-Collision.

Parameters **ttc** (*float*) – the Time-to-Collision

CHAPTER

TWO

SENSOR

The Sensors are the eyes and ears of the system. Here is a LiDAR Sensor implemented to be used for measuring the distance to the object in the front.

class lib.sensor.**SensorInterface**

An interface to implement a sensor with all the needed methods to function properly

abstract change_addr(*new_addr: int*)

A method to change the i2c address to be used with multiple devices on one bus

Parameters **new_addr** (*int*) – the new address that should be set

abstract close()

A method to close the sensor/bus

abstract configure(*mode: int*)

A method to apply any configuration to the sensor

Parameters **mode** (*int*) – the mode that should be set

abstract measure(*rec_bias_corr: bool = True*)

Method to tell the sensor to measure a value

Parameters **rec_bias_corr** (*bool*) – wether to measure with bias correction

abstract read_measurements() → float

Method to retrieve the measured data from the sensor

Returns the measured data

Return type float

class lib.sensor.**Sensor**(*i2c_bus: int = 1*, *max_range: int = 50*)

The Class that handles a LiDAR Sensor

Param i2c_bus: the bus number on which the sensor is running, defaults to Bus 1

change_addr(*new_addr: int*)

Method to change the I2C Address of the sensor

Parameters **new_addr** (*int*) – the new address that should be set

close()

Method to close the bus

configure(*mode: int = 0*)

Method to initialize the sensor to different modi. Must be done before the sensor can be used

configuration: Defaults to **0**.

0: Default mode, balanced performance
1: Short range, high speed. Uses `0x1d` maximum acquisition count
2: Default range, higher speed short range. Turns on quick termination detection for faster measurements at short range (with decreased accuracy)
3: Maximum range. Uses `0xff` maximum acquisition count
4: High sensitivity detection. Overrides default valid measurement detection algorithm, and uses a threshold value for high sensitivity and noise
5: Low sensitivity detection. Overrides default valid measurement detection algorithm, and uses a threshold value for low sensitivity and noise

Parameters **mode** (*int*) – the selected mode

measure(*rec_bias_corr: bool = True*)
Method to tell the sensor to take a measurement

Parameters **rec_bias_corr** (*bool*) – Wether the measurement should be taken with or without receiver bias correction; defaults to True

read_measurements() → float
Method to obtain the measured distance in cm

Returns the measured value in cm. Returns 0 if timeouted

Return type int

class lib.sensor.**SensorGroup**(*i2c_bus: int*, *sensors: Optional[*lib.sensor.SensorInterface*] = None*, *sensor_names: Optional[List[str]] = None*, *init_mode: int = 0*)
Class to connect to multiple sensors at once.

This class can be used in a context manager (`with` statement). It returns itself and then all *Sensors* which are being set (default: 3).

If not used in a `with` -Statement the bus must be closed manually using the *close()* method

```
with SensorGroup(i2c_bus=1) as (group, *sensors):
    ...
```

Parameters

- **i2c_bus** (*int*) – the raspberry pi bus the sensors are running on
- **sensor_names** (*Union[List[str], None]*) – a list of names, to identify the sensors. Defaults to `["left", "center", "right"]` if `None`
- **init_mode** (*int*) – the mode the sensors should be initialised with. Defaults to mode 0. See *Sensor.configure()* method

close()
Method do close and delete all sensors from the group. This method is also called when exiting the `with`-Statement

set_mode(*mode_num: int*, *sensors: Optional[List[str]] = None*)
Method to set the mode for specific or all Sensors in the group

Parameters

- **mode_num** – the mode number, referring to the mode if the *configure()* method of the *Sensor*

- **sensors** – the sensornames of which the mode should be changed

Raises **TypeError** – TypeError when the is no such sensor in the group

start()

Method to start the measurements of the sensors. Launches a Thread for each sensor, where it measures continously in a loop

CHAPTER

THREE

DRIVER

The driver is the interface between the assistant system and the vehicle itself. The Driver will be used to retrieve the current speed and set parameters e.g. the throttle.

class lib.driver.**Driver**
An interface to be used by the *noCollide* class get and set parameters regarding the vehicle

abstract get_speed() → *lib.data.Speed*
A method to retrieve the current speed of the car

Returns the current speed of the car

Return type *Speed*

abstract set_brake(*val*)
A method to set the brake amount of the vehicle

Parameters **val** (*float*) – the brake amount

abstract set_throttle(*val: float*)
A method to set the throttle of the vehicle

Parameters **val** (*float*) – the throttle amount

abstract warn()
A method to warn the user for possible collisions

class lib.driver.**CanConfig**
Class to better store config details of the can bus if necessary

class lib.driver.**CanBus**(*conf:* lib.driver.CanConfig, *q_size: int = 1*)

get_speed() → *lib.data.Speed*
Method to retrieve the speed from the queue

Returns the latest speed sent via CAN

Return type float

run_forever()
Method to keep in touch with the CAN-Bus. Runs in a thread, to avoid blocking

set_brake(*val*)
A method to set the brake amount of the vehicle

Parameters **val** (*float*) – the brake amount

set_throttle(*val: float*)
A method to set the throttle of the vehicle

Parameters **val** (*float*) – the throttle amount

warn()
A method to warn the user for possible collisions

CHAPTER

FOUR

DATA

To better and more easily calculate distances, and speed and to always have values and the corresponding times in one place, a custom Data class was created.

class lib.data.**Data**(*value: float*, *time: float*)

A class to store a value and its corresponding time. Calculation with +, −, ⋆, / and all types of comparisons can be applied directly to the class.

Parameters

- **value** (*float*) – the value that should be stored
- **time** (*float*) – the corresponding time to the value

class lib.data.**Speed**(*value: float*, *time: float*)

A class to extend the *Data*. This class stores velocity values and the acceleration can be easily calculated

acceleration(*other:* lib.data.Speed) → *lib.data.Data*

A method to calculate the acceleration

Parameters **other** – the Speed before with which the Acceleration should be calculated

Returns the Acceleration

Return type *Data*

class lib.data.**Distance**(*value: float*, *time: float*)

A class to extend the *Data*. This class stores distance values and the velocity can be easily calculated

velocity(*other:* lib.data.Distance) → *lib.data.Speed*

A method to calculate the velocity. Returns Speed with value 1 if the times are the same

Parameters **other** – the Distance before with which the speed should be calculated

Returns the speed

Return type *Speed*

CHAPTER

FIVE

SIMULATION INTEGRATION

To establish a neatless simulation, the code that is intended to use hardware must be extended/changed to use the simulation's sensors. That's why simulation classes are needed, to adapt to the challanges of merging different usecases in one API.

class lib.sim_interfaces.**SimNoCollide**(*hud*, **args*, ***kwargs*)

A class to use the NoCollide in the Carla-Simulator. Simply integrates the HUD of the simulator to warn via the HUD rather than to warn via some kind of Bus

Parameters

- **hud** – The HUD of the Simulation
- **args** – Arguments to be passed to the parent class
- **kwargs** – Keyword arguments to be passed to the parent class

warn(*ttc: float*)

Overwrites the parent warn Method to warn via the HUD rather than some kind of Bus :param ttc: the Time-to-Collision in seconds :type ttc: float

class lib.sim_interfaces.**SimSensor**(*sensor*, *max_range=40*)

A class to implement the Carla-Simulator object detection sensor. Due to the constant pushing nature of the Carla sensor, the last retrieved value and time must be saved to be ready when the NoCollide Algorithm requests the Value.

Parameters

- **sensor** – the Carla sensor to use
- **max_range** – the maximum range of the Carla Sensor

callback(*data*)

The callback method that is called, whenever the Carla sensor has measured new data. Simply stores the data, so taht i can be requested any time

Parameters **data** – the data measured by the carla sensor

change_addr(*addr: int*)

A method to change the i2c address to be used with multiple devices on one bus

Parameters **new_addr** (*int*) – the new address that should be set

close()

A method to close the sensor/bus

configure(*mode: int*)

A method to apply any configuration to the sensor

Parameters **mode** (*int*) – the mode that should be set

destroy()
Method to destroy the Sensor to free up memory when the simulation has ended

listen()
A Method to tell the Carla sensor to use the *callback()* method whenever a new value is measured

measure(*rec_bias_corr: bool = True*)
Method to tell the sensor to measure a value

Parameters **rec_bias_corr** (*bool*) – wether to measure with bias correction

read_measurements() → *lib.data.Distance*
The method ot retrieve the newest Data of the Sensor. If the Carla Sensor hasn't measured anything yet, a *Distance* object will be returned with the maximum range. :return: the newest Distance measured by the Carla sensor :rtype: *Distance*

stop()
A method that is needed when stopping the simulation.

class lib.sim_interfaces.**SimSensorGroup**(*i2c_bus: int, sensors: Optional[lib.sensor.SensorInterface] = None, sensor_names: Optional[List[str]] = None, init_mode: int = 0*)
Class to overwrite the *SensorGroup* class to use the simulated sensor when retrieving the distance

get_closest() → *lib.data.Distance*
Class to overwrite the get_closest() method to use the simulated sensor when retrieving the distance :return:

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

W