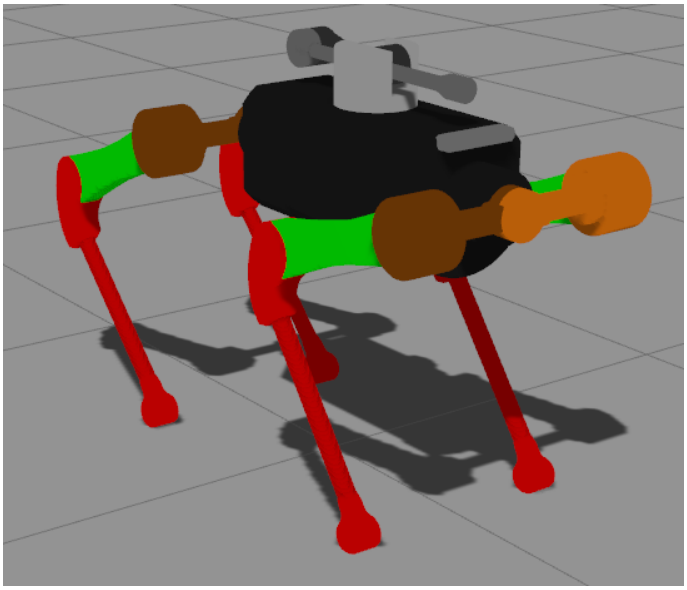
Mini Project Report

Christos Vasileios Kokas

11-10-2021



Contents

1 Abstract

The goal of this project is to build a simulation framework for ARGOS, a quadruped robot with three actuated degrees of freedom on each leg. In addition, with the use of Champ and gmapping ROS packages the robot will be able to move to a designated spot on the map while avoiding obstacles detected by a stereo camera mounted on its body. This project covers in detail the steps taken to model the robot using the XACRO XML macro language and with the use of Champ Setup Assistant create a ROS package for the control of ARGOS. In turn, the pid gains and the gait parameters have to be tuned. In order for the robot to be able to avoid obstacles while navigating the map, gmapping ROS packages will be used which provide laser-based SLAM (Simultaneous Localization and Mapping). Since the robot is equipped with stereo cameras only, a laserscan has to be produced from them with the use of the appropriate ROS packages so that ARGOS can detect objects in close proximity.

2 Modelling of ARGOS

The first step in this project is the modelling of the robot and, following that, Champ Setup Assistant will be used to produce the ROS package.

2.1 Xacro Language

Xacro (XML Macros) Xacro is an XML macro language [1]. With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions. After creating all the xacro files needed for the model of the robot a urdf [2] file can be generated using the command shown in Figure 1.

```
rosrun xacro xacro file.urdf.xacro > file.urdf
```

Figure 1: Command to generate a URDF file from XACRO

2.2 URDF file

A URDF (Universal Robot Description Format) file is an XML file that describes the robot and contains properties of each part such as mass or inertia. Additionally, it describes the position of every link and joint as well as any equipment that the robot has e.g.: a stereo camera. To model ARGOS a URDF file will be created.

2.2.1 Mass Properties

Using Solidworks, the Mass Properties tool can be used to calculate the mass and inertia for each part. The mass properties, i.e: Mass, Center of Mass from the origin, Inertia Matrix, for the body of the robot are presented in Figure 2.

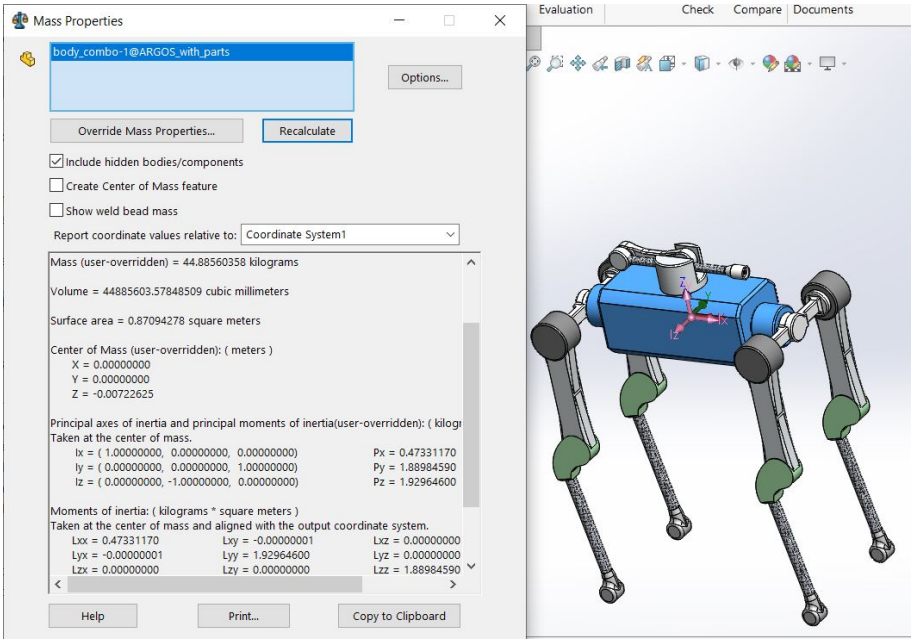


Figure 2: Body Mass Properties

The mass properties calculated for each part are summarized in Table 1

Link	Mass	Center of Mass(xyz)	Inertia Matrix
Body	44.89	0.0 0.0 -0.007	0.473 0.0 0.0 0.0 1.929 0 0.0 0.0 1.889
Leg Roll Segment	2.338	0.001 0.195 0.0	0.017 0.0 0.0 0.0 0.005 0 0.0 0.0 0.017
Upper Leg Segment	2.906	0.0 0.0 -0.197	0.07 0.0 0.0 0.0 0.0721 0 0.0 0.0 0.003
Lower Leg Segment	1.817	-0.025 0.0 -0.155	0.093 0.0 0.0 0.0 0.096 0 0.0 0.0 0.003
Manipulator Base	1.633	0.0 -0.011 0.049	0.0003 0.0 0.0 0.0 0.007 0 0.0 0.0 0.007
Manipulator Segment 1	0.657	0.194 0.0 0.0	0.0002 0.0 0.0 0.0 0.02 0 0.0 0.0 0.02
Manipulator Segment 2	0.457	0.194 0.0 0.0	0.0001 0.0 0.0 0.0 0.02 0 0.0 0.0 0.02

Table 1: Mass Properties of Each Part

2.2.2 Creating the Model

One of the advantages of the xacro language is the ability to include separate files, e.g.: robot properties, legs, manipulator, in one main file. The file containing all the robot properties like mass, distance of the center of mass from the origin, inertia values and the path to the mesh file for the part are presented in Figure 3.

```
<xacro:property name="path_to_base" value =
   "package://mini_project/meshes/body.dae"/>
<xacro:property name="ixx_body" value = "0.47331170488629"/>
<xacro:property name="ixy_body" value = "-0.00000001187694"/>
<xacro:property name="ixz_body" value = "0.0"/>
<xacro:property name="iyy_body" value = "1.92964600184763"/>
<xacro:property name="iyz_body" value = "0.0"/>
<xacro:property name="izz_body" value = "1.88984590052932"/>
<xacro:property name="body_mass" value = "44.88560358"/>
<xacro:property name="body_center_of_mass" value = "0 0
   -0.00722624877"/>
```

Figure 3: Robot Properties Using Xacro

Links are connected with joints to build the robot model. Each Link has the following elements [5]:

- Inertial (Required for Gazebo Simulation) : The inertial properties of the link.
  - Origin : This is the pose of the inertial reference frame, relative to the link reference frame. The origin of the inertial reference frame needs to be at the center of gravity.
  - Mass : The mass of the link is represented by the value attribute of this element.
  - Inertia : The 3x3 rotational inertia matrix, represented in the inertia frame. Because the rotational inertia matrix is symmetric, only 6 above-diagonal elements of this matrix are specified here, using the attributes ixx, ixy, ixz, iyy, iyz, izz.
- Visual : The visual properties of the link. This element specifies the shape of the object (box, cylinder, etc.) for visualization purposes.
  - Origin : The reference frame of the visual element with respect to the reference frame of the link.
  - Geometry : The shape of the visual object. This can be one of the following:
    - Box

        * Cylinder
        * Sphere
        * Mesh File
    - Material : The material of the visual element.

- Collision : The collision properties of a link. To reduce computation time a simplified mesh can be used.
    - Origin : The reference frame of the collision element, relative to the reference frame of the link.
    - Geometry : The shape of the collision object.

The Kinematics and Dynamics Library (KDL) defines a tree structure to represent the kinematic and dynamic parameters of a robot mechanism. kdl_parser provides tools to construct a KDL tree from an XML robot representation in URDF. KDL does not support a root link with an inertia, so the root link (base_link) of the model cannot have inertia or mass properties so a secondary link (base_inertia) is created to have the mass properties of the body. The two links are displayed in Figure 4.

```
<link name="base_link">
    <visual>
        <geometry>
            <mesh filename="${path_to_base}" scale = "0.001 0.001
                0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0 0"/>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_base}" scale = "0.001 0.001
                0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0 0"/>
    </collision>
</link>

<link name="base_inertia">
    <inertial>
        <origin xyz="${body_center_of_mass}" rpy="0 0 0"/>
        <mass value="${body_mass}" />
        <inertia ixx="${ixx_body}" ixy="${ixy_body}"
            ixz="${ixz_body}"
                  iyy="${iyy_body}" iyz="${iyz_body}"
                  izz="${izz_body}" />
    </inertial>
</link>
```

Figure 4: Base Link with Secondary Link

ARGOS is a quadruped robot with four identical legs. The XACRO language gives the ability to create macros which are helpful for identical parts. Macros provide the ability to create a part once, e.g. a leg, and then use that part multiple times. In addition, if statements can be used to change values of variables of the macro depending on which part is created. In this case, the variables changed are the position_hip(the distance from the hip to the body), which value changes if the referenced leg is the front or the hind, and the side variable (side of the robot), which value changes if the referenced leg is the right or the left. The if statement is presented in Figure

5, where "RF" is Right Front, "LF" is Left Front, "RH" is Right Hind and "LH" is Left Hind.

```
<xacro:if value="${leg == 'RF'}">
    <xacro:property name="side" value = "-1"/>
    <xacro:property name="position_hip" value = "0.43"/>
</xacro:if>
<xacro:if value="${leg == 'LF'}">
    <xacro:property name="side" value = "1"/>
    <xacro:property name="position_hip" value = "0.43"/>
</xacro:if>
<xacro:if value="${leg == 'RH'}">
    <xacro:property name="side" value = "-1"/>
    <xacro:property name="position_hip" value = "-0.43"/>
</xacro:if>
<xacro:if value="${leg == 'LH'}">
    <xacro:property name="side" value = "1"/>
    <xacro:property name="position_hip" value = "-0.43"/>
</xacro:if>
```

Figure 5: Xacro If Statement

Each leg has 3 parts, the leg roll segment, which is connected to the body, the upper leg segment and the lower leg segment. These parts with their origins are displayed in Figures 6 - 9.

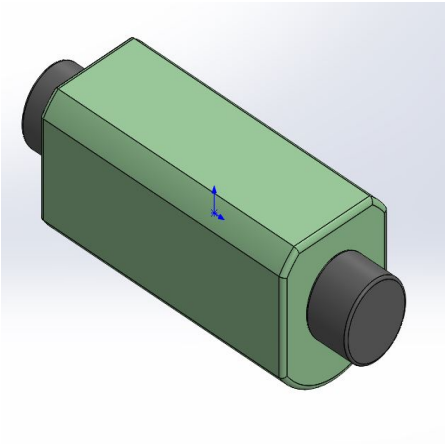


Figure 6: Body of the Robot

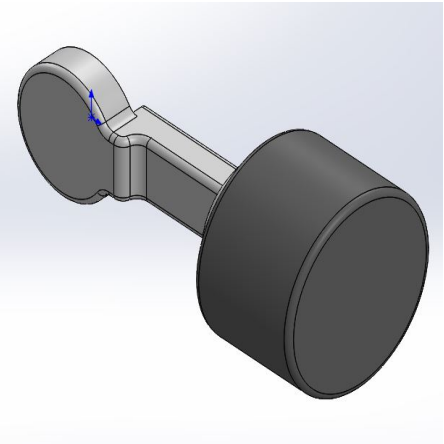


Figure 7: Leg Roll Segment

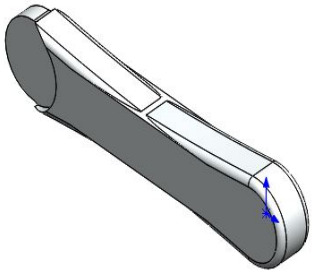


Figure 8: Upper Leg Segment

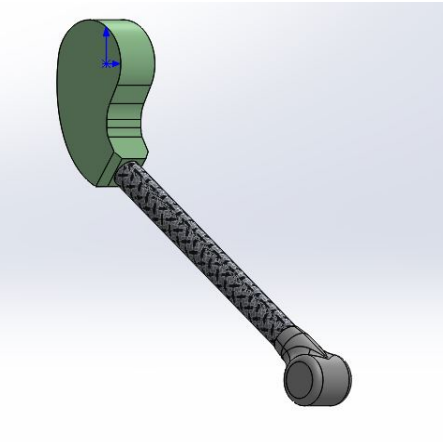


Figure 9: Lower Leg Segment

The measuring tool is used to calculate the distance from the center of the body to the hip as displayed in Figure 10.

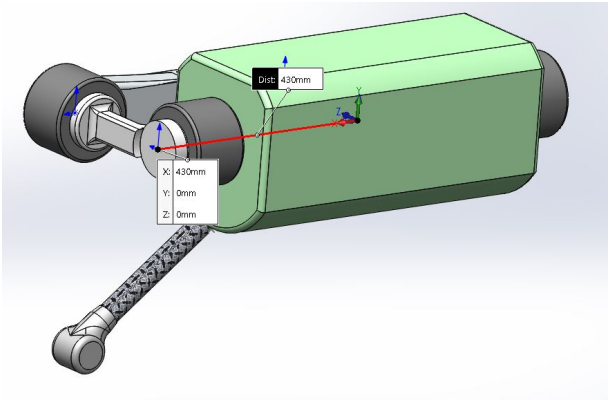


Figure 10: Distance from the Body to Hip

The body with one leg is presented in Figure 11.

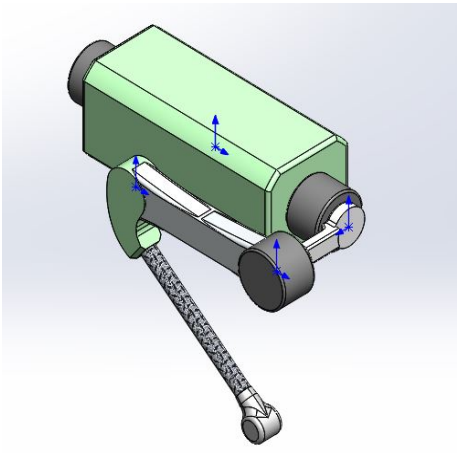


Figure 11: Body with One Leg

The links can be created for each part as displayed in Figures 12-14. Each link has a different colour to be easily distinguished.

```
<link name="${leg}_leg_roll">
    <inertial>
        <origin xyz = "${leg_roll_center_of_mass}" rpy="0 0 0"/>
        <mass value="${leg_roll_mass}" />
        <inertia ixx="${ixx_leg_roll}" ixy="${ixy_leg_roll}"
            ixz="${ixz_leg_roll}"
             iyy="${iyy_leg_roll}" iyz="${iyz_leg_roll}"
             izz="${izz_leg_roll}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_leg_roll}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 ${pi*side/2}" />
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_leg_roll}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 ${pi*side/2}"/>
    </collision>
</link>
```

Figure 12: Leg Roll Segment Link

```
<link name="${leg}_upper_leg">
    <inertial>
        <origin xyz = "${thigh_center_of_mass}" rpy="0 0 0"/>
        <mass value="${thigh_mass}" />
        <inertia ixx="${ixx_thigh}" ixy="${side*ixy_thigh}"
            ixz="${ixz_thigh}"
             iyy="${iyy_thigh}" iyz="${side*iyz_thigh}"
             izz="${izz_thigh}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_upper_leg}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} ${-pi/2} 0"/>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_upper_leg}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} ${-pi/2} 0"/>
    </collision>
</link>
```

Figure 13: Upper Leg Link

```
<link name="${leg}_lower_leg">
    <inertial>
        <origin xyz = "${lower_leg_center_of_mass}" rpy="0 0 0"/>
        <mass value="${lower_leg_mass}" />
        <inertia ixx="${ixx_lower_leg}" ixy="${ixy_lower_leg}"
            ixz="${ixz_lower_leg}"
             iyy="${iyy_lower_leg}" iyz="${iyz_lower_leg}"
             izz="${izz_lower_leg}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_lower_leg}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0 0.0 0" rpy="${pi/2} ${pi/2-0.182960291} 0"/>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_lower_leg}" scale = "0.001
                0.001 0.001"/>
        </geometry>
        <origin xyz="0 0.0 0" rpy="${pi/2} ${pi/2-0.182960291} 0"/>
    </collision>
</link>
```

Figure 14: Lower Leg Link

The links for the manipulator's parts are presented in Figures 15-17.

```
<link name="manipulator_base">
    <inertial>
        <origin xyz="${man_base_center}" />
        <mass value="${man_base_mass}" />
        <inertia ixx="${ixx_man_base}" ixy="${ixy_man_base}"
            ixz="${ixz_man_base}"
                iyy="${iyy_man_base}" iyz="${iyz_man_base}"
                izz="${izz_man_base}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_manipulator_base}" scale =
                "0.001 0.001 0.001"/>
        </geometry>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_manipulator_base}" scale =
                "0.001 0.001 0.001"/>
        </geometry>
    </collision>
</link>
```

Figure 15: Manipulator Base Link

```
<link name="manipulator_segment_1">
    <inertial>
        <origin xyz="${man_seg_1_center}" rpy="0 0.0 0"/>
        <mass value="${man_seg_1_mass}" />
        <inertia ixx="${ixx_man_seg_1}" ixy="${ixy_man_seg_1}"
            ixz="${ixz_man_seg_1}"
                iyy="${iyy_man_seg_1}" iyz="${iyz_man_seg_1}"
                izz="${izz_man_seg_1}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_manipulator_segment_1}" scale
                = "0.001 0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 ${pi}"/>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_manipulator_segment_1}" scale
                = "0.001 0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} 0.0 ${pi}"/>
    </collision>
</link>
```

Figure 16: Manipulator Segment 1 Link

```
<link name="manipulator_segment_2">
    <inertial>
        <origin xyz="${man_seg_2_center}" rpy="0 0 0.0"/>
        <mass value="${man_seg_2_mass}" />
        <inertia ixx="${ixx_man_seg_2}" ixy="${ixy_man_seg_2}"
            ixz="${ixz_man_seg_2}"
                iyy="${iyy_man_seg_2}" iyz="${iyz_man_seg_2}"
                izz="${izz_man_seg_2}" />
    </inertial>
    <visual>
        <geometry>
            <mesh filename="${path_to_manipulator_segment_2}" scale
                = "0.001 0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} ${pi} 0.0"/>
    </visual>
    <collision>
        <geometry>
            <mesh filename="${path_to_manipulator_segment_2}" scale
                = "0.001 0.001 0.001"/>
        </geometry>
        <origin xyz="0.0 0.0 0.0" rpy="${pi/2} ${pi} 0.0"/>
    </collision>
</link>
```

Figure 17: Manipulator Segment 2 Link

The links are connected with joints and they follow a tree structure by using the kdl_parser [6]. Joints have the following elements [7] :

- Origin : This is the transform from the parent link to the child link. The joint is located at the origin of the child link.
  - xyz : Represents the offset. All positions are specified in meters.
  - rpy : Represents the rotation around fixed axis: first roll around x, then pitch around y and finally yaw around z. All angles are specified in radians.
- Parent Link (Required) : The name of the parent link in the robot tree structure.

- Child Link (Required) : The name of the child link.
- Axis : This is the axis of rotation for revolute joints, the axis of translation for prismatic joints, and the surface normal for planar joints.
- Calibration : The reference positions of the joint, used to calibrate the absolute position of the joint.
- Dynamics : An element specifying physical properties of the joint.
  - Damping : The physical damping value of the joint.
  - Friction : The physical static friction value of the joint.
- Limit (Required for revolute and prismatic joints) : Contains the following attributes :
  - Effort (Required) : Maximum joint effort.
  - Lower : An attribute specifying the lower joint limit.
  - Upper : An attribute specifying the upper joint limit.
  - Velocity (Required) : An attribute for enforcing the maximum joint velocity.
- Mimic : This tag is used to specify that the defined joint mimics another existing joint.
- Safety_Controller : An element can contain the following attributes :
  - soft_lower_limit : An attribute specifying the lower joint boundary where the safety controller starts limiting the position of the joint.
  - soft_upper_limit : An attribute specifying the upper joint boundary where the safety controller starts limiting the position of the joint.
  - k_position : An attribute specifying the relation between position and velocity limits.
  - k_velocity : An attribute specifying the relation between effort and velocity limits.

Base_link and base_inertia links are connected together using a fixed joint which can be viewed in Figure 18.

```
<joint name="base_link_to_base_inertia" type="fixed">
    <parent link="base_link"/>
    <child link="base_inertia"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
```

Figure 18: Joint of Base Link with Base Inertia

The joints for the leg's links are presented in Figures 19-21.

```
<joint name="${leg}_hip_joint" type="revolute">
    <parent link="base_link"/>
    <child link="${leg}_leg_roll"/>
    <axis xyz="1.0 0.0 0.0"/>
    <origin xyz="${position_hip} 0.0 0.0"/>
    <limit effort="250" lower="-${pi}" upper="${pi}" velocity="5.0"
        />
    <dynamics damping="0.0" friction="0.0" />
</joint>
```

Figure 19: Base Link Connection to Leg Roll Segment

```
<joint name="${leg}_upper_leg_joint" type="revolute">
    <parent link="${leg}_leg_roll"/>
    <child link="${leg}_upper_leg"/>
    <axis xyz="0.0 1.0 0.0"/>
    <origin xyz="0.0 ${side*0.233} 0.0"/>
    <limit effort="250" lower="-${pi}" upper="${pi}" velocity="5.0"
        />
    <dynamics damping="0.0" friction="0.0" />
</joint>
```

Figure 20: Leg Roll Segment Connection to Upper Leg

```
<joint name="${leg}_lower_leg_joint" type="revolute">
    <parent link="${leg}_upper_leg"/>
    <child link="${leg}_lower_leg"/>
    <axis xyz="0.0 1.0 0.0"/>
    <origin xyz="0.0 0.0 -0.45"/>
    <limit effort="250" lower="-${pi}" upper="${pi}" velocity="5.0"
        />
    <dynamics damping="0.0" friction="0.0" />
</joint>
```

Figure 21: Upper Leg Connection to Lower Leg

The joints for the manipulator's links are presented in Figures 22-24.

```
<joint name="${base_name}_to_manipulator_base" type="fixed">
    <parent link="${base_name}"/>
    <child link="manipulator_base"/>
    <origin xyz="0.0 0.0 0.13"/>
</joint>
```

Figure 22: Body Connection to Manipulator Base

```
<joint name="manipulator_base_to_manipulator_segment_1"
    type="fixed">
    <parent link="manipulator_base"/>
    <child link="manipulator_segment_1"/>
    <origin xyz="0.0 0.04 0.075" rpy="0 0 0"/>
</joint>
```

Figure 23: Manipulator Base Connection to Manipulator Segment 1

```
<joint name="manipulator_segment_1_to_manipulator_segment_2"
    type="fixed">
    <parent link="manipulator_segment_1"/>
    <child link="manipulator_segment_2"/>
    <origin xyz="-0.25 -0.04 0.0" rpy="0.0 0.0 0.0"/>
</joint>
```

Figure 24: Manipulator Segment 1 Connection to Manipulator Segment 2

An empty link is created at the toe, required by Champ Setup Assistant described in Paragraph 2.3.1, to determine the contact point of the robot with the ground with a fixed joint connecting it to the lower leg as shown in Figure 25.

```
<link name="${leg}_foot"/>
<joint name="${leg}_foot_joint" type="fixed">
    <parent link="${leg}_lower_leg"/>
    <child link="${leg}_foot"/>
    <origin xyz="0.0 0.0 -0.61"/>
</joint>
```

Figure 25: Foot Link with Joint

For the robot to be able to move, transmissions need to be added on all revolute joints. The transmission element is an extension to the URDF robot description model that is used to link actuators to joints [8]. The transmission has one attribute :

- Name (Required) : A unique name for the transmission.

And the following elements :

- Type : The transmission type.
- Joint : A joint the transmission is connected to. The joint is defined by it's name attribute, and the following sub-elements:
  - HardwareInterface : A supported joint-space hardware interface.

- Actuator : An actuator the transmission is connected to. The actuator is defined by it's name attribute, and the following sub-elements:
  - MechanicalReduction : Mechanical reduction at the joint/actuator transmission.
  - HardwareInterface : A supported joint-space hardware interface.

The Transmission element is displayed in Figure 26.

```
<transmission name="${leg}_hip_joint_trans">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="${leg}_hip_joint">
        <hardwareInterface>hardware_interface/
            EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="${leg}_hip_joint_motor">
        <hardwareInterface>hardware_interface/
            EffortJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
```

Figure 26: Transmission Element

The XACRO files for the legs and the manipulator can be included in the main xacro file as presented in Figure 27.

```
<xacro:include filename="$(find
    mini_project)/robots/legs.urdf.xacro"/>
<xacro:argos_leg leg="LF"/>
<xacro:argos_leg leg="RF"/>
<xacro:argos_leg leg="LH"/>
<xacro:argos_leg leg="RH"/>
<xacro:include filename="$(find
    mini_project)/robots/manipulator.urdf.xacro"/>
<xacro:manipulator base_name="base_link"/>
```

Figure 27: Creating The Parts

Gazebo provides a tool to evaluate the model's mass properties. From the view menu in Gazebo by choosing the option for inertias and center of mass it is possible to check if the values are realistic. The pink boxes are the inertia of each part and they should cover most of the part as shown in Figure 28. The center of mass for each part is displayed in Figure 29.

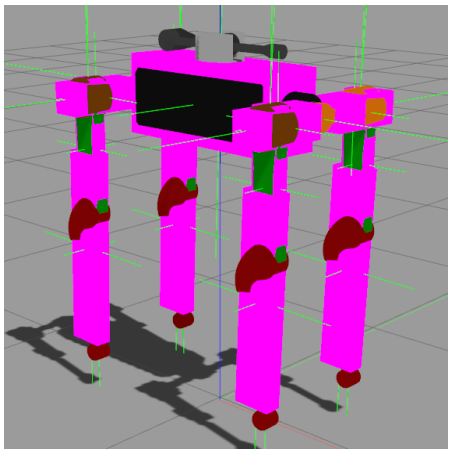


Figure 28: Inertia of Robot

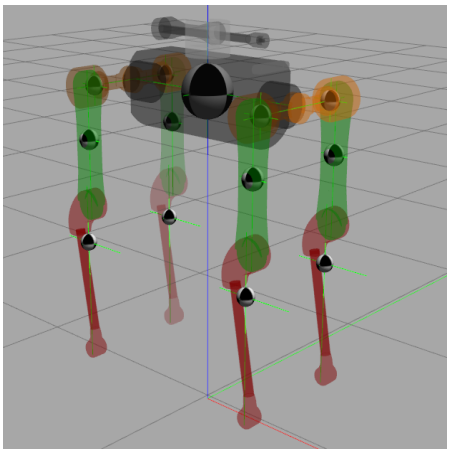
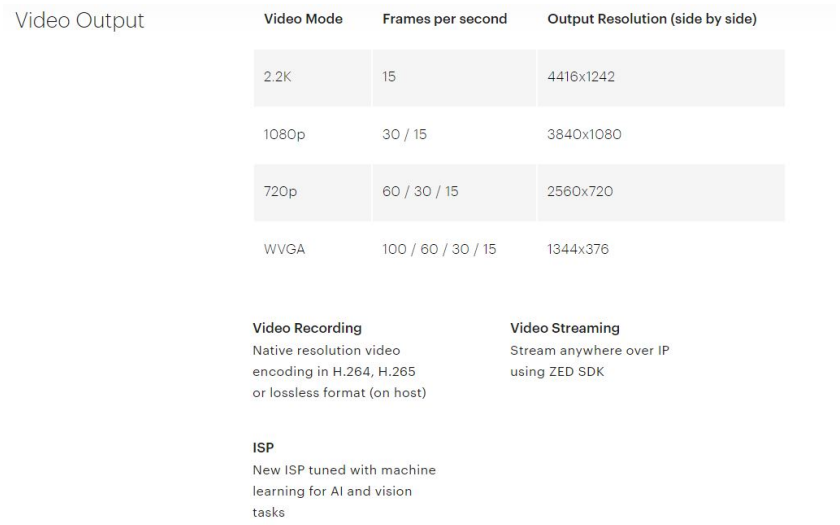


Figure 29: Centers of Mass

For obstacle detection and avoidance the model is equipped with a stereo camera. A stereo camera is a type of camera with two lenses with a separate image sensor for each lens. This allows the camera to simulate human binocular vision, and therefore gives it the ability to capture three-dimensional images, a process known as stereo photography [11]. For ARGOS the ZED2

camera was chosen. ZED2 specifications are summarized in Figure 30 and Figure 31.



Video Output

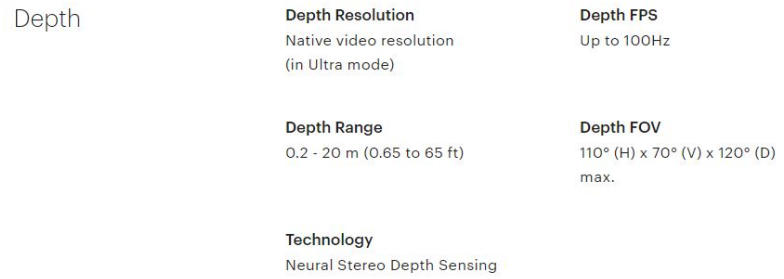
Video Mode	Frames per second	Output Resolution (side by side)
2.2K	15	4416x1242
1080p	30 / 15	3840x1080
720p	60 / 30 / 15	2560x720
WVGA	100 / 60 / 30 / 15	1344x376

Video Recording
Native resolution video encoding in H.264, H.265 or lossless format (on host)

Video Streaming
Stream anywhere over IP using ZED SDK

ISP
New ISP tuned with machine learning for AI and vision tasks

Figure 30: ZED2 Video Properties



Depth

Depth Resolution
Native video resolution (in Ultra mode)

Depth FPS
Up to 100Hz

Depth Range
0.2 - 20 m (0.65 to 65 ft)

Depth FOV
110° (H) x 70° (V) x 120° (D) max.

Technology
Neural Stereo Depth Sensing

Figure 31: ZED2 Depth Properties

To equip the model with the ZED2 camera, the center and the left camera frame are created as links and are connected to the body using joints as presented in Figure 32.

```
<link name="camera_center">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <mesh filename="package://mini_project/
            meshes/${model}.stl" />
        </geometry>
        <material name="${model}_mat" />
    </visual>
</link>
<link name="left_camera_frame" />
<link name="left_camera_optical_frame"/>
<joint name="camera_center_joint" type="fixed">
    <parent link="$(arg base_frame)"/>
    <child link="camera_center"/>
    <origin xyz="$(arg cam_pos_x) $(arg cam_pos_y) $(arg
        cam_pos_z)" rpy="$(arg cam_roll) $(arg cam_pitch) $(arg
        cam_yaw)" />
</joint>
<joint name="left_camera_joint" type="fixed">
    <parent link="camera_center"/>
    <child link="left_camera_frame"/>
    <origin xyz="0 ${baseline/2} 0" rpy="0 0 0" />
</joint>
<joint name="left_camera_optical_joint" type="fixed">
    <origin xyz="0 0 0" rpy="-${M_PI/2} 0.0 -${M_PI/2}"/>
    <parent link="left_camera_frame"/>
    <child link="left_camera_optical_frame"/>
```

Figure 32: Camera Links and Joints

2.2.3 ROS TF Tree

The tf system in ROS keeps track of multiple coordinate frames and maintains the relationship between them in a tree structure. To evaluate the relationship between the transforms(positions) of the links (frames) the ROS TF tree is used. Every frame has one parent and an unlimited number of children. The base_link frame, which is the body of ARGOS, should be a parent of all the remaining ARGOS parts and a child to the odom frame.

The odom frame is used to represent an absolute position in the world, typically (0.0,0.0,0.0), and contains the pose of the robot's base_link frame in the world as reported by the state estimator. The state estimator receives as inputs the measurements from the imu, the positions of the joints and the positions of the foot contacts and computes an estimate of the position of the base_link frame (/argos/odom/raw) and the base_to_footprint_pose which is the position of the base_link with height(z coordinate) = 0 as presented in Figure 33.

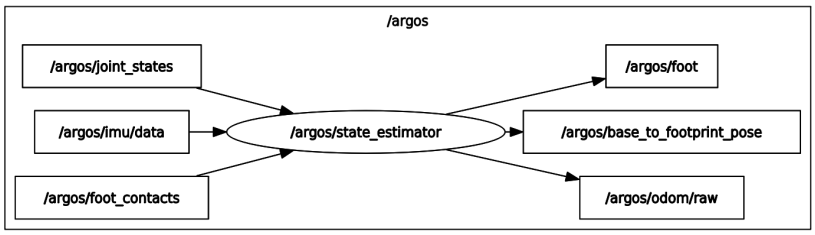


Figure 33: Inputs and Outputs of State Estimator

Directly or indirectly, all other frames are defined with respect to the odom frame. The location of a frame is defined relative to it's parent so the base_link frame is defined relative to the odom frame and all the remaining ARGOS parts are defined relative to the base_link frame. In Figure 34 the ROS TF frame tree is presented.

Figure 34: ROS TF Frame Tree

2.2.4 Gazebo References and Plugins

The toes of the robot need to have the appropriate friction to avoid slipping on the ground. Gazebo can add friction to a link by using the <gazebo> element [9]. The <gazebo> element is an extension to the URDF used for specifying additional properties needed for simulation purposes in Gazebo. The <gazebo> element for links has the following elements to add friction:

- kp,kd : Contact stiffness kp and damping kd for rigid body contacts as defined by the Open Dynamics Engine (ODE) [12]
- mu1,mu2 : Friction coefficients μ for the principal contact directions along the contact surface as defined by ODE
- minDepth : minimum allowable depth between the surfaces before contact correction impulse is applied
- maxContacts : Maximum number of contacts allowed between two entities

The values choosen are shown in Figure 35.

```
<gazebo reference="${leg}_lower_leg">
    <kp>10000000.0</kp>
    <kd>1.0</kd>
    <mu1>0.9</mu1>
    <mu2>0.9</mu2>
    <minDepth>0.005</minDepth>
    <maxContacts>1</maxContacts>
    <material>Gazebo/Red</material>
</gazebo>
```

Figure 35: Friction of Lower Leg Link

Gazebo additionally offers plugins that give your URDF models functionality and can tie in ROS messages and service calls for sensor output and motor input [10]. Gazebo supports the following plugin types :

- ModelPlugins : to provide access to the physics::Model API, e.g.: gazebo_ros_control (parses the transmission tags and loads the appropriate hardware interfaces and controller manager).

- SensorPlugins : to provide access to the sensors::Sensor API, e.g.: camera_controller (provides ROS interface for simulating cameras by publishing the CameraInfo and Image ROS messages).
- VisualPlugins : to provide access to the rendering::Visual API e.g.: display_video_controller (displays a ROS image stream on an OGRE Texture inside gazebo).

Sensors in Gazebo are meant to be attached to links, so the <gazebo> element describing that sensor must be given a reference to the camera link. Since ARGOS is equipped with a stereo camera, the sensor type chosen was the multicamera (Figure 36).

```
<gazebo reference="camera_center">
    <sensor type="multicamera" name="stereo_camera">
```

Figure 36: multicamera sensor

The Left and Right camera specifications match the ZED2 camera specifications and are shown in Figures 37 and 38 (The Right camera has the same specifications except the pose).

```
<camera name="left">
    <pose>0 0.06 0 0 0 0</pose>
    <horizontal_fov>1.91986218</horizontal_fov>
    <vertical_fov>1.22173048</vertical_fov>
    <diagonal_fov>2.0943951</diagonal_fov>
    <image>
    <width>1920</width>
    <height>1080</height>
    <format>B8G8R8</format>
    </image>
    <clip>
    <near>0.2</near>
    <far>20</far>
    </clip>
    <noise>
    <type>gaussian</type>
    <mean>0.0</mean>
    <stddev>0.007</stddev>
    </noise>
</camera>
```

Figure 37: Left Camera

```
    <pose>0 -0.06 0 0 0 0</pose>
```

Figure 38: Right Camera Pose

To publish the stereo camera outputs on topics a plugin is needed that is shown in Figure 39.

```
<plugin name="stereo_camera_controller"
   filename="libgazebo_ros_multicamera.so">
   <alwaysOn>true</alwaysOn>
   <updateRate>0.0</updateRate>
   <cameraName>camera</cameraName>
   <imageTopicName>image_raw</imageTopicName>
   <cameraInfoTopicName>camera_info</cameraInfoTopicName>
   <frameName>camera_center</frameName>
   <!--<rightFrameName>right_camera_optical_frame</
      rightFrameName>-->
   <hackBaseline>0.07</hackBaseline>
   <distortionK1>0.0</distortionK1>
   <distortionK2>0.0</distortionK2>
   <distortionK3>0.0</distortionK3>
   <distortionT1>0.0</distortionT1>
   <distortionT2>0.0</distortionT2>
</plugin>
```

Figure 39: Plugin for Stereo Camera Control

Lastly, to control the movement and the stability of the robot the gazebo ros control plugin is used that is displayed in Figure 40.

```
<gazebo>
   <plugin filename="libgazebo_ros_control.so"
      name="gazebo_ros_control">
      <legacyModeNS>true</legacyModeNS>
   </plugin>
</gazebo>
```

Figure 40: ROS Control Plugin

2.3 Champ

CHAMP is an open source development framework for building new quadrupedal robots and developing new control algorithms [3]. Core Features include :

- Setup-assistant to configure newly built robots.
- Collection of pre-configured URDFs like Anymal, MIT Mini Cheetah, Boston Dynamic's Spot and LittleDog.
- Compatible with DIY quadruped projects like SpotMicroAI and Open-Quadruped.
- Demo Applications like TOWR and chicken head stabilization.
- Lightweight C++ header-only library that can run on both SBC and micro-controllers.

2.3.1 Champ Setup Assistant

Champ Setup Assistant is a software that, if provided with a correct URDF file, auto generates a configuration package to make a quadruped robot walk [4]. In Figure 41 the Champ Setup Assistant Software is presented.

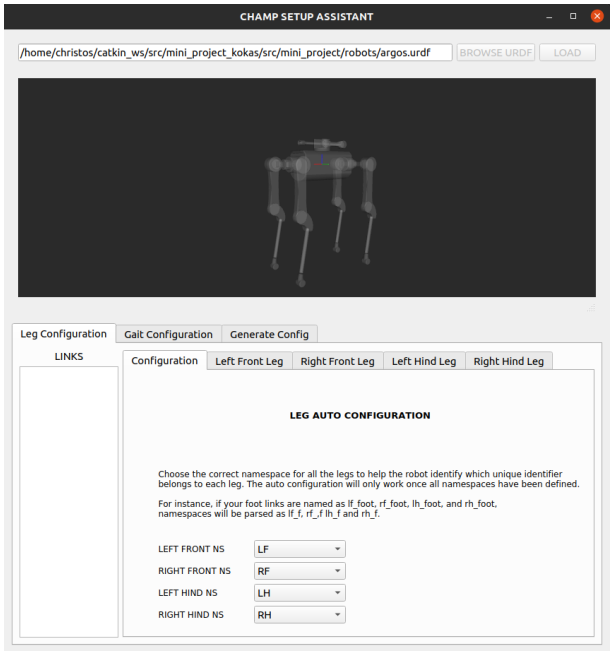


Figure 41: Champ Setup Assistant Software

2.3.2 Champ Setup Assistant Assumptions

The following assumptions have been made to avoid fragmentation across different robots as there can be thousands of ways to create robot's URDF :

- There are no rotation between frames (joint's origin-rpy are all set to zero).
- Hip joints rotate in the X axis.
- Upper Leg joints rotate in the Y axis.
- Lower Leg joints rotate in the Y axis.
- Origins of actuators' meshes are located at the center of rotation.
- All joints at zero position will results the robot's legs to be fully stretched towards the ground. From frontal and sagittal view, all legs should be perpendicular to the ground.

Figures 42-45 shows the above assumptions. AXES: +X:Red, +Y:Green, +Z:Blue.

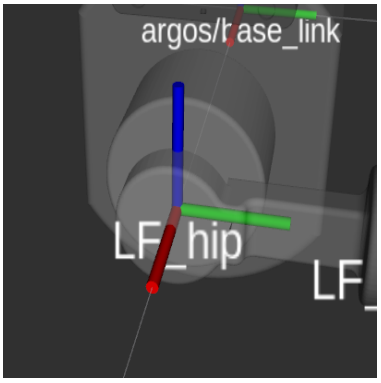


Figure 42: Hip Joint Rotation on X Axis

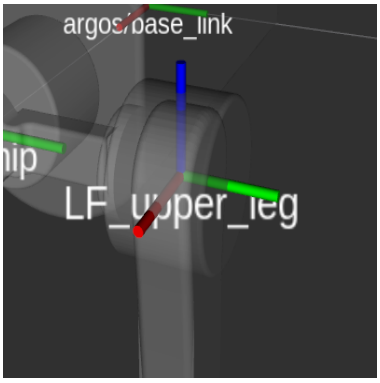


Figure 43: Upper Leg Joint Rotation on Y Axis

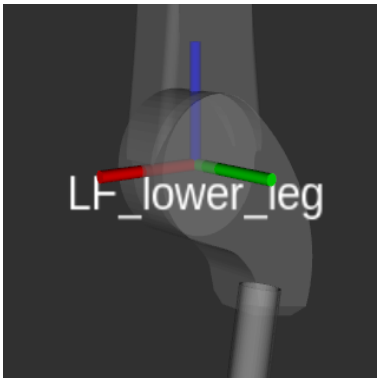


Figure 44: Upper Leg Joint Rotation on Y Axis

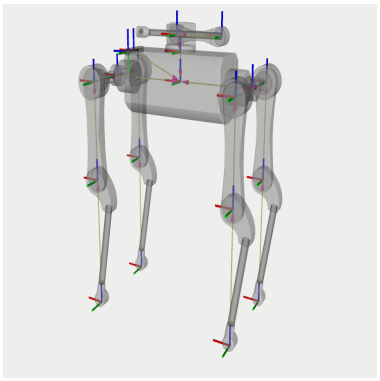


Figure 45: ARGOS Following the Above Assumptions

2.4 ROS control

The ros_control [13] package was chosen to control ARGOS. The ros_control packages takes as input the joint state data from your robot's actuator's encoders and an input set point. It uses a generic control loop feedback mechanism, typically a PID controller, to control the output, typically effort,

sent to your actuators. The data flow diagram of ROS Control is presented in Figure 46.

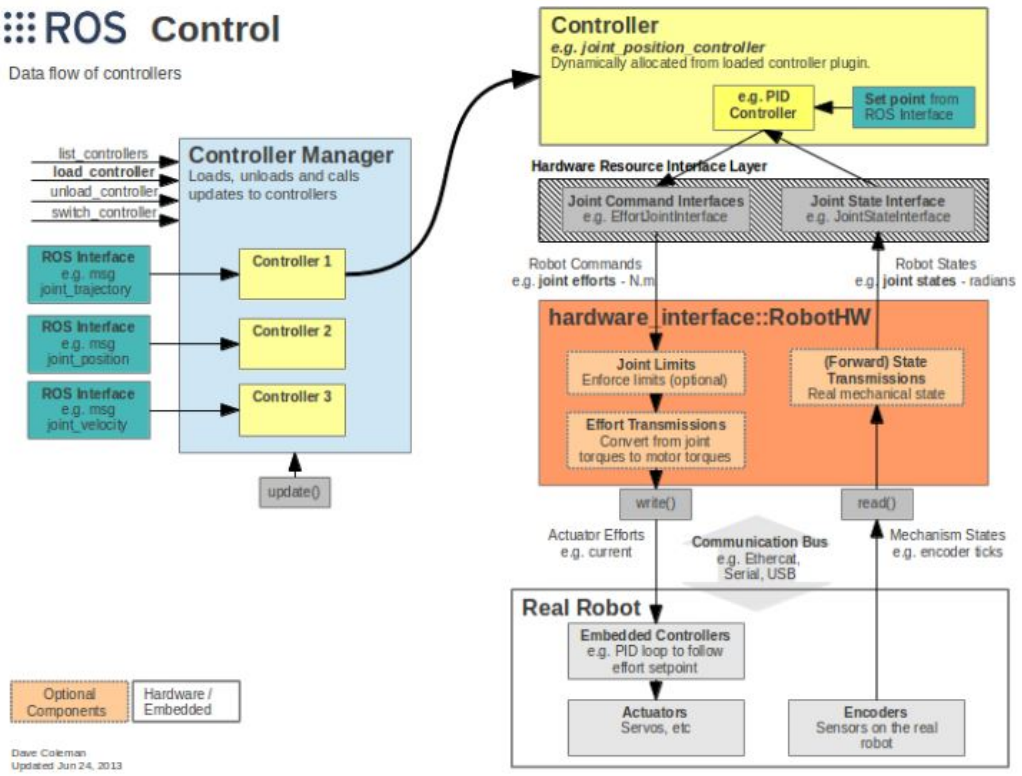


Figure 46: ROS Control Data Flow Diagram

3 Stereo Camera to Laserscan

Gmapping packages [14] require laserscans as input to produce SLAM (Simultaneous Localization and Mapping) [15].

3.1 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of a robot's location within it. As mentioned in Paragraph 2.2.3, the odom frame contains an estimate of the pose of the

robot in the world computed by the state estimator. Due to errors in odometry (estimated position of the robot), the odom pose accumulates error over time. Gmapping packages introduce a transform to correct for the error, between the map frame (created by gmapping) and the odom frame. This transform is the correction that has been computed by gmapping based on sensor input from a laserscan. Gmapping compares past laserscan messages with present, and calculates the correction of the estimated position of the robot.

3.2 Disparity Image To Laserscan

One method to produce a laserscan from a stereo camera is converting a disparity image to a laserscan.

3.2.1 Stereo Camera Topics

ARGOS is equipped with a stereo camera which publishes the topics presented in Figure 47. The stereo camera publishes its measurements of the left camera with namespace /camera/left and the measurements of the right camera with namespace /camera/right.

```
/argos/camera/left/image_raw
/argos/camera/left/image_raw/compressed
/argos/camera/left/image_raw/compressed/parameter_descriptions
/argos/camera/left/image_raw/compressed/parameter_updates
/argos/camera/left/image_raw/compressedDepth
/argos/camera/left/image_raw/compressedDepth/parameter_descriptions
/argos/camera/left/image_raw/compressedDepth/parameter_updates
/argos/camera/left/image_raw/theora
/argos/camera/left/image_raw/theora/parameter_descriptions
/argos/camera/left/image_raw/theora/parameter_updates
```

Figure 47: Stereo Camera Outputs

All topics with their corresponding message type is shown in Table 2.

Topic	Message Type
image_raw	sensor_msgs/Image
compressed	sensor_msgs/CompressedImage
compressed/parameter_descriptions	dynamic_reconfigure/ConfigDescription
compressed/parameter_updates	dynamic_reconfigure/Config
compressedDepth	sensor_msgs/CompressedImage
compressedDepth/parameter_descriptions	dynamic_reconfigure/ConfigDescription
compressedDepth/parameter_updates	dynamic_reconfigure/Config
theora	theora_image_transport/Packet
theora/parameter_descriptions	dynamic_reconfigure/ConfigDescription
theora/parameter_updates	dynamic_reconfigure/Config

Table 2: Message Type of Each Topic

3.2.2 Stereo Vision to Laserscan

A stereo camera has two image sensors, this allows the camera to simulate human binocular vision and therefore gives it the ability to perceive depth. The human binocular vision perceives depth by using Stereo disparity which refers to the difference in the location of an object seen by the left compared to the right eye. By using feature matching (the task of establishing similarities between two images of the same object, presented in Figure 48) on the images from the stereo camera a disparity image can be created as is presented in Figure 49.

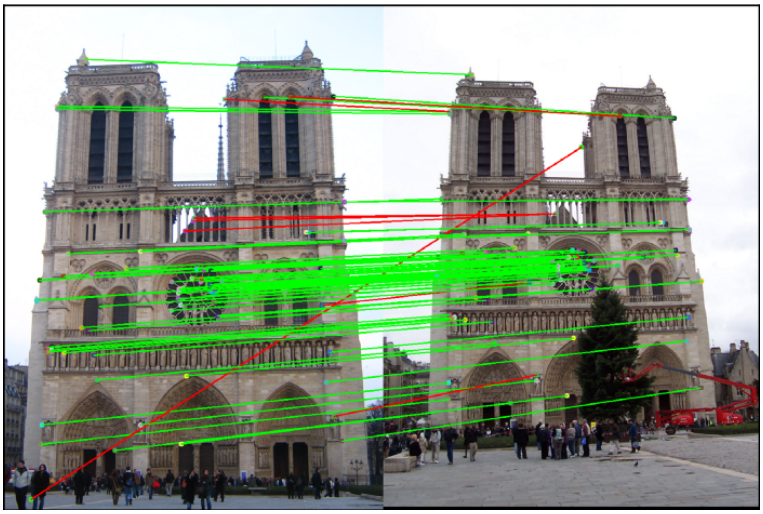


Figure 48: Feature Matching

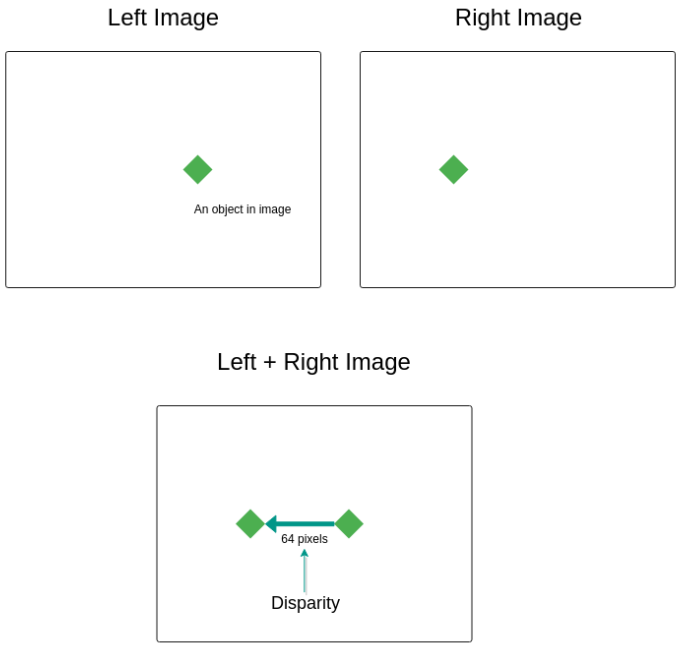


Figure 49: Stereo Disparity

A disparity image is an image that shows the distance between two cor-

responding points in the left and right image of a stereo camera as shown in Figure 50.

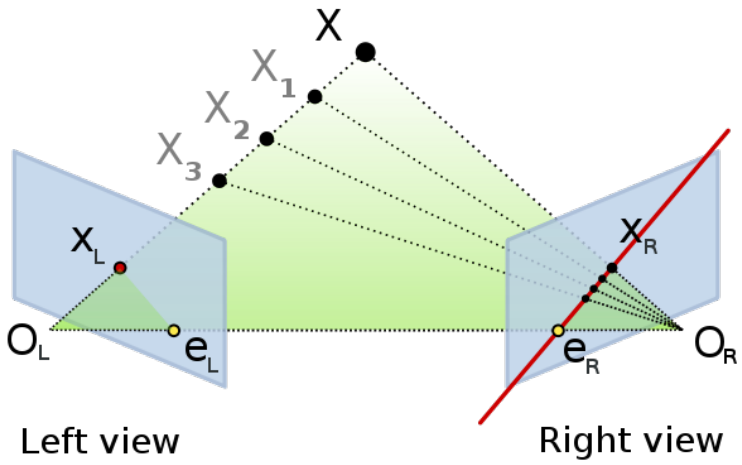


Figure 50: Disparity

The disparity image can be calculated using the Equation 1.

$$\boxed{disparity = |x| + |x'| = \frac{Bf}{Z}} \tag{1}$$

x and x' is the distance between the matched points from each image plane and their camera center. B is the distance between two cameras (also known as the baseline) and f is the focal length of the camera. Z is the distance of the stereo camera from the detected object as presented in Figure 51.

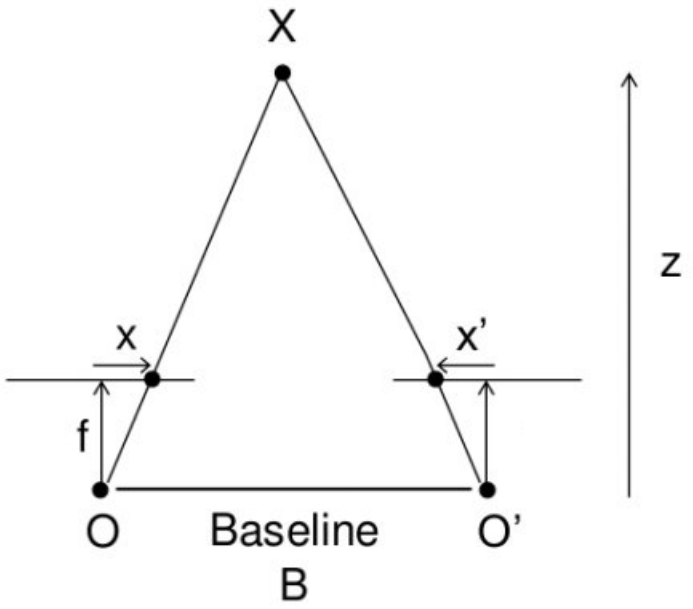


Figure 51: Disparity to Depth

The depth image can be calculated using Equation 2 for each matched point.

$$\boxed{Z = \frac{Bf}{disparity}} \tag{2}$$

With the distance from each object from the depth camera, the laserscan message (the colored line) can be calculated as presented in Figure 52 with the color red representing the closest object and the color purple representing the furthest object.

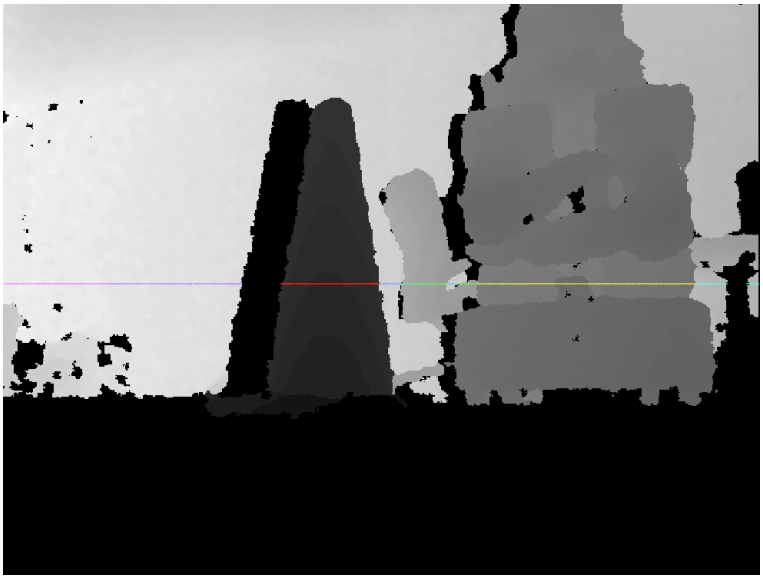


Figure 52: Depth Image to Laserscan

The colors of the laserscan points depend on the distance of the robot from each point in the laserscan message. The different colors are presented in Figure 53.

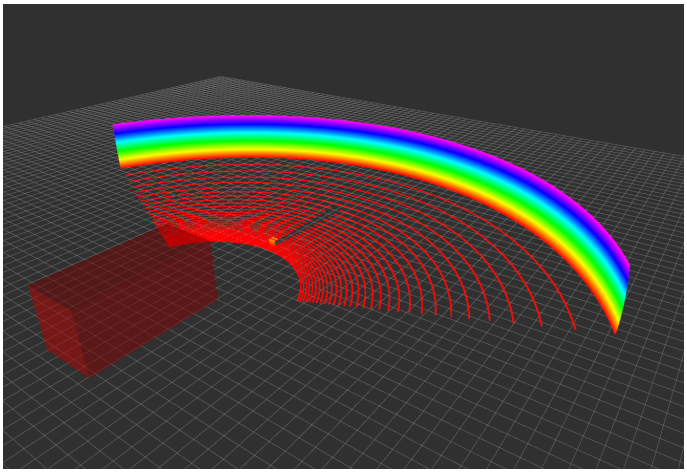


Figure 53: Laserscan Colors

3.2.3 Packages Used to Convert a Disparity Image to Laserscan

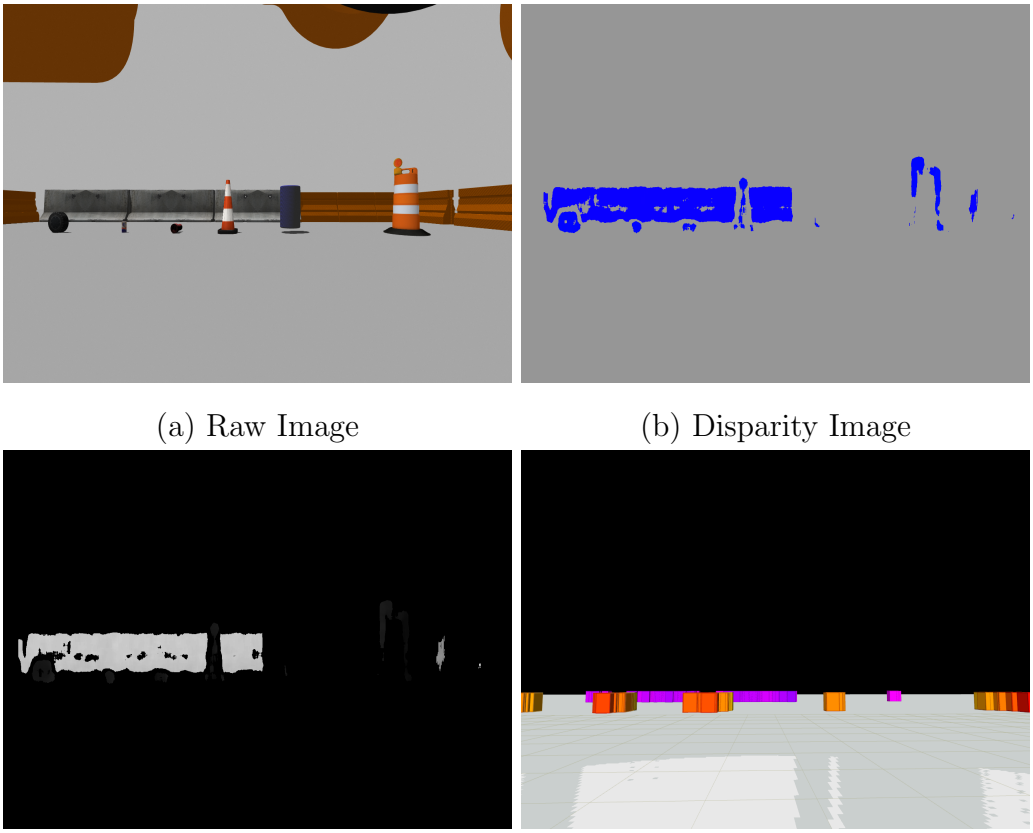
The published topics from the stereo camera (Figure 47) are used to produce a laserscan message and the packages needed are :

- **stereo_image_proc** : Processes the images from both cameras, undistorting and colorizing the raw images and will also compute disparity images.

- **disparity_image_proc** : Package that can convert ROS disparity images to depth images. The depth image produced has values that represent the distance of the object from the stereo camera.

- **depthimage_to_laserscan** [18] : This package generates a 2D laser scan from a depth image based on the provided parameters.

In Figure 58 the raw, disparity and depth images are presented as well as the laserscan produced.



(a) Raw Image (b) Disparity Image

(c) Depth Image (d) Laserscan

Figure 58: Raw (a) Disparity (b) Depth(c) Laserscan (d)

The computation in ROS is performed by using a network of processes called ROS nodes. A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using messages that are a data structure containing a typed field, which can hold a set of data and can be sent to another node. Each message in ROS is transported using topics. When a node sends a message through a topic, the node is publishing a topic, and when a node receives a message through a topic the node is subscribing to a topic. The communication of the nodes are presented in a ROS Communication Graph in Figure 59.

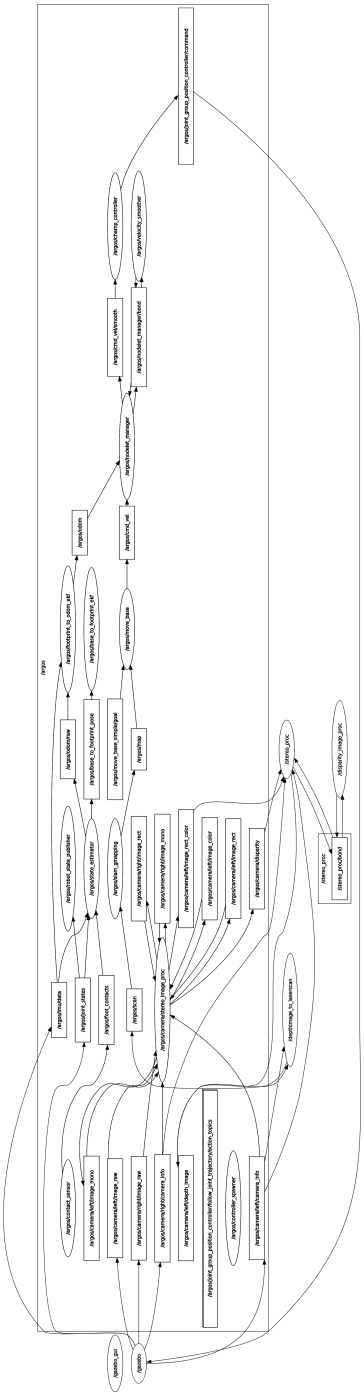


Figure 59: Complete ROS Graph

The rectangles represent the topics and the ellipses represent the nodes. The arrows represent the communication between nodes. If the arrow is inbound then the node is subscribed to the topic, whereas if the arrow is outbound then the node is publishing to the topic. The ROS Graph of the communication of the packages used to convert the images to laserscan are presented in Figures 60-62.

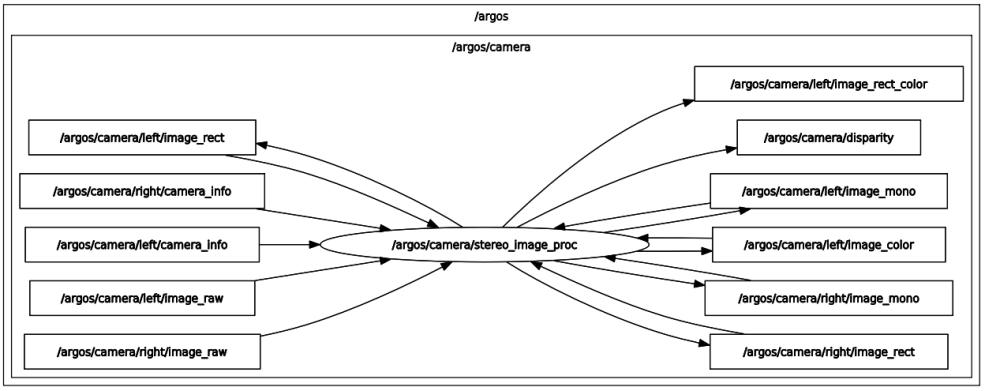


Figure 60: stereo_image_proc

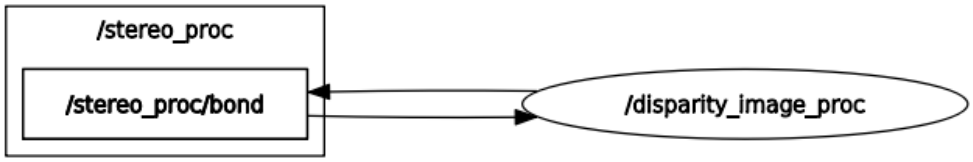


Figure 61: disparity_image_proc

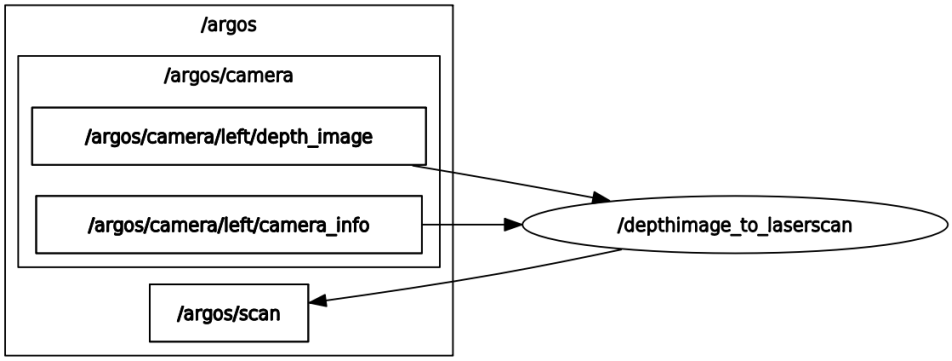


Figure 62: depth_to_laserscan

3.2.4 Launch Files

To use the stereo_image_proc package the namespace of the stereo camera and the parameters of the stereo_image_proc package have to be defined on the launch file as shown in Figure 63.

```
<node ns="argos/camera" pkg="stereo_image_proc"
    type="stereo_image_proc" name="stereo_image_proc">
    <param name="prefilter_size"        value="5"/>
    <param name="prefilter_cap"         value="25"/>
    <param name="correlation_window_size" value="21"/>
    <param name="min_disparity"         value="-59"/>
    <param name="disparity_range"       value="224"/>
    <param name="uniqueness_ratio"      value="69.0"/>
    <param name="texture_threshold"     value="10"/>
    <param name="speckle_size"          value="0"/>
    <param name="speckle_range"         value="2"/>
</node>
```

Figure 63: Stereo Image Proc Launch

The stereo_image_proc parameters [19] are :

- prefilter_size, prefilter_cap : they control the pre-filtering phase, which normalizes image brightness and enhances texture in preparation for block matching.

- correlation_window_size : controls the size of the sliding SAD (sum of absolute differences) window used to find matching points between the left and right images.
- min_disparity : controls the offset from the x-position of the left pixel at which to begin searching.
- disparity_range : How many pixels to slide the window over. The larger it is, the larger the range of visible depths, but more computation is required.
- uniqueness_ration : controls another post-filtering step. For each disparity filter out the disparities on a circle around it with this parameter representing the radius of the circle.
- texture_threshold : filters out areas that don't have enough texture for reliable matching.
- speckle_size : the number of pixels below which a disparity blob is dismissed as a "speckle".
- speckle_range : controls how close in value disparities must be to be considered part of the same blob.

The parameter values for the stereo_image_proc package in Figure 63 were chosen so the obstacles were clearly distinguished.

The disparity image is converted to a depth image using the disparity_image_proc package. To use this package the topics of the launch file need to be remapped to agree with the topics published from the stereo_image_proc package as presented in Figure 64.

```
<node name="disparity_image_proc" pkg="nodelet" type="nodelet"
    args="load disparity_image_proc/depth_image stereo_proc"
    output="screen">
  <remap from="/right/camera_info"
      to="/argos/camera/right/camera_info"/>
  <remap from="/left/camera_info"
      to="/argos/camera/left/camera_info"/>
  <remap from="/left/image_rect_color"
      to="/argos/camera/left/image_rect_color"/>
  <remap from="/disparity" to="/argos/camera/disparity"/>
  <remap from="/depth_image" to="/argos/camera/left/depth_image"/>
</node>
```

Figure 64: Disparity Image Proc Launch

The produced laserscan message from the depthimage_to_laserscan package is published by default on the topic /scan. The robot is named ARGOS so all topics published from the robot's accessories have a namespace of /argos e.g. the stereo camera publishes its measurements on topics with namespace argos/camera/. Gmapping searches for the laserscan message at the topic named /argos/scan. For that reason the /scan topic, which is the default topic in the depthimage_to_laserscan package, has to be remapped to /argos/scan. To accomplish that, the executable file DepthImageToLaserScanNodelet was loaded from the arguments on the launch file, and following that, the /scan topic can be remapped to /argos/scan as presented in Figure 65.

```
  <node name="depthimage_to_laserscan"
      pkg="depthimage_to_laserscan" type="depthimage_to_laserscan"
      args="load
      depthimage_to_laserscan/DepthImageToLaserScanNodelet">
    <remap from="image"     to="/argos/camera/left/depth_image"/>
    <remap from="camera_info" to="camera/color/camera_info"/>
    <remap from="scan" to="argos/scan"/>
```

Figure 65: Scan topic Remapped in DepthImage to Laserscan Launch File

A launch file was created to include all parameters, remappings and ar-

guments. Firstly the arguments with their default values are presented in Figure 66.

```
<arg name="robot_name"      default="argos"/>
<arg name="rviz"            default="false"/>
<arg name="lite"            default="false" />
<arg name="ros_control_file" default="$(find
   argos_config)/config/ros_control/ros_control.yaml" />
<arg name="gazebo_world"    default="$(find
   argos_config)/worlds/outdoor_new.world" />
<arg name="gui"             default="true"/>
<arg name="world_init_x"    default="0.0" />
<arg name="world_init_y"    default="0.0" />
<arg name="world_init_z"    default="1.08" />
<arg name="world_init_heading" default="0.0" />
<param name="use_sim_time" value="true" />
```

Figure 66: Arguments in Launch File

Following that, the bringup.launch file and the gazebo.launch file are launched to spawn the model, start the controllers and start gazebo as presented in Figure 67.

```
<include file="$(find argos_config)/launch/bringup.launch">
    <arg name="robot_name"          value="$(arg robot_name)"/>
    <arg name="gazebo"              value="true"/>
    <arg name="lite"                value="$(arg lite)"/>
    <arg name="rviz"                value="$(arg rviz)"/>
    <arg name="joint_controller_topic"
        value="joint_group_position_controller/command"/>
    <arg name="hardware_connected"  value="false"/>
    <arg name="publish_foot_contacts" value="false"/>
    <arg name="close_loop_odom"     value="true"/>
</include>
<include file="$(find champ_gazebo)/launch/gazebo.launch">
    <arg name="robot_name"      value="$(arg robot_name)"/>
    <arg name="lite"            value="$(arg lite)"/>
    <arg name="ros_control_file" value="$(arg ros_control_file)"/>
    <arg name="gazebo_world"    value="$(arg gazebo_world)"/>
    <arg name="world_init_x"    value="$(arg world_init_x)" />
    <arg name="world_init_y"    value="$(arg world_init_y)" />
    <arg name="world_init_z"    value="$(arg world_init_z)" />
    <arg name="world_init_heading" value="$(arg
        world_init_heading)" />
    <arg name="gui"             value="$(arg gui)" />
</include>
```

Figure 67: Gazebo and Controllers Launch

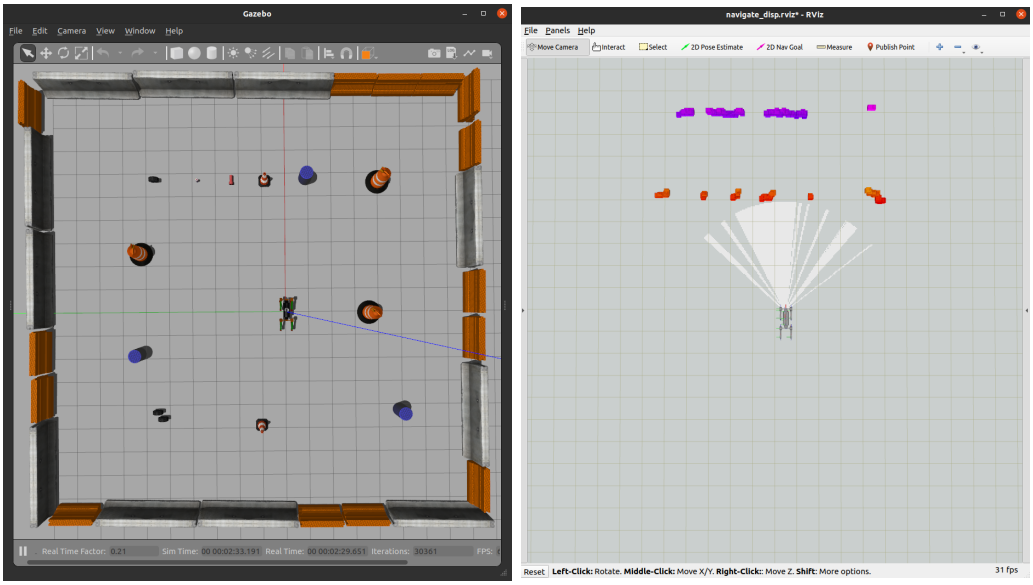
Lastly, the packages for the image processing are launched to produce the laserscan for the gmapping packages in Figure 68.

```
<node ns="argos/camera" pkg="stereo_image_proc"
    type="stereo_image_proc" name="stereo_image_proc"/>
<include file="$(find
    argos_config)/launch/disparity_to_depth.launch" />
<include file="$(find argos_config)/launch/depth_to_laser.launch"
    />
```

Figure 68: Image Processing Packages

3.2.5 Produced Laserscan

The Gazebo world with the resulting laserscan is presented in Figure 71.

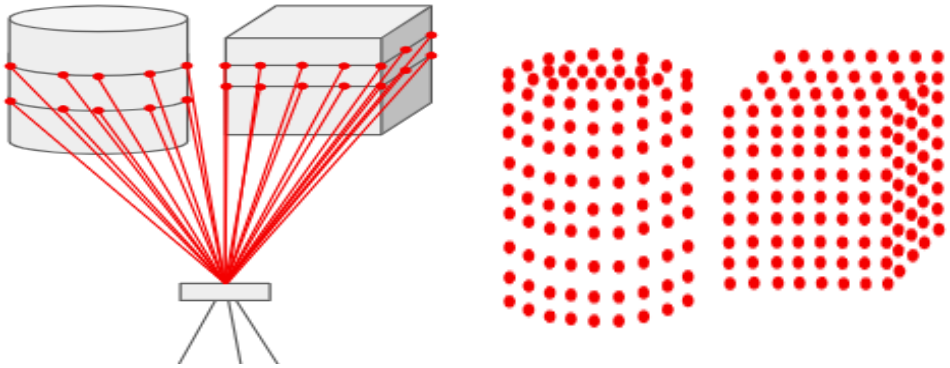


(a) Gazebo World (b) Laserscan

Figure 71: Gazebo World (a) and Produced Laserscan (b) (Disparity to Laserscan)

3.3 PointCloud To Laserscan

Another method of producing a laserscan is by converting a pointcloud message to Laserscan. Point clouds are sets of points that describe an object or surface and it's position. The measurements are usually made by 3D laser scanners as presented in Figure 74.



(a) 3D Laser scanner computing a Pointcloud

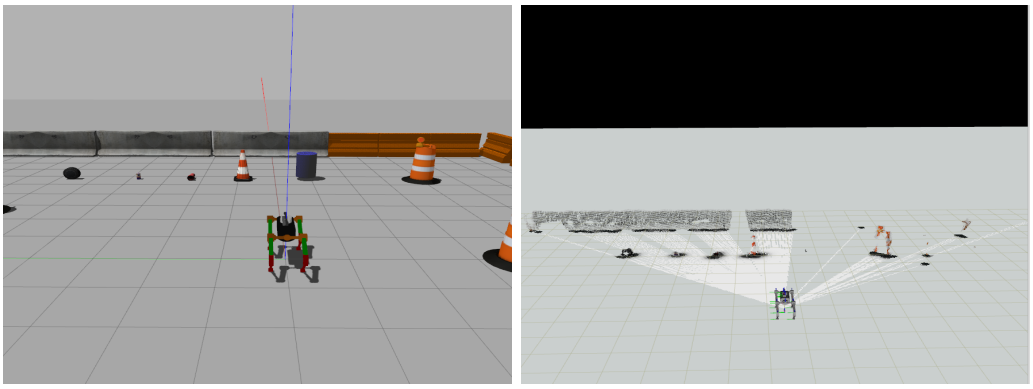
(b) Computed Pointcloud

Figure 74: 3D Laser Scanner (a) and Computed PointCloud (b)

The distance of the objects described by the pointcloud are known, because their position is known, and as a result a laserscan message can be calculated that describes the distances of the robot from the objects.

3.3.1 Packages Used to Convert a PointCloud to Laserscan

The stereo_image_proc package, described in Paragraph 3.2.3, has the ability to compute a pointcloud message as presented in Figure 77.



(a) Gazebo World

(b) PointCloud

Figure 77: Gazebo World (a) and Produced PointCloud (b)

The computed Pointcloud is published on the topic in Figure 78.

```
/argos/camera/points2
```

Figure 78: PointCloud Topic

To convert the computed pointcloud to a laserscan the pointcloud_to_laserscan package [20] is used, which can convert a 3D Point Cloud into a 2D laserscan by evaluating the distance of the different points from the stereo camera.

3.3.2 Launch Files

The stereo_image_proc package launch matches the launch from Paragraph 3.2.4. The pointcloud is converted to a laserscan using the pointcloud_to_laserscan node. The cloud_in parameter is remapped to the /camera/points2 topic (the topic from Figure 78) and the parameters angle_min, angle_max are changed to match the ZED2 camera's field of view. The launch file for the pointcloud_to_laserscan node is presented in Figure 79.

```
<node pkg="pointcloud_to_laserscan"
    type="pointcloud_to_laserscan_node"
    name="pointcloud_to_laserscan">

    <remap from="cloud_in" to="camera/points2"/>
    <remap from="scan" to="scan"/>
    <rosparam>
        target_frame: argos/camera_center # Leave disabled to
            output scan in pointcloud frame
        transform_tolerance: 0.01
        min_height: -1.0 # -1 because it is -1 from the
            camera_center position not the foot of the robot
        max_height: 1.0

        angle_min: -0.95993109
        angle_max: 0.95993109
        angle_increment: 0.0087 # M_PI/360.0
        scan_time: 0.3333
        range_min: 0.2
        range_max: 20.0
        use_inf: true
        inf_epsilon: 1.0

        # Concurrency level, affects number of pointclouds queued
            for processing and number of threads used
        # 0 : Detect number of cores
        # 1 : Single threaded
        # 2->inf : Parallelism level
        concurrency_level: 2
    </rosparam>

</node>
```

Figure 79: PointCloud_to_laserscan node launch file

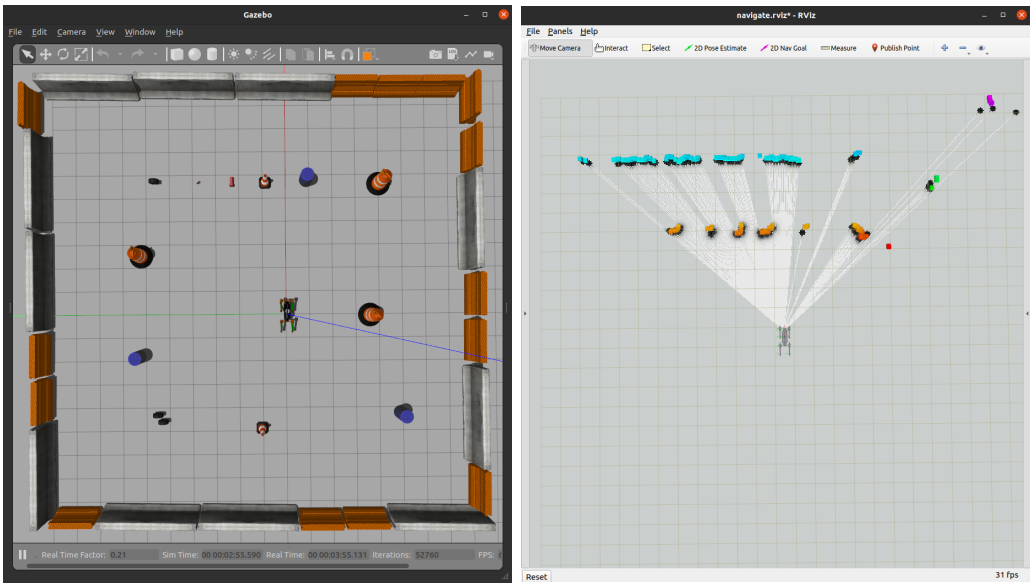
The pointcloud_to_laserscan node launch file is launched from the gazebo launch file with namespace /argos, as presented in Figure 80.

```
<group ns="argos">
    <include file="$(find
        argos_config)/launch/pointcloud_to_laserscan/
    point_to_laser_node.launch" />
</group>
```

Figure 80: pointcloud_to_laserscan launch

3.3.3 Produced Laserscan

The produced laserscan is presented in Figure 83.



(a) Gazebo World (b) Produced Laserscan

Figure 83: Gazebo World (a) and Produced Laserscan (b) (PointCloud to Laserscan)

3.4 Depth Camera To Laserscan

3.4.1 URDF for Depth Camera

ARGOS can be additionally equipped with a depth camera, which produces a depth image that can be converted to a laserscan. To equip ARGOS

with additional accessories, e.g. a depth camera, the new accessory needs to be included in the URDF file that describes the robot. The camera and the gazebo plugin for the depth camera are presented in Figures 84 and 85 respectively. The values for the parameter were chosen to match the ZED2 depth camera specifications.

```
<sensor type="depth" name="openni_camera_camera">
<always_on>1</always_on>
<visualize>true</visualize>
<camera>
    <horizontal_fov>1.91986218</horizontal_fov>
    <vertical_fov>1.22173048</vertical_fov>
    <diagonal_fov>2.0943951</diagonal_fov>
    <image>
        <width>1920</width>
        <height>1080</height>
        <format>R8G8B8</format>
    </image>
    <clip>
        <near>0.1</near>
        <far>25.0</far>
    </clip>
</camera>
</sensor>
```

Figure 84: Depth Camera URDF

```
<plugin name="camera_plugin"
   filename="libgazebo_ros_openni_kinect.so">
   <robotNamespace>argos</robotNamespace>
   <baseline>0.2</baseline>
   <alwaysOn>true</alwaysOn>
   <!-- Keep this zero, update_rate in the parent <sensor> tag
      will control the frame rate. -->
   <updateRate>0.0</updateRate>
   <cameraName>depth_camera</cameraName>
   <imageTopicName>/depth/image_rect_color</imageTopicName>
   <cameraInfoTopicName>/argos/depth/camera_info
      </cameraInfoTopicName>
   <depthImageTopicName>/argos/depth/depth_registered
      </depthImageTopicName>
   <depthImageCameraInfoTopicName>/argos/depth/image_raw/
      camera_info</depthImageCameraInfoTopicName>
   <pointCloudTopicName>/argos/depth/point_cloud/cloud_registered
      </pointCloudTopicName>
   <frameName>left_camera_optical_frame</frameName>
   <pointCloudCutoff>0.4</pointCloudCutoff>
   <!-- does nothing it seems
   <pointCloudCutoffMax>20.0</pointCloudCutoffMax> -->
   <distortionK1>0</distortionK1>
   <distortionK2>0</distortionK2>
   <distortionK3>0</distortionK3>
   <distortionT1>0</distortionT1>
   <distortionT2>0</distortionT2>
   <CxPrime>0</CxPrime>
   <Cx>0</Cx>
   <Cy>0</Cy>
   <focalLength>0</focalLength>
   <hackBaseline>0</hackBaseline>
</plugin>
```

Figure 85: Depth Camera Plugin URDF

3.4.2 Packages Used to Convert Depth Image to Laserscan

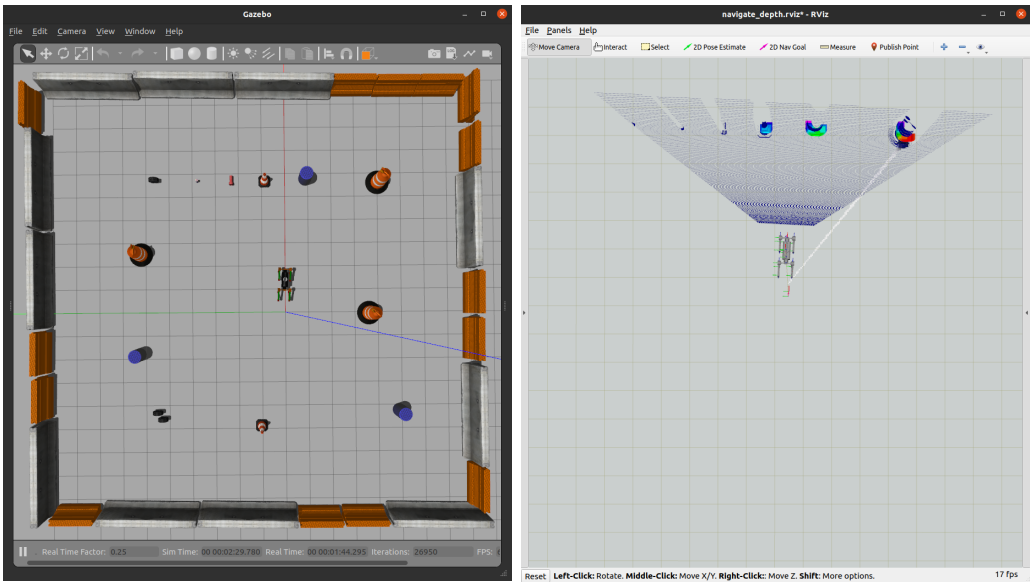
To Convert the depth image to a laserscan the depthimage_to_laserscan package is used, described in Paragraph 3.2.3. The package is launched with the topics remapped to match the depth camera's published topics, as presented in Figure 86.

```
<remap from="image"     to="/argos/depth/depth_registered"/>
<remap from="camera_info" to="/argos/depth/camera_info"/>
<remap from="scan" to="argos/scan"/>
```

Figure 86: Depth Camera Launch file

3.4.3 Produced Laserscan

The resulting laserscan is presented in Figure 89 with the blue lines representing the depth image.



(a) Gazebo World (b) Produced Laserscan

Figure 89: Gazebo World (a) and Produced Laserscan (b) (Depth Camera to Laserscan)

3.5 Disparity To Laserscan With Ground Truth Position

As mentioned in Paragraph 3.1 gmapping tries to correct errors in odometry by transforming the position of the robot, resulting in position "jumps" of the robot. These "jumps" happen in RVIZ (3D visualization tool for ROS), which is the position the robot thinks it is currently on (estimated position), not the real position of the robot. To avoid these sudden movements in RVIZ, instead of using a state estimator, the real position can be used, which can be published by gazebo.

3.5.1 Ground Truth URDF

To obtain the real position of the robot the gazebo plugin in Figure 90 needs to be included in the URDF file.

```
<gazebo>
  <plugin name="p3d_base_controller"
     filename="libgazebo_ros_p3d.so">
       <alwaysOn>true</alwaysOn>
       <updateRate>50.0</updateRate>
       <bodyName>base_link</bodyName>
       <topicName>odom/ground_truth</topicName>
       <gaussianNoise>0.001</gaussianNoise>
       <frameName>world</frameName>
       <xyzOffsets>0 0 0</xyzOffsets>
       <rpyOffsets>0 0 0</rpyOffsets>
    </plugin>
</gazebo>
```

Figure 90: Ground Truth Plugin

This gazebo plugin publishes the real position of the base_link, with respect to the world frame, on the topic /odom/ground_truth. The published message is of type nav_msgs/Odometry (Figure 91).

```
# This represents an estimate of a position and velocity in free
    space.
# The pose in this message should be specified in the coordinate
    frame given by header.frame_id.
# The twist in this message should be specified in the coordinate
    frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Figure 91: nav_msgs/Odometry file

However, gmapping is subscribed to the /tf topic [14] and requires a message of type tf2_msgs/TFMessage (Figure 92), which contains all the transforms in the ROS TF tree (Paragraph 2.2.3).

```
geometry_msgs/TransformStamped[] transforms
```

Figure 92: TFMessage file

3.5.2 Packages Used to Convert Message Type

To convert a message of type nav_msgs/Odometry to a tf2_msgs/TFMessage the package **message_to_tf** [21] is used, which translates pose information from nav_msgs/Odometry, geometry_msgs/PoseStamped or sensor_msgs/Imu message types to tf2_msgs/TFMessage. The message_to_tf package has the following parameters :

- frame_id : The world frame.
- stabilized_frame_id : The name of the stabilized base frame (takes into account the imu data to stabilize any movement of the base).
- footprint_frame_id : The name of the base footprint frame.
- child_frame_id : The name of the child frame (base_link).
- imu_topic : The name of the topic the imu publishes it's data on.

and takes as argument the topic name on which the message to be converted is being published.

3.5.3 Launch file

The message_to_tf package is launched as presented in Figure 93.

```
<arg name="world_frame"         default="argos/world"/>
<arg name="namespace"           default="argos"/>
<arg name="child_frame_id"      default="argos/base_link"/>
<arg name="imu_topic"           default="argos/imu/data"/>

<node name="odom_to_tf" pkg="message_to_tf" type="message_to_tf"
    args="argos/odom/ground_truth">
    <param name="~frame_id"             value="$(arg world_frame)"
        />
    <param name="~stabilized_frame_id"  value="/$(arg
        namespace)/base_stabilized" />
    <param name="~footprint_frame_id"   value="/$(arg
        namespace)/base_footprint" />
    <param name="~child_frame_id"       value="$(arg
        child_frame_id)" />
    <param name="~imu_topic"            value="$(arg imu_topic)"/>
</node>
```

Figure 93: Message to Tf Launch

The ROS TF tree using the message_to_tf package is presented in Figure 94.

Figure 94: ROS TF Frame Tree with Ground Truth

4 Project Challenges

Choosing the parameters for the controllers as well as the parameters for the movement of the robot was challenging.

4.1 Controller Parameters

Each actuator is controlled by a PID controller. For ARGOS the PD controller was chosen. The P gain for each controller was chosen with a value of 2500 to be able to stabilize the weight of the robot while at the same time keeping joint effort under $250Nm$ (the robot's weight is approximately 80kg) and velocity under $5.0\frac{rad}{s}$ for each actuator. Lower values for the P gain resulted in instability as presented in Figure 95 with a P gain of 500.

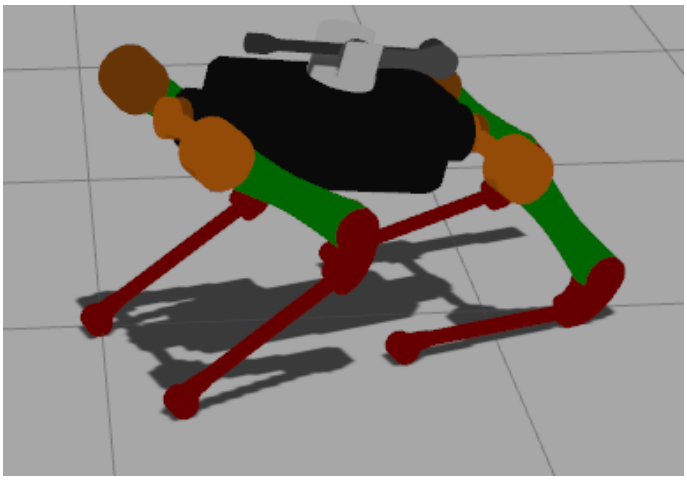
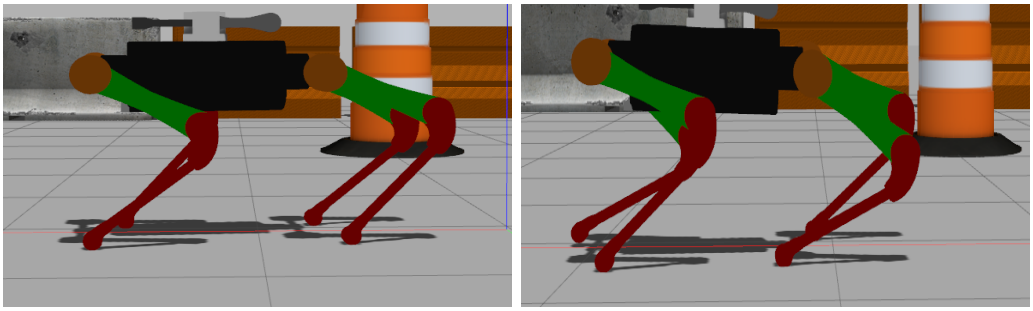


Figure 95: Robot Instability

ARGOS with a P value of 1500 was not able to stabilize the orientation of the robot's body while moving towards a goal because the weight was distributed to two of the four legs as presented in Figure 98.



(a) Robot Stationary with P = 1500 (b) Robot Walking with P = 1500

Figure 98: Robot Stationary (a) and Robot Walking (b)

The D gain for each controller was chosen with a value of 80 to remove unwanted oscillations. Lower values meant that ARGOS would oscillate for a long period of time. In Figures 99-100 the efforts and the velocities of each actuator are presented when the robot is stationary and when the robot is moving and it can be observed that the oscillations are minimized rapidly.

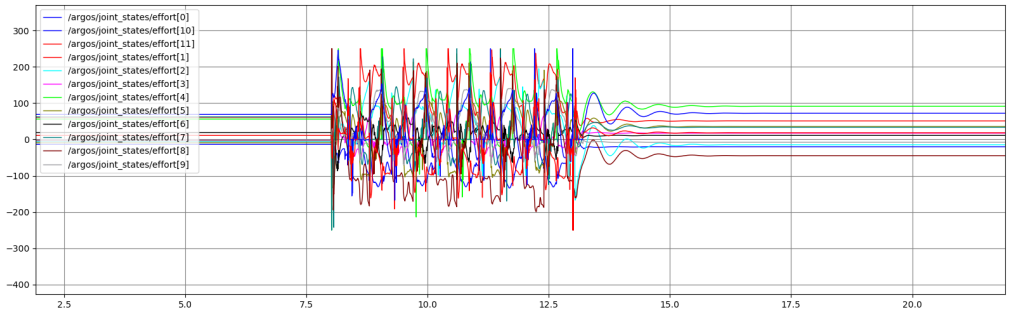


Figure 99: Joint Efforts

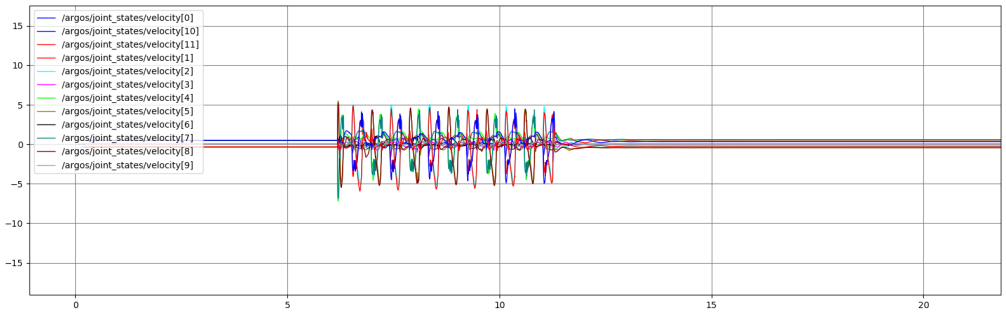


Figure 100: Joint Velocities

The values chosen for each actuator are displayed in Figure 101.

```
gains:
  LF_hip_joint       : {p: 2500, d: 80.0, i: 0.0}
  LF_upper_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  LF_lower_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  RF_hip_joint       : {p: 2500, d: 80.0, i: 0.0}
  RF_upper_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  RF_lower_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  LH_hip_joint       : {p: 2500, d: 80.0, i: 0.0}
  LH_upper_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  LH_lower_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  RH_hip_joint       : {p: 2500, d: 80.0, i: 0.0}
  RH_upper_leg_joint : {p: 2500, d: 80.0, i: 0.0}
  RH_lower_leg_joint : {p: 2500, d: 80.0, i: 0.0}
```

Figure 101: PID Gains

4.2 Gait Parameters

The gait parameters for CHAMP are :

- Knee Orientation : There are 4 configurable orientations : .>> .>< .<< .<> where dot is the front side of the robot.
- Max Linear Velocity X (meters/second) : Robot's maximum forward/reverse speed.
- Max Linear Velocity Y (meters/second) : Robot's maximum speed when moving sideways.

- Max Angular Velocity Z (radians/second) : Robot's maximum rotational speed.
- Stance Duration (seconds) : How long should each leg spend on the ground while walking.
- Leg Swing Height (meters) : Toe Trajectory height during swing phase.
- Leg Stance Height (meters) : Toe Trajectory depth during stance phase.
- Robot Walking Height (meters) : Distance from hip to the ground while walking.
- CoM X Translation (meters) : This parameter is used to compensate for the weight of the robot if the center of mass is not in the middle of the robot.
- Odometry Scaler : This parameter can be used as a multiplier to the calculated velocities for dead reckoning. This can be useful to compensate odometry errors on open-loop systems.

The gait parameters are summarized in Figure 102.

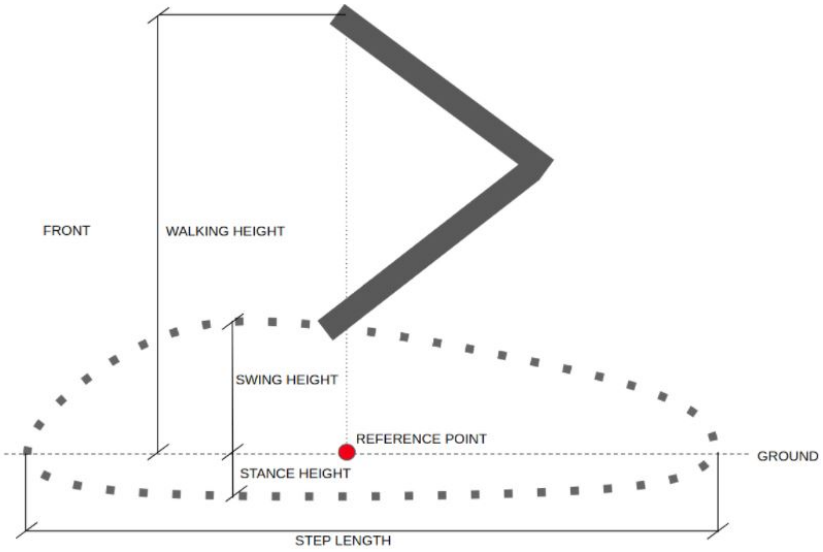


Figure 102: Gait Parameters

The parameter values that were chosen for ARGOS are presented in Figure 103.

```
knee_orientation : ">>"
pantograph_leg : false
odom_scaler: 1.8
max_linear_velocity_x : 1.5
max_linear_velocity_y : 0.25
max_angular_velocity_z : 1.0
com_x_translation : -0.07
swing_height : 0.15
stance_depth : 0.0
stance_duration : 0.65
nominal_height : 0.7
```

Figure 103: Chosen Gait Parameters

5 Results

5.1 Robot's Stability and Movement

In this section the results are presented of the robot's stability when idle, and when moving while avoiding obstacles detected by the stereo camera. The simulation is launched using the command in Figure 104.

```
rolaunch argos_config gazebo.launch
```

Figure 104: Gazebo Launch

This command launches the gazebo world and spawns the robot from height of 1.08 meters (which is the height of the robot with stretched legs). In addition it launches the packages needed for the conversion of the stereo camera messages to the laserscan. In Figure 105 the robot is at its starting position.

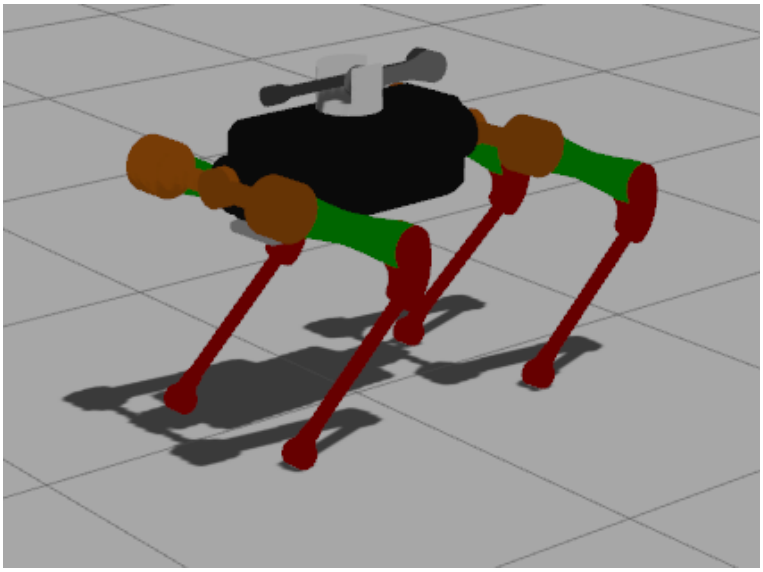


Figure 105: Starting Position of the Robot

To launch the obstacle avoidance using gmapping packages the command in Figure 106 is executed.

```
roslaunch argos_config slam.launch
```

Figure 106: Command for Obstacle Avoidance with Gmapping Packages

This command launches the gmapping packages needed, as well as rviz to be able to move the robot. With the command 2D Nav Goal the robot can be instructed to move to a position in the map with a certain orientation. The goal that was set for ARGOS is shown in Figure 107.

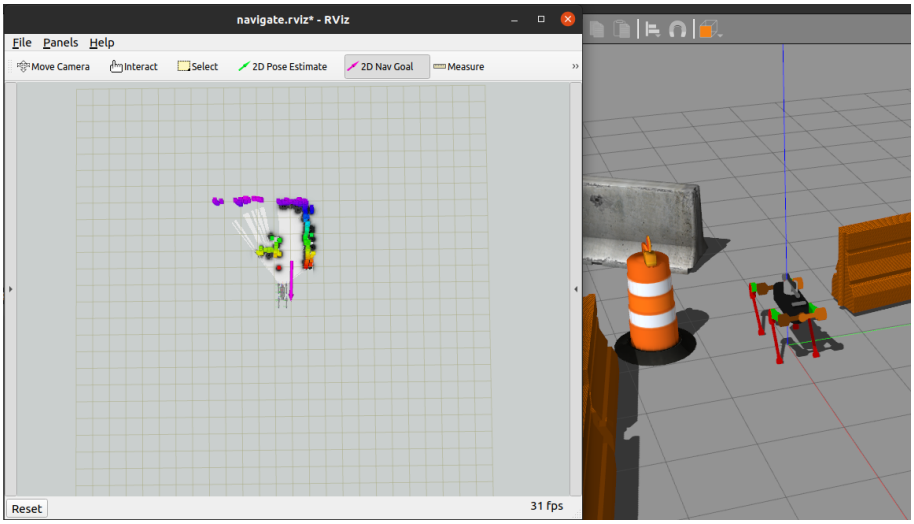


Figure 107: Robot Goal with Orientation (left) and ARGOS Position (right)

The robot after given the instruction calculates a path to follow and starts the movement towards the goal as shown in Figure 108 (left to right, top to bottom).

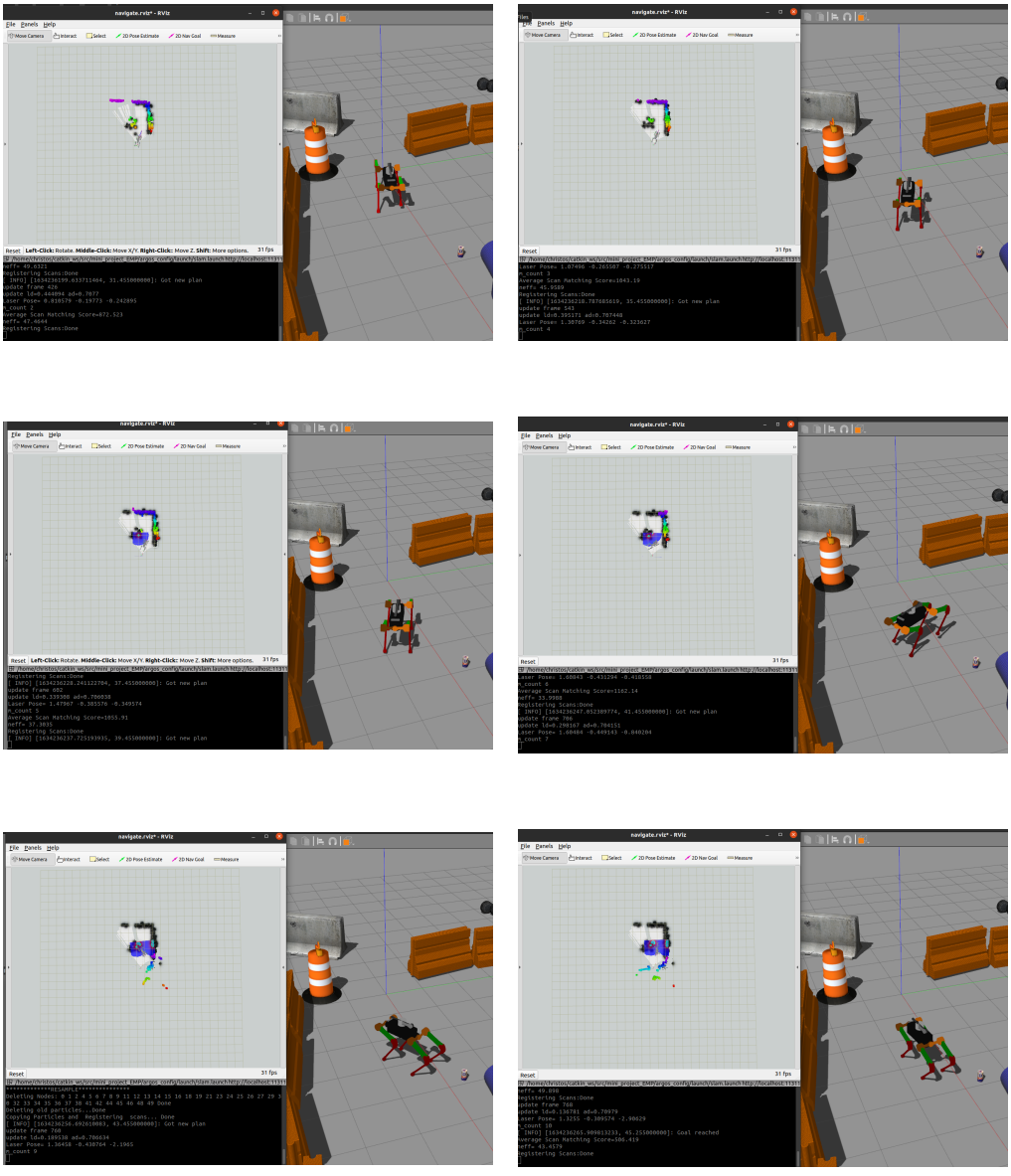


Figure 108: Movement of the Robot Towards Goal

The path planning also avoids obstacles detected by the laserscan as shown in Figure 109.

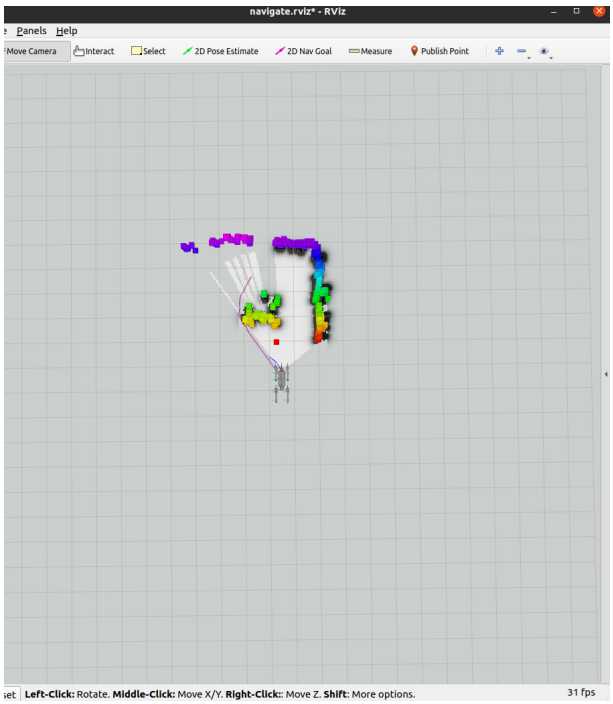


Figure 109: Obstacle Avoidance

5.2 Obstacle Detection : Disparity Image to Laserscan

In this section the results for obstacle detection using the disparity image to laserscan approach are presented. To distinguish the obstacles that can be detected, a new gazebo world was created with obstacles of different size (biggest obstacle is a barrel, smallest obstacle is a drink can). The gazebo world is presented in Figure 110.

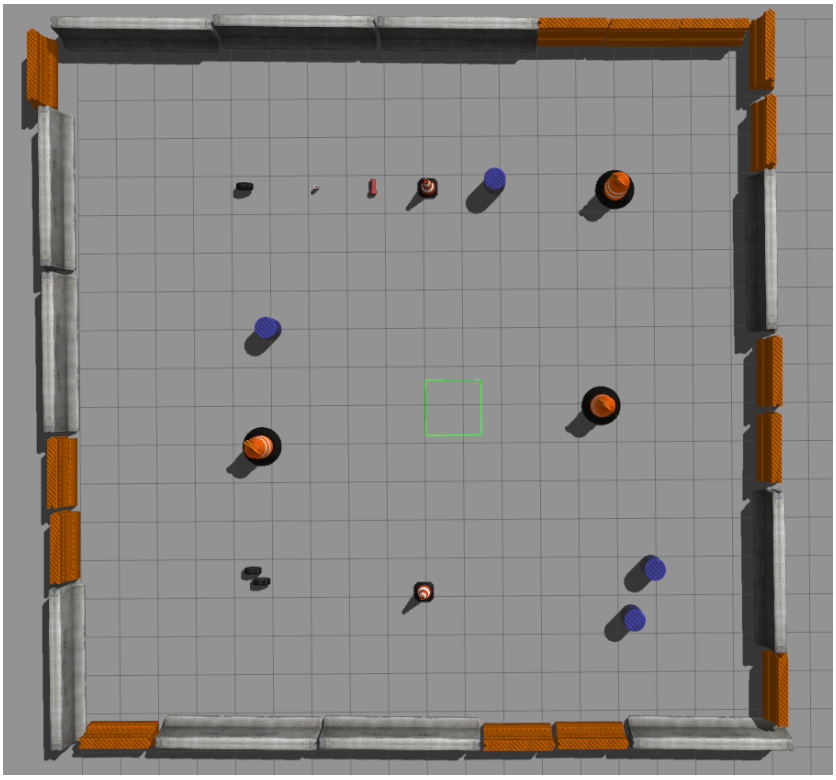


Figure 110: Gazebo Obstacle World

To spawn ARGOS in the new gazebo world the argument gazebo_world in the launch file is changed to match the newly created world as presented in Figure 111.

```
<arg name="gazebo_world"    default="$(find
    argos_config)/worlds/arena_world.world" />
```

Figure 111: Gazebo World Argument

The commands in Figure 112 are executed in separate terminals to launch the gazebo world, spawn ARGOS with the disparity image to laserscan packages and launch the gmapping packages.

```
roslaunch argos_config disp_gazebo.launch

roslaunch argos_config slam.launch
```

Figure 112: Commands to Launch Gazebo and Gmapping : Disparity Image To Laserscan

The gazebo obstacle world with ARGOS is presented in Figure 113.

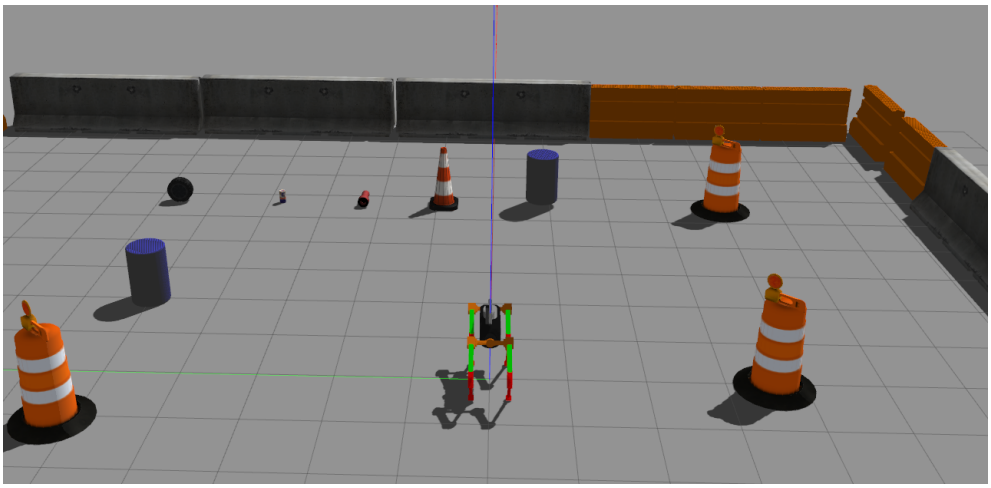
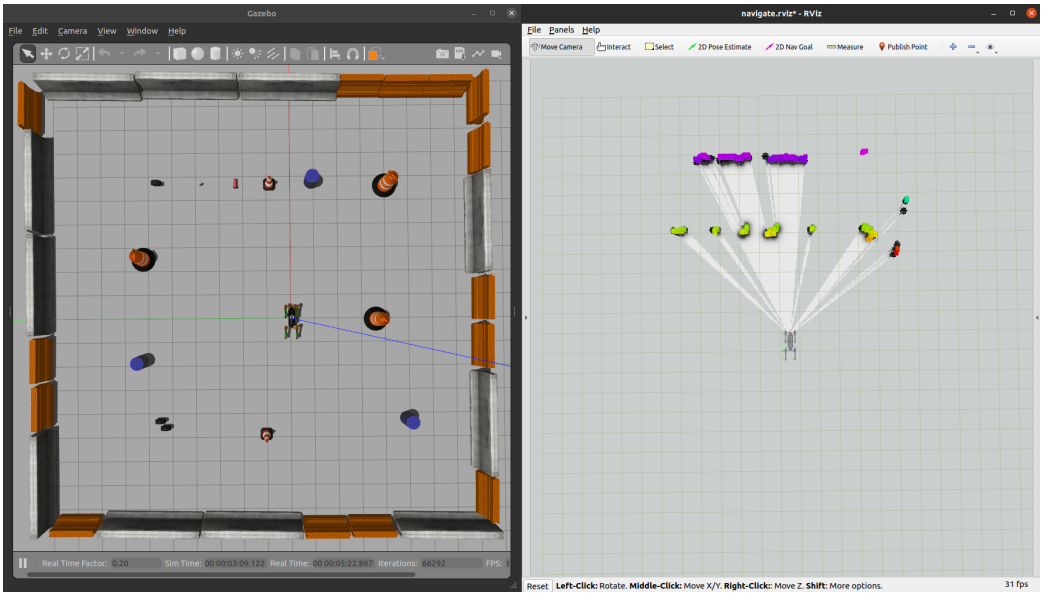


Figure 113: ARGOS in Gazebo Obstacle World

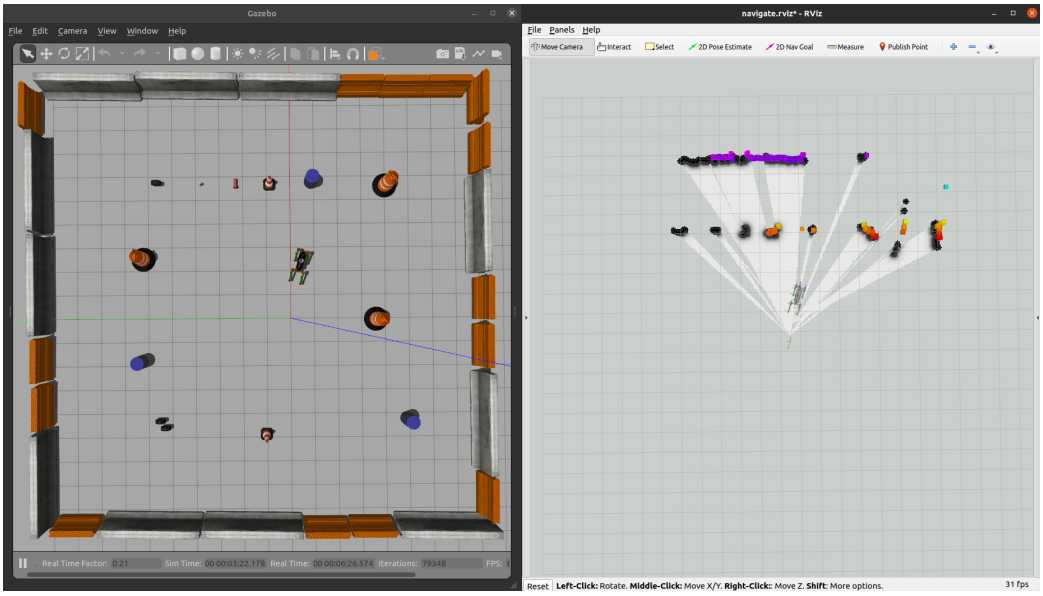
In Figure 114 ARGOS is at starting position and the laserscan can detect all obstacles of different size.



(a) Gazebo World (b) Obstacles Detected by Laserscan

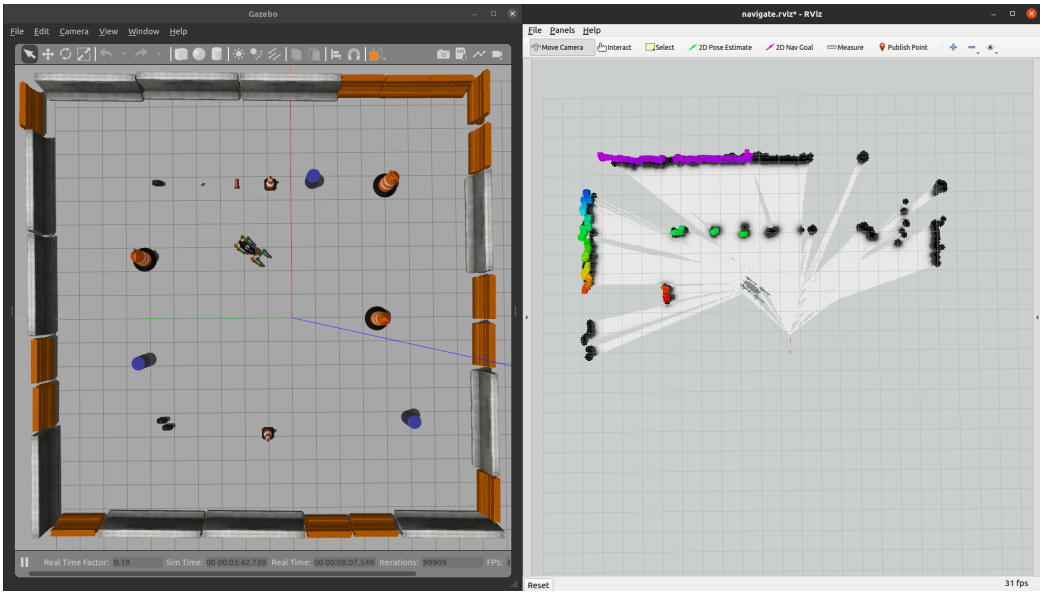
Figure 114: Obstacle Detection : Starting Position

In Figures 115-116 ARGOS is instructed to move closer to the obstacles to verify that the laserscan can still detect them.



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 115: Obstacle Detection : Big Obstacles



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 116: Obstacle Detection : Small Obstacles

In can be verified from Figures 114-116 that the laserscan can detect all obstacles in this gazebo world.

5.3 Obstacle Detection : PointCloud to Laserscan

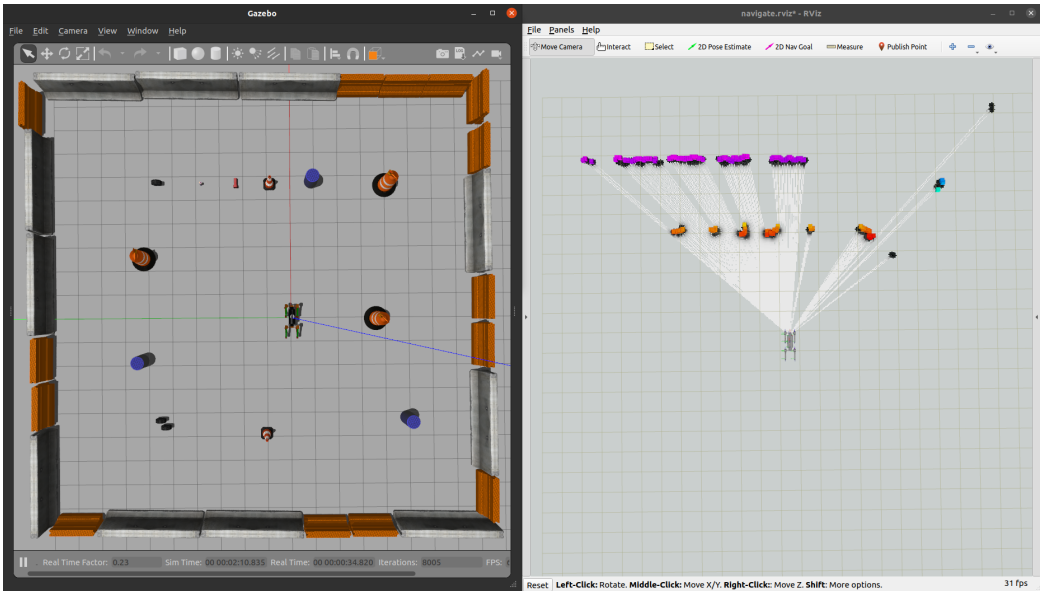
In this section the results for the obstacle detection using the pointcloud to laserscan approach are presented. The commands in Figure 117 are executed in separate terminals to launch the gazebo world, spawn ARGOS with the pointcloud to laserscan packages and launch the gmapping packages.

```
roslaunch argos_config point_gazebo.launch

roslaunch argos_config slam.launch
```

Figure 117: Commands to Launch Gazebo and Gmapping : Pointcloud To Laserscan

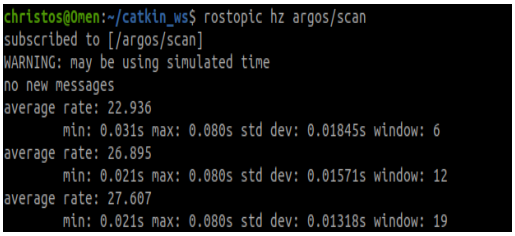
In Figure 118 ARGOS is at starting position and the laserscan can detect all obstacles of different size.



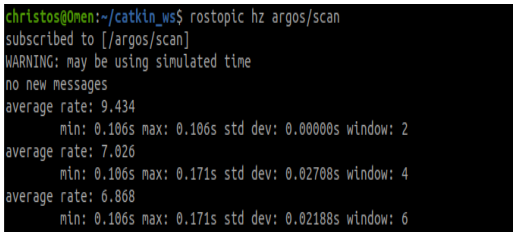
(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 118: Obstacle Detection : Starting Position

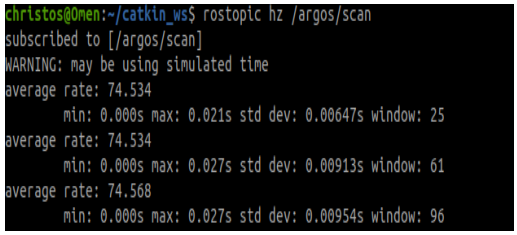
It is worth noting that processing pointclouds takes a long time so the publishing rate of the laserscan topic /argos/scan is slower (average rate 8Hz) than the other approaches (Disparity Image to Laserscan (average rate 25Hz), Depth Camera to Laserscan (average rate 74Hz)) as presented in Figure 122 using the *rostopic hz* command(returns the publishing rate of a topic).



(a) Disparity to Laserscan



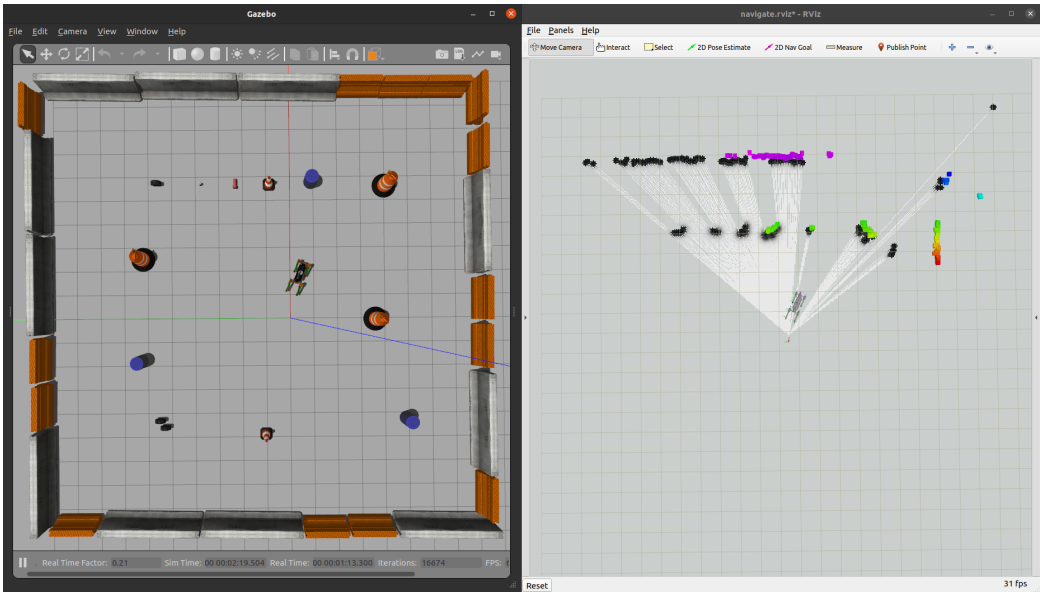
(b) PointCloud to Laserscan



(c) Depth Camera to Laserscan

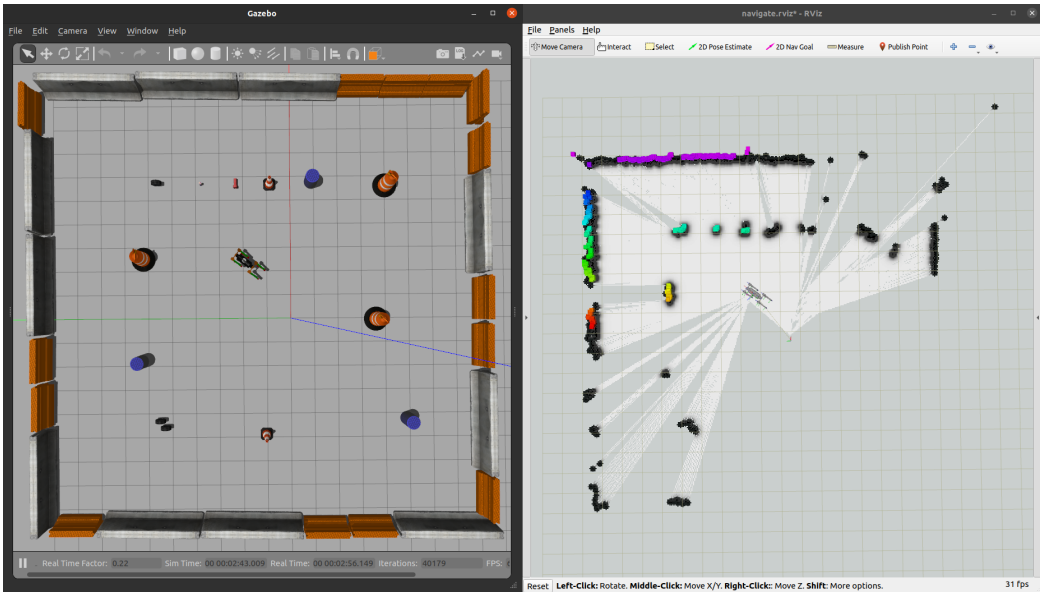
Figure 122: Publishing Rate Disparity to Laserscan (a), Publishing Rate PointCloud to Laserscan (b) and Publishing Rate Depth Camera to Laserscan (c)

In Figures 123-124 ARGOS is instructed to move closer to the obstacles to verify that the laserscan can still detect them.



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 123: Obstacle Detection : Big Obstacles



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 124: Obstacle Detection : Small Obstacles

In can be verified from Figures 118-124 that the laserscan can detect all obstacles in this gazebo world.

5.4 Obstacle Detection : Depth Camera to Laserscan

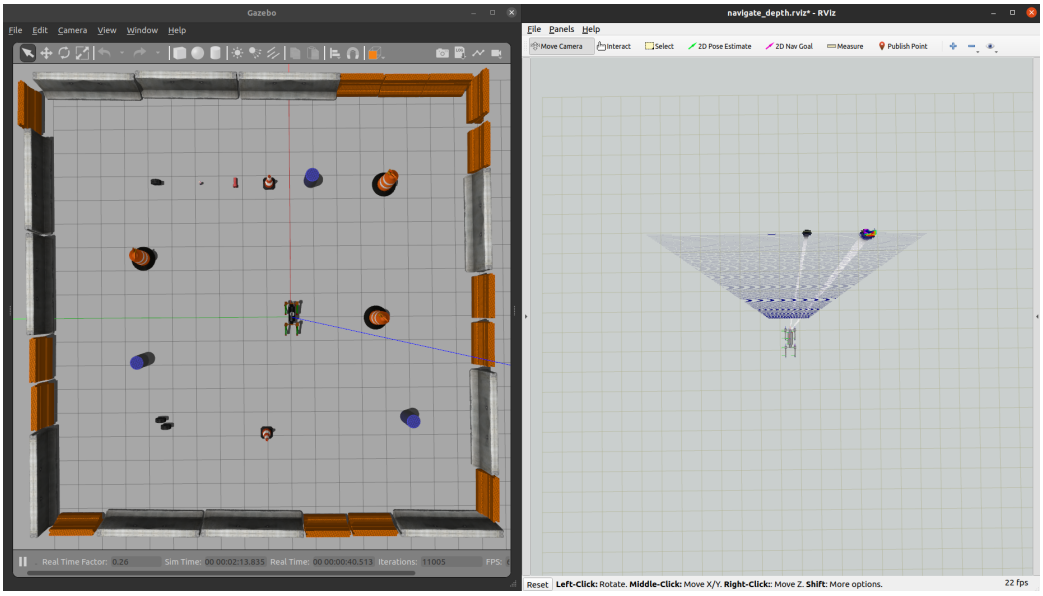
In this section the results for the obstacle detection using the depth camera to laserscan approach are presented. The commands in Figure 125 are executed in separate terminals to launch the gazebo world, spawn ARGOS with the depth camera to laserscan packages and launch the gmapping packages.

```
roslaunch argos_config dc_gazebo.launch

roslaunch argos_config dc_slam.launch
```

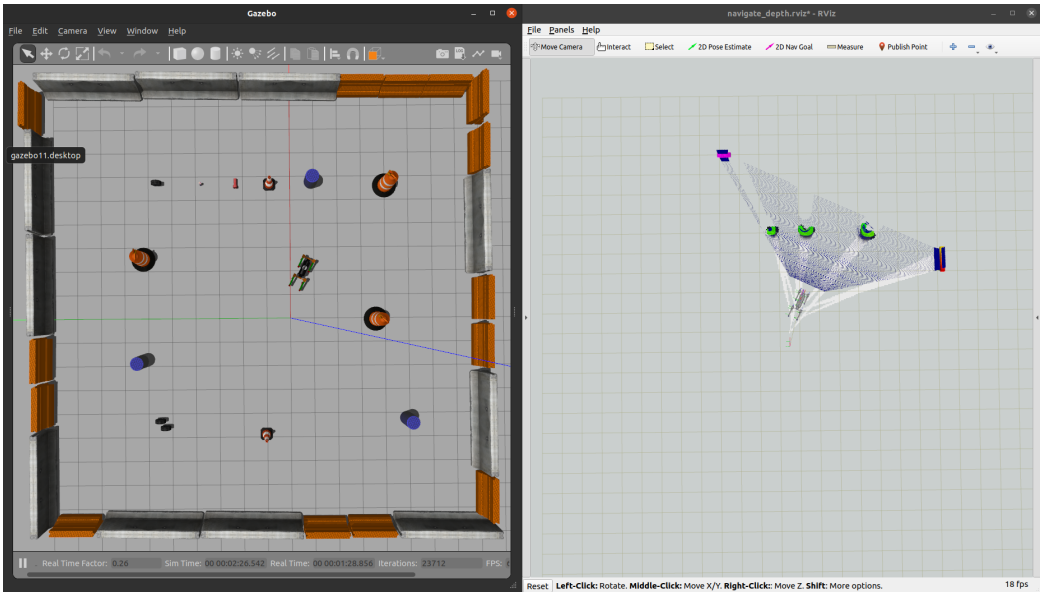
Figure 125: Commands to Launch Gazebo and Gmapping : Depth Camera To Laserscan

The results for the detected obstacles are presented in Figures 126-128.



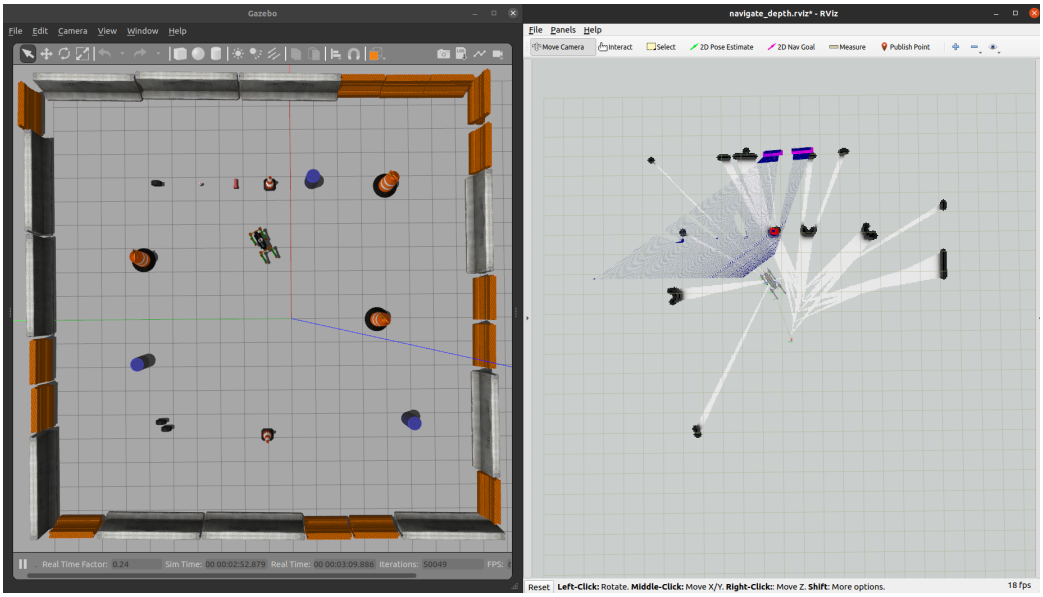
(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 126: Obstacle Detection : Starting Position



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 127: Obstacle Detection : Big Obstacles



(a) Gazebo World (b) Obstacles Detected by Laserscan

Figure 128: Obstacle Detection : Small Obstacles

Although the depth camera can detect obstacles with greater detail than the other approaches it fails to detect small obstacles as presented in Figure 128.

5.5 Ground Truth Movement

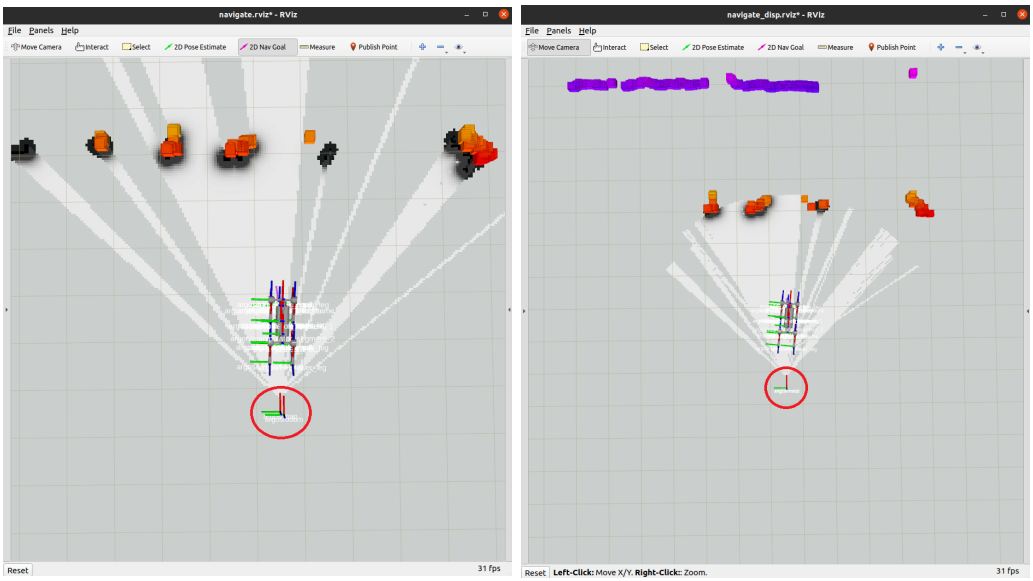
In this section the results for the movement of the robot using the real position (ground truth) of the robot instead of an estimate. The commands in Figure 129 are executed in separate terminals to launch the gazebo world, spawn ARGOS with the ground truth packages (using disparity to laserscan) and launch the gmapping packages.

```
roslaunch argos_config gt_gazebo.launch

roslaunch argos_config gt_slam.launch
```

Figure 129: Commands to Launch Gazebo and Gmapping : Ground Truth Position

Using this approach the robot does not have abrupt movements as is presented in Figure 132.



(a) Estimated Position (b) Real Position

Figure 132: ARGOS with State Estimator (a) ARGOS with Real Position (b)

The red circle indicates the movement of the odom frame (result of gmapping, described in Paragraph 3.1) on Figure 132(a), while in Figure 132(b) the odom frame remains static because gmapping knows the real position of ARGOS and as a result there are no odometry errors.

6 Conclusion

The Disparity Image to Laserscan approach had better results than the other approaches because the laserscan can detect all obstacles, it has a higher publishing frequency than the PointCloud to Laserscan approach and does not need a depth camera, in contrast to the Depth Camera to Laserscan approach. Using the real position of the robot increases the ARGOS's ability to avoid obstacles but loses robustness because in the real world it is impossible to know the true position of the robot.

List of Figures

References

[1] XACRO XML macro language, ROS Wiki.

[2] URDF, ROS Wiki.

[3] Champ,GitHub.

[4] Champ Setup Assistant, GitHub.

[5] Link, ROS Wiki.

[6] kdl_parser, ROS Wiki.

[7] Joint, ROS Wiki.

[8] Transmission in URDF, ROS Wiki.

[9] Gazebo Reference Tag, GazeboSim.

[10] Gazebo Plugins, GazeboSim.

[11] Stereo Camera, Wikipedia.

[12] Open Dynamics Engine, ODE.

[13] Gazebo ROS Control, ROS Wiki.

[14] Gmapping Packages, ROS Wiki.

[15] SLAM, Wikipedia.

[16] Stereo Image Proc package, GitHub.

[17] Disparity Image Proc package, GitHub.

[18] Depthimage To Laserscan package, GitHub.

[19] Stereo Image Proc Parameters, ROS Wiki.

[20] PointCloud To Laserscan, ROS Wiki.

[21] Message To Tf Package, ROS Wiki.