(1a)

Gazebo

→ It is used for simulating robots.
- → Rapidly test algorithms
- → Design robot
- → Perform regression testing
- → Train AI systems using realistic scenarios.

→ Simulate population of robot in complex indoor and outdoor environments.

★ Gazebo components

① World file

→ Contains all the elements in a simulation, including robots, lights, sensors & static objects.

→ This file is formatted using SDF & typically has a .world extension.

(Simulation description format)

→ The Gazebo server (gzserver) reads this file to generate and populate a world.

② Model File

→ A model file uses the same SDF format as world files, but should contain a single <model> ... </model>.

→ The purpose of these files is to facilitate model reuse, and simplify world files.

↳ Once the model file is created, it can be included in a world file.

③ Environment Variables

→ Gazebo uses a number of environment variables to locate files, and set up communications between the server and clients.

GAZEBO_MODEL_PATH

↳ Colon-Separated set of directories where Gazebo will search for models.

GAZEBO_RESOURCE_PATH

↳ Colon-Separated set of directories where Gazebo will search for other resource such as world and media files.

GAZEBO_MASTER_URI

↳ URI of the Gazebo master (IP address & Port)

GAZEBO_PLUGIN_PATH

→ Colon-separated set of directories where Gazebo will search for the plugin shared libraries at runtime.

GAZEBO_MODEL_DATABASE_URI

→ URI of the online model database where Gazebo will download models from.

⑤ Gazebo Server

→ Server is the workhorse of Gazebo.

→ It parses a world description file given on the command line, and then simulates the world using a physics & sensors engine.

gzserver <world_filename>

Can be
→ relative to the current directory.
→ an absolute path
→ relative to a path component in GAZEBO_RESOURCE_PATH.
→ worlds/<worldname>, where <worldname> is installed with Gazebo.

⑤ Graphical Client

→ The graphical client connects to a running gz server and visualizes the elements.

→ The graphical client is run using:

```
gzclient
```

⑥ Server + Graphical Client in One

→ The `gazebo` command combines server and client in one executable.

⑦ Plugins

→ Plugins provide a simple and convenient mechanism to interface with Gazebo.

→ Plugins can either be loaded on the commandline as specified in an SDF file.

```
gz server -s <plugin_filename>
```

`-s` → indicates it is a system plugin

`<plugin_filename>` → name of a shared library found in GAZEBO_PLUGIN_PATH

→ Same mechanism is used by the graphic client

```
gzclient --gui-client-plugin <graphic_plugin>
```

`<graphic_plugin>` → {shared file}

(1b)

Overview of Gazebo Plugin

→ 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 functionalities of gazebo through the standard C++ classes.

→ Plugins are useful because they:
- → Let developers control almost any aspect of Gazebo.
- → are self-contained routines that are easily shared.
- → Can be inserted and removed from a running system.

→ You should use a plugin when you want to programmatically alter a simulation.

Ex: move models, respond to events etc..

① Plugin Types

⇒ There are currently 6 types of plugins

1. World → {Control world property}
2. Model → {Control a specific model}
3. Sensor → {To acquire Sensor info & control sensor properties}
4. System → {Control over the startup process}
5. Visual
6. GUI

② Using a Plugin

→ Once you have a plugin compiled as a shared library, you can attach it to SDF file.

→ On startup, Gazebo parses the SDF file, locates the plugin and loads the code.

---

* **Physics library**

  Open Dynamics Engine (ODE)
  Bullet
  Simbody
  Dynamic Animation & Robotics Toolkit (DART)

* **Rendering Libraries**

  OGRE

* **Sensor generation**

  Rendering Libraries & Physics Libraries.

* **GUI**

  Rendering Library Qt

Gazebo Architecture

① Introduction

⇒ Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, communication & sensor generation.

② Communication between processes

⇒ The Communication library currently uses the open source Google Protobuf for the message serialization and boost::ASIO for the transport mechanism.

⇒ It supports publisher/subscriber Communication paradigm.

③ System

* Gazebo Master
  → This is essentially a topic name ~~master~~ Server.
  → It provides name lookup, and topic management.
* Communication Library
  Protobuf and boost::ASIO

Connect to ROS (Gazebo)

(2a)

ROS Integration Overview

=> A set of ROS packages named gazebo_ros_pkgs Provides wrappers around the Stand-alone Gazebo.

gazebo_ros | gazebo_msgs | gazebo_plugins

* Launch files (↘)

(2b)

Using roslaunch to start Gazebo, world files and URDF models

Example

roslaunch gazebo_ros emptyworld.launch

Arguments → Paused
use_sim_time
gui
headless --- recording
debug
Verbose

```
<launch>
  <include file="$(find gazebo_ros)/launch
                 /empty_world.launch">
    <arg name = "world_name"
         value = "word/mud.world" />
    <arg name = "Paused"
         value="true" />
    <arg name = "use_sim_time"
         value =" true" />
    <arg name = "recording"
         value =" false" />
    <arg name = "debug"
         value="false" />
  </include>
</launch>
```

{world file you need to create}

* Creating your own Gazebo ROS package

=> Hierarchy structures for using ROS with Gazebo :-

/MYROBOT_description
- Package.xml
- CMakeLists.txt
- /urdf
  - MYROBOT.urdf
- /meshes
  - mesh1.dae
  - mesh2.dae
- /materials
- /Cad

/MYROBOT_gazebo
- /Launch
  - MYROBOT.launch
- /world
  - MYROBOT.world
- /models
  - world_object1.dae
  - world_object2.stl
  - world_object3.urdf
- /materials
- /Plugins.

★ Using roslaunch to Spawn URDF Robot

⇒ There are two ways to launch your URDF-based robot into Gazebo using roslaunch:

① Ros service call Spawn method (recommended)

→ ROS service call using a small (Python) script.

② Model Database Method

→ Allows you to include your robot within the .world file, which seems cleaner and more convenient but requires you to add your robot to the Gazebo model database by ~~finding~~ setting an environment variable.

Robot Spawn Method (Ros Service Call)

```
rosrun gazebo_ros spawn_model -file
`rospack find MYROBOT_description`/urdf/MYROBOT.urdf
-urdf -x 0 -y 0 -z 1 -model MYROBOT
```

⇒ To see all the available arguments for spawn_model

```
rosrun gazebo_ros spawn_model -h
```

⇒ Using launch file:

```
<node name="spawn_urdf" pkg="gazebo_ros"
  type="spawn_model" args="-file $(find baxter
  -description)/urdf/baxter.urdf -urdf -z 1
  -model baxter" />
```

Robot Spawn Method (Model Database)

⇒ Need to Create model.config file.

⇒ Set Environment Variable for path.

~~Using a Gazebo Depth Camera~~ with ROS

URDF in Gazebo

URDF → ROS

(Universal Robot description format)

Gazebo ← SDF

(Simulation Description Format)

=> To use a URDF file in Gazebo, some additional simulation-specific tags must be added to work properly with Gazebo.

↳ Under the hood, Gazebo will then convert the URDF to SDF automatically.

=> URDF can only specify the kinematic and dynamic properties of a single robot in isolation.

↳ Cannot specify the pose of the robot itself within the world.

⇒ To deal with this issue, a new format called the Simulation Description Format (SDF) was created for use in Gazebo to ~~store~~ solve the shortcomings of URDF.

(1) Overview of Converting to Gazebo

* **Required**

→ An <inertia> element within each <link> element must be properly specified and configured.

* **Optional**

→ Add a <gazebo> element for each <link>

→ Add a <gazebo> element for each <joint>

→ Add a <gazebo> element for the <robot> element.

→ Add a <link name="world"/> link if the robot should be rigidly attached to the world/base_link.

② The <gazebo> element

⇒ The <gazebo> element is an extension to the URDF word for specifying additional properties needed for simulation purposes in Gazebo.

⇒ There are three different types of <gazebo> elements :-

① One for <robot>

② One for <link>

③ One for <joint>

⇒ It is important that while Converting your robot to work in Gazebo, you don't break Rviz or other ROS-application functionality.

① Header of a URDF file

⇒ All you need in your robot root element tag is the name of the robot and optionally the xml namespace for xacro if you are using that.

Example

```
<robot name="mybot" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

* <gazebo> element for the tag

⇒ If a <gazebo> element is used without a reference="" property, it is assumed the <gazebo> element is for the whole robot model.

⇒ The elements for a <robot> inside the <gazebo> tag are listed in the following table

Static	bool	If set to true, the model is immovable. Otherwise the model is simulated in the dynamic engine.

* Rigidly Fixing a model to the world.

```
<link name="world"/>
<joint name="fixed" type="fixed">
    <Parent link="world"/>
    <child link="link1" />
</joint>
```

④ <gazebo> Element for Links

⇒ List of Elements that are individually parsed:

matcid	value	Matcid of Visual Element.
gravity	bool	Use gravity
dampingFactor	double	Exponential velocity decay of the link velocity.
maxVel	double	maximum contact correction velocity truncation term
manDpth	double	minimum allowable depth before Contact correction impulse is applied
mu1 mu2	double	Friction coefficients μ for the principal contact directions along the Contact surface as defined by the Open Dynamic Engine (ODE)
fdir1	string	3-tuple specifying direction of mu1 in the collision local reference frame.
Kp Kd	double	Contact stiffness Kp and damping Kd for rigid body Contacts as defined by ODE.
selfCollide	bool	If true, the link can collide with other links in the model.
maxContacts	int	Maximum number of contacts allowed between two entities.
laserRetro	double	Intensity value returned by laser sensor.

⑤ Joints

→ The <origin>, <Parent> and <child> are required.

→ <calibration> and <safety_controller> are ignored.

→ In the <dynamics> tag, only the damping property is used for gazebo and earlier. Gazebo5 and up also uses the friction property.

→ All the properties in the <limit> tag are optional.

⑥ Verifying the Gazebo Model Works

gz sdf -p MODEL.urdf

This will show you the SDF that has been generated from your input URDF as well as any warnings about missing information required to generate the SDF

Using gazebo plugins with ROS

⇒ Gazebo plugins give your URDF models greater functionality and can tie in ROS messages and service calls for (sensor output) and (motor input).

⇒ There are three plugin type that can be referenced through a URDF file:

1. Model plugin
2. Sensor plugin
3. Visual plugin

① Adding model plugin

⇒ Model plugin is inserted in the URDF inside the <robot> element. It is wrapped with the <gazebo> pill, to indicate information passed to Gazebo.

```
<robot>
  --- robot description ---
  <gazebo>
    <plugin name="differential_drive_controller"
            filename="libdiffdrive-plugin.so">
      --- plugin parameters ---
    </plugin>
  </gazebo>
  ---- robot description ---
</robot>
```

② Adding a Sensor plugin

⇒ Specifying Sensor plugin is slightly different.

⇒ Sensors are meant to be attached to links so the <gazebo> element describing the Sensor must be given a reference to the link.

```
<robot>
 --- robot description ---
  <link name="sensor_link">
   --- link description ---
  </link>

  <gazebo reference="sensor_link">
    <sensor type="camera" name="camera1">
     ---- Sensor parameters ---
      <plugin name="camera controller"
              filename="libgazebo_ros_camera.so">
       --- Plugin parameters ---
      </plugin>
    </sensor>
  </gazebo>
</robot>
```

ROS Control and Gazebo

ros_control

→ A set of packages that include controller interfaces, Controller manager, transmissions and hadware_interface.

⇒ The ros_control packages takes as input the joint state data from your robot's actuator's encoders and an input set point

↳ Uses a generic control loop feedback mechanism, to control the output.

typically effort

typically PID

⇒ The hadware interfaces are used by ROS Control in Conjunction with one of the above ROS controllers to send and receive Commands to hadware.

⇒ A transmission is an element in your ~~cost~~ control pipeline that transforms efforts/flow Variables such that their product (Power) remains Constant.

① Data flow of ros_control & Gazebo

=> Simulating a robot's controllers in Gazebo can be accomplished using ros_control and a Simple Gazebo plugin adapter.

② Add transmission elements to a URDF

=> The <transmission> element is used to link actuators to joints.

=> For the purposes of gazebo_ros_control in its current implementation, the only important information in these transmission tags are:

① <joint name=" ">

② <type>
↳ Currently only "transmission_interface/SimpleTransmission" is implemented.

③ <hardwareInterface>
↳ Within the <actuator> and <joint> tags, this tells the gazebo_ros_control plugin what hardware interface to load
↳ Currently only effort interfaces are implemented

⇒ The rest of the name and elements are currently ignored.

③ Add the gazebo_ros_control plugin

⇒ An addition to the transmission tags, a Gazebo plugin needs to be added to your URDF that actually parses the transmission tags and loads the appropriate hardware interface and controller manager.

⇒ By default the gazebo_ros_control plugin is very simple, though it is also extensible via an additional plugin architecture to allow power user to create their own custom robot hardware interface between ros_control and Gazebo.

```
<gazebo>
  <plugin name="gazebo_ros_control"
          filename="libgazebo_ros_control.so">
    <robotNamespace>/MYROBOT</robotNamespace>
  </plugin>
</gazebo>
```

⇒ The gazebo_ros_control <Plugin> tag also has the following optional child elements.

① <robotNamespace>
→ default to robot name in URDF/SDF.

② <controlPeriod>
→ The period of the controller update (in seconds), default to Gazebo's period.

③ <robotParam>
→ The location of the robot_description (URDF) on the parameter server, default to /robot_description.

④ <robotSimType>
→ The pluginlib name of a custom robot sim interface to be used defaults to "DefaultRobotHWSim".

ROS Communication

→ Gazebo provides a set of ROS API's that allows user to modify and get information about various aspects of the simulated world.

body ⟶ rigid body

model ⟶ Conglomeration of bodies connected by "joints".

gazebo_ros_api_plugin

→ It integrates the ROS callback scheduler with Gazebo's internal scheduler to Provide the ROS interfaces.

gazebo_ros_paths_plugin

→ Allows Gazebo to find ROS resources i.e. resolving ROS package pathnames.

/use_sim_time	Bool	Notifies ROS to use published /clock topic for ROS time.

① Gazebo Subscribed Topic

ⓐ ~/set_link_state

→ gazebo_msgs/LinkState

→ Sets the state (Pose/twist) of a link.

ⓑ ~/set_model_state

→ gazebo_msgs/ModelState

→ Set the state (Pose/twist of the model.

② Gazebo Published Topics

ⓐ /clock

→ rosgraph_msgs/Clock

→ Publish simulation time, to be used with /use_sim_time Parameter.

ⓑ ~/link_state

→ gazebo_msgs/LinkStates

→ Publishes states of all the links in simulation.

⑥ ~/model_states

→ gazebo_msgs/ModelStates

→ Publishes states of all the models in simulation.

---

1)

⑥

Gazebo {Build a Robot}

ⓐ Model structure and requirements

⇒ Gazebo is able to dynamically load models into simulation either programmatically or through the GUI.

⇒ Models exists on your computer, often they have been downloaded or created by you.

Models →
- Physical entity with dynamic, kinematic and visual properties.
- An addition, a model may have one or more plugins, which affects the model's behavior.
- A model can represent anything from a simple to a complex robot; even the ground is a model.

⇒ Model database repository:

https://bitbucket.org/osrf/gazebo_models

* Model database Structure

⇒ A model database must abide by a specific directory and file structure.

⇒ The root of a model ~~directory~~ database contains:

- → One directory for each model
- → database.config file with information about the model database.

⇒ Each model directory has:

- → model.config file that contains metadata about the model.
- → SDF of the model
- → any mesh, materials & plugins.

Example of typical model database directory

→ Database
- * database.config
- * model_1
  - * model.config
  - * model.sdf
  - * model.sdf.osb
  - * meshes (COLLADA & STL)
  - * materials
    - * texture (jpg, png)
    - * script (OGRE)
  - * Plugins

* Database Config

=> The database.config files is ~~not~~ only required for online repositories.

└> A directory full of models on your local computer does not need a database.config file.

* Model Config

=> Each model have a model.config file in the model's root directory that contains meta information about the model.

```
<?xml version="1.0"?>
<model>
  <name>My Model Name 2</name>
  <version>1.0</version>
  <sdf version="1.5">model.sdf</sdf>

  <author>
    <name>My name</name>
    <email>name@gmail.com</email>
  </author>

  <description>
    A description of the model.
  </description>
</model>
```

④ How to contribute a model

1. You should have an account on Bitbucket.
2. Fork and Clone the OSRf/gazebo_models repository.
3. Add your model directory to the repository.
4. Commit change and push.
5. Create a pull request.

⑤ Make a model

URDF may work with additional tags.