WS 16/17
Autonomous Driving Lab

Simulator Manual

Abstract

This document is a manual, and code documentation for certain parts of the Autonomous Driving Lab project (WS 16/17). Specifically, the covered packages of this document are the `simulator` module, the `vehicle` module and a `util` module which contains auxiallary code but also a plotter for testing and reviewing purposes. We discuss the design, the desired functionality and the implementation of our modules. Futhermore, we give a introduction how to reuse each of our modules and how to run a simulation. We provide the functionality for running a time-discrete simulation of physical objects and the corresponding physical computations for simulating a car.

For more technical information, please also use the **JavaDoc Documentation**.

Contents

Typeset Monday 27th March, 2017

1 Introduction

The Autonomous Driving Lab project (WS 16/17) is a software project, mainly written in Java, about autonmously driving cars. The project consists of several modules such as a visualization module, a controller, a module taking care of computer vision and decision making and a simulation module. We provide a subset of the functionality of the simulation module.

The core functionality of our code is a simulator, simulating physical objects in a time-descrete way. We focus on the simulation of cars. Nevertheless, our simulator is capable of simulating a variety of physical objects. Conceptionally, after creating a simulator instance, one is able to add physical objects to this instance. The simulation asks each object for updating its properties in each time step. This is usually done according to a certain initilization and certain physical rules.

1.1 System Layout

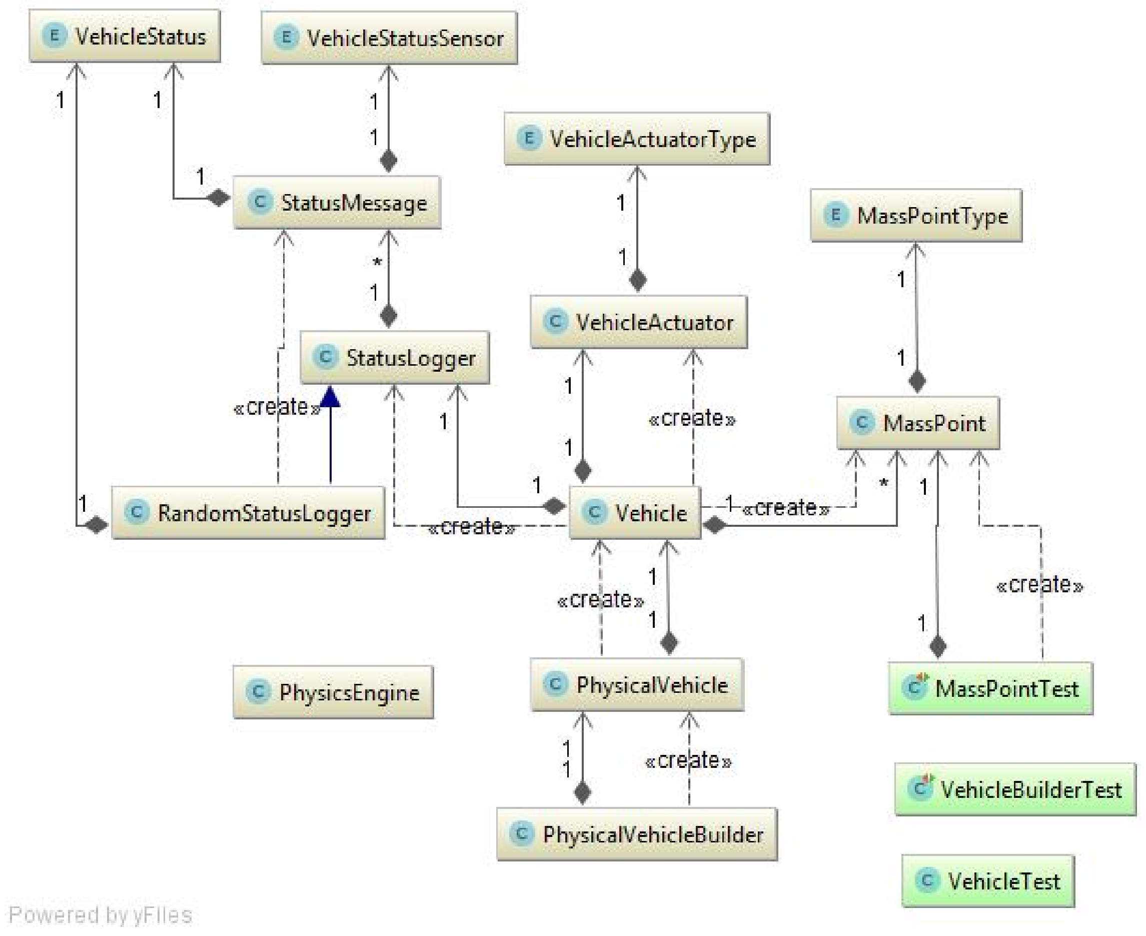

Figure 1: Class diagram of the `vehicle` module

In Figure 1 you can see the overall structure of the `vehicle` module. The core of it is the `Vehicle` class (see section 3.1), which is contained in a `PhysicalVehicle` (see section 3.4). The `Vehicle` contains multiple `MassPoint` classes for the wheels and a `StatusLogger` (3.6). Besides that, the `Vehicle` also contains one `VehicleActuator` (section 3.2) of every `VehicleActuatorType`. The `PhysicsEngine` (section 3.3) is also included in the `vehicle` module but it does not have any connections to any other classes of the module.

1.2 Module Overview

Our code is mainly located in three different submodules, in the `simulator` module, the `vehicle` module and the `util` module. The `simulator` module contains the simulation loop, provides a configurable interface and takes care of threading and synchronization. This module is discussed in detail in section 2.

The `vehicle` module contains a class to model a vehicle. Such a vechicle is setup using a builder. Furthermore, derived physical properties and a suitable computation engine is provided. The details are discussed in section 3.

The last module, the `util` module, contains auxilliary functionalities such as logging and static math functionalities. Additionally, a two dimensional plotter is included. It is utilized to verfiy and test the behavior of a simulated car. A more detailed view on the `util` module is given in section 4.

2 Simulation Module

The simulation module mainly provides the `Simulator` class that is responsible for the management of the simulation. Besides that, the `SimulationPlotter2D` provides a way to plot logs generated during the simulation.

2.1 `Simulator`

The `Simulator` class is responsible for running the simulation, managing the simulated objects and for notifying others about the progress of the simulation. The simulator uses the Singleton pattern, i.e., there is always only one shared instance of the simulator existing. During the simulation the simulator executes the *simulation loop* to update the state of simulated objects.

2.1.1 Configuring a Simulation

To access the shared instance, the `Simulator.getSharedInstance()` method is used. Configuring the simulator requires basically three decisions:

1. At which frequency the simulation should be executed, i.e., how many frames should be computed per second. This value can be set using `setSimulationLoopFrequency()`. Note that this is the optimum frequency. The actual number of frames generated may be lower depending on the hardware the simulation is executed on and the simulation mode discussed in the next step. The actual time between the last two frames can always be retrieved during the simulation using `getTimeBetweenLastIterations()`.

2. How to calculate the frames. Frames may be calculated using the modes defined in the `SimulationType` enum. Before starting the simulation, the simulation type can be set using `setSimulationType()`.

   `SIMULATION_TYPE_FIXED_TIME` guarantees the time between two frames is always the frequency set in the first step. The time between two frames does not need to match the actual time passed between the calculation of the frames. This mode produces reproducible results and should be preferred for testing.

   `SIMULATION_TYPE_REAL_TIME` computes the frames in real time, i.e., the simulation frequency should be as close as possible to the actually generated frames in one second of real time.

   `SIMULATION_TYPE_MAX_FPS` calculates as many frames as possible during the given simulation time. This mode ignores the simulation frequency. This mode should be preferred for calculating video output in real time.

3. Whether or not the simulation should be executed asynchronously in its own thread. Setting `setSynchronous- Simulation()` to `true` will cause the simulation to be executed in the thread calling `startSimulation()`. In synchronous mode, `startSimulation()` is blocking. Setting `setSynchronousSimulation()` to `true` will cause the simulation to be executed in its own thread.

2.1.2 Object Management

To add objects to the simulation, `registerSimulationObject()` is used. All objects in the simulation need to implement the `SimulationLoopExecutable` interface. During the simulation these objects will be asked to update themselves and be provided with the amount of simulated time that passed since the last update request. To remove an object from the `unregisterSimulationObject()` is used. Adding and removing objects can be done at any time – even during the execution of the simulation.

2.1.3 Sending Notifications

Similarly to adding and removing objects to the simulation, objects can be registered to be notified about certain events in the simulation such as that start of the simulation, executions of the simulation loop, or the stop of the simulation. To (un)register an object for these notifications, `registerLoopObserver()` and `unregisterLoopObserver()` are used.

2.1.4 Running a Simulation

Before starting a simulation, the duration of the simulation needs to be set. `stopAfter()` takes the desired simulation duration in milliseconds. The simulation can then be started using `startSimulation()`. Since `startSimulation()` is non-blocking in asynchronous mode, the simulator provides methods for waiting either a certain amount of simulated time (using `waitForTime()`) or until the simulation is finished (using `waitUntilSimulationFinished()`). Both of these methods block the calling thread until the desired time has passed or the simulation has finished. After the simulation has finished, the simulation can be continued by extending the simulation duration using `extendSimulationTime()` and starting the simulation again. If the simulation should be extended later on, `setPausedInFuture()` needs to be set accordingly. This ensures that objects that should be notified about the end of the simulation are only notified once after the last stop of the simulation. To start a new simulation, the simulator can be reset using `resetSimulator()`.

2.1.5 Full Example

The following listing provides and example how to set up and run a simulation using the simulator module.

```java
//1. Configure
Simulator sim = Simulator.getSharedInstance();
sim.setSimulationType(SimulationType.SIMULATION_TYPE_FIXED_TIME);
sim.setSimulationLoopFrequency(30);
sim.setSynchronousSimulation(true);
sim.setPausedInFuture(true);

//2. Add an object to the simulation
PhysicalVehicleBuilder physicalVehicleBuilder = PhysicalVehicleBuilder.
    getInstance();
PhysicalVehicle physVehicle = physicalVehicleBuilder.
    buildPhysicalVehicle();
sim.registerSimulationObject(physVehicle);

//3. Create a plotter object that receives notifications
SimulationPlotter2D plotter2D = new SimulationPlotter2D();
sim.registerLoopObserver(plotter2D);

//4. Run a simulation
sim.stopAfter(1000);
sim.startSimulation();

//Continue for the same amount of time as before
extendSimulationTime(1000);
sim.setPausedInFuture(false);
sim.startSimulation();
```

Listing 1: Setting up and executing a simulation.

This example code first configures the simulator to run at 30 frames per simulated second and to use a synchronous simulation. In the second step a new vehicle is added to the simulation using a factory class. After that a plotter is added and registered to receive notifications about the simulation progress. Lastly, the simulation is executed for 1 second and then continued for another second. Since the simulation is executed synchronously, `startSimulation()` is blocking and therefore no waiting-method calls are needed. Note that before the last execution of the simulation `setPausedInFuture()` is set to `false`.

2.2 `SimulationPlotter2D`

The `SimulationPlotter2D` class collects the logging data related to the `PhysicalVehicle` state update during the simulation, such as the position and velocity of its center of mass and that of the four wheels, and stores them in separate datasets. These data sets are then used by `Plotter2D` class to create separate 2D charts representing the evolution of these data along the simulation.

Operation `SimulationPlotter2D` implements the `SimulationLoopNotifiable` interface which consists of a number of functions that are executed before, during, and after the simulation starts to allow certain functionalities. As the purpose here is to create plots of logging data, the approach is to: - Collect logging data while the simulation is running - Send these data to the plotter once the simulation ends The functions of interest to us therefore are the ones run during and at the end of simulation. Therefore we override the following functions:

`willExecuteLoopForObject()` Retrieves the position of the vehicle center of mass and that of the four wheels at every time step of the simulation and stores it for plotting by the `Plotter2D` function. To be executed for every iteration in the simulation and takes as parameter the `PhysicalVehicle` object used in the current simulation.

`simulationStopped()` Once the simulation is over, this function that creates plots using instances of the `Plotter2D` class with arguments consisting of the stored data by the `willExecuteLoopForObject()` function:

- Position of vehicle center of mass,
- Position of wheels center of mass,
- Vehicle center of mass velocity,
- Time steps,
- Eventually, the user has to input a character string along with the simulation data in order to choose which data set he/she wants to plot.

3 Automobile Simulation

3.1 `Vehicle`

The vehicle is responsible for the representation of the real vehicle in the environment. It holds its dimensions, data about the wheels, e.g. wheel radius and the actuators of the car (see 3.2). In addition to that, the `Vehicle` holds its fault memory (see 3.6). As the only physical important values, the `Vehicle` contains the `WheelMassPoints` of the vehicle. These mass points represent the four mass points of the physical movable object. As the only other methods the `Vehicle` class offers setter and getter for the `StatusLogger`, which manages the fault memory, the wheel mass points and the dimensions.

3.2 `VehicleActuator`

The `VehicleActuator` reflects the different actuators of a car. In our implementation these are the motor, the brakes and the steering. An actuator has a `VehicleActuatorType` to identify it and five other important attributes:

- `actuatorValueMin`: This is the lowest value an actuator can have. E. g. for the brakes this value would be 0.
- `actuatorValueMax`: Analogous to the `actuatorValueMin` this is the maximal value an actuator can have.
- `actuatorValueCurrent`: This is the current value of the actuator.
- `actuatorValueTarget`: This is the value the actuator should have. This is set by the controller of the vehicle and reflects the drivers wishes.
- `actuatorChangeRate`: This is the rate with which the value of the actuator can change. This prevents for example the motor going from no to full acceleration in one time step.

If the controller of the vehicle now wants to accelerate, the method to set the actuator target value of the motor is called. It will then increase or decrease the value by the given change rate until the target value is reached, if it is in the allowed actuator value range.

3.3 `PhysicsEngine`

The `PhysicsEngine` computes all the forces, which are not directly produced by the vehicle. It basically uses the formula $\tau \leftarrow \Sigma_i r_i \times f_i$, to sum up different forces applying on the vehicle. The forces of

acceleration and brakes are computed in the `PhysicalVehicle` itself and the regarding methods get called from the `PhysicsEngine`. All the forces regarding gravity, centripetal force and air friction are computed directly in the `PhysicsEngine`. The forces are different for every mass point i. e. the wheels, and are computed based on the properties of them, e. g. velocity or position. The `PhysicsEngine` gets called to compute the movement of every `PhysicalObject` in the simulation but at the moment only cars can move, which makes the forces to be calculated for everything else equal to none.
The forces calculated in the `PhysicsEngine` are:

- Gravity: The force of gravity is applied by the formula $F_g = mass\ *\ 9.81$. In addition to the pure force of gravity on flat earth, there is also a downhill force implemented. This force results from adding a normal force vector that is perpendicular to the ground surface with an amount of $F_N = mass\ *\ 9.81$ to the gravity force vector.
- Air friction: The force is calculated by the formula $F_a = -0.5 *\ air\ density\ *\ velocity^2\ *\ drag\ coefficient\ *\ area\ hit\ by\ wind$ with a constant car drag coefficient of 0.3.
- Centripetal force: The force is calculated by the formula $F_c = mass\ *\ acceleration\ centrifugal = mass * (angularVelocity \times (angularVelocity \times radiusVector))$
- Road friction: The total amount road friction depends on the direction of the applied normal force vector F_N. The friction for the rolling resistance of the wheels is approximated by the formula $F_1 = 0.005 + tire\ pressure^{-1} * (0.01 + 0.0095 * (3.6 * forward\ velocity\ /100)^2)$ with an average tire pressure of 2.5 bar [1]. The sideways road friction is computed with the formula $F_2 = \mu * F_N$ for sideways movement of the vehicle with a constant μ of 0.7 for dry roads and 0.4 for wet roads.

3.4 `PhysicalVehicle`

The physical vehicle is responsible for all the needed physical computations regarding forces coming from the vehicle itself. The physical vehicle contains a `Vehicle` object. To move the vehicle the Euler loop is implemented according to the slides of the Computer Science Department for Computer Graphics of the University of Freiburg [2].

3.4.1 Euler loop computations

To execute this Euler loop, several values have to be pre-computed:

[1] http://www.engineeringtoolbox.com/rolling-friction-resistance-d_1303.html
[2] http://cg.informatik.uni-freiburg.de/course_notes/sim_06_rigidBodies.pdf

- $M \leftarrow \Sigma_i m_i$
- $\bar{x}_{CM} \leftarrow \frac{1}{M}\Sigma_i \bar{x}_i m_i$
- $\bar{r_i} \leftarrow \bar{x_i} - \bar{x}_{CM}$
- $\bar{I}^{-1} \leftarrow (\Sigma_i - m_i \tilde{\bar{r}}_i \tilde{\bar{r}}_i)^{-1}$

In addition to that, some values have to be initialized:

- $x_{CM},\ v_{CM},\ A,\ L$
- $I^{-1} \leftarrow A\bar{I}^{-1}A^T$
- $\omega \leftarrow I^{-1}L$

One loop iteration is then applied by the formulas:

1. $F \leftarrow \Sigma_i f_i$
2. $x_{CM} \leftarrow x_{CM} + \Delta t * v_{CM}$
3. $v_{CM} \leftarrow v_{CM} + \Delta t * F/M$
4. $A \leftarrow A + \Delta t * \tilde{\omega} A$
5. $L \leftarrow L + \Delta t * \tau$
6. $I^{-1} \leftarrow A\bar{I}^{-1}A^T$
7. $\omega \leftarrow I^{-1}L$
8. $r_i \leftarrow A * \bar{r}_i$
9. $x_i \leftarrow x_{CM} + r_i$
10. $v_i \leftarrow v_{CM} + \omega \times r_i$

Following these 11 steps, the movement of a rigid body can be simulated. For every simulation step you have to jump from step 11 back to step 1 in the end. The step, where external forces are summed up, moved to the `PhysicsEngine` (3.3). The forces, which are computed in the `PhysicalVehicle` and included in the computation are:

- Brake force: The brake force is equal to the actual acceleration but in the opposite direction. So the Brake force is calculated by the formula $F_b = acceleration\ *\ -1$
- Acceleration force: The acceleration force is simply applied by the formula $F_a = mass * acceleration$

3.4.2 Further explanations `PhysicalVehicle`

Regarding methods and constants, the `PhysicalVehicle` class is rather extensive. It contains multiple physical constants, e.g. air density or the gravity constant. For all the forces listed in section 3.4.1 there are methods to compute them.Initially a `PhysicalVehicle` is set up by the `PhysicalVehicleBuilder` (see Section 3.5).

3.5 `PhysicalVehicleBuilder`

To avoid a large number of contructors with complex parametrizations, we introduced the `PhysicalVehicleBuilder` for creating `PhysicalVehicle` objects. The implementation is done according to the *builder pattern*. Furthermore, the builder is implemented as a singleton. The builder is capable to contruct a `PhysicalVehicle` without the need of further parametrization. This can be done by calling its `buildPhysicalVehicle` method directly, without taking care about its setter methods. All parameters are set to reasonable default values.

For customization of the resulting `PhysicalVehicle` object, one has to use the provided setter methods. In the following, we will discuss the most important setter functionalities.

- `setDimensions`, `setGlobalPosition`, `setGlobalRotation` With this setters one can customize the size so length, width and height of a vehicle. Furthermore, the current position in the global coordinate system and the rotation, so the orientation of the car can be adjusted.
- `setActuatorProperties` With the `setActuatorProperties` one can adjust actuator parameters, i.e., the minimum value, the maximum value and the change rate. Currently there are three actuator types for the engine, the brakes and the steering.
- `setWheelProperties` In this setter, the radius and the distance of the wheels may be adjusted. Additionally, two mass values can be set, the sum of the mass of the front wheels and the sum of the mass of the back wheels. Be aware of the fact, that in the simulation the wheels are assumed as the mass points, i.e., the sum of the wheels masses is the overall weight of the car.

During the building process the values can be reset to the default values by calling the `resetPhysicalVehicle` method. Additionally it is possible to serialize and store a `PhysicalVehicle`, including all set parameters, in JSON format on hard disk. To store the currently build `PhysicalVehicle` the method `storeJSONInFile` method has to be invoked. To restore a previously serialized `PhysicalVehicle` use the `loadPropertiesFromFile` method. The JSON serializer and deserilizer used is based on the GSON library. For the ease of use a middleware is used `ParsableVehicleProperties`, this is to be able to directly use GSONs automatic serialization and deserialization engine. All static properties of a `PhysicalVehicle` are represented in a object of the middleware class, then are serialized to a JSON string. When restoring from such a serialized JSON string first a `ParsableVehicleProperties` object is build and from this the `PhysicalVehicleBuilder` is parametrized appropriately.

3.6 Fault Memory

The simulator allows to also simulate diagnostic messages of vehicles. The class `VehicleStatusSensor` represents a sensor that monitors the status of a component of the car, such as the motor or the brakes. Using `VehicleStatus`, the condition of the monitored component can be expressed. Every `VehicleStatus` also includes a *severeness* that indicates how well the condition of the monitored component is. Use the function `isWorseThanOrEqual()` to check if a status is at least as severe as a given level.

To create status messages, the class `StatusMessage` is used. It allows to create status messages that, besides the `VehicleStatus` and the `VehicleStatusSensor`, also include an error code, a timestamp and an optional (preferably human-readable) message.

3.6.1 `StatusLogger`

The aggregation of status messages for a specific car, i.e., the fault memory component, is represented by an object of type `StatusLogger`. Use `setStatusLogger()` to assign a specific status logger to a specific `Vehicle` object. Besides the option to add status messages, `StatusLogger` of course also allows to clear or read out the status memory. In addition to that, it also provides the functionality to read out only status messages that are at least as severe as a given `VehicleStatus` or to read out only those messages that belong to a given `VehicleStatusSensor`.

Because the car does not create status messages by itself by default, one can use a `RandomStatusLogger`, which is a subclass of `StatusLogger`. This class in certain (randomized) time intervals creates status messages that simulate the damage or failure of a component.

4 Utilities

In the utilities module auxilliary functionalities are located. This includes functionalities for application wide inter-object communication via a broadcast service, a plotter, static math auxilliary functionality and logging functionality.

4.1 Logging

The logging class is a further abstraction of Javas logging framework located in `java.util.log`. The `Log` class provides mainly one static core method the `log()` method. It takes a log level as first argument and the message to log as second. This method is used for logging purposes in all of our submodules. For convenience the `Log` class provides static methods for logging a message with a certain level directly, for instance, the static method `info()` takes the log message as string as only arguement and logs the message with log level info. This is done by passing the appropriate arguements to the `log()` method.

Internally, the `Log` class is structured in the inner class `LoggingModule`. This class takes care of initialization and management of the actual `Logger` object. All messages logged via the `log()` method are passed to this logger. It is capable of writing a log file to disk. This can be configured via the static method `setWriteToDiskEnabled()`. Enabling logging to disk causes the logger to write a logfile named *simulation.log* to the current working directory.

4.2 App-wide Communication

The utilities contain a set of classes that can be used for broadcasting or offering information anonymously to the application.

4.2.1 `NotificationCenter`

This class allows to broadcast information within the application. It is essentially a re-implementation of the `NSNotificationCenter` [3] found on iOS and MacOS. Similar concepts are also used in the so-called "intents" found on Android.

To listen to or stop listening to specific broadcasts, the `registerListener()` and `removeListener()` methods are used. The caller can here define a method to be called on a specific object, whenever the given notification is broadcasted. The methods to be called are required to take an object of type `Object` as parameter. It is used to optionally send additional information with the broadcast. This is done using the so-called *Method references* introduced in Java 8. For example, `registerListener("Title",`

[3] `https://developer.apple.com/reference/foundation/nsnotificationcenter`

`rec1::receiveMsg, rec1)` registers the method `receiveMsg` to be called on object `rec1` whenever `Title` is broadcasted. The used broadcast titles should be stored in `Notification` class.

To broadcast a message `postNotification()` is used. This method broadcasts a message with a specific title. Optionally, an object can be passed as additional information to the objects receiving the broadcast.

4.2.2 `InformationService`

The `InformationService` is used to offer information anonymously within the program. To offer a specific information, an object calls `offerInformation()`. There, a title for the information is defined as well as a method that can provide the information whenever the information is requested. The used information titles should be stored in `Information` class. To stop providing the information, set ”null” as information provider.

To retrieve a specific information, an object may call `requestInformation()`. As a parameter the title of the desired information is supplied. The `InformationService` then requests the information from the object the supplier and returns it to the object that asked for the information.

This service can be used to, e.g., request the current simulation time in objects that do not have access to the `Simulator` class.

4.3 `Plotter2D`

`Plotter2D` class is based on the `JFree` plotting tools for Java. It is executed after the end of simulation and displays 2D graphs of position and/or velocity-based on an input string by the user. It creates a `JFreeChart` object for the data in question which will be displayed, and it is based mainly on the following two functions:

1. `VehiclePositionDataset/VehicleVelocityDataset/WheelsDataset`

   Creates XY datasets for the input individual data sets obtained during simulation, and returns an XY series representing the relative position/velocity of the `PhysicalVehicle` in the plane for plotting For example, for every wheel and also for the vehicle's center of mass, it returns a data set of the form $Y = f(X)$, where Y and X represent the coordinates of the quantity in question. Other time related data sets are available as well such as $X = f(t), Y = f(t)$.

2. `VehiclePositionChart/VehicleVelocityChart`

   Creates the vehicle position or velocity 2D plot charts using datasets of created previously by their respective functions. It creates the chart for display and defines their properties such as naming of the axis and properties of the renderer (symbols used for plotting, colours to distinguish between different sets, etc.), and in the case of vehicle oosition chart, it combines more than one dataset together to display in one graph: Vehicle's center of mass and wheels center of mass.

4.4 `MathHelper`

The `MathHelper` is a final class, providing some static methods to execute some math operations. All of the offered methods are checked for correctness and log their results. The class offers for example the orthonormalization of a three-dimensional matrix. This is important for the simulation, because the Euler loop uses a rotation matrix, which has to stay orthonormal during the whole computation. Because the used computations are not exact and just approximate the right results, the matrix will not stay orthonormal unless it is re-orthonormalized after every simulation step.