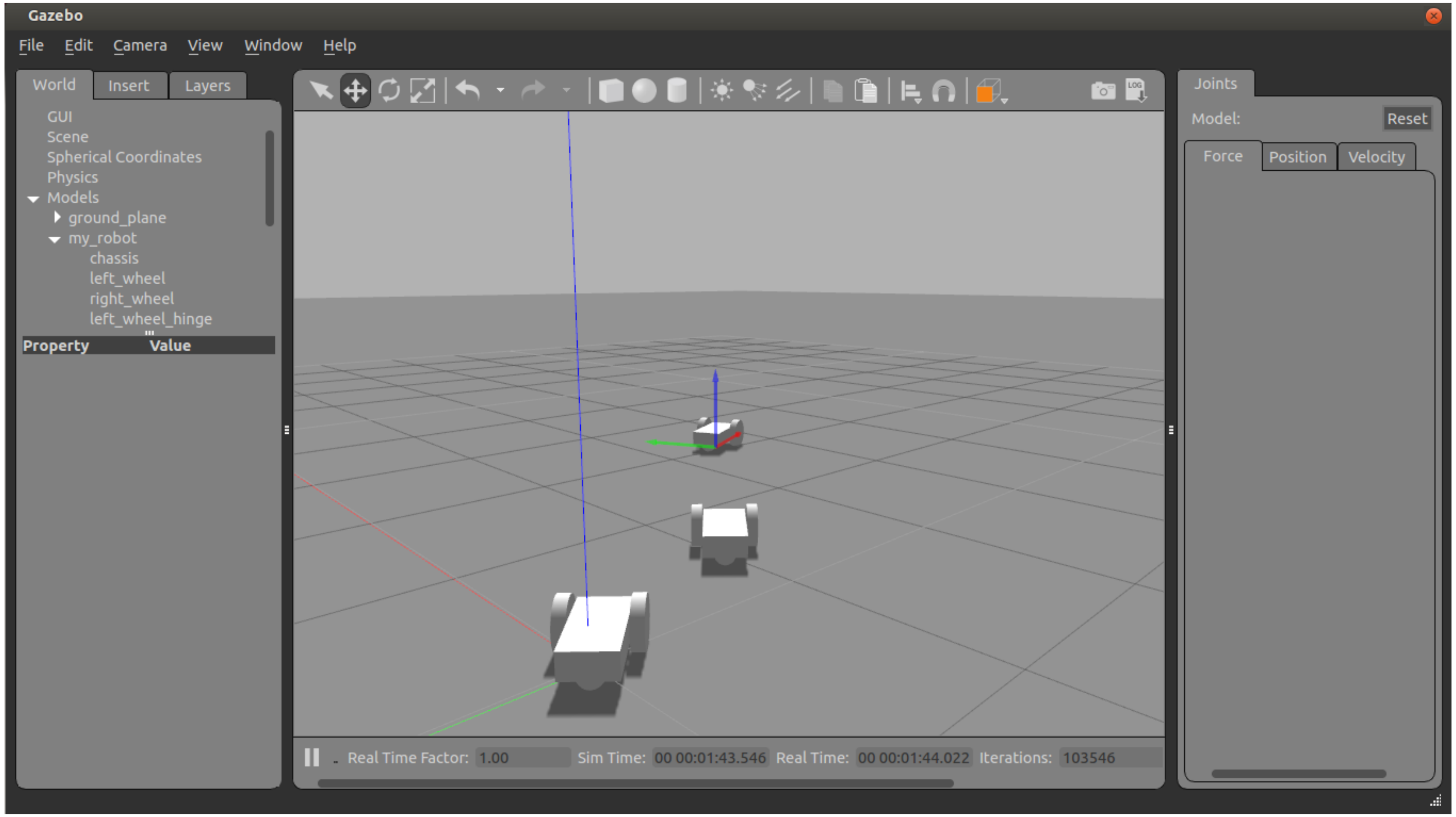
DETAIL OF THE MODEL

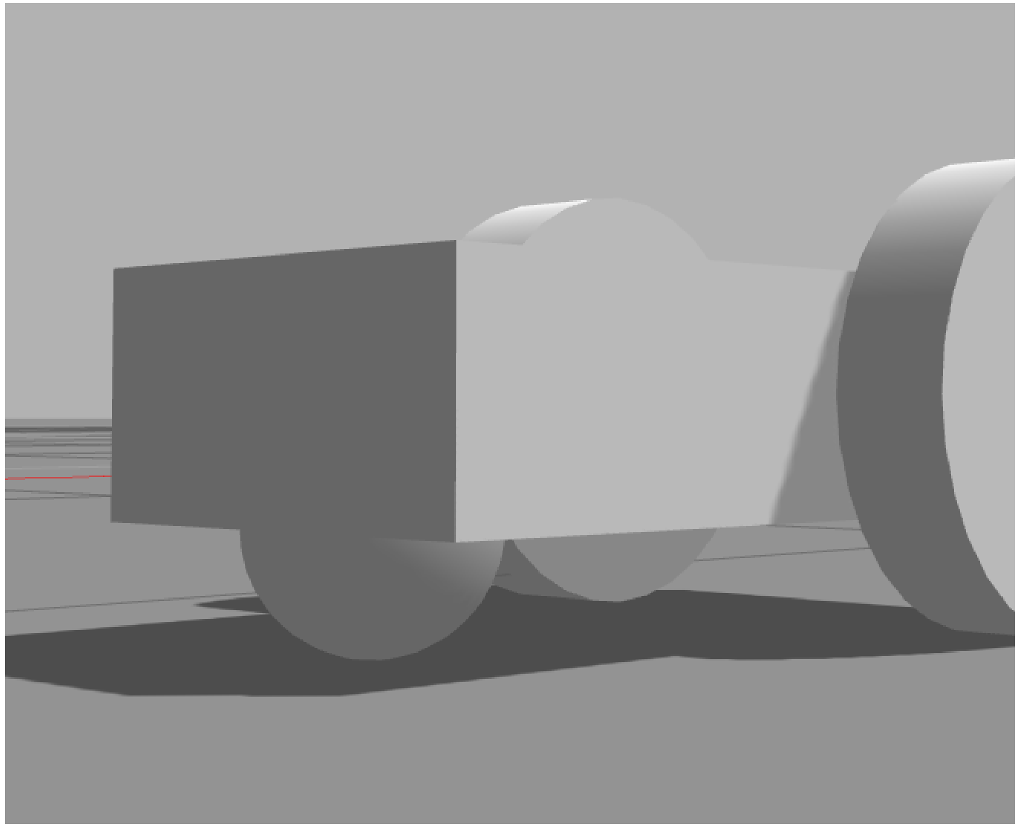
ROBOTICS

POLITECNICO
MILANO 1863

IN THE PREVIOUS EPISODE…



THE CASTER WHEEL

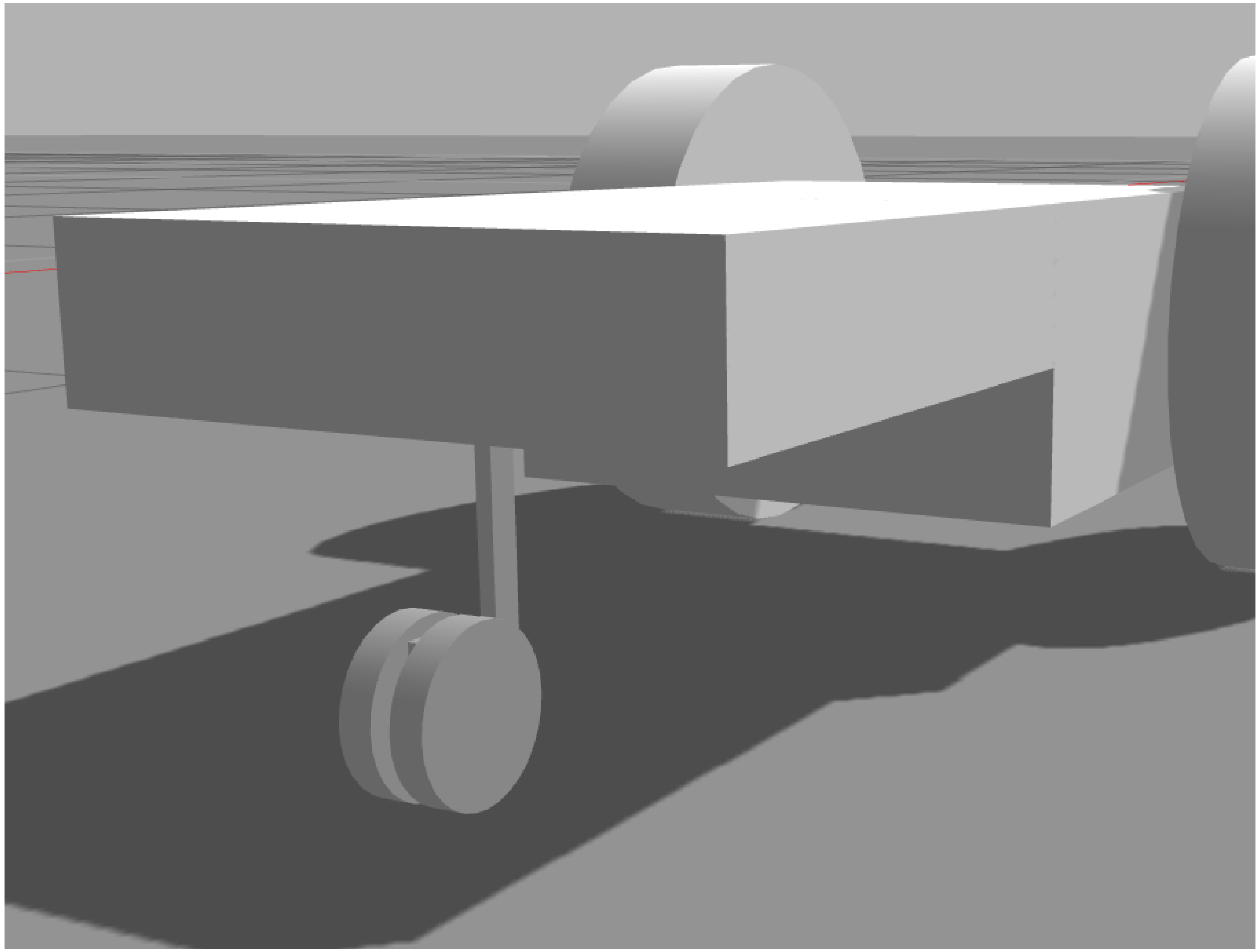


VS

YOUR TURN

How would you design a caster wheel?

I TRIED TOO…



FIELD TEST

Let's see it in action

It's terrible! Why?

EFFORT VS PERFORMANCE

Building a perfect model may require a lot of time and effort
You always have to measure the complexity of the model against the task

You need a detailed model when

- testing field performance
- testing specific environment
- analyzing a specific behavior
- working on low level tasks (i.e. localization)

You DON'T need a detailed model when

- performance doesn't matter
- analyzing the general behavior
- working in a structured environment
- working on high level tasks (i.e. planning)

SENSORS AND PLUGINS

ROBOTICS

POLITECNICO
MILANO 1863

TYPES OF SENSORS

Many ways to differentiate sensors, in a simulation: proprioceptive vs exteroceptive

Proprioceptive sensors:

- provide information about the robot
- only the model of the robot is required
- examples: GPS, accelerometer, gyroscope, odometer, torque sensor, …

Exteroceptive sensors:

- provide information about the environment
- require some form of interaction
- examples: laser scanner, contact and proximity sensor, camera, sonar, …

ERRORS

Perfect sensors doesn't exist

Every measurement is subject to some error

ERRORS

Different types of errors:

- Systemic errors (predictable, can be removed)
  - Bias, removed through calibration
  - Drift, caused by use (i.e. rising temperatures)
- Random errors (unpredictable, can be estimated)
  - Noise, intrinsic error of the measurement tool
  - Random event disrupting the measurement

IN REAL SENSORS

Gyroscope and accelerometer: bias

Magnetometer: distortion and varying Earth magnetic field

GPS: Absence of measurements and multipath

Laser scanner: reflection

Odometer: drift (short term and long term)

Contact sensor: detection fail and response time

Camera: distortion, lack of focus, compression errors

All of them: noise with different characteristics depending on the sensor

SENSORS IN GAZEBO

In SDF sensors have they own tag: **`sensor`**

child of **`link`** or **`joint`**

type of sensor specified by the attribute **`type`**

> **`altimeter, camera, contact, depth, force_torque, gps, gpu_ray, imu, logical_camera, magnetometer, multicamera, ray, rfid, rfidtag, sonar, wireless_receiver and wireless_transmitter`**

Each type has its own tag to define the parameters of the sensor

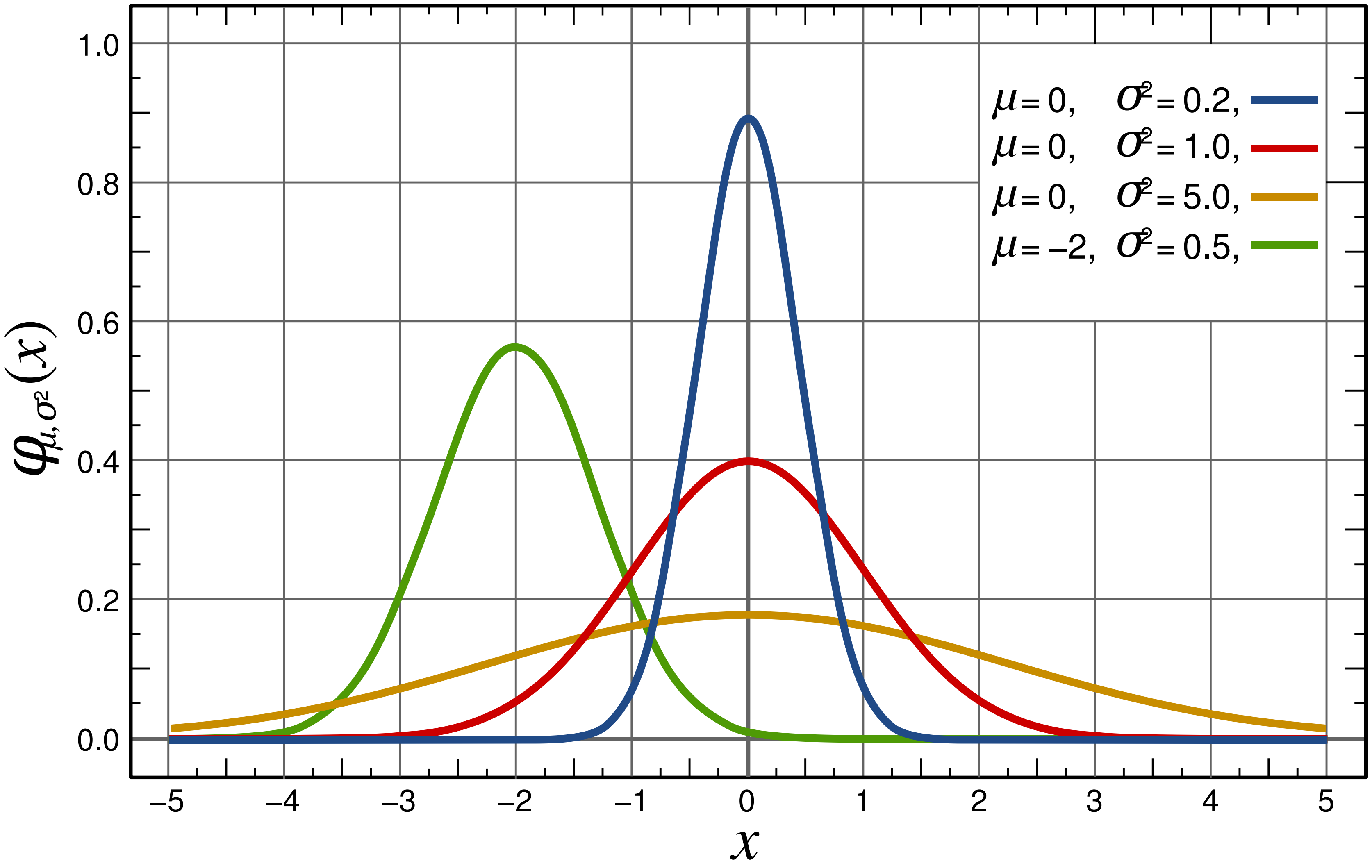
Everything not on the list have to be modeled "by hand"

A COUPLE OF EXAMPLES

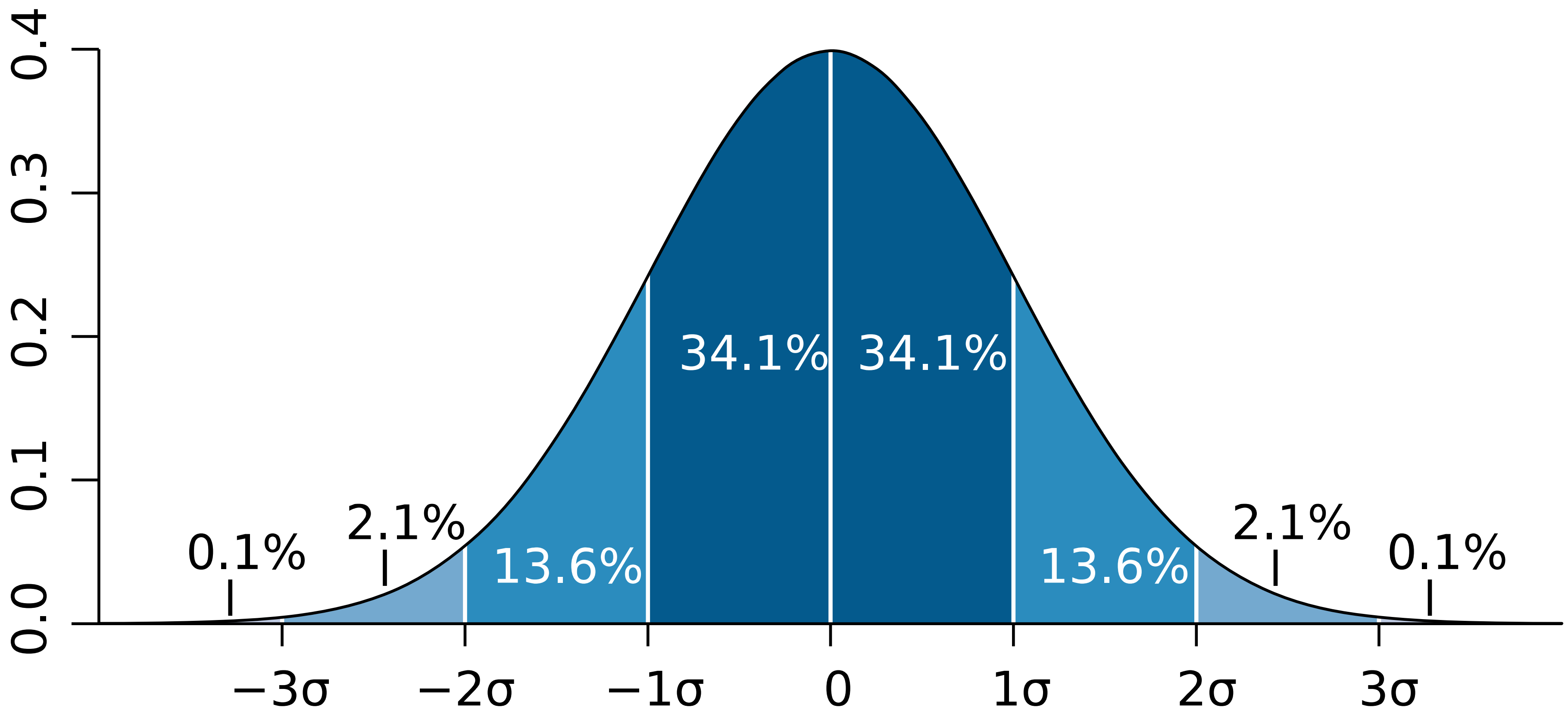
```xml
<sensor type="camera" name="camera1">
    <camera>
        <horizontal_fov>1.047</horizontal_fov>
            <image>
                <width>320</width>
                <height>240</height>
            </image>
            <clip>
                <near>0.1</near>
                <far>100</far>
            </clip>
    </camera>
    <always_on>1</always_on>
    <update_rate>30</update_rate>
    <visualize>true</visualize>
</sensor>
```

```xml
<sensor type="gps" name="mGPS">
    <gps>
        <position_sensing>
            <horizontal><noise type="gaussian">
                <mean>0.0</mean>
                <stddev>0.5</stddev>
            <noise></horizontal>
            <vertical><noise type="gaussian">
                <mean>0.0</mean>
                <stddev>5.0</stddev>
            </noise></vertical>
        </position_sensing>
    </gps>
    <always_on>1</always_on>
    <update_rate>10</update_rate>
</sensor>
```

UNDERSTANDING THE GAUSSIAN NOISE



UNDERSTANDING THE GAUSSIAN NOISE

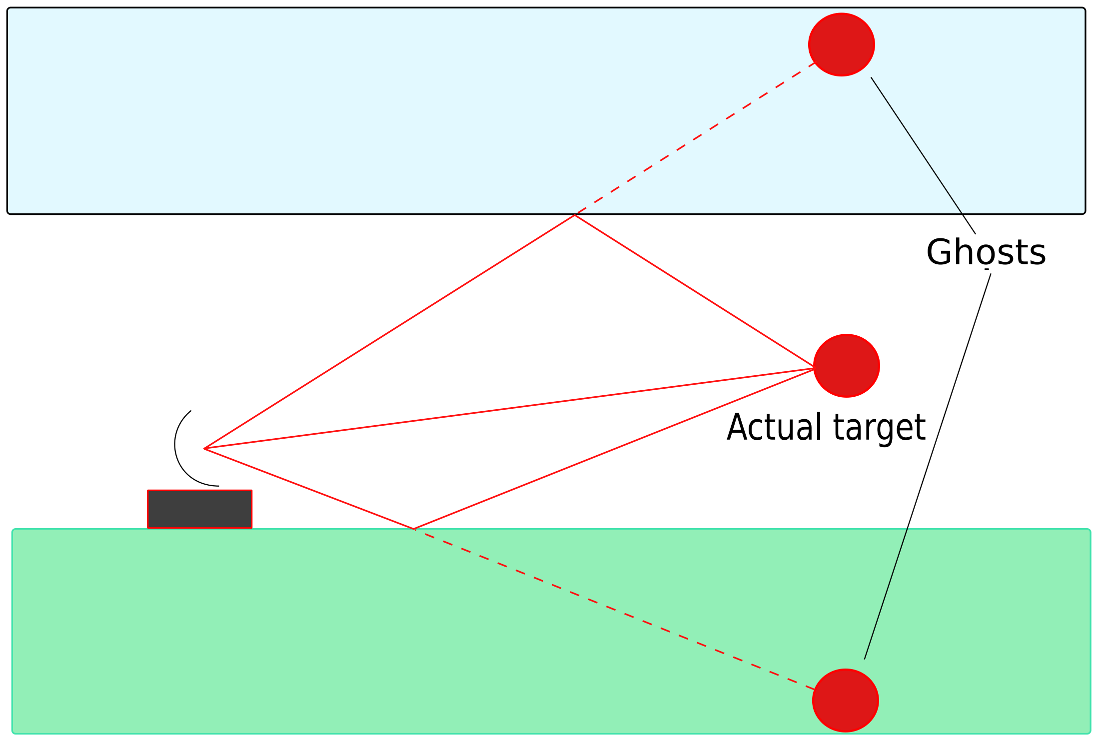


FAULTY BEHAVIOR

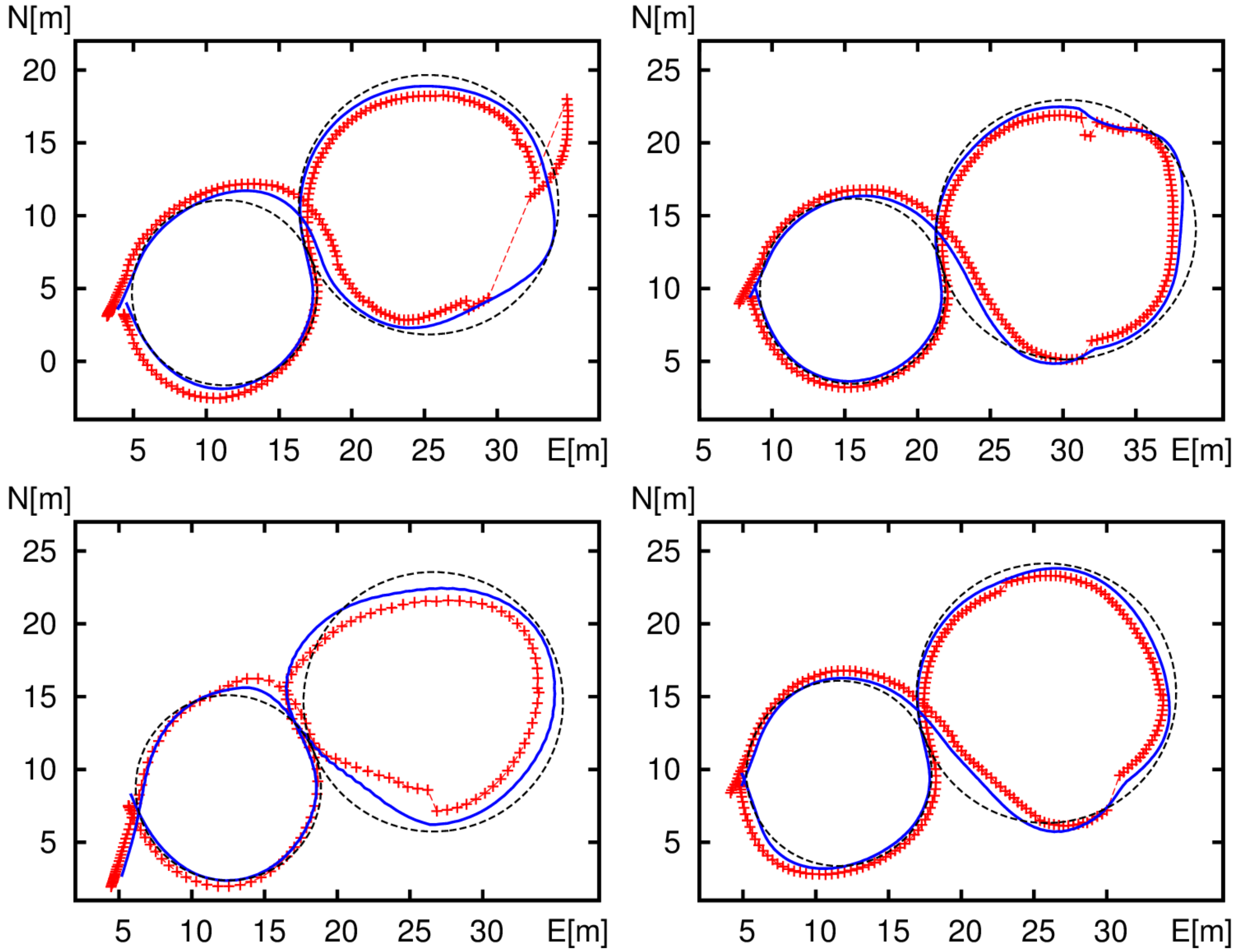
Some sensors have more than bias and noise, a common example is GPS multipath and loss of signal

Loss of signal: a GPS receiver needs clear view of the sky to contact satellites. Trees or building may cause interruption in the communication

Multipath: messages from the satellites are reflect by buildings, the ground or the atmosphere. The GPS receiver collect multiple signal from the same satellite and miscalculate the position



FAULTY BEHAVIOR



YOUR TURN

How would you simulate the multipath and the loss of signal?

MODEL SPECIFIC BEHAVIORS IN GAZEBO

It is possible to customize the behavior of the simulation using plugins

A plugin is a chunk of code that is compiled as a shared library and inserted into the simulation. The plugin has direct access to all the functionality of Gazebo through the standard C++ classes.

Six different types of plugins depending on the associated object: world, model, sensor, system, visual, GUI

HELLO WORLD

```
#include <gazebo/gazebo.hh>
namespace gazebo {
  class HelloWorldPlugin : public WorldPlugin {
    public: HelloWorldPlugin() : WorldPlugin() {
      printf("Hello World!\n");
    }
    public: void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf){}
  };
  GZ_REGISTER_WORLD_PLUGIN(HelloWorldPlugin)
}
```

LET'S ANALYZE THE CODE

```
#include <gazebo/gazebo.hh>
namespace gazebo {
```

Various includes depending on the feature used (i.e. math or sensors)

Every plugin must be in the **gazebo** namespace

```
 class HelloWorldPlugin : public WorldPlugin {
    public: HelloWorldPlugin() : WorldPlugin() {
      printf("Hello World!\n");
    }
```

Each plugin must inherit from a plugin type, which in this case is the **WorldPlugin** class.

We print our "Hello world!" in the constructor method

LET'S ANALYZE THE CODE

```
public: void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf){}
```

This is the only mandatory function, receives an SDF element that contains the elements and attributes specified in loaded SDF file.

In our case it's only a placeholder since we have no extra logic.

```
GZ_REGISTER_WORLD_PLUGIN(HelloWorldPlugin)
```

This macro register the plugin in the simulator, the only requested parameter is the plugin name

Each plugin has it's own register macro: `GZ_REGISTER_MODEL_PLUGIN`, `GZ_REGISTER_SENSOR_PLUGIN`, `GZ_REGISTER_SYSTEM_PLUGIN`, `GZ_REGISTER_WORLD_PLUGIN` and `GZ_REGISTER_VISUAL_PLUGIN`.

A MORE INTERESTING EXAMPLE

```
#include <boost/bind.hpp>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <stdio.h>

namespace gazebo {
  class ModelPush : public ModelPlugin {
    public: void Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/) {
      this->model = _parent;
      this->updateConnection = event::Events::ConnectWorldUpdateBegin(
                                  boost::bind(&ModelPush::OnUpdate, this, _1));
    }
```

CONT.

```
    public: void OnUpdate(const common::UpdateInfo & /*_info*/) {
      this->model->SetLinearVel(math::Vector3(.03, 0, 0));
    }

    private: physics::ModelPtr model;

    private: event::ConnectionPtr updateConnection;
  };
  GZ_REGISTER_MODEL_PLUGIN(ModelPush)
}
```

BACK TO SENSORS

Let’s see now how it’s possible to add a faulty behavior to the GPS

We try to implement a simple way to randomly shut down the sensor

FUALTY GPS

```
namespace gazebo {
  class GAZEBO_VISIBLE FaultyGPSPlugin : public SensorPlugin {
    public: FaultyGPSPlugin();
    public: virtual ~FaultyGPSPlugin();
    public: void Load(sensors::SensorPtr _parent, sdf::ElementPtr _sdf);
    protected: virtual void OnUpdate(sensors::GpsSensorPtr _sensor);
    protected: virtual void OnWorldUpdate(const common::UpdateInfo &_info);
    protected: sensors::GpsSensorPtr parentSensor;
    private: event::ConnectionPtr connection;
    private: event::ConnectionPtr updateConnection;
  };
}
```

CONT.

```
#include "FaultyGPSPlugin.hh"

using namespace gazebo;

GZ_REGISTER_SENSOR_PLUGIN(FaultyGPSPlugin)

FaultyGPSPlugin::FaultyGPSPlugin() {}

FaultyGPSPlugin::~FaultyGPSPlugin() {
    this->parentSensor->DisconnectUpdated(this->connection);
    this->parentSensor.reset();
}
```

CONT.

```
void FaultyGPSPlugin::Load(sensors::SensorPtr _parent, sdf::ElementPtr _sdf) {
  this->parentSensor = std::dynamic_pointer_cast<sensors::GpsSensor>(_parent);

  if (!this->parentSensor)
    gzthrow("FaultyGPSPlugin requires a gps sensor as its parent.");

  this->connection = this->parentSensor->ConnectUpdated(
              std::bind(&FaultyGPSPlugin::OnUpdate, this, this->parentSensor));

  this->updateConnection = event::Events::ConnectWorldUpdateBegin(
              boost::bind(&FaultyGPSPlugin::OnWorldUpdate, this, _1));
}
```

CONT.

```
void FaultyGPSPlugin::OnUpdate(sensors::GpsSensorPtr _sensor) {
  if(math::Rand::GetDblUniform() > 0.1)
    _sensor->SetActive(false);
}

void FaultyGPSPlugin::OnWorldUpdate(const common::UpdateInfo & /*_info*/) {
  if(!this->parentSensor->IsActive()) {
    if(math::Rand::GetDblUniform() > 0.995)
      this->parentSensor->SetActive(true);
  }
}
```

HOW TO USE THE PLUGIN

This plugin have to be added to a sensor in the simulation, three steps:

1. Compile the code using make
2. Add the compiled library to an sdf file
3. Tell Gazebo where is the library

CMAKE AND MAKE

I put the plugin code in the same directory as the model of the sensor

Use a different structure if you want to create a more general plugin

```
mkdir faultygps
cd faultygps
gedit CMakeLists.txt
mkdir build
cd build
cmake ..
make
```

The result is a library file called: `libFaultyGPSPlugin.so`

CMAKE AND MAKE

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

find_package(gazebo REQUIRED)
include_directories(${GAZEBO_INCLUDE_DIRS})
link_directories(${GAZEBO_LIBRARY_DIRS})
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GAZEBO_CXX_FLAGS}")

add_library(FaultyGPSPlugin SHARED FaultyGPSPlugin.cc)
target_link_libraries(FaultyGPSPlugin ${GAZEBO_libraries})
```

ADDING THE PLUGIN

```
<?xml version='1.0'?>
<sdf version='1.5'>
  <model name="gps">
    <static>true</static>
    <link name="link">
      <visual name='box'> <geometry>
          <box> <size>.05 .05 .05</size> </box>
      </geometry> </visual>
      <sensor type="gps" name="mGps">
        . . .
        <plugin name="faulty_behavior" filename="libFaultyGPSPlugin.so"/>
      </sensor>
    </link>
</model>
</sdf>
```

RUNNING GAZEBO

To run the plugin we need to tell Gazebo where to find the library

```
export GAZEBO_PLUGIN_PATH=/path/to/plugin/build:$GAZEBO_PLUGIN_PATH
```

This command have to be run in every time you want to run gazebo with a plugin in a new terminal

If you want to see any output result generated with **`printf`** or **`std::cerr`**, run Gazebo with this command

```
gazebo --verbose
```