# Gazebo

## 虚拟机中无法运行Gazebo解决

在虚拟机中的linux上运行Gazebo的时候，Gazebo窗口闪退，并提示问题：

VMware: vmw\_ ioctl\_command error Invalid argument.

运行如下ROS+Gazebo命令也会报错：

roscore & rosrun gazebo\_ros gazebo

解决办法：

* **关闭硬件加速（亲测可用）**

echo " export SVGA\_VGPU10=0" >> ~/.bashrc

source ~/.bashrc

* 关闭虚拟机的3D图形加速（备选）

export LIBGL\_ ALWAYS\_SOFTWARE=0

## 创造实体

### Start Simulation of an Empty World

Start an empty world simulation

* roslaunch gazebo\_worlds empty\_world.launch

### Create a Simple Box URDF

Create an URDF for an object and save it as **object.urdf**:

[切换行号显示](http://wiki.ros.org/simulator_gazebo/Tutorials/SpawningObjectInSimulation)

<robot name="simple\_box">

<link name="my\_box">

<inertial>

<origin xyz="2 0 0" />

<mass value="1.0" />

<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="100.0" iyz="0.0" izz="1.0" />

</inertial>

<visual>

<origin xyz="2 0 1"/>

<geometry>

<box size="1 1 2" />

</geometry>

</visual>

<collision>

<origin xyz="2 0 1"/>

<geometry>

<box size="1 1 2" />

</geometry>

</collision>

</link>

<gazebo reference="my\_box">

<material>Gazebo/Blue</material>

</gazebo>

</robot>

Here the origins of the inertial center, visual and collision geometry centers are offset in +x by 2m relative to the model origin. The box geometry primitive is used here for visual and collision geometry, and has sizes 1m wide, 1m deep and 2m tall. Note that visual and collision geometries do not always have to be the same, sometimes you want to use a simpler collision geometry to save collision detection time when running a dynamic simulation. The box inertia is defined as 1kg mass and principal moments of inertia ixx=izz=1 kg\*m2 and iyy=100 kg\*m2.

### Spawn Model in Simulation

To spawn above URDF object at height **z** = 1 meter and assign the name of the model in simulation to be **my\_object**:

rosrun gazebo spawn\_model -file `pwd`/object.urdf -urdf -z 1 -model my\_object

# URDF/xacro

## URDF

### start with the rviz

$ roslaunch urdf\_tutorial display.launch model:=PATH\_NAME/XXX.urdf

OR

$ roslaunch urdf\_tutorial display.launch model:='$(find PACKAGE\_NAME)/PATH\_NAME/XXX.urdf'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

the “display.launch” is :

<launch>

<arg name="model" />

<arg name="gui" default="true" />

<arg name="rvizconfig" default="$(find urdf\_tutorial)/rviz/urdf.rviz" />

<param name="robot\_description" command="$(find xacro)/xacro.py $(arg model)" />

<param name="use\_gui" value="$(arg gui)"/>

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher" />

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="state\_publisher" />

<node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true" />

</launch>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HOW IT WORKS IN RVIZ:

As you move the sliders around in the GUI, the model moves in Rviz. How is this done? First the [GUI](http://wiki.ros.org/joint_state_publisher) parses the URDF and finds all the non-fixed joints and their limits. Then, it uses the values of the sliders to publish [sensor\_msgs/JointState](http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html) messages. Those are then used by [robot\_state\_publisher](http://wiki.ros.org/robot_state_publisher) to calculate all of transforms between the different parts. The resulting transform tree is then used to display all of the shapes in Rviz.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Practical Usage

<?xml version="1.0"?>

<robot name="physics">

<material name="blue">

<color rgba="0 0 0.8 1"/>

</material>

<material name="black">

<color rgba="0 0 0 1"/>

</material>

<material name="white">

<color rgba="1 1 1 1"/>

</material>

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

<material name="blue"/>

</visual>

<collision>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</collision>

<inertial>

<mass value="10"/>

<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>

</inertial>

</link>

<link name="right\_leg">

<visual>

<geometry>

<box size="0.6 0.1 0.2"/>

</geometry>

<origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>

<material name="white"/>

</visual>

<collision>

<geometry>

<box size="0.6 0.1 0.2"/>

</geometry>

<origin rpy="0 1.57075 0" xyz="0 0 -0.3"/>

</collision>

<inertial>

<mass value="10"/>

<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>

</inertial>

</link>

<joint name="base\_to\_right\_leg" type="fixed">

<parent link="base\_link"/>

<child link="right\_leg"/>

<origin xyz="0 -0.22 0.25"/>

</joint>

…....

</robot>

#### mesh

The meshes can be imported in a number of different formats. STL is fairly common, but the engine also supports DAE, which can have its own color data, meaning you don’t have to specify the color/material. Often these are in separate files. These meshes reference the .tif files also in the meshes folder.

format: package://NAME\_OF\_PACKAGE/path notation

eg.

<link name="left\_gripper">

<visual>

<origin rpy="0.0 0 0" xyz="0 0 0"/>

<geometry>

<mesh filename="package://urdf\_tutorial/meshes/l\_finger.dae"/>

</geometry>

</visual>

</link>

#### Movable Robot Model

four types of joints: fixed, continuous, revolute and prismatic.

#### fixed

can't move

#### continuous

It can take on any angle from negative infinity to positive infinity.

The only additional information we have to add is the axis of rotation, here specified by an xyz triplet, which specifies a vector around which the head will rotate. Since we want it to go around the z axis, we specify the vector "0 0 1".

eg.

<joint name="head\_swivel" type="continuous">

<parent link="base\_link"/>

<child link="head"/>

<axis xyz="0 0 1"/>

<origin xyz="0 0 0.3"/>

</joint>

#### revolute

This means that they rotate in the same way that the continuous joints do, but they have strict limits. Hence, we must include the limit tag specifying the upper and lower limits of the joint (in radians). We also must specify a maximum velocity and effort for this joint but the actual values don't matter for our purposes here.

eg.

<joint name="left\_gripper\_joint" type="revolute">

<axis xyz="0 0 1"/>

<limit effort="1000.0" lower="0.0" upper="0.548" velocity="0.5"/>

<origin rpy="0 0 0" xyz="0.2 0.01 0"/>

<parent link="gripper\_pole"/>

<child link="left\_gripper"/>

</joint>

#### prismatic

This means that it moves along an axis, not around it.

eg.

<joint name="gripper\_extension" type="prismatic">

<parent link="base\_link"/>

<child link="gripper\_pole"/>

<limit effort="1000.0" lower="-0.38" upper="0" velocity="0.5"/>

<origin rpy="0 0 0" xyz="0.19 0 0.2"/>

</joint>

#### other

There are two other kinds of joints that move around in space. Whereas the prismatic joint can only move along one dimension, a planar joint can move around in a plane, or two dimensions. Furthermore, a floating joint is unconstrained, and can move around in any of the three dimensions. These joints cannot be specified by just one number, and therefore aren’t included in this tutorial.

### Adding Physical and Collision Properties to a URDF Model

#### Collision

eg.

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

<material name="blue">

<color rgba="0 0 .8 1"/>

</material>

</visual>

<collision>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</collision>

</link>

In many cases, you’ll want the collision geometry and origin to be exactly the same as the visual geometry and origin. However, there are two main cases where you wouldn’t.

Quicker Processing - Doing collision detection for two meshes is a lot more computational complex than for two simple geometries. Hence, you may want to replace the meshes with simpler geometries in the collision element.

Safe Zones - You may want to restrict movement close to sensitive equipment. For instance, if we didn’t want anything to collide with R2D2’s head, we might define the collision geometry to be a cylinder encasing his head to prevent anything from getting to near his head.

#### Inertia

eg.

<link name="base\_link">

<visual>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

<material name="blue">

<color rgba="0 0 .8 1"/>

</material>

</visual>

<collision>

<geometry>

<cylinder length="0.6" radius="0.2"/>

</geometry>

</collision>

<inertial>

<mass value="10"/>

<inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.4" iyz="0.0" izz="0.2"/>

</inertial>

</link>

This information can be provided to you by modeling programs such as [MeshLab](http://wiki.ros.org/MeshLab). The inertia of geometric primitives (cylinder, box, sphere) can be computed using Wikipedia's [list of moment of inertia tensors](http://en.wikipedia.org/wiki/List_of_moment_of_inertia_tensors) (and is used in the above example).

The inertia tensor depends on both the mass and the distribution of mass of the object. A good first approximation is to assume equal distribution of mass in the volume of the object and compute the inertia tensor based on the object's shape, as outlined above.

If unsure what to put, a matrix with ixx/iyy/izz=1e-3 or smaller is often a reasonable default for a mid-sized link (it corresponds to a box of 0.1 m side length with a mass of 0.6 kg). Although often chosen, the identity matrix is a particularly bad default, since it is often much too high (it corresponds to a box of 0.1 m side length with a mass of 600 kg!).

You can also specify an origin tag to specify the center of gravity and the inertial reference frame (relative to the link's reference frame).

When using realtime controllers, inertia elements of zero (or almost zero) can cause the robot model to collapse without warning, and all links will appear with their origins coinciding with the world origin.

## Xacro

### how to use the xacro

convert Xacro to URDF:

rosrun xacro xacro model.xacro > model.urdf

### grammer

#### begin

At the top of the URDF file, you must specify a namespace in order for the file to parse properly. For example, these are the first two lines of a valid xacro file:

<?xml version="1.0"?>

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

#### Constants

eg.

<xacro:property name="width" value="0.2" />

<xacro:property name="bodylen" value="0.6" />

<link name="base\_link">

<visual>

<geometry>

<cylinder radius="${width}" length="${bodylen}"/>

</geometry>

<material name="blue"/>

</visual>

<collision>

<geometry>

<cylinder radius="${width}" length="${bodylen}"/>

</geometry>

</collision>

</link>

eg.

<xacro:property name=”robotname” value=”marvin” />

<link name=”${robotname}s\_leg” />

#### math

eg.

<cylinder radius="${wheeldiam/2}" length="0.1"/>

<origin xyz="${reflect\*(width+.02)} 0 0.25" />

#### macro

##### simple macro

eg.

<xacro:macro name="default\_origin">

<origin xyz="0 0 0" rpy="0 0 0"/>

</xacro:macro>

<xacro:default\_origin />

##### parameterized macro

1. The parameters act just like properties, and you can use them in expressions .

eg.

<xacro:macro name="default\_inertial" params="mass">

<inertial>

<mass value="${mass}" />

<inertia ixx="1.0" ixy="0.0" ixz="0.0"

iyy="1.0" iyz="0.0"

izz="1.0" />

</inertial>

</xacro:macro>

1. You can also use entire blocks as parameters too.

eg.

<xacro:macro name="blue\_shape" params="name \*shape">

<link name="${name}">

<visual>

<geometry>

<xacro:insert\_block name="shape" />

</geometry>

<material name="blue"/>

</visual>

<collision>

<geometry>

<xacro:insert\_block name="shape" />

</geometry>

</collision>

</link>

</xacro:macro>

<xacro:blue\_shape name="base\_link">

<cylinder radius=".42" length=".01" />

</xacro:blue\_shape>