Vehicle Modeling with Chronos

A. Giavaras

Contents

1 Vehicle Modeling with Chronos

In this section we will develop and simulate a vehicle model using the open source physics engine chronos.

```
http://projectchrono.org/
```

Specifically, we will simulate a vehicle that moves in straight line. Figure 1 shows a snapshot of the simulation

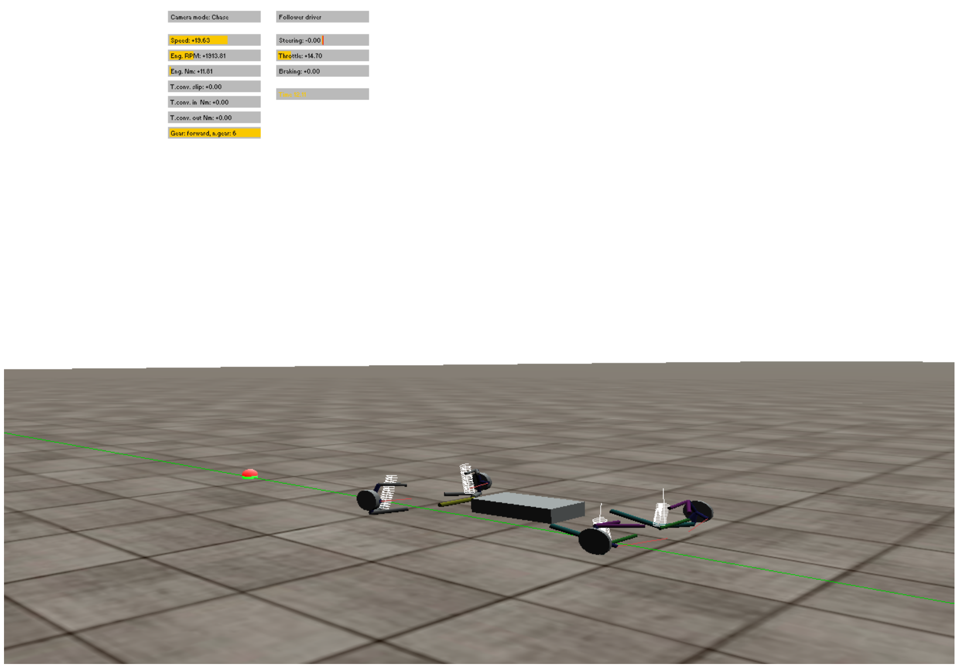

Fig. 1: Straight line vehicle motion.

The following link can be used to consult for further information

```
http://api.projectchrono.org/tutorial_install_project.html
```

The reference manual can be found at

```
http://api.projectchrono.org/manual_root.html
```

1.1 The Chrono::Vehicle library

The Chrono::Vehicle is a C++ middleware library for the modeling, simulation, and visualization of wheeled and tracked ground vehicles. It consists of two core modules:

- The ChronoEngine_vehicle
  - Defines the system and subsystem base classes
  - Provides concrete, derived classes for instantiating templates from JSON specification files
  - Provides miscellaneous utility classes and free functions for file I/O, Irrlicht vehicle visualization, steering and speed controllers, vehicle and subsystem test rigs, etc.
- The ChronoModels_vehicle
  - Provides concrete classes for instantiating templates to model specific vehicle models

The following dependencies should be satisfied in order to use the library.

- The Chrono::Engine required
- The Chrono:: Irrlicht and the Irrlicht library, Chrono::OpenGL and its dependencies. Both are optional
- The Chrono::FEA and Chrono::MKL (optional)

The Chrono::Engine supports the notion of a system. In our case, the following components are considered a system

- Powertrain
- Tire
- Terrain
- Driver
- Vehicle

Chrono::Vehicle encapsulates templates for systems and subsystems in polymorphic C++ classes:

- A base abstract class for the system/subsystem type (e.g. Chrono::ChSuspension)
- A derived, still abstract class for the system/subsystem template (e.g. Chrono::ChDoubleWishbone)
- Concrete class that particularize a given system/subsystem template (e.g. Chrono::HMMWV_DoubleWishboneFront)

1.2 The chrono:: vehicle :: ChVehicle class

Vehicles in Chrono inherit from the base class chrono:: vehicle :: ChVehicle. This class provides the interface between the vehicle system and other systems (tires, driver, etc.)

The reference frame for a vehicle follows the ISO standard. Namely, Z-axis up, X-axis pointing forward, and Y-axis towards the left of the vehicle. The following figure illustrates the asseumed reference frames.

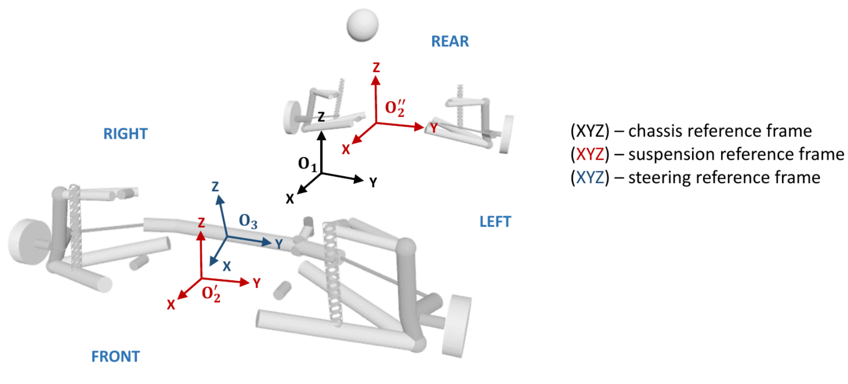

Fig. 2: Vehicle ISO reference frames.

A chrono::vehicle::ChVehicle has

- ChSystem* m_system pointer to the Chrono system
- std::shared_ptr<ChChassis> m_chassis handle to the chassis subsystem
- **bool** m_ownsSystem true if system created at construction
- **double** m_stepsize integration step-size for the vehicle system

Deferring to its constituent subsystems as needed, a chrono::vehicle::ChVehicle provides accessors for:

- Underlying chrono::vehicle::ChSystem
- Handle to the vehicle chassis
- Chassis state (reference frame and COM)
- Angular speed of the vehicle driveshaft (connection to powertrain)

A chrono::vehicle::ChVehicle intermediates communication between other systems (e.g., powertrain, driver, etc.) and constituent subsystems (e.g., suspensions, brakes, etc.)

1.3 The chrono::vehicle::ChChassis class

This is the ase class for the chassis vehicle subsystem. The class documentation can be found at

```
http://api.projectchrono.org/classchrono_1_1vehicle_1_1_ch_chassis.html
```

A chassis has the following attributes

- std::shared_ptr<ChBodyAuxRef> m_body; a handle to the chassis body
- **bool** m_fixed; a flag indicating if the chassis body is fixed to the ground

It provides access to the following properties

- Chassis mass and inertia properties
- Chassis state reference frame and COM
- Vehicle speed reference frame and COM
- Driver position
- Absolute acceleration of a point specified in local reference frame

A chassis system can be specified in a JSON file.

1.4 The chrono::vehicle::ChDriver class

Base class for a vehicle driver system. A driver system must be able to report the current values of the inputs (throttle, steering, braking). A concrete driver class must set the member variables:

- m_throttle
- m_steering
- m_braking

Since these are the main quantities that a driver can interact with a vehicle, this class has to be adapted when we want to incorporate autonomy.

1.5 Setup simulation

1.5.1 Setup the vehicel chrono::vehicle::sedan::Sedan

Now that we went over the basics of the Chrono::Vehicle library let's try to set up a basic simulation; namely a vehicle that moves in straight line. Concretely, we will use an instance of the chrono::vehicle::sedan::Sedan class.

The following code initializes the vehicle instance for the simulation

```
// Create the vehicle, set parameters, and initialize
    Sedan vehicle;
    vehicle.SetContactMethod(contact_method);
    vehicle.SetChassisFixed(false);
    vehicle.SetInitPosition(ChCoordsys<>(initLoc, initRot));

    vehicle.SetTireType(tire_model);
    vehicle.SetTireStepSize(tire_step_size);
```

```
    vehicle.SetVehicleStepSize(step_size);
    vehicle.Initialize();

    vehicle.SetChassisVisualizationType(chassis_vis_type);
    vehicle.SetSuspensionVisualizationType(suspension_vis_type);
    vehicle.SetSteeringVisualizationType(steering_vis_type);
    vehicle.SetWheelVisualizationType(wheel_vis_type);
    vehicle.SetTireVisualizationType(tire_vis_type);
```

1.5.2 Create the application

```
// Create the vehicle Irrlicht application
ChVehicleIrrApp app(&vehicle.GetVehicle(), &vehicle.GetPowertrain(),
                    L"Steering_XT_Controller_Demo",
        irr::core::dimension2d<irr::u32>(800, 640));

app.SetHUDLocation(500, 20);
app.SetSkyBox();
app.AddTypicalLogo();

irr::core::vector3df v1(-150.f, -150.f, 200.f);
irr::core::vector3df v2(-150.f, 150.f, 200.f);
irr::core::vector3df v3(150.f, -150.f, 200.f);
irr::core::vector3df v4(150.0f, 150.f, 200.f);
app.AddTypicalLights(v1, v2, 100, 100);
app.AddTypicalLights(v3, v4, 100, 100);
app.EnableGrid(false);
app.SetChaseCamera(trackPoint, 6.0, 0.5);
app.SetTimestep(step_size);
```

Our application will use an instance of the Chrono::ChPathFollowerDriverXT class. This is a driver model that uses a path steering controller and a speed controller. The steering controller adjusts the steering input to follow the prescribed path. The output from the speed controller is used to adjust throttle and braking inputs in order to maintain the prescribed constant vehicle speed.

1.5.3 Advance the vehicle

Each system base class declares a virtual function Advance() with a single parameter, the time interval between two communication points (Δt). A particular system may take as many intermediate steps (constant or variable step-size) as needed to advance the state of the system by (Δt). If the system has no internal dynamics, this function can be a no-op

```
driver_follower.Advance(step);
driver_gui.Advance(step);
terrain.Advance(step);
vehicle.Advance(step);
app.Advance(step);
```

1.5.4 Vehicle state information

When running a simulation, we would like to be able to view various quantites that describe the state of the vehicle. Let's see how we can obtain some of them.

Get the vehicle position coordinates and vehicle speed

```
//This is the global location of the chassis reference frame origin.
ChVector<> pos = vehicle.GetVehicle().GetVehiclePos();

//Return the speed measured at the origin of the chassis reference frame.
double speed = vehicle.GetVehicle().GetVehicleSpeed();
```

References

[1] Åström K. J., Murray R. M. *Feedback Systems. An Introduction for Scientists and Engineers*

[2] Philip , Florent Altche1, Brigitte dAndrea-Novel, and Arnaud de La Fortelle *The Kinematic Bicycle Model: a Consistent Model for Planning Feasible Trajectories for Autonomous Vehicles?* HAL Id: hal-01520869, https://hal-polytechnique.archives-ouvertes.fr/hal-01520869

[3] Marcos R. O., A. Maximo *Model Predictive Controller for Trajectory Tracking by Differential Drive Robot with Actuation constraints*