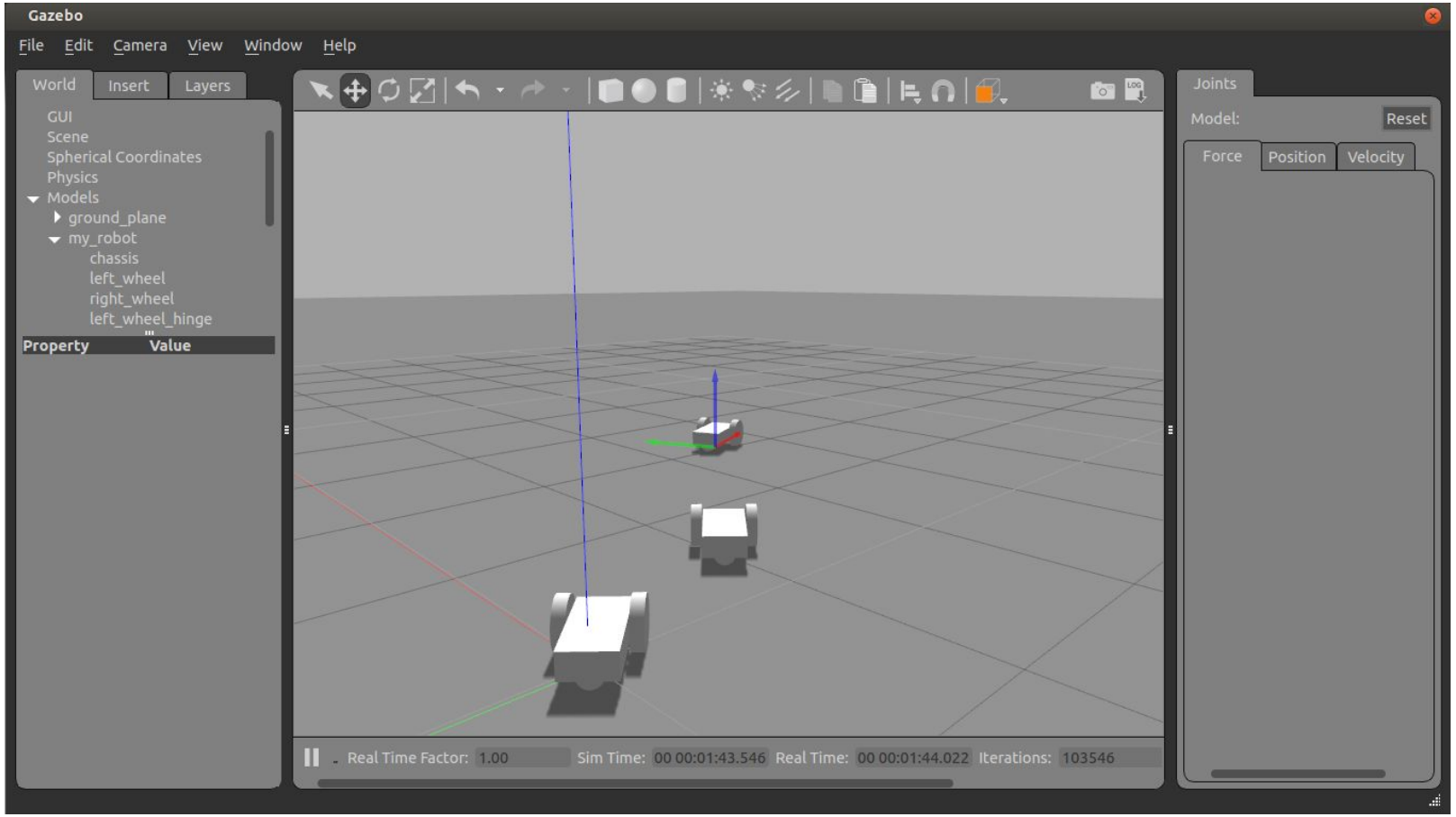
DETAIL OF THE MODEL

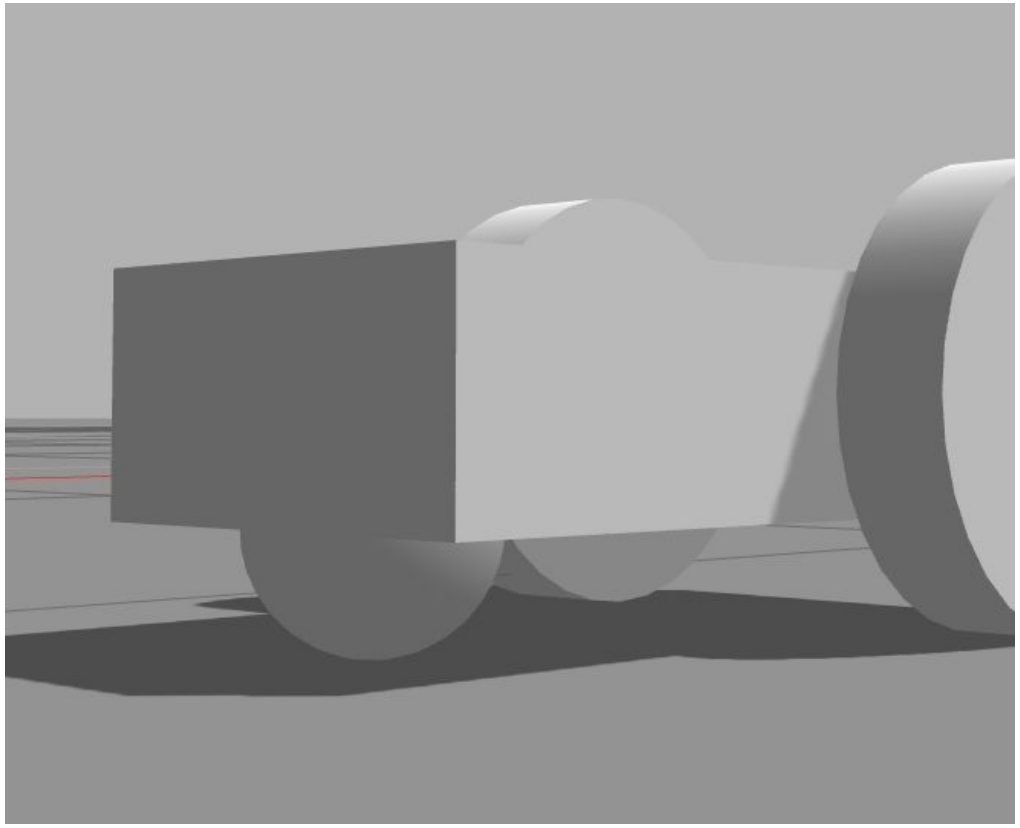
ROBOTICS

POLITECNICO
MILANO 1863

IN THE PREVIOUS EPISODE...



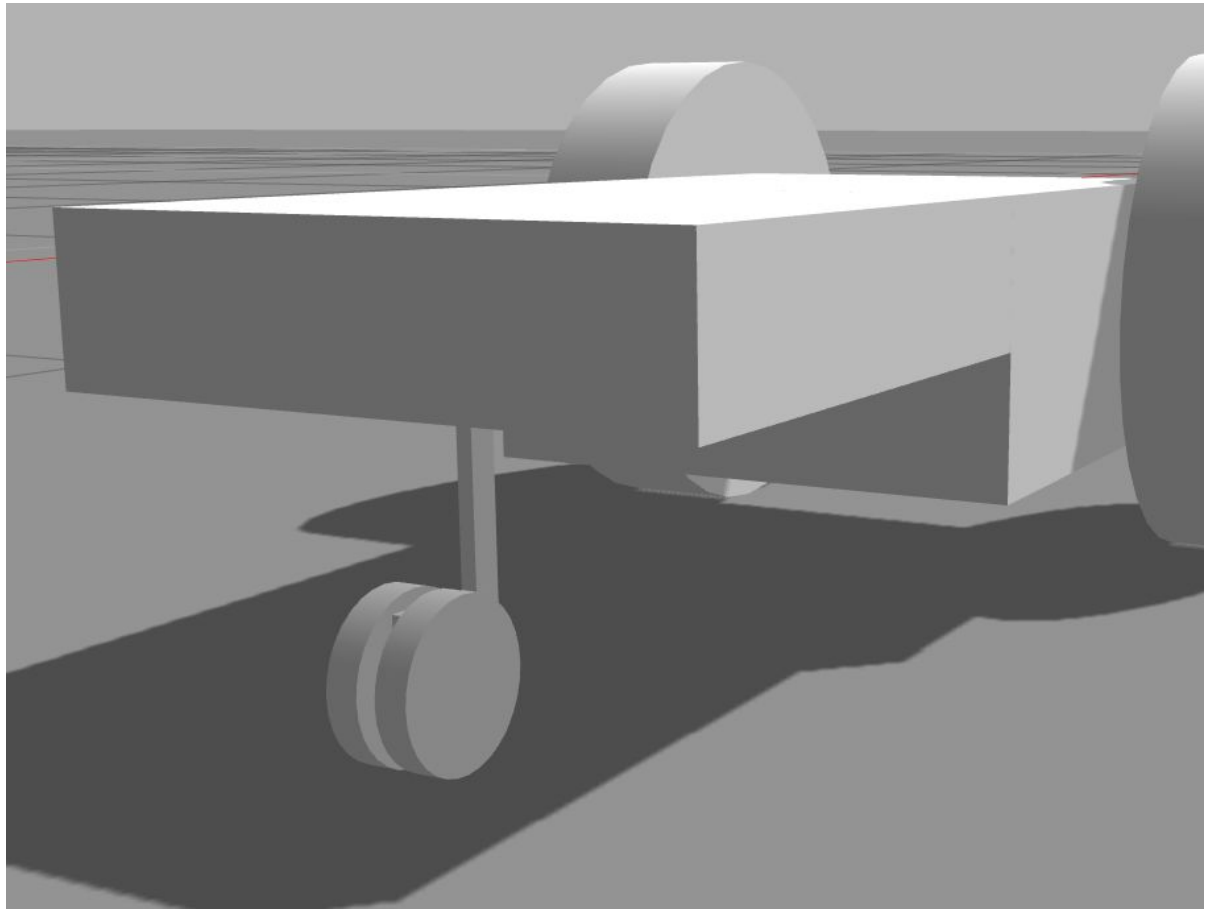
THE CASTER WHEEL



VS

IN THE SIMULATION

https://goo.gl/GonArW



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: 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, ...

TYPES OF SENSORS

Many ways to differentiate sensors: passive vs active

Passive sensors:

- use energy from the environment
- examples: temperature probe, accelerometer

Active sensors:

- Emits energy, then measure the reaction
- examples: 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 tags to define the parameters of the sensor:

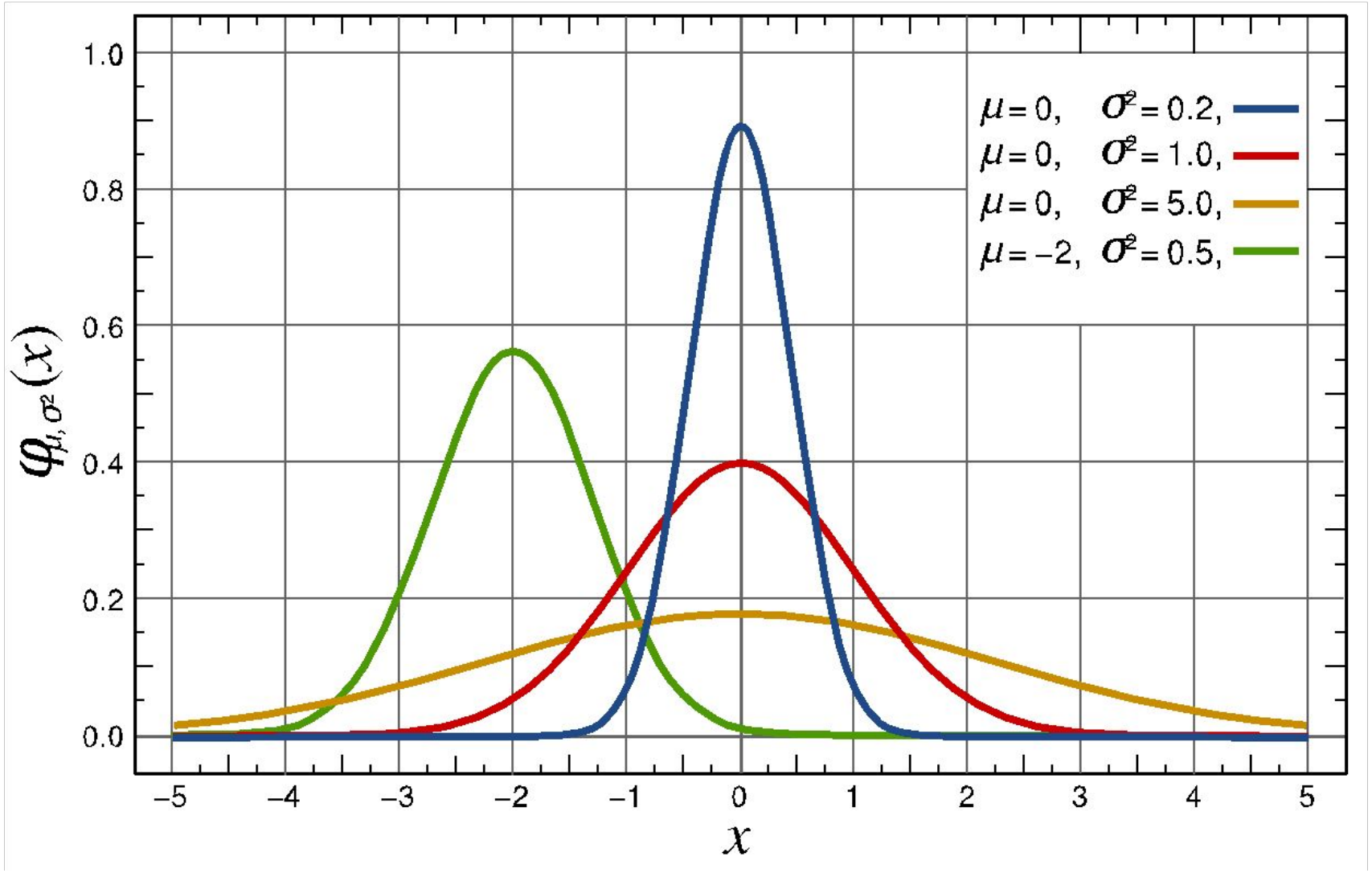
http://sdformat.org/spec?elem=sensor

Everything not on the list have to be modeled “by hand”

SENSORS IN GAZEBO

```
<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>
```

UNDERSTANDING THE GAUSSIAN NOISE



NOISY SENSORS IN GAZEBO

In gazebo most of the sensors have the option to add noise, usually defining the gaussian mean and standard deviation.

But we can only add noise, no other complex behaviour like sensor faults, latency, etc.

Still useful for fast sensor modelling

We will test a noisy laser:

create a directory for your new project: mkdir -p ~/.gazebo/models/noisy_laser

and a file model.config inside

NOISY LASER

```
<?xml version="1.0"?>
<model>
	<name>Noisy laser</name>
	<version>1.0</version>
	<sdf version='1.6'>model.sdf</sdf>

	<author>
		<name>Bob TheBuilder</name>
		<email>bob@thebuilder.net</email>
	</author>

	<description>Noisy laser.</description>
</model>
```

Next create a file model.sdf

NOISY LASER

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="noisy">
    <link name="link">
    <pose>0 0 .1 0 0 0</pose>
      <gravity>false</gravity>
      <visual name="visual">
        <geometry>
            <sphere>
            <radius>.02</radius>
          </sphere>
          </geometry>
      </visual>
```

NOISY LASER

```
<sensor name="laser" type="ray">
      <pose>0.01 0 0.03 0 -0 0</pose>
      <ray>
       <scan>
        <horizontal>
         <samples>640</samples>
         <resolution>1</resolution>
         <min_angle>-2.26889</min_angle>
         <max_angle>2.268899</max_angle>
        </horizontal>
       </scan>
       <range>
        <min>0.08</min>
        <max>10</max>
        <resolution>0.01</resolution>
       </range>
```

NOISY LASER

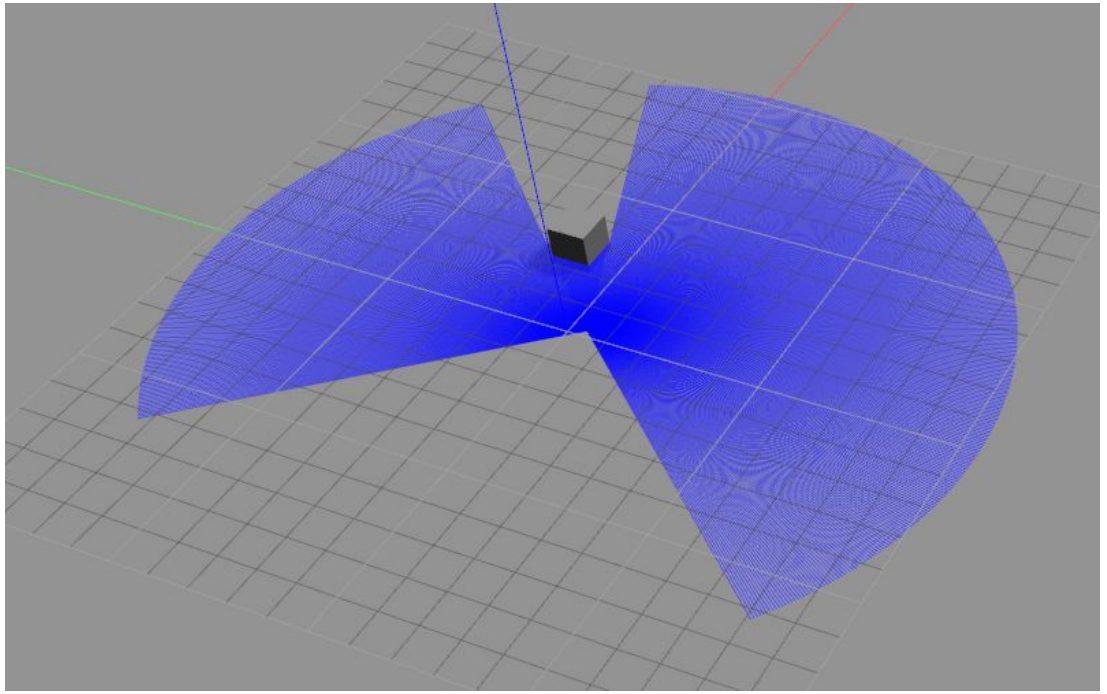
```
        <noise>
         <type>gaussian</type>
         <mean>0.0</mean>
         <stddev>0.01</stddev>
        </noise>
       </ray>
       <plugin name="laser" filename="libRayPlugin.so" />
       <always_on>1</always_on>
       <update_rate>30</update_rate>
       <visualize>true</visualize>
     </sensor>
   </link>
  </model>
</sdf>
```

NOISY LASER

Now place your noisy sensor in Gazebo and add some obstacles for the laser

Now to visualize the sensor output go to:

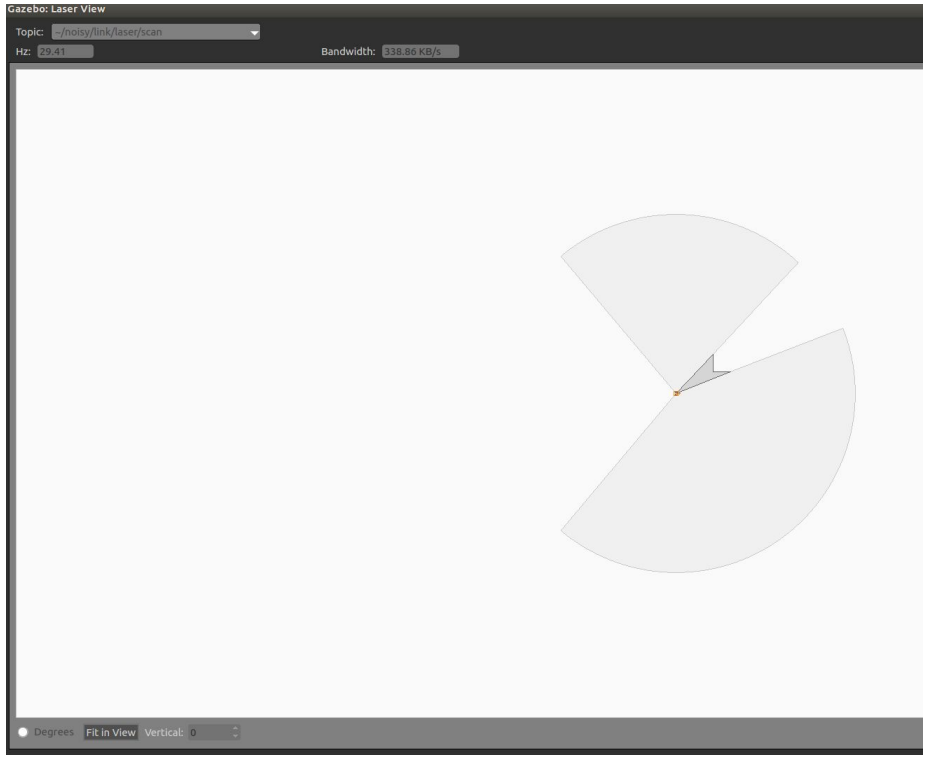
Window->Topic Visualization

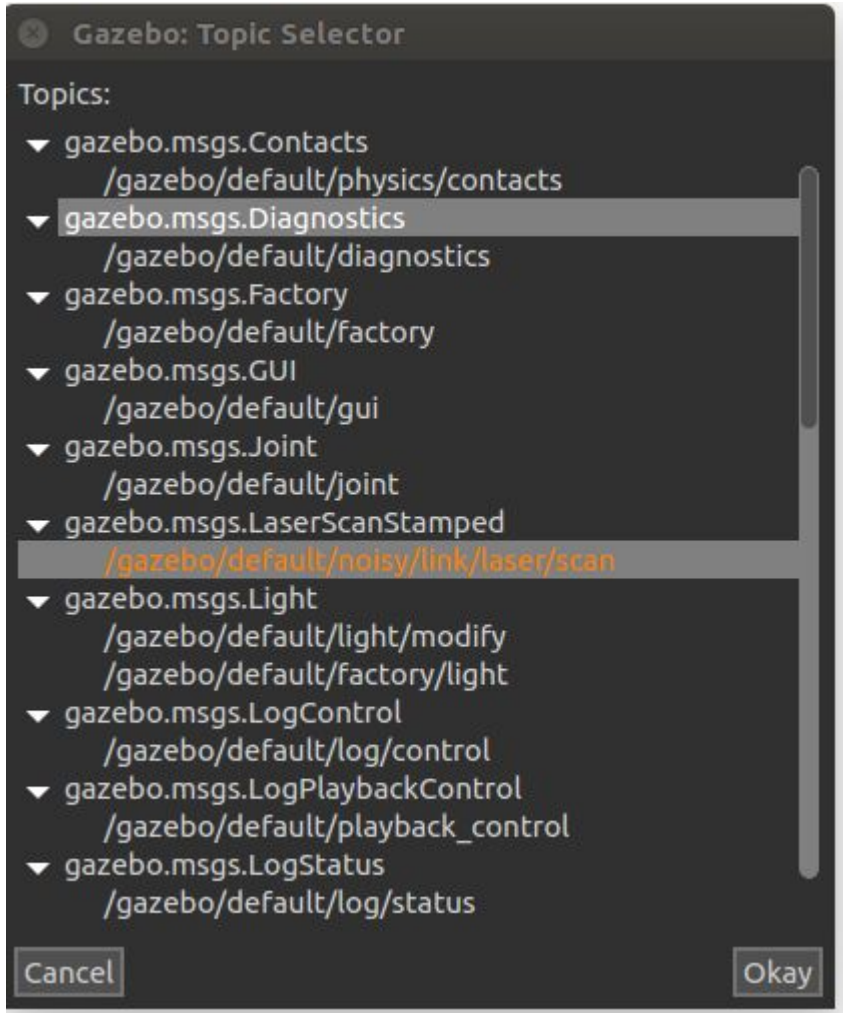


NOISY LASER

Next select your sensor topic, which will have type:
gazebo.msgs.LaserScanStamped

and click okay to visualize it



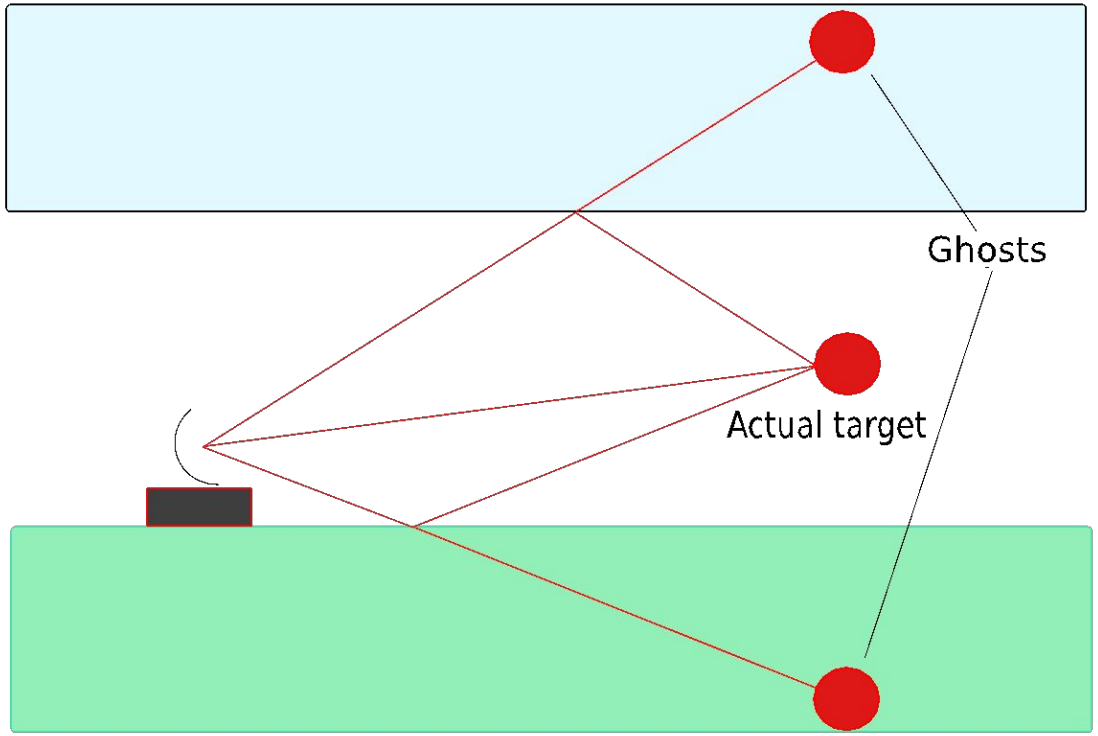


FAULTY BEHAVIOR

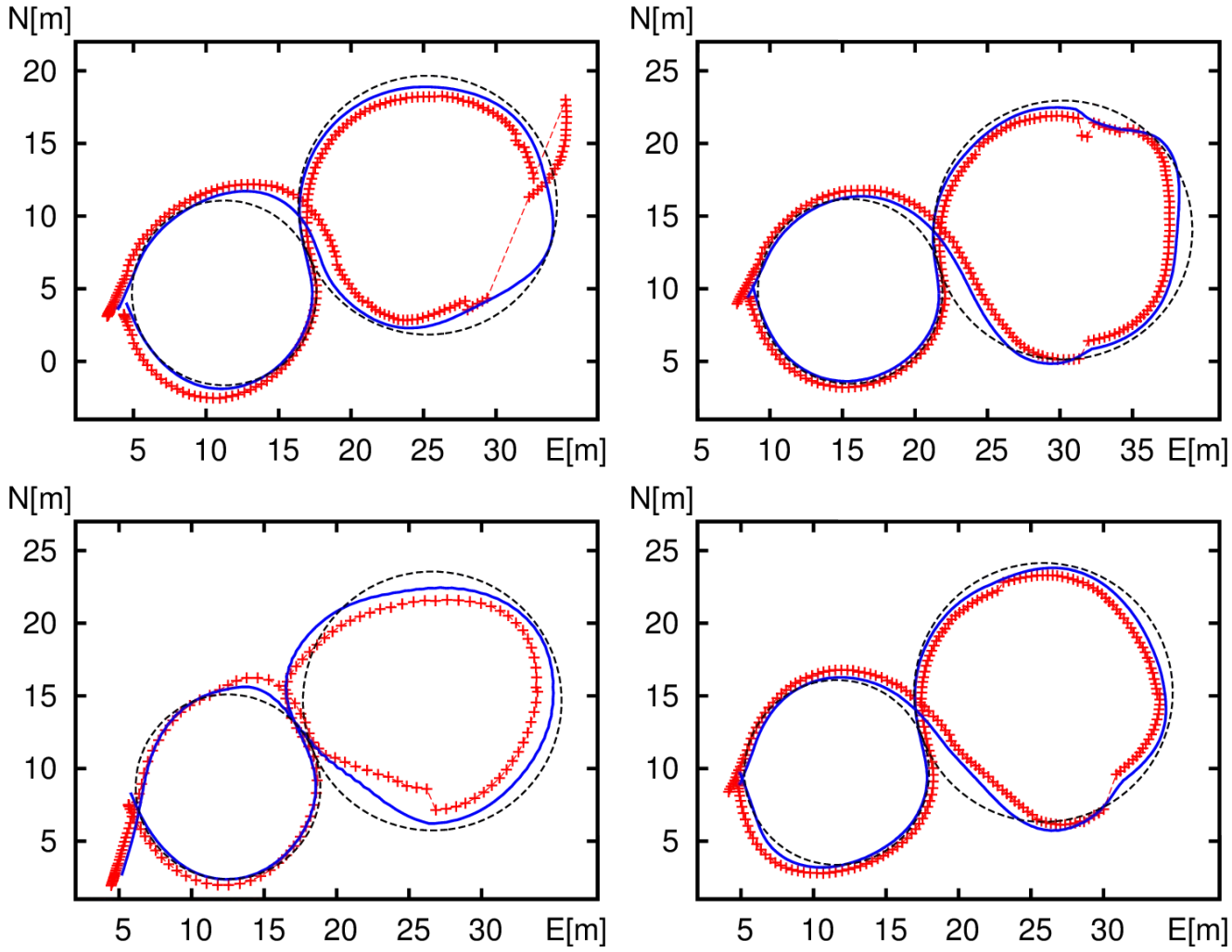
Some sensors have more than bias and noise, a common example is GPS multipath and loss of signal

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



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 PLUGIN

Create a folder where you will save the plugin

mkdir hello

cd hello

Next create a file called “hello_world.cc” and paste the plugin code

gedit hello_world.cc

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>
```

Various includes depending on the feature used (i.e. math or sensors), gazebo.hh include the core function, if you need specific features like physics, rendering, etc. you have to also include them

```
namespace gazebo {
```

Every plugin must be in the gazebo namespace

```
class HelloWorldPlugin : public WorldIPlugin {
  public: HelloWorldPlugin() : WorldIPlugin() {
    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.

The next line has different parameters for different inherited plugin space

```
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.

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

Create a folder where you will save the plugin

mkdir hello

cd hello

Next create a file called “hello_world.cc” and paste the plugin code

gedit hello_world.cc

Now we have a folder with c++ code, we have to compile it

Create a file called CMakeLists.txt and open it

gedit CMakeLists.txt

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})
list(APPEND CMAKE_CXX_FLAGS "${GAZEBO_CXX_FLAGS}")

add_library(hello_world SHARED hello_world.cc)
target_link_libraries(hello_world ${GAZEBO_LIBRARIES})
```

CMAKE AND MAKE

Now we can compile our plugin

mkdir build

cd build

cmake ..

make

The result is a library file called: libHelloWorldPlugin.so

ADDING THE PLUGIN

Next we create a minimal world file to test our plugin

gedit hello.world

```
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">
    <plugin name="hello_world" filename="libhello_world.so"/>
  </world>
</sdf>
```

RUNNING GAZEBO

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

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/path_to_plugin/build
```

Example: `export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/hello/build`

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

Now we can run the world file, calling only the gazebo server

```
$ gzserver ~/path_to_plugin/hello.world --verbose
```

Example: `$ gzserver ~/hello/hello.world --verbose`

A (little) MORE INTERESTING EXAMPLE

Create a new folder called push and a file called “model_push.cc”

This plugin is more interesting, so we have more include:

```
#include <functional>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <ignition/math/Vector3.hh>
```

A (little) MORE INTERESTING EXAMPLE

```
namespace gazebo
{
  class ModelPush : public ModelPlugin
  {
    public: void Load(physics::ModelPtr _parent, sdf::ElementPtr /*_sdf*/)
    {
      this->model = _parent;
      this->updateConnection = event::Events::ConnectWorldUpdateBegin(
          std::bind(&ModelPush::OnUpdate, this));
    }
public: void OnUpdate()
    {
          this->model->SetLinearVel(ignition::math::Vector3d(.3, 0, 0));
    }
 private: physics::ModelPtr model;

 private: event::ConnectionPtr updateConnection;
  };

  GZ_REGISTER_MODEL_PLUGIN(ModelPush)
}
```

A (little) MORE INTERESTING EXAMPLE

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

Save pointer to the model

Listen to the update event and connect a collback to the world update start signal

A (little) MORE INTERESTING EXAMPLE

```
public: void OnUpdate()
  {
      this->model->SetLinearVel(ignition::math::Vector3d(.3, 0, 0));
  }
private: physics::ModelPtr model;

private: event::ConnectionPtr updateConnection;
};

GZ_REGISTER_MODEL_PLUGIN(ModelPush)
}
```

Callback

apply linear velocity to the model

pointer to the object

A (little) MORE INTERESTING EXAMPLE

Now you can compile this new file as previously shown

create a build folder

create a CMakeLists.txt file (you can cp the previous one changing the file name)

cmake ..

make

As previously add the path of your new lib folder:

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/path_to_plugin/build
```

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/push/build
```

Next we will create a world file with an object to test the plugin

```
$ gedit model_push.world
```

A (little) MORE INTERESTING EXAMPLE

Defining our world

```
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">

    <!-- Ground Plane -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <uri>model://sun</uri>
    </include>
```

A (little) MORE INTERESTING EXAMPLE

Defining our object

```
<model name="box">
    <pose>0 0 0.5 0 0 0</pose>
    <link name="link">
        <collision name="collision">
            <geometry>
                <box>
                    <size>1 1 1</size>
                </box>
            </geometry>
        </collision>
        <visual name="visual">
            <geometry>
                <box>
                    <size>1 1 1</size>
                </box>
            </geometry>
        </visual>
    </link>
```

A (little) MORE INTERESTING EXAMPLE

Adding the plugin

```
    <plugin name="model_push" filename="libmodel_push.so"/>
  </model>
  <physics type='ode'>
      <real_time_factor>1.000000</real_time_factor>
      <real_time_update_rate>1000.000000</real_time_update_rate>
      <gravity>0.000000 0.000000 -9.800000</gravity>
    </physics>
 </world>
</sdf>
```

A (little) MORE INTERESTING EXAMPLE

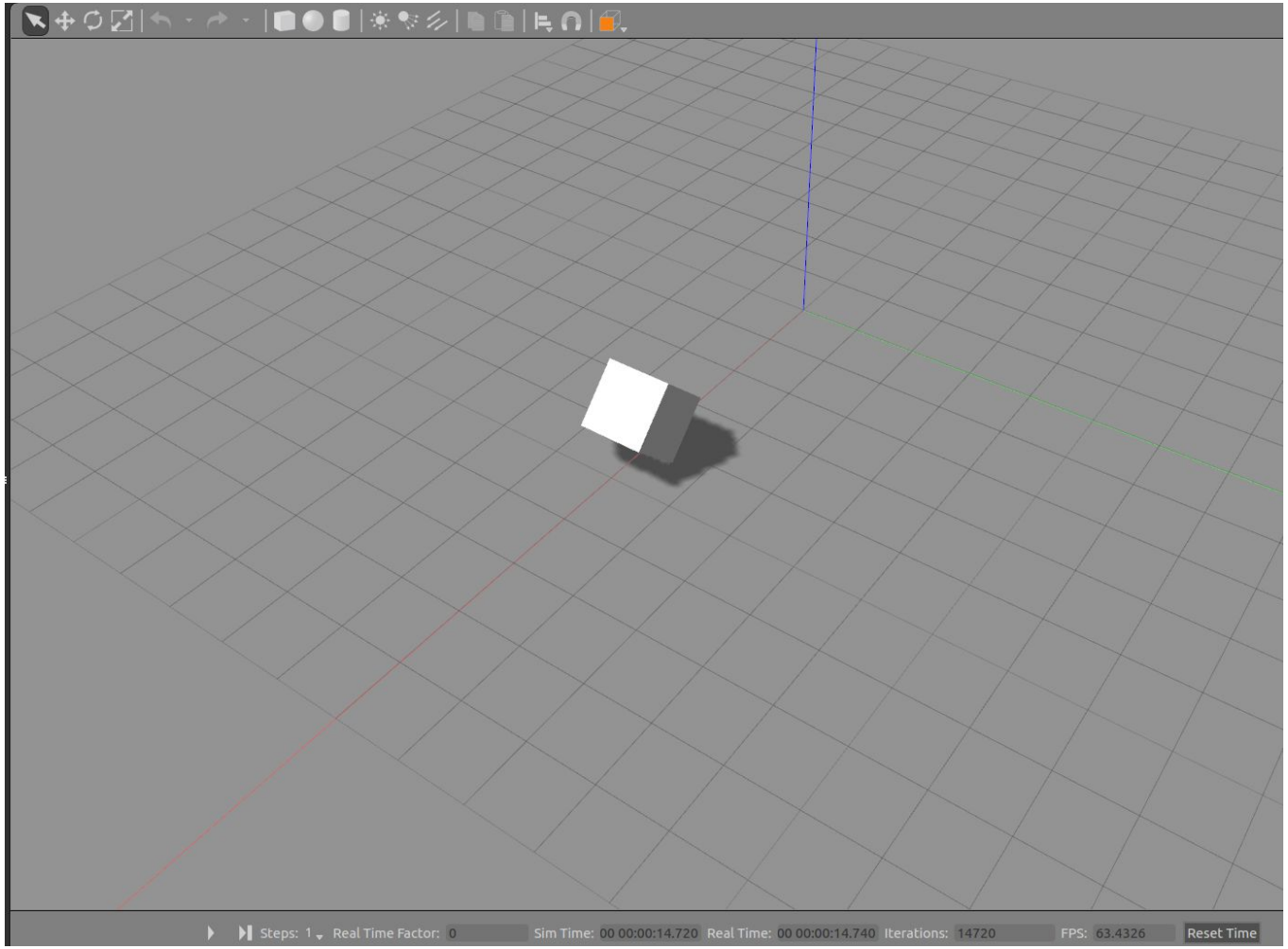
To start the simulation first call:

$ gzserver -u model_push.world

And then to start the gui:

$ gzclient

Then start the simulation pushing the play button in the bottom panel



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

FAULTY GPS (FaultyGPSPlugin.hh)

```
#ifndef _GAZEBO_FAULTY_GPS_PLUGIN_HH_
#define _GAZEBO_FAULTY_GPS_PLUGIN_HH_

#include "gazebo/common/Plugin.hh"
#include "gazebo/sensors/sensors.hh"
#include "gazebo/common/Events.hh"
#include "gazebo/math/gzmath.hh"

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;
   };
}
#endif
```

FAULTY GPS (FaultyGPSPlugin.cc)

```
#include "FaultyGPSPlugin.hh"

using namespace gazebo;

GZ_REGISTER_SENSOR_PLUGIN(FaultyGPSPlugin)

FaultyGPSPlugin::FaultyGPSPlugin() {}

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

FAULTY GPS (FaultyGPSPlugin.cc)

```
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));
}
```

FAULTY GPS (FaultyGPSPlugin.cc)

```
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);
  }
}
```

LET'S ANALYZE THE CODE

GZ_REGISTER_SENSOR_PLUGIN(FaultyGPSPlugin)

Register the plugin as a sensor plugin

This can be done everywhere in the code

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

Destructor implement a tear down routine for our pluging

Remove the connection and reset the sensor

LET'S ANALYZE THE CODE

```
this->parentSensor = std::dynamic_pointer_cast<sensors::GpsSensor>(_parent);
```

Save locally the sensor provided by the Load method

Cast it from a general sensor to a GPS

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

Raise an exception if there is any problem with the parent sensor

LET'S ANALYZE THE CODE

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

Connect the OnUpdate method to the update cycle of the sensor

The method get called each time the sensor get a new measurement

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

Connect the OnWorldUpdate method to the update cycle of the simulation

Each time a new simulation loop begins this method get called

LET'S ANALYZE THE CODE

```
if(math::Rand::GetDblUniform() > 0.1)
  _sensor->SetActive(false);
```

At each sensor update randomly shut down the sensor

```
if(!this->parentSensor->IsActive()) {
  if(math::Rand::GetDblUniform() > 0.995)
    this->parentSensor->SetActive(true);
```

At each world update randomly activate the sensor

Works only when the sensor is not active

FAULTY GPS

Now we can create the CMakeLists.txt file and compile our plugin

cmake ..

make

Remember to add your folder plugin to the global variables:

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/path_to_plugin/build
```

Now we will create our world:

$ gedit model.config

$ gedit model.world

FAULTY GPS (model.config)

```
<?xml version="1.0"?>
<model>
  <name>gps</name>
  <version>1.0</version>
  <sdf version='1.4'>model.world</sdf>

  <author>
   <name>Bob builder</name>
   <email>bob@build.it</email>
  </author>

  <description>A GPS, not a good one</description>
</model>
```

FAULTY GPS (model.world)

```
<?xml version='1.0'?>
<sdf version='1.5'>
<world name="default">
    <model name="gps">
        <static>true</static>
        <link name="link">
        <visual name='box'>
            <geometry>
                <box>
                    <size>.05 .05 .05</size>
                </box>
            </geometry>
        </visual>
```

FAULTY GPS (model.world)

```
<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>
```

FAULTY GPS (model.world)

```
        <always_on>1</always_on>
        <update_rate>10</update_rate>
        <plugin name="faulty_behavior" filename="libFaultyGPSPlugin.so"/>
      </sensor>
    </link>
  </model>
  </world>
</sdf>
```

BUT, IS IT REALLY WORKING?

Start the simulation as usual:

gzserver model.world --verbose

Then in another terminal type:

gz topic -l

to show all the messages from gazebo, then:

gz topic -e /gazebo/default/gps/link/mGps

to see the gps messages

Latitude, longitude and altitude are not constant values

gz topic -z /gazebo/default/gps/link/mGps

to see frequency

```
time {
  sec: 217
  nsec: 0
}
link_name: "gps::link"
latitude_deg: 1.8004324191014234e-06
longitude_deg: 8.007811765432573e-06
altitude: 1.6467317352071404
velocity_east: 0
velocity_north: 0
velocity_up: 0

time {
  sec: 218
  nsec: 0
}
link_name: "gps::link"
latitude_deg: 3.2542824662349013e-06
longitude_deg: 2.5973975811382929e-06
altitude: 2.264546686783433
velocity_east: 0
velocity_north: 0
velocity_up: 0

time {
  sec: 219
  nsec: 0
}
link_name: "gps::link"
latitude_deg: -5.901070439893769e-06
longitude_deg: 2.1183575684405225e-06
altitude: -4.9942365847527981
velocity_east: 0
velocity_north: 0
velocity_up: 0
```