How to Create a New Model of a Mobile Robot in ROS/Gazebo Environment: An Extended Tutorial

Alexandra Dobrokvashina [1]*, Roman Lavrenov [1], Evgeni Magid [1, 2], Yang Bai [3], and Mikhail Svinin [3]

[1] Intelligent Robotics Department, Kazan Federal University, Kazan, Russia; Email: lavrenov@it.kfu.ru (R.L.)
[2] HSE University, Moscow, Russia; Email: magid@it.kfu.ru (E.M.)
[3] Information Science and Engineering Department, Ritsumeikan University, Kyoto, Japan;
Email: {yangbai, svinin}@fc.ritsumei.ac.jp (Y.B., M.S.)
*Correspondence: dobrokvashina@it.kfu.ru (A.D.)

***Abstract*—With a growth of a main stream simulation tools variety and virtual experiments popularity in a role of a first R&D stage in robotics, researchers faced a need to model their own robotic platforms. Edge cutting simulators for mobile robots, e.g., Gazebo, Webots and CoppeliaSim, contain a limited number of well documented robot models, which were constructed by robots' manufacturers or associated research groups. Yet, to create a new model of a complex robot is not an easy task for a beginner. This tutorial paper describes a step-by-step process of creating of a new robot model in the Gazebo simulator. The process starts from the model construction and physics setup, and ends up with sensors, Robot Operating System (ROS) based control integration and creating a ROS-project with the model. The process is illustrated with an example of a crawler-type robot Servosila Engineer. The example is supplied with an open source code of the ROS-packages that are via a public Gitlab repository.**

***Keywords*—modelling, simulation, Gazebo, ROS, tutorial**

I. Introduction

Nowadays a number of robotic applications, their hardware and software complexity increase every single day, which allows to automate dangerous and repetitive processes [1]. Robots plays important roles in manufacturing [2], entertainment [3] and service [4], search and rescue [5], transportation [6], human-robot interaction [7], medicine [8] and healthcare [9].

Prior to integration of new approaches and algorithms into control systems of real robots, typically, they are tested in simulators. Virtual experiments in simulators became a fundamental part of research activities since easy created and reproducible complicated physical setups for testing reduce valuable time and resource spendings. The usefulness of simulation became the reason for companies to create simulation models alongside with real robots. Unfortunately, not all manufacturers provide a proper simulation model for their robots, if any, which forces researchers to create such models on their own.

Figure 1. Servosila Engineer robot at Laboratory of Intelligent Robotic Systems, Intelligent Robotics Department, Institute of Information Technology and Intelligent Systems, Kazan Federal University.

This paper presents a tutorial on creating a simulation model of a real (existing) robot. It contains information about modelling parts of the robot, shows different ways to cope with collision meshes problem, explains a procedure of controllers and sensors integration. The model was constructed in the Gazebo simulation environment and employs Robot Operating System (ROS, [10]) for control purposes. All modelling steps are illustrated using our project of Servosila Engineer modelling [11], [12], which is a crawler mobile robot (Fig. 1) produced by Russian company Servosila [13].

II. Instruments

This article presents a step-by-step simulation process using ROS/Gazebo environment. ROS is a broad set of tools and libraries packed as a framework that is used for robot application, both for real robots and simulations, which are widely used by leading robotics companies, e.g., such as PAL robotics [14] or Robotis [15]. Gazebo is a robot simulator, integrated with ROS [16]. A vast majority of robot simulation models and plugins with

Manuscript received October 27, 2022; revised December 14, 2022; accepted March 24, 2023.

ROS were created for the Gazebo simulator. RViz is used together with the Gazebo as a ROS visualizer for data that a robot receives from its sensors [17].

To work with a 3D model, we recommend using Blender software [18]. It is a free 3D computer graphics software, which is used for modelling, animation, and computer games. It appears to be quite popular and useful for scientific research, visualization and modelling [19].

III. Creating a Robot Model

Few steps should be done before creating a robot model. A physically realistic model construction requires reliable data about the robot, including dimensions of robot links and their weights. Another important element for a robot description are default hardware and software limits, e.g., joint limits provide information about a workspace of a manipulator in real life.

A. Visual Meshes

The first step is creating a reliable visual 3D model of a robot. Often, a CAD model of the robot could be obtained from a manufacturer as these models are used at robot design and production stages. Such model could be used as visual meshes. Otherwise, it should be created manually, which is a time-consuming procedure that requires experience and execution of a full-stack modelling process. Physical accuracy is ensured solely by a good CAD model and documentation (in the first case) or thorough measurements (in the second). For multiple reasons, it could be discovered that a manufacturers' CAD model does not precisely correspond to a real robot, and it is a responsibility of a modelling designer to verify measurements and update the CAD model accordingly.

Modelling could be done using any popular 3D engine, such as Maya [20], Blender [21], 3DsMax [22] etc. With some efforts, using an existing software, a model could be transferred from one file format (associated with a particular file extension) to another.

B. Collision Meshes

Calculating physics of a visual model using only its geometry could be quite efficient. For this reason, simulators require additional meshes for every link of a robot, called a *collision mesh*. It is a mesh that is maximally simplified relatively to a visual mesh. There are two methods to create the collision mesh: *generating* models from visual meshes with automatic tools or *creating* models manually.

Automatic generation of collision meshes is an easy and fast solution. It suits for research teams that do not have a qualified 3D modelling specialist or are severely limited in time. There exist a large variety of graphical applications that provide users an ability of an automatic polygon decimation. In our case, Blender open-source solution was employed. *Decimate* function is released in Blender as one of available modifiers. First, the model is imported using *File-Import-(type of file with your model)* tab. Then, if the model is complicated and contains multiple parts, it is recommended to decide which parts could be deleted (for example, small-size pins, insignificant elements of a decor or inner elements). Each remaining part of the model should be supplied with a corresponding *Decimate* modifier. Modifiers appear in a right menu shown in the Fig. 2. Using parameters of this modifier the model could be significantly simplified. Other instruments that could be helpful for this task, are *ProOptimizer* modifier in 3DsMax or *Mesh-Reduce* option in Maya.

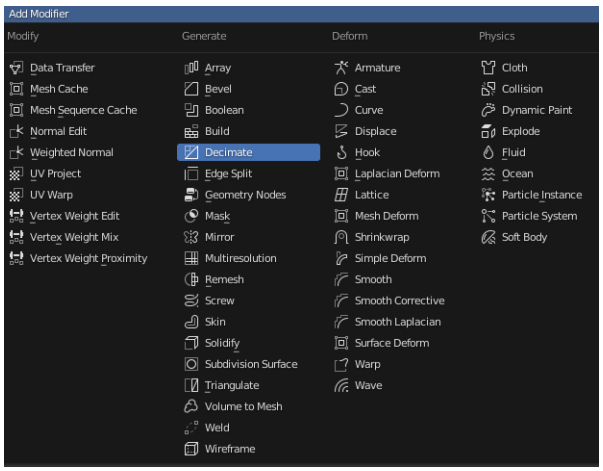


Figure 2. List of modifier options in Blender.

The second approach for collision meshes construction is a **manual creation** of required models. This option requires some expertise in 3D modelling. Creating collision meshes in most cases means covering a visual model with a new mesh while excluding small details and keeping only main geometry of objects. The expertise and experience are important in order to decide which details of the original model could be omitted.

Fig. 3 demonstrates three different models. The one on the left is a visual model of a front sub-crawler (flipper) of the crawler-type robot Servosila Engineer. The others are collision meshes, which were created using different options. The one in the center was created manually [11] and the one on right was generated automatically [23]. Table I presents a comparison of the two approaches. When possible, we strongly recommend a manual modelling.

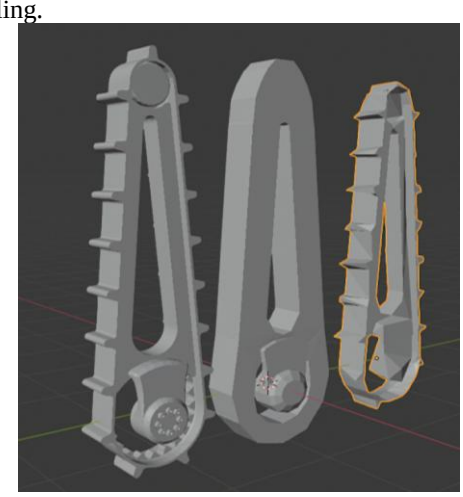


Figure 3. Mesh examples of the Servosila Engineer robot's front sub-crawler: an original mesh (left), a simplified manually created mesh (center), an automatically generated mesh (right).

TABLE I. CREATING COLLISION MESHES METHODS COMPARISON

Comparation criteria	Mode	
	Manual	Automatic
Time consumption	Low	High
Quality	High	Low (most cases)
Optimization	High	Low (most cases)
Modelling skills requirements	High	Low

IV. BUILDING A ROBOT

A. Writing Description File

Unified Robot Description Format (URDF) is a main instrument for a robot description in ROS/Gazebo environment. One of the URDF realizations is Xacro (XML Macros) that became popular among users because of several improvements such as parametrization and macros. It makes a description more readable and easier to construct. The Xacro description could be valuable for a large and complicated project. Moreover, it significantly decreases a size of a file.

Each element of the robot should be properly described. Information about links' length and joint limits allows reaching a good level of similarity between a simulation model and its real world counterpart. A part of a XACRO file code[1] that describes the Servosila Engineer robot is presented in Code 1; it contains a description of two links of the manipulator – a waist link (lines 1-18) and a shoulder (lines 28-48) link – and a shoulder joint (lines 50-58) between them.

```
1 : <!-- waist -->
2 : <link name="waist_link">
3 : <visual>
4 : <geometry>
5 : <mesh filename="package://(path-to-the-directory-with-visual-
6 : meshes)/cronstain.dae"/>
7 : </geometry>
8 : </visual>
9 : <collision>
10: <geometry>
11: <mesh filename="package://(path-to-the-directory-with-collision-
12: meshes)/Cronstain.dae"/>
13: </geometry>
14: <xacro:cuboid_inertia mass="${waist_mass}" length="0.08"
width="0.08"
15: height="0.08">
16: <origin xyz="0 0 0" rpy="0 0 0"/>
17: </xacro:cuboid_inertia>
18: </link>
19:
20: <gazebo reference="waist_link">
21: <selfCollide>false</selfCollide>
22: <kp>${kp}</kp>
23: <kd>${kd}</kd>
24: <mu1>100</mu1>
25: <mu2>50</mu2>
26: </gazebo>
27:
28: <!-- shoulder -->
29: <link name="shoulder_link">
30: <visual>
31: <geometry>
32: <mesh filename="package://(path-to-the-directory-with-visual-
33: meshes)/shoulder.dae"/>
34: </geometry>
35: </visual>
36: <collision>
37: <geometry>
38: <mesh filename="package://(path-to-the-directory-with-collision-
39: meshes)/Shoulder.dae"/>
40: </geometry>
41: </collision>
42: <inertial>
43: <mass value="${shoulder_mass}"/>
44: <origin xyz="-0.0303 -0.0001 0.1511" rpy="0 0 0"/>
45: <inertia ixx="0.0295383" ixy="-0.0000001" ixz="-0.0004068"
46: iyy="0.0292352" iyz="0.0000004" izz="0.0011211"/>
47: </inertial>
48: </link>
49:
50: <joint name="shoulder" type="revolute">
51: <parent link="waist_link"/>
52: <child link="shoulder_link"/>
53: <axis xyz="1 0 0"/>
54: <dynamics friction="${friction}" damping="${damping}"/>
55: <origin xyz="0.036 0.051 -0.07" rpy="${pi} 0 0"/>
56: <limit lower="${shoulder_llimit}" upper="${shoulder_ulimit}"
57: effort="${shoulder_mass * 50}" velocity="${joints_vlimit}"/>
58: </joint>
59:
60: <gazebo reference="shoulder_link">
61: <selfCollide>false</selfCollide>
62: <kp>${kp}</kp>
63: <kd>${kd}</kd>
64: <mu1>100</mu1>
65: <mu2>50</mu2>
66: </gazebo>
```

Code 1. Code listing of the file *engineer_arm.xacro*.

A description of every link of the robot contains paths to its visual and collision meshes and an inertial unit, which includes information of weight and inertia. Joints have information about links they connect, a rotational axis and an origin, friction data and limits of position and velocity.

B. Setting up Inertia

Inertias are one of the most significant parts of a robot link description. Incorrectly tuned inertia can make a model unrealistic and even destroy it. An example of improperly tuning of the Servosila Engineer robot inertias that caused incorrect model behavior after its spawning in a Gazebo world is shown in Fig. 4. A proper visual model of the robot initially appeared at a predefined height of a Gazebo world 3D space and under gravitation force glided down; upon its contact with a ground plane the parts of the model felt apart.

An inertial block describes a mass of a link, its center of mass (6 coordinates) and a matrix of inertia tensors. The inertial block code example appears in Fig. 4, lines 42–47.

[1] A full version of this file and other support files with comments to each command line could not be included into the paper due to space limitations and are available in [24]

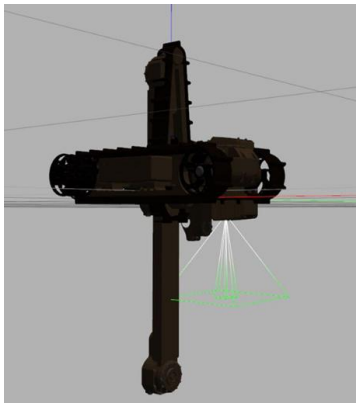
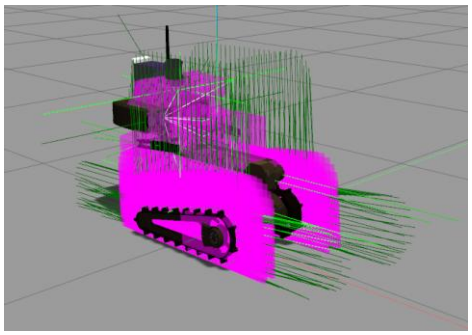


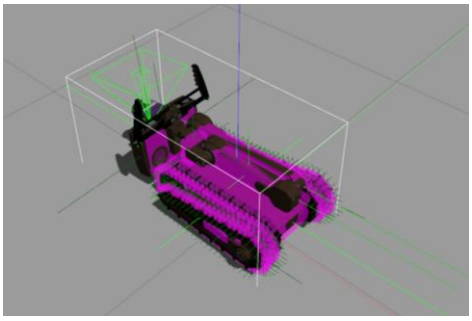
Figure 4. The robot model felt into parts due to wrong inertia settings.

Fig. 5 presents two attempts of setting the inertial block of the Servosila Engineer robot model in Gazebo. Properly tuned inertia should have a shape, which is maximally close to an object it is attached to. In Fig. 5(a) we can note that inertia of the robot head (magenta color) nearly three times exceeds the original parts of the robot (brown and black colors). Fig. 5(b) presents corrected inertial data. In addition, while setting up inertia, it is important to remember about weights (mass) that do not have visual parts; the weights should be set according to the real robot technical information

There are several ways to set up inertial blocks. First of all, inertias could be calculated using precise measurements and standard formulas. This option might be laborious, especially for complicated models. Another solution is to select parameters using visualization in Gazebo (Fig. 6). It could be faster than the first option but still it takes time and optimality of this method is quite questionable.



(a)



(b)

Figure 5. Inertial blocks for Servosila Engineer model are shown with magenta color.

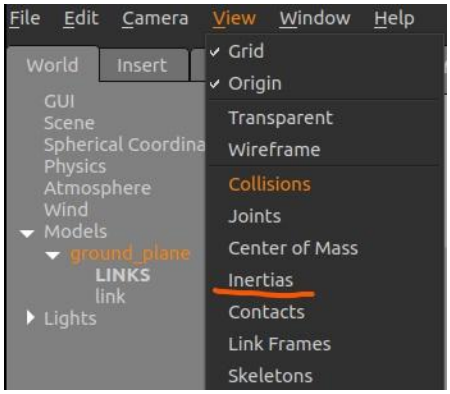


Figure 6. Inertias option tab of View menu in Gazebo simulator.

The third option, which we recommend, is using a corresponding software, e.g., MeshLab [25]. It allows to easily calculate inertia values within three steps:

- Import a model using *File-Import Mesh* tab
- Open a console for a log output with *View-Show Layer Dialog*
- Calculate inertia value with option *Filters-Quality Measure and Computations-Compute Geometric Measures*

Table II compares the three options of setting up inertial blocks by their time consumption and resulting quality.

TABLE II. INERTIA SETTING METHODS COMPARISON

Comparison criteria	***Approach***		
	Inertia formulas	Visual approximation	Software
Time consumption	Low	Low (most cases)	High
Quality	High	Low (most cases)	High

C. Adding ROS Controllers

Controllers are used for moving a robot within a simulation. They are connected to model joints and move them according to given commands. Adding controllers contains three steps:

1) Add transmissions to every moving joint. The transmission contains information about a type of a joint, an interface, an actuator and a name of a joint it is connected to. An example of code with a transmission description is listed in Code 2.

```
01: <transmission name="waist_shoulder_trans">
02: <type>transmission_interface/SimpleTransmission</type>
03: <actuator name="waist_shoulder_motor">
04: <mechanicalReduction>1</mechanicalReduction>
05: </actuator>
06: <joint name="shoulder">
07: <hardwareInterface>
08: hardware_interface/EffortJointInterface
09: </hardwareInterface>
10: </joint>
11: </transmission>
```

Code 2. Code listing of the file *engineer_arm.xacro*.

2) Create a YAML file that contains parameters of the controllers. It contains a controller type, a joint name and PID parameters. An example of such description is listed in Code 3.

```
1: shoulder_position_controller:
2: type: effort_controllers/JointPositionController
3: joint: shoulder
4: pid: {p: 100.0, i: 0.01, d: 10.0}
```

Code 3. Code listing of the file *engineer_control.yaml*.

3) Create a launch file for the controllers. It should contain a loader of a controllers' list created in the previous step and a launcher of the controllers. An example of a launch file is listed in Code 4.

```
1: <rosparam file=
2: "$(find engineer_control)/config/engineer_control.yaml"
command="load"/>
3: <node name="controller_spawner" pkg="controller_manager"
type="spawner" respawn="false" output="screen"
args="shoulder_position_controller"/>
```

Code 4. Code listing of the file *engineer_control.launch*.

Finally, after launching the model and its controllers, several ROS-topics are required to control the robot in a simulated environment. For example, to control *shoulder* joint there is a topic named *shoulder position controller* (refer the code in Code 1). It works with a message of *std/msgsFloat64* type. To send a command to the controller next command is used:

```
rostopic pub -1 \/shoulder\_position\_controller std\_msgs\ Float64
"data: 0.5"
```

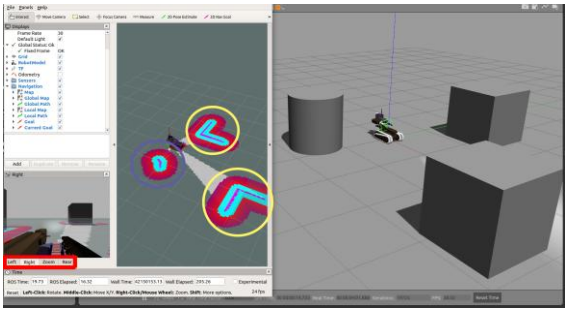
```
01: <gazebo reference="camera${number}_link"> <sensor
type="camera" name="camera_${number}">
<update_rate>${fps}</update_rate>
02: <camera name="head_${number}">
<horizontal_fov>1.3962634</horizontal_fov> <image>
03: <width>${width}</width>
04: <height>${height}</height>
05: <format>R8G8B8</format>
06: </image>
07: <clip>
08: <near>0.02</near>
09: <far>300</far>
10: </clip>
11: <noise>
12: <type>gaussian</type>
13: <mean>0.0</mean>
14: <stddev>0.007</stddev>
15: </noise>
16: </camera>
17: <plugin name="camera${number}_controller"
18: filename="libgazebo_ros_camera.so">
19: <alwaysOn>true</alwaysOn>
20: <updateRate>0.0</updateRate>
21: <cameraName>camera${number}</cameraName>
22: <imageTopicName>image${number}_raw</imageTopicName>
23: <cameraInfoTopicName>camera_info</cameraInfoTopicName>
24: <frameName>camera${number}_link_optical</frameName>]
25: <hackBaseline>0.0</hackBaseline>
26: <distortionK1>0.0</distortionK1>
27: <distortionK2>0.0</distortionK2>
28: <distortionK3>0.0</distortionK3>
29: <distortionT1>0.0</distortionT1>
30: <distortionT2>0.0</distortionT2>
31: <CxPrime>0</CxPrime>
32: <Cx>0.0</Cx>
33: <Cy>0.0</Cy>
34: <focalLength>0.0</focalLength>
35: </plugin>
36: </sensor>
37: </gazebo>
```

Code 5. Code listing of the file *engineer_arm.xacro*.

D. Adding Sensors

Robots are typically equipped with several types of onboard sensors, including cameras, laser range finders (LRF), IMUs and others. A number of sensors already have simulation models for the ROS/Gazebo environment. Adding a sensor to a robot simulation model means adding it to a robot description in a XACRO or URDF file. Every sensor type has its own description pattern incapsulated into a corresponding plugin [26]. An example of a sensor description in Code 5 corresponds to a mono camera of the Servosila Engineer robot.

An example of working sensors is demonstrated in Fig. 7. Data from working cameras and LRF are presented in RViz. LRF scan data are shown in the left subfigure with red dots, corresponding to a cylinder and couple cube obstacles of the simulated Gazebo world Fig. 7 (b). Visual data are captured by the four cameras of the robot, and a user could switch between video streams of the cameras by going through the corresponding tabs in the bottom of the camera window Fig. 7 (a); in the figure a dynamically updated frame from the right camera demonstrates the ball and the cube obstacles.



(a) (b)

Figure 7. Example of the working cameras and the LRF in the simulation. b: side view of the robot and the environment in the Gazebo simulation. a: RViz window with data from the right camera in simulation; the red rectangular emphases tabs of switchable camera views; the yellow circle emphases LRF data that corresponds to the cube obstacles; the magenta circle emphases LRF data that corresponds to the cylinder obstacle.

E. Project Hierarchy

Using Robot Operating System (ROS) includes setting up a proper hierarchy of a newly created project. In most cases ROS-project is a single ROS-package or a set of several ROS-packages. An example of a ROS-package is presented in Fig. 8, which will be used in this section to explain the hierarchy visually. Each package contains source files (Fig. 8, /src block) and instructions on how to build a current package (*CMakeList.txt*) and what packages and dependencies the current project is using for compilation and running (*package.xml*). Source files could be not only source code files of the project (such as *.cpp* or *.py ;* in Fig. 8 they correspond to /src block) but also service and messages information files (*.srv* and *.msg* in Fig. 8 they correspond to /src and /msg blocks respectively), instruction for the execution files (*.launch* in Fig. 8 they correspond to /launch block) and

information about robot structure files (*.urdf* and *.xacro* in Fig. 8 they correspond to /urdf block).

For projects with simulation model of robots there is already an established type of an architecture. Mostly such projects contain several packages. Each of them is responsible for different parts of the virtual model. A basic set of the packages is:

- *(name-of-the-robot)_description* – the package contains main information about the model such as visual data and collision 3D models and textures, urdf and *xacro* files with information about the robot structure. In most cases this package also contains launch files and configurations for the RViz visualization.
- *(name-of-the-robot)_gazebo* – the package consist of additional information for the simulation. These are world files, which set up simulation scene for the robot and launch files, which could start the simulation.
- *(name-of-the-robot)_control* – the package, which is strongly connected to the *(name-of-the-robot)_gazebo* package. It contains configuration files for the controllers that are used to perform joint movements of the robot in a simulated environment.

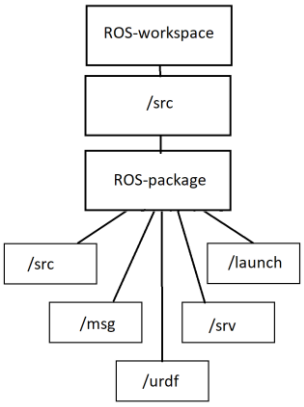


Figure 8. Example of the hierarchy of the ROS-project.

There are also optional packages that could be presented in projects with a robot virtual model, such as:

- *(name-of-the-robot)_navigation* – the package that contains configuration files for the navigation stack. In most cases it has several launch files to perform a navigation in the simulation. Also it could have some source code files for custom navigation algorithms or add-ons for existing algorithms (that are already a part of ROS).
- *(name-of-the-robot)_teleop* – the package contains launch files and (sometimes) source code that allows to control the virtual robot using an input from an operator (a keyboard, a joystick, a mouse etc.). In rare cases the same functionality could be met in *(name-of-the-robot)_control* package.
- *(name-of-the-robot)_msgs* – the package that can present any custom ROS-messages that may be required for further simulation. These could be non-standard control messages.
- *(name-of-the-robot)_viz* – could be performed as a separated package for visualization. Mainly contains configuration files for the RViz visualization and rqt. Also it could contain launch files to execute RViz and rqt with presented configs automatically. In rare cases the same functions can be performed by *(name-of-the-robot)_description* package.

All of the presented above packages allow to make a project modular. Such approach is supposed to improve and easy code understanding. In case of improvements it becomes clear where a user could find a source code, or where his/her new modules should be placed.

In our case we needed to create only three base packages: *engineer_description*, *engineer_gazebo* and *engineer_control*. Creating a new package could be performed using next command:

catkin_create_pkg engineer_description std_msgs rospy roscpp

Here we should specify a name of a package (in the example it is *engineer_description*) and packages, which the new package will depend from (in the example case these are *std_msgs*, *rospy* and *roscpp*).

Figure 9. Console output of a project compiling command (with an error highlighted in red color).

This command automatically creates configurational files, such as *CMakeList.txt* and *package.xml*. Moreover, it adds all of the mentioned above dependencies into proper places in these files.

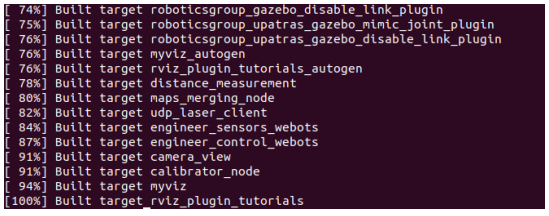


Figure 10. Console output of a project compiling command.

All of the packages should be created in */src* directory of the ROS-workspace. After filling the new package with the necessary files it needed to be compiled using command:

catkin_make

This command should be run in the directory of the workspace. It will build all of the packages in the workspace. Most of the problems with source code and dependencies in it will arise at this stage.

```
1 : <launch>
2 : <!-- Input parameters -->
3 : <arg name="gui" default="True"/>
4 : <arg name="debug" default="False"/>
5 : <arg name="paused" default="True"/>
6 : <arg name="headless" default="False"/>
7 : <arg name="use_sim_time" default="True"/>
8 : <arg name="world_name"
9 : default="$(find engineer_gazebo)/world/empty_world.world"/>
10:
11: <arg name="robot_namespace" default="engineer"/>
12: <arg name="x" default="0.0"/>
13: <arg name="y" default="0.0"/>
14: <arg name="z" default="1.0"/>
15: <arg name="roll" default="0"/>
16: <arg name="pitch" default="0.0"/>
17: <arg name="yaw" default="0.0"/>
18:
19: <!-- Launch Gazebo with the specified world -->
20: <include file="$(find gazebo_ros)/launch/empty_world.launch">
21: <arg name="world_name" value="$(arg world_name)"/>
22: <arg name="gui" value="$(arg gui)"/>
23: <arg name="debug" value="$(arg debug)"/>
24: <arg name="paused" value="$(arg paused)"/>
25: <arg name="headless" value="$(arg headless)"/>
26: <arg name="use_sim_time" value="$(arg use_sim_time)"/>
27: </include>
28:
29: <param name="tf_prefix" value="engineer"/>
30:
31: <arg name="model"
32:default="$(find
engineer_description)/robots/engineer_default.urdf.xacro"/>
33:
34: <!-- Load Jackal's description -->
35: <include
36: file="$(find
engineer_description)/launch/engineer_description.launch">
37: <arg name="model" value="$(arg model)"/>
38: </include>
39:
40: <!-- Spawn robot in Gazebo -->
41: <node name="urdf_spawner" pkg="gazebo_ros"
type="spawn_model"
42: args="-x $(arg x) -y $(arg y) -z $(arg z) -R $(arg roll)
43: -P $(arg pitch) -Y $(arg yaw) -urdf -model $(arg
robot_namespace)
44: -param robot_description" respawn="false" output="screen"/>
45:
46: </launch>
```

Code 6. Code listing of the file *engineer_gazebo.launch*.

For example, Fig. 9 presents one of possible errors that may occur during project compiling. It refers a source code file *mimic_control.cpp* and points at a problem with a variable (*subscriber1*), which was not initialized. It also suggests a way to fix this problem by renaming a problematic variable into another one within the same file (*subscriber2*), which was successfully initialized. When no errors appear, a successful build will reach 100 percent (Fig. 10).

V. Simulation Launch

The example of the launch file for the controllers was already shown in Code 4. As a first step, the simulation is started and the model is spawned. Only after that it is possible to launch the created controller. To start the simulation and spawn the model, a new launch file is created; Code 6 demonstrates a listing of such launch file.

The launch file performs the following actions:

1. Initialize several parameters for the simulation (Lines 3–9), such as options if it needs to launch a graphic user interface for the simulation (Line 3) or if it is necessary to pause the simulation at the start (Line 5). Also they specify a position and an orientation of the robot at the beginning (Lines 12-17).
2. Launch the Gazebo simulator with the created parameters (Lines 20–27).
3. Introduce *tf_prefix* that will be a namespace for the robot (Line 29).
4. Initialize an argument which, contains a path to the description of the robot (Lines 31–32).
5. Launch a loader for the description of the robot (Lines 35–38) so that in the future it could be easier to launch the visualization in *RViz* or *rqt*.
6. Spawn the robot in the simulated environment in a specified position and orientation and with a specific namespace using *urdf_spawner* (Lines 41–44).

To decrease a number of launchers for a manual execution, a launch of the controllers could be added in the end of the launch file (presented in Code 6). It could be added using a part of the code that is presented in Code 7.

```
<include file="$(find
engineer_control)/launch/engineer_control_all.launch">
<arg name="robot_namespace" value="$(arg robot_namespace)"/>
```

Code 7. A part of the code, which allows to call an existing launch file from another launch file.

The last step is launching the created file. It could be performed using next command:

```
roslaunch engineer_gazebo engineer_gazebo.launch
```

Before launching this command, it is necessary to ensure that the project was successfully compiled using the previously shown command, and then all compiled packages were loaded using the next command:

```
source devel/setup.bash
```

So that *roslaunch* command will be able to locate the previously created launch.

As a result, the Gazebo simulator starts and several seconds later a widow of the simulator appears. The window contains a created environment. The virtual model of the described in *urdf* file robot appears in the viewport (an example is shown in Fig. 7).

VI. Conclusions

This paper presented a step-by-step tutorial that explains a construction of a virtual model for a mobile robot within the Gazebo simulator using Robot Operating System (ROS). It described an entire process that starts from a visual model construction and physics setup, and

ends up with sensors setup, ROS-based control integration and creating a ROS-project with the model. Information about hierarchy of different ROS-packages and stages of execution is demonstrated as a part of the project. The process is illustrated with an example of a crawler-type robot Servosila Engineer modelling. Examples of software implementation with code, detailed comments, explanations and corresponding video files are available via Gitlab as open source supporting files of the paper

CONFLICT OF INTEREST

The authors declare no conflict of interest.

AUTHOR CONTRIBUTIONS

AD conducted the research, modelling and programming; RL conducted programming; EM supervised the research and wrote the paper; YB validated the models; MS performed existing solutions analysis; all authors had approved the final version.

FUNDING

The reported study was funded by the Russian Science Foundation (RSF) and the Cabinet of Ministers of the Republic of Tatarstan according to the research project No. 22-21-20033.

REFERENCES

[1] S. Smys and G. Ranganathan, "Robot assisted sensing control and manufacture in automobile industry," *Journal of ISMAC*, vol. 1, no. 03, pp. 180–187, 2019.
[2] V. Voronin, M. Zhdanova, E. Semenishchev, A. Zelenskii, Y. Cen, and S. Agaian, "Action recognition for the robotics and manufacturing automation using 3-d binary micro-block difference," *Int J Adv Manuf Techno*, no. 17, pp. 2319–2330, 2021.
[3] E. A. Martinez-Garcia, O. Akihisa et al., "Crowding and guiding groups of humans by teams of mobile robots," *IEEE Workshop on Advanced Robotics and Its Social Impacts*, pp. 91–96, 2005.
[4] D. Ryumin, I. Kagirov, A. Axyonov, N. Pavlyuk, A. Saveliev, I. Kipyatkova, M. Zelezny, I. Mporas, and A. Karpov, "A multimodal user interface for an assistive robotic shopping cart," *Electronics*, vol. 9, no. 12, p 2093, 2020.
[5] R. R. Murphy, "Rescue robotics for homeland security," *Communications of the ACM*, vol. 47, no. 3, pp. 66–68, 2004.
[6] D. Koung, O. Kermorgant, I. Fantoni, and L. Belouaer, "Cooperative multi-robot object transportation system based on hierarchical quadratic programming," *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6466-6472, 2021.
[7] E. Chebotareva, R. Safin, K. H. Hsia, A. Carballo, and E. Magid, "Person-following algorithm based on laser range finder and monocular camera data fusion for a wheeled autonomous mobile robot," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, no. 12336, pp. 21-33, 2020.
[8] E. Magid, A. Zakiev, T. Tsoy, R. Lavrenov, and A. Rizvanov, "Automating pandemic mitigation," *Advanced Robotics*, vol. 35, no. 9, pp. 572–589, 2021.
[9] D. Kolpashchikov, O. Gerget, and R. Meshcheryakov, "Robotics in healthcare," *Handbook of Artificial Intelligence in Healthcare*, Springer, pp. 281–306, 2022.
[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al., "Ros: an open-source robot operating system," in *Proc. ICRA Workshop on Open Source Software*, vol. 3, no. 3.2. Kobe, Japan, p. 5, 2009.
[11] A. Dobrokvashina, R. Lavrenov, E. A. Martinez-Garcia, and Y. Bai, "Improving model of crawler robot Servosila Engineer for simulation in ROS/Gazebo," in *Proc. 2020 13th International Conference on Developments in eSystems Engineering (DeSE)*, IEEE, 2020, pp. 212–217.
[12] A. Dobrokvashina, R. Lavrenov, T. Tsoy, E. A. Martinez-Garcia, and Y. Bai, "Navigation stack for the crawler robot Servosila Engineer," in *Proc. 2021 IEEE 16th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 1907–1912, 2021.
[13] "Servosila official site," [Online]. Available: https://www.servosila.com/en/index. shtml, accessed 30-March-2020].
[14] J. Pages, L. Marchionni, and F. Ferro, "Tiago: the modular robot that adapts to different research needs," in *Proc. International Workshop on Robot Modularity*, IROS, 2016.
[15] C. N. Thai, "Robotis' robot systems," *Exploring Robotics with ROBOTIS Systems*, Springer, pp. 5–21, 2017.
[16] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004, pp. 2149–2154.
[17] H. R. Kam, S. H. Lee, T. Park, and C. H. Kim, "Rviz: A toolkit for real domain data visualization," *Telecommunication Systems*, vol. 60, no. 2, pp. 337–345, 2015.
[18] J. V. Gumster, *Blender for Dummies*, John Wiley & Sons, 2020.
[19] B. R. Kent, *3D Scientific Visualization with Blender*, Morgan & Claypool Publishers San Rafael, CA, 2015.
[20] K. Murdock. Autodesk Maya 2019 Basics Guide. SDC Publications. 2018.
[21] R. Hess, *Blender Foundations: The Essential Guide to Learning Blender 2.5*, Routledge. 2013.
[22] S. Tickoo, *Autodesk 3Ds Max 2021: A Comprehensive Guide*, Cadcim Technologies. 2020.
[23] M. Sokolov, I. Afanasyev, R. Lavrenov, A. Sagitov, L. Sabirova, and E. Magid, "Modelling a crawler-type UGV for urban search and rescue in Gazebo environment," in *Proc. Artificial Life and Robotics (ICAROB)*, 2017, pp. 360-362.
[24] Support files for the tutorial, Laboratory of Intelligent Robotic Systems, Intelligent Robotics Department, Institute of Information Technology and Intelligent Systems, Kazan Federal University – [Online]. Available: https://gitlab.com/LIRS_Projects/public_examples/-/tree/main/ModellingInstructionsExample
[25] P. Cignoni, G. Ranzuglia, M. Callieri, M. Corsini, F. Ganovelli, N. Pietroni, M. Tarini, MeshLab, 2011.
[26] "Gazebo plugins in ros," [Online]. Available: http://gazebosim.org/tutorials?tut=ros_gzplugins, [Online; accessed 28-March-2022].