ARDEE: A GENERAL AGRICULTURAL ROBOTIC DEVELOPMENT AND EVALUATION ENVIRONMENT

BY

HUNTER YOUNG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Agricultural and Biological Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Abstract

When evaluating any algorithm, it is essential that the data used for evaluation be collected from the target operating environment, as well as conditions, in order to get an accurate representation of the algorithm's performance in that environment. This is especially important when extrinsic sensor measurements are used for developing and evaluating autonomous control and perception algorithms intended for agricultural applications. Unfortunately, there are many obstacles that can considerably hinder the development process, most notably the 7-8 months in which most crops are not in season.

The work presented in this thesis allows the year-round development and evaluation for a wide variety of autonomous control and perception algorithms for agricultural field robotic applications, using a set of developed simulation tools in combination with an open-source simulation platform, Gazebo. The custom set of tools was designed such that any number of user-specific agricultural environments can be simulated, and the sensor/robot configuration can be easily customized, being useful for a wide range of agricultural research interests.

The fundamental contributions of this work are the following: (1) a collection of sufficiently accurate simulated crop models for three different crop species (corn, sorghum, and tobacco), (2) user-friendly tools for generating a user-customizable agricultural field environment, (3) a collection of simulated, commonly-used, sensors that can be attached to any simulated robot platform, (4) a simulated model of an ultra-compact robot platform, and (5) a set of socket-based, or UDP, tools used for testing algorithm performance on-board target hardware and with the simulated sensors and field. Finally, a few core autonomous control and perception algorithms, which reflect the range of field robot research areas that could be used, are executed on-board an ultra-lightweight ground robot platform, and the performance is evaluated and compared in both a real-world and a simulated, agricultural environment.

Contents

Chapter 1

Introduction

According to United Nations projections, the entire world's human population of 7.3 billion, as of mid-2015, is expected to increase to be between 9.2 and 10.1 billion by the year 2050 [63]. The Food and Agriculture Organization of the United Nations suggests the global production of food needs to increase, from the 2009 production levels, by 70%, by the year 2050 [17], to properly sustain the reported growth in the world population. To fulfill this great of an increase in food production levels, there must be a significant amount of improvement in the current state-of-the-art agricultural techniques and practices, in addition to developing innovative solutions. A potential solution, which has gained interest in recent years, is the use of automation in agriculture to help solve this increased demand in global food production. In addition to food production, use of automation can benefit the agricultural community, because it can enable available resources to be used more effectively, it has the capacity for improving the efficiency of current agricultural practices, and it can help reduce production and labor costs.

The use of autonomous systems and more specifically small and lightweight autonomous field robots, in agriculture can be a great asset to farmers, biologists, and breeders because they can be useful for many agricultural applications. Some examples include, the weed scouting robot platform, called AgBotII, introduced by Hall et al.[26], the multi-purpose Thorvald ground robot platform useful for seeding, weeding, and harvesting applications[24], the wheat precision seeding robot introduced by Haibo et al.[25], and the Robotanist robot platform developed for automated phenotyping and is capable of navigating underneath the crop-canopy within individual rows of crops[43]. In addition to supporting versatile applications, autonomy in agriculture can enable site-specific applications (e.g. location-specific management, remote field sensing, etc.) to be carried out much more effectively, and at a finer resolution, with the help of small autonomous robots. For example, a team of small autonomous robots could be used to execute, much more effectively, various field maintenance and surveillance tasks by allowing high priority areas to be targeted first before going to areas of lower priority, which would not be possible with a commonly used large and expensive tractor for pulling the necessary farm implement across an entire field.

Another benefit to using automation in agriculture is it can be used to improve the efficiency of current agricultural practices. For example, current phenotyping practices, which are extremely manual labor intensive, and prone to human measurement errors, suffer from what is known as the "phenotyping bottleneck" ([19], [16], [51]). Automation could help solve this "phenotyping bottleneck" by enabling essential phenotypic measurements to be continuously collected reliably, and consistently since machines do not suffer from fatigue as humans do.

The previously mentioned advantages of automation in agriculture contribute to a third benefit, its ability to help reduce the costs associated with agricultural production and labor. Current agricultural practices rely predominantly on the usage of large, and heavy, farm equipment (e.g. tractor) for maintaining and harvesting of commodity crops (i.e. corn, wheat, etc.) in bulk, however, they tend to be expensive. For applications where large farm equipment is inadequate, such as the harvesting of specialty crops (i.e. strawberries, bell peppers, etc.) which requires fine manipulation of harvested fruits, humans labors are typically hired to perform the manual tasks required by the specific applications. Sustaining the food production necessary for a growing world population will become much more difficult, due to the continued increase in labor costs resulting from the limited availability of willing manual laborers, in addition to common expensive large farm equipment making it difficult to purchase the additional machines required to produce and maintain the necessary crops.

Current autonomy still needs to be developed and evaluated further to fully realize the effectiveness of automation in agriculture through safe and reliable methods. This critical development of autonomous methods is hindered by many obstacles including unfavorable environmental conditions, limited resource availability, and complex environmental conditions resulting in unreliable or inadequate algorithm performance.

The performance of field robotic methods being developed, such as perception and autonomous control algorithms, are environmentally-driven, or rely heavily on the interactions with the surrounding environment, and is critical for developing robust autonomy. As a result, these methods need to be tested outside in their target field environments, in order to properly evaluate their performance. This requirement can be presented with major obstacles that cannot be easily prevented or foreseen, such as unfavorable weather and field conditions, which can effectively halt any type of development or analysis. These types of scenarios are detrimental to the developmental process primarily because in most cases there is little that can be done to move the development of the perception, or control, method along. Another important factor is the crops within these field environments experience drastic morphological changes throughout the season in a relatively short time frame (approximately 4-5 months), making development and evaluation of safe and reliable field robotic methods time-dependent, and difficult especially for methods that are not sufficiently reliable or robust for deployment into the field. The relatively short time window of a crop's season can be a major obstacle, in which effective development and field evaluation can be halted for 7-8 months out of the year, annually, once a crop's season is over (i.e. crops have been harvested).

Unlike the environmental obstacles which cannot be easily prevented, resource-dependent obstacles can pose a problem for field robotic development, such as when the essential robotic hardware is unavailable due to limited funding or damaged components. For example, often times when developing innovative compact field robots, the robotic components essential to robotic operation (i.e actuators, sensors, etc.) can fail due to various reasons, including extended use in harsh field conditions, being inadequately designed for the target environments, or unforeseen field conditions (i.e puddles, rocks, etc.). Although these types of scenarios are preventable, they can still happen, which can lead to the loss of precious time, and money, being spent for repairing, or replacing, the necessary equipment, instead of being used towards the actual development of the sensing or control, algorithms.

Current field robotic research is trying to solve the issue of being able to reliably sense, interact with, and navigate about their surrounding environment. Achieving this can be difficult for autonomous field robotics due to the complexity (i.e.highly cluttered, unstructured, constantly changing, and difficult to model) of the agricultural environments in which they are expected to operate in.

In addition to these inherent complexities, developing safe and reliable autonomy in these types of environments, which can also contain high-value crops, expensive equipment, and human laborers, raises the concern of properly evaluating newly developed methods in real field environments while preventing the damage to any crops, resources, or humans, in processes. Particularly in the early stages of their development, this concern can become a hindrance for autonomous algorithms which are unable to properly handle or foresee, all the environmental complexities which can result in failure or damage. For example, when testing vision-based algorithms developed for the autonomous harvesting of specialty crops, variable lighting conditions can present a challenge that can result in damage, or complete destruction, of high-value crops, if these algorithms are unable to properly handle these irregularities.

A potential solution that can help mitigate the aforementioned obstacles is the use of simulation environments to supplement autonomous field robotic development, and they have been shown to be effective development and evaluation tools in other areas of autonomous and general robotic research ([72], [29]). The capabilities of simulation environments, when applied to robotic development and evaluation, include, but is not limited to, the following: (1) enables quick testing of developed methods in various different environments, (2) enables exhaustive testing, which leads to increased confidence in the robustness, of developed solutions, (3) reduction in the costs of development, and (4) reduces the time required for integration and troubleshooting of target hardware.

A desirable feature of simulation environments is they can allow developed robotic methods to be evaluated in various different environments. This capability is essential for developing safe and reliable autonomy for various robotic applications, particularly for those operating in environments which may be difficult, or ill-advised, to re-create. For example, the USARSim simulator allows autonomous algorithms developed for urban search and rescue applications to be tested and evaluated in different environments which would be expensive and ill-advised to re-create in the real world (i.e. disaster scenarios involving explosions, fires, chemical spills, and human victims)[5]. Although not related to robotic development, the simulation system, proposed by Uno and Kashiyama, shows an example of a simulation environment which enables the development and evaluation of methods associated with natural disaster-related evacuations[64]. Specifically, the presented simulation environment was used to estimate the number of potential victims, in correspondence with the time evacuation begins, for a flood disaster in an urban environment. In both previously mentioned works, the usefulness of simulation environments is shown to capable of allowing different algorithms and methods to be developed and evaluated, by simulating different environments that are impossible, or difficult, to re-create.

In addition to evaluation in different virtual environments, simulation environments have the capability of enabling exhaustive testing strategies. This gives simulation environments an advantage, which can lead to increased confidence in the robustness of proposed solutions, through exhaustive testing of developed methods in different environments and with a variety of associated conditions. For example, when developing autonomous robotic methods for missions in space, thorough testing of these methods to validate safe and reliable operation, is required before they can be implemented. Boge and Ma provide an example where the use of simulation environment that can enable autonomous methods developed for space satellite docking missions to be thoroughly tested, and is necessary before the real missions can be carried out[9]. Similar to autonomy in space, the usefulness of simulation environments to support extensive testing and development of safe reliable autonomy holds true for the automotive industry as well. For example, Gietelink et al. show an example where a simulation environment enables newly developed advanced driver assistance systems to be extensively tested, safely, in passenger vehicles[21]. This capability allows new methods to be extensively tested, and further developed to improve their robustness, in a safe environment. In both examples, extensive testing with the help of simulations supports validation of developed methods' reliable operation, which is important for ensuring the safety of any potential humans and costly equipment that may interact with these systems.

Another benefit to using simulation environments is they can be used to facilitate the design and evaluation of robotic solutions, some of which may not have been physically implemented yet. For example, a method, which simulates the dynamics associated with the contact and handling of different payloads and virtual objects, was proposed which can be used evaluate different designs of virtual manipulator systems for Space Station applications [41]. This work shows that simulations have the capability of reducing the costs associated with the development of potentially expensive robotic systems, by allowing potential design candidates to be tested and optimal robotic designs to be determined, without the need for essential hardware.

Another advantage of simulation environments is they can reduce the time associated with the implementation of developed robotic solutions on target hardware platforms. For example, a design approach was proposed for the development of chassis control systems in passenger vehicles, which can be used to reduce algorithm development times using efficient testing and development procedures[69]. This work shows that simulations have the ability to support efficient integration and evaluation of target hardware with developed algorithms, via Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) testing, and has the advantage of allowing rapid development for robotic solutions.

From the previously discussed advantages, simulation environments could be a viable option for supporting autonomous field robotic development and evaluation for agricultural applications. In order for any meaningful and effective development of safe and reliable autonomy in agriculture to be achieved, virtual field environments must be simulated with sufficient accuracy. Unfortunately, sufficiently accurate models of virtual field environments (i.e. fields, terrains, and crops) can be difficult to acquire, due to their complexity and the time required to create them. Furthermore, the limited availability of sufficiently accurate virtual field models and open-source field robotic development software can prevent effective development and healthy collaboration within the field robotic academic community.

1.1 Objectives and Approaches

The work presented in this thesis attempts to take the first step towards an open-source simulation environment capable of simulating sufficiently accurate agricultural environments intended to support multi-purpose development and research in the field robotic community. To address the aforementioned obstacles relating to safe and reliable field robotic development and evaluation, this thesis attempts to fulfill three primary objectives: (1) provide a collection of sufficiently realistic simulated field and crop models, (2) support user-specific customization of simulated fields and robotic platforms with relative ease, and (3) support the integration of user-specific target hardware of physical robotic platforms.

The first objective is to allow the possibility for simulating any elements essential for developing and evaluating field robotic methods, such as sensors, robotic platform dynamics, crop/field conditions, when the necessary physical components are unavailable, and they should be simulated with a sufficient degree of accuracy. The second objective is to allow the essential components for any user-specific field environments (i.e. crops species, placement of crops, and terrain), and robotic systems (i.e. robot platform, a specific configuration of attached sensors, custom robot dynamics), to be to generated and simulated with relative ease. The final objective is to enable any given field robotic method to be developed and evaluated on its target robotic hardware, using a simulation of its intended field environment (i.e. simulated sensor measurements, simulated robot dynamics, and simulated field-robot interactions).

To fulfill these objectives, this thesis presents a novel development environment that uses a 3D physics-based simulation environment combined with distributable communication mechanisms and user-friendly functionality, enabling time-efficient development and evaluation of new Autonomous Field Robot (AFR) perception and control methods for various developers and users.

1.2 Overview

This thesis is organized as follows: **Chapter 2** discusses some related work that has been done in the area of field robots and simulations, in addition to providing an overview on some of the underlying methods used in this work; **Chapter 3** provides an overview of the robotic platform used and the various components that make up the development environment/pipeline presented in this work; **Chapter 4** discusses how the various development pipeline components were created and developed; **Chapter 5** shows how the developed tools presented were used for facilitating in the development and evaluation of a Lidar-based algorithm used for late-season autonomous navigation within rows of crops in corn and sorghum fields, by investigating the performance of the Lidar-based method in both the real fields and the development environment simulated fields; finally, **Chapter 6** provides a summary of the work presented in this thesis, the experimental results obtained, and any future work necessary for further establishing the usefulness of the presented agricultural field robotic development environment for other areas of agricultural research.

Chapter 2

Background

This chapter aims to provide an overview of some the research that has been done in the areas of research related to the use of simulations in agricultural field robotics, in addition to any fundamental information related to the underlying components that are used in the work presented in this thesis.

Specifically, the following chapter discusses the following topics: **Section 2.1** discusses previous research done relevant to the use of simulation platforms in agricultural field robotics, **Section 2.2** provides an overview of the Robot Operating System (ROS) framework which is used for bridging the developed simulated field environment with the physical robot platform/hardware, and **Section 2.3** provides an introduction to the Gazebo simulation environment used in this work as the basis for a field robot simulation and testing environment for agricultural and biological applications.

2.1 Related Work

In this section the following will be discussed: first a handful of well-known general-purpose robotic simulation platforms will be briefly introduced, and lastly an overview of some of the relevant research that has previously been done in area relating to the use of simulation platforms for assisting in the development and evaluation of field robotic methods for agricultural applications.

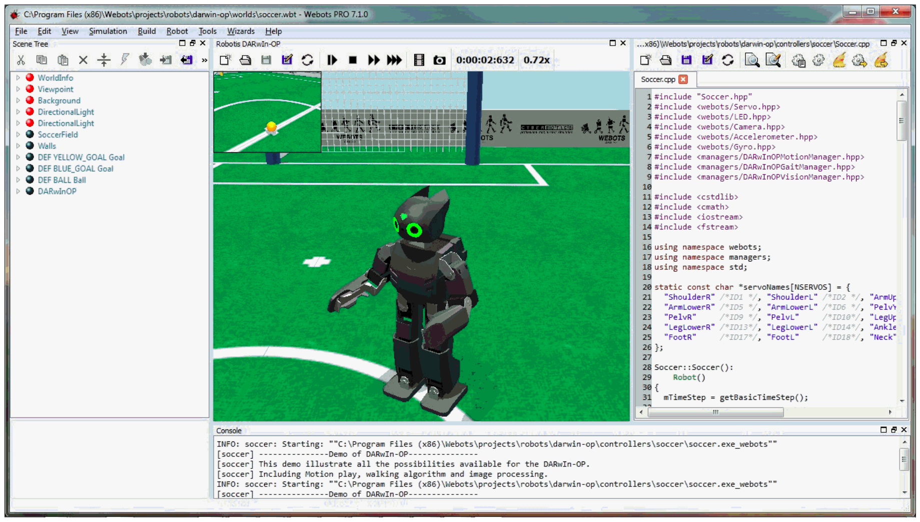
2.1.1 Simulation Platforms

When testing, or evaluating, any kind of algorithm that requires any interaction with, or control over, any kind of hardware, it is preferred to use the target hardware in order to ensure that the performance of the algorithm being developed most accurately reflects its true behavior. Unfortunately, there are many factors that can make acquiring the necessary physical components difficult, and in some cases impossible (i.e limited funding, faulty components, and interfacing difficulties, etc.), which can hinder the developmental progress.

In many cases, a viable approach to evaluating an algorithm's performance in the absence of the targeted physical system is to simulate the dynamics of that system using a mathematical model, e.g simulating the dynamics of a four-wheeled robot using a Dubin's car model [46] in Matlab. In general, the necessary complexity and accuracy of the simulated model depends on the specific algorithms used and the priorities of the developers. For some complex systems, acquiring a sufficiently accurate model can be impractical or impossible. In such cases, a 3D physics-based simulation platform may be a more practical approach.

A 3D physics-based simulation platform, or simulation platform for brevity, is a type of software package capable of simulating the physical interactions with, and visual features of any, 3D objects in a virtual environment. Additionally, simulation platforms can not only be used to alleviate some of the hassles of simulating, and modeling, all of the necessary aspects relating to any given dynamic, or static, system, but they can also provide other features that can make developing and evaluating algorithms easier, such as simulating various sensors, simulating multiple robot systems, simulating the interactions of all the objects in any given simulated environment. There are quite a few simulation platforms available that, in addition to providing many useful features, are able to simulate, relatively accurately, the dynamics and kinematics of many different types of, simple and/or complex, physical systems through the use of a physics engine, such as the widely used open-source Open Dynamics Engine (ODE) [58].

For example, Webots [42] is a, fairly feature-rich, 3D simulation environment capable of accurately simulating multi-robot physics using an ODE-based physics engine. Webots is capable of simulating a wide range of sensors and actuators, and can interface with the widely-used Robot Operating System (ROS) middleware [50], allowing quick testing of developed algorithms. Another benefit to Webots is that it has multi-OS support (Windows, macOS, Linux, and even mobile devices), as well as supporting multiple programming languages and having a user-friendly Graphical-User-Interface (GUI), which makes it easy for multiple research teams to collaborate on a project. An example Webots simulation can be seen in **Figure 2.1**. Being able to accurately visualize and render 3D models is a must have for any simulation platform where vision-based, and/or Lidar-based, algorithms are being developed. This is possible using the open-source Object-Oriented Graphics Rendering Engine (OGRE) [36], which Webots uses for rendering its simulated objects.


C:\Program Files (x86)\Webots\projects\robots\darwin-op\worlds\soccer.wbt - Webots PRO 7.1.0
File Edit View Simulation Build Robot Tools Wizards Help
Scene Tree
WorldInfo
Viewpoint
Background
DirectionalLight
DirectionalLight
SoccerField
Walls
DEF YELLOW_GOAL Goal
DEF BLUE_GOAL Goal
DEF BALL Ball
DARwInOP
Robotis DARwIn-OP
0:00:02:632 0.72x
...x86)\Webots\projects\robots\darwin-op\controllers\soccer\Soccer.cpp
Soccer.cpp
Console

```
#include "Soccer.hpp"
#include <webots/Servo.hpp>
#include <webots/LED.hpp>
#include <webots/Camera.hpp>
#include <webots/Accelerometer.hpp>
#include <webots/Gyro.hpp>
#include <managers/DARwInOPMotionManager.hpp>
#include <managers/DARwInOPGaitManager.hpp>
#include <managers/DARwInOPVisionManager.hpp>

#include <cstdlib>
#include <cmath>
#include <iostream>
#include <fstream>

using namespace webots;
using namespace managers;
using namespace std;

static const char *servoNames[NSERVOS] = {
  "ShoulderR" /*ID1 */, "ShoulderL" /*ID2 */, "ArmUp
  "ArmLowerR" /*ID5 */, "ArmLowerL" /*ID6 */, "PelvY
  "PelvR"     /*ID9 */, "PelvL"     /*ID10*/, "LegUp
  "LegLowerR" /*ID13*/, "LegLowerL" /*ID14*/, "Ankle
  "FootR"     /*ID17*/, "FootL"     /*ID18*/, "Neck"
};

Soccer::Soccer():
    Robot()
{
  mTimeStep = getBasicTimeStep();
```

```
INFO: soccer: Starting: ""C:\Program Files (x86)\Webots\projects\robots\darwin-op\controllers\soccer\soccer.exe_webots""
[soccer] ---------------Demo of DARwIn-OP---------------
[soccer] This demo illustrate all the possibilities available for the DARwIn-OP.
[soccer] Including Motion play, walking algorithm and image processing.
INFO: soccer: Starting: ""C:\Program Files (x86)\Webots\projects\robots\darwin-op\controllers\soccer\soccer.exe_webots""
[soccer] ---------------Demo of DARwIn-OP---------------
```

Figure 2.1: Figure showing example Webots environment.[1]

Although there are many things that Webots does well, one drawback is that it is not free, which makes it less desirable, and reducing the overall quality of its community support, as opposed to its open-source counterparts. Additionally, even though Webots uses a state-of-the-art rendering engine, OGRE, the supported 3D model mesh file formats that are used to describe and define the 3D rendered objects is limited.

The Microsoft Robotics Developer Studio (MRDS) [34], is another example of an available simulation platform that has many of the same capabilities as Webots, such as a range of supported sensors and interface-ability with real hardware, however, unlike Webots, MRDS is free to use. MRDS, like Webots, has a relatively intuitive User-Interface (UI) shown in **Figure 2.2**. Additionally,

[1]Image courtesy of `http://support.robotis.com/en/product/darwin-op/simulation/webots/windows_os_installation/running.htm`

MRDS uses a physics engine developed by Nvidia, called PhysX, for sensor and robot-world interaction simulations, as opposed to Webots ODE-based engine.

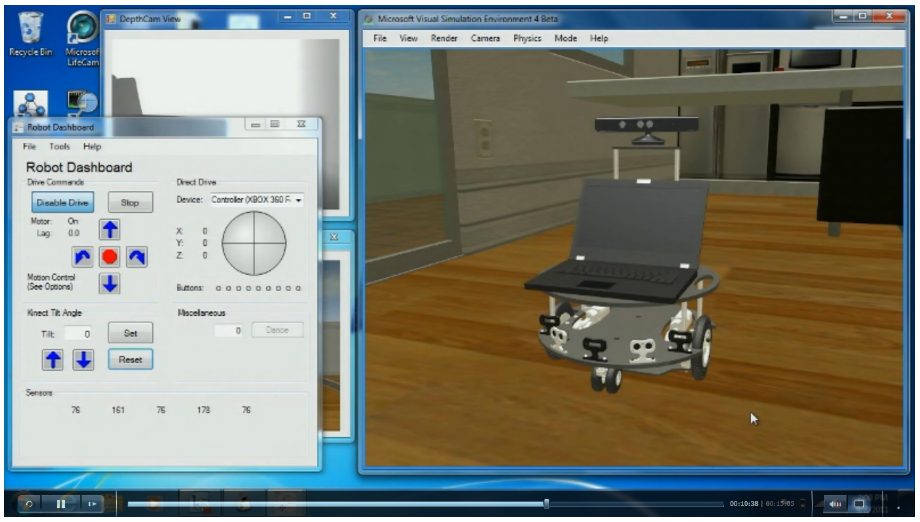
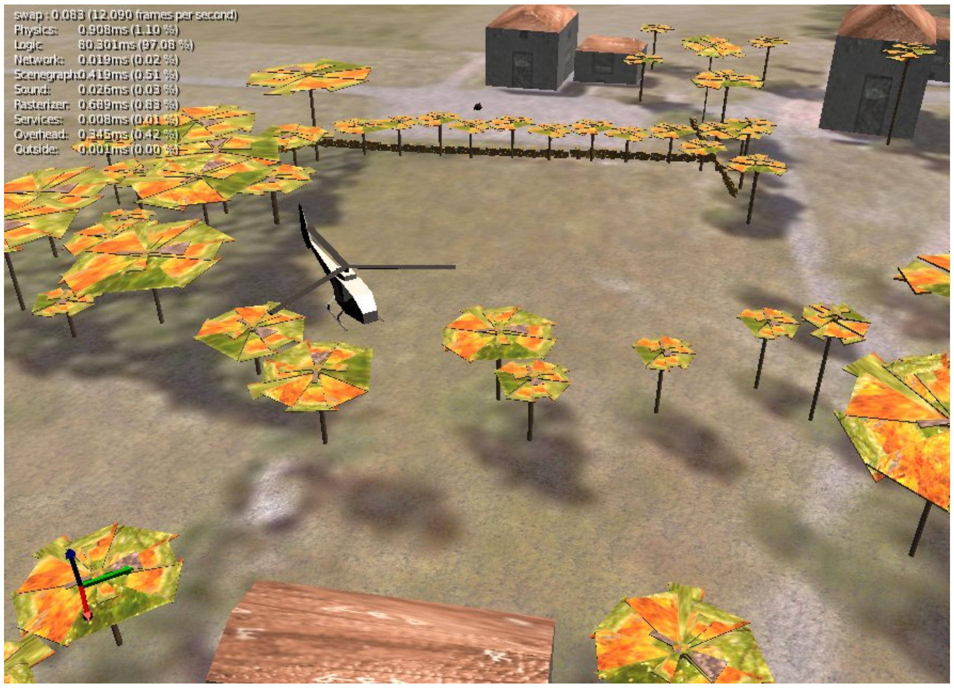

DepthCam View
Microsoft Visual Simulation Environment 4 Beta
File View Render Camera Physics Mode Help
Robot Dashboard
File Tools Help
Drive Commands
Disable Drive
Stop
Direct Drive
Device: Controller (XBOX 360 F
Motion Control (See Options)
Kinect Tilt Angle
Set
Reset
Buttons
Miscellaneous
Dance
Sensors

Figure 2.2: Figure showing example MRDS environment.[2]

Although MRDS is free to use, it is not entirely open-source, and as a result, much of its internal components are not openly available to the public. In addition to MRDS not being fully open-source, the MRDS platform, which is only available for Windows OS, stopped being updated and maintained, resulting in a, relatively, underdeveloped simulation platform, compared to more state-of-the-art, actively maintained, simulation platforms, as well as limited community support.

Unlike MRDS, The Modular OpenRobots Simulation Engine (MORSE) platform [13], is an example of a fully open-source 3D multi-robot simulation platform. MORSE is targeted for experienced computer scientists and the academic community, whereas MRDS targets hobbyists. Similar to MRDS, MORSE has limited OS support outside of the Linux OS. In addition to supporting ROS, MORSE is able to support other middlewares, such as YARP, ProcoLibs, and more. MORSE also come with a human avatar that can be used during simulations to aid in human-machine interaction (HMI) research, that can interact with items in the simulated environment. In addition to an interact-able human avatar, MORSE is capable of simulating multiple robots with sensors in the same environment using a state-of-the-art physics engine, Bullet[10].

[2]Image courtesy of `https://medium.com/@titan.haryawan90/microsoft-robotics-developer-studio-simulator-untuk-membuat-robot-3da8e033b2bd`



```
swap : 0.083 (12.090 frames per second)
Physics:       0.908ms (1.10 %)
Logic          80.301ms (97.08 %)
Network:       0.019ms (0.02 %)
Scenegraph:0.419ms (0.51 %)
Sound:         0.026ms (0.03 %)
Rasterizer:    0.689ms (0.83 %)
Services:      0.008ms (0.01 %)
Overhead:      0.345ms (0.42 %)
Outside:       0.001ms (0.00 %)
```

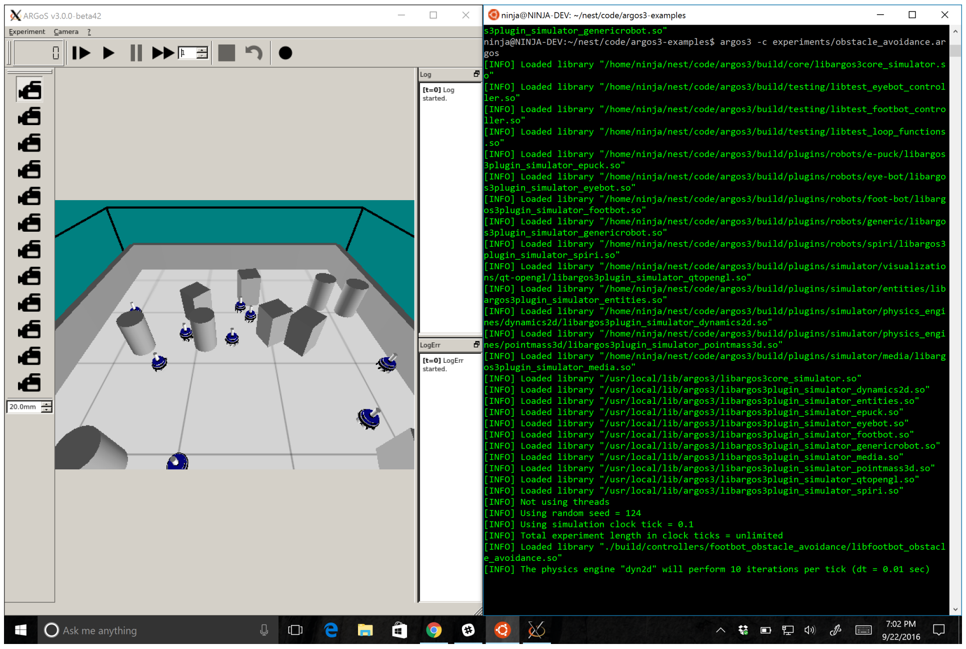
Figure 2.3: Figure showing example MORSE environment.[3]

Since MORSE is targeted for experienced computer scientist, it relies primarily on the use of a command-line interface for user-control and interaction, and has almost no GUI capabilities, **Figure 2.3** shows an example simulation using MORSE with the command-line interface.

Unlike MORSE and MRDS, where a single OS is primarily supported, The Autonomous Robots Go Swarming (ARGoS) platform [47], is an example of an open-source 3D multi-robot simulation platform that is available on Mac OS and Linux. Similar to MRDS and MORSE, ARGoS comes with default physics engine for simulating objects and interactions, although it is very limited in its capabilities, however, it is capable of supporting other physics-engines, such as ODE, Bullet, and more. Additionally, ARGoS allows for more than one physics-engine to be used in parallel during simulation runtime for user-specified roles. Similar to the previously mentioned simulation platforms, ARGoS is capable of simulating multiple robots and objects at a time, however, it is able to do so without any major drawbacks to computation and runtime. **Figure 2.4** shows

[3]Image courtesy of `https://www.openrobots.org/morse/doc/latest/what_is_morse.html`.

an example simulation using ARGoS. ARGoS does not have as many features as the other simulation platforms, such as being able to import custom 3D meshes and it has a limited database of pre-made robot models.



```
s3plugin_simulator_genericrobot.so"
ninja@NINJA-DEV:~/nest/code/argos3-examples$ argos3 -c experiments/obstacle_avoidance.ar
gos
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/core/libargos3core_simulator.s
o"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/testing/libtest_eyebot_control
ler.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/testing/libtest_footbot_contro
ller.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/testing/libtest_loop_functions
.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/robots/e-puck/libargos
3plugin_simulator_epuck.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/robots/eye-bot/libargo
s3plugin_simulator_eyebot.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/robots/foot-bot/libarg
os3plugin_simulator_footbot.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/robots/generic/libargo
s3plugin_simulator_genericrobot.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/robots/spiri/libargos3
plugin_simulator_spiri.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/simulator/visualizatio
ns/qt-opengl/libargos3plugin_simulator_qtopengl.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/simulator/entities/lib
argos3plugin_simulator_entities.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/simulator/physics_engi
nes/dynamics2d/libargos3plugin_simulator_dynamics2d.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/simulator/physics_engi
nes/pointmass3d/libargos3plugin_simulator_pointmass3d.so"
[INFO] Loaded library "/home/ninja/nest/code/argos3/build/plugins/simulator/media/libarg
os3plugin_simulator_media.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3core_simulator.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_dynamics2d.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_entities.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_epuck.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_eyebot.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_footbot.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_genericrobot.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_media.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_pointmass3d.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_qtopengl.so"
[INFO] Loaded library "/usr/local/lib/argos3/libargos3plugin_simulator_spiri.so"
[INFO] Not using threads
[INFO] Using random seed = 124
[INFO] Using simulation clock tick = 0.1
[INFO] Total experiment length in clock ticks = unlimited
[INFO] Loaded library "./build/controllers/footbot_obstacle_avoidance/libfootbot_obstacl
e_avoidance.so"
[INFO] The physics engine "dyn2d" will perform 10 iterations per tick (dt = 0.01 sec)
```

Figure 2.4: Figure showing example ARGoS environment.[4]

The Virtual Robot Experimentation Platform (V-Rep) [11], is an example of an open-source, multi-robot 3D simulation platform that is a feature-rich, all-in-one simulation platform, similar to Webots. V-Rep is not free to everyone, only members of the academic community, such as students and universities, can acquire V-Rep for free. V-Rep is capable of accurately simulating the physics of multiple robots and world objects, even custom particles like air and water, with the support of a select number of state-of-the-art physics engines. V-Rep can support the ODE, Bullet, Vortex [57], and Newton [33] physics engines, however, unlike ARGoS with multi-engine support, V-Rep can only use one physics engine during simulation runtime. V-Rep is very user-friendly, allowing users to edit/modify both code and objects in the simulated environment within its GUI, as seen in **Figure 2.5**. Another V-Rep feature that can be useful is the ability to manipulate and modify, for example reducing rendering complexity,

[4]Image courtesy of `http://www.terriblesysadmin.com/?p=76`.

within the GUI, without the need for external 3D graphics editing software, such as Blender [8]. V-Rep also allows interfacing with ROS through the use of the “RosInterface” function calls, available in V-Rep’s API, in the embedded scripts used throughout V-Rep.

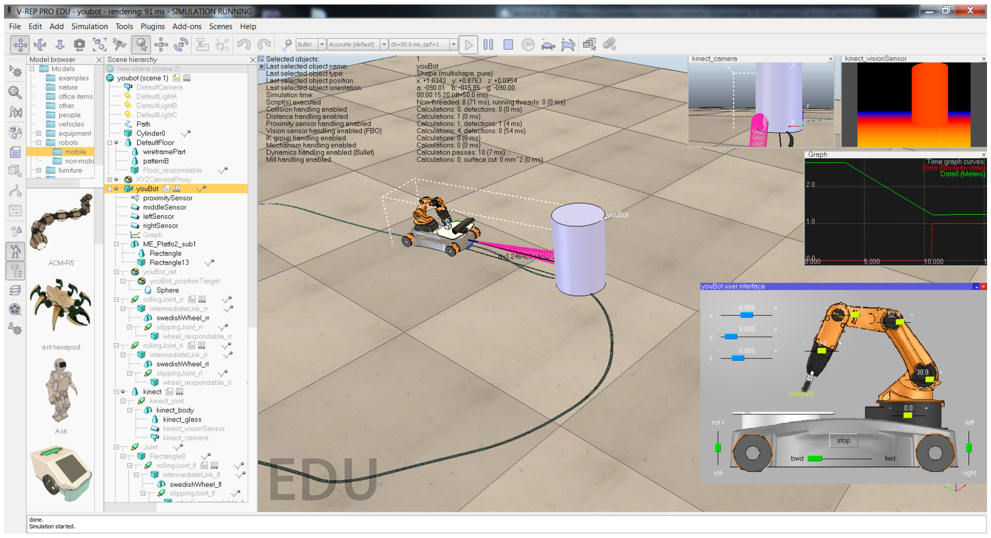

V-REP PRO EDU - youbot - rendering: 91 ms - SIMULATION RUNNING
File Edit Add Simulation Tools Plugins Add-ons Scenes Help
Model browser
Scene hierarchy
kinect_camera
kinect_visionSensor
Graph
youBot user interface
EDU

Figure 2.5: Figure showing example V-Rep environment.[5]

Although V-Rep has many desirable features that make simulating robots in custom environments user-friendly and easy, such as the use of custom user-interfaces and data recording and visualization, V-Rep can be very computationally expensive, and even making the simulation of multiple robots at one time not feasible in situations that ARGoS is capable of handling. Even though V-Rep gives users the ability to customize objects in the simulated environment, the modified simulation environment is saved as a V-Rep special format, which makes dynamically changing the simulated world programmatically difficult, if not impossible.

Although it does not have as many user-friendly features as V-Rep, Gazebo [37] is another 3D multi-robot simulation platform that has many of the same capabilities as V-Rep. Similar to V-Rep, Gazebo supports four state-of-the-art physics engines, ODE, Bullet, Simbody [12], and DART [39]. Gazebo gives users the ability to interact with the simulated robots and world objects, such

[5]Image courtesy of `https://www.aepwebmasters.it/my-product/installazione-e-configurazione-del-simulatore-di-robotica-v-rep-esempio-di-\realizzazione-di-una-cella-robotizzata-con-kuka-youbot/`.

as playing/pausing the simulation and adding/editing simulated objects in real-time, via "drag-n-drop" type methods, editing simulated models and worlds, and more, through its GUI, shown in **Figure 2.6**. Similar to V-Rep's embedded scripts, Gazebo's "plugins" give users the ability to programmatically control and modify most, if not all, aspects of the objects in the simulation environment, as well as generate and utilize custom information or actions. Unlike V-Rep, Gazebo's API is almost entirely ROS-integrated, which allows users to use all the functionality of ROS using ROS standard functions. This makes it easier for users to integrate with and provides the same capabilities as, ROS. Although it can be used on multiple operating systems, Gazebo is primarily Linux supported.

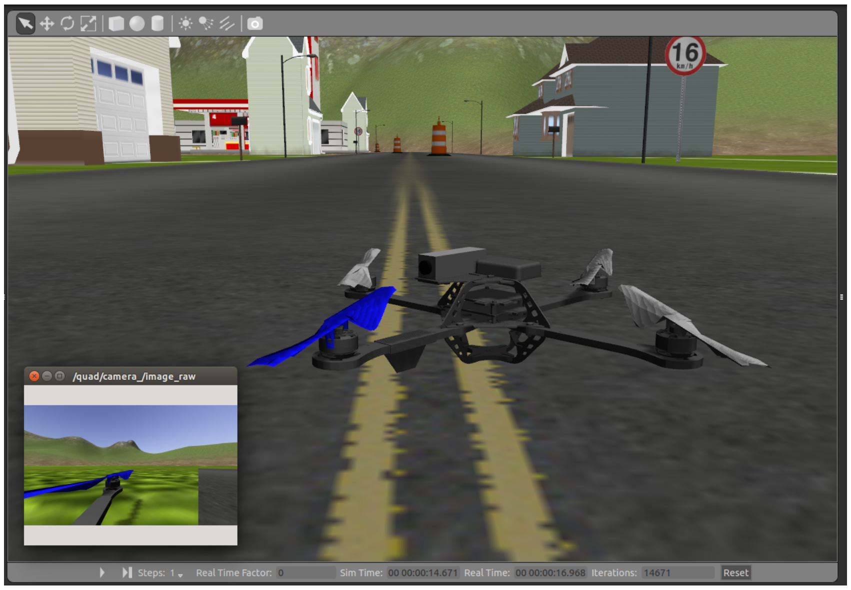


Figure 2.6: Figure showing example Gazebo environment.[6]

2.1.2 Literature Review

The development of Autonomous Field Robots (AFR) is an important area of research that can have a major positive impact for many different applications, such as search and rescue, military, and forestry surveying.

For example, Ball et al.[6] proposed the application of a multi-robot system used in an attempt to help solve the agricultural problem of resistant weed main-

[6]Image courtesy of `http://bestzzz.info/cliparts/quadrotor-control-simulator/`.

tenance for “zero-tillage” practices and further highlights the utility of AFR to benefit agricultural applications. The work proposed the use of a novel robotic platform, in combination with a multi-robot path planner, which was used to investigate the spray coverage of agricultural fields. The proposed platform is based on an agricultural robot platform, called the John Deere TE Gator, and was modified to enable autonomous field operation by using the ROS framework, in combination with GPS-based local navigation, vision-based obstacle detection, and a multi-robot path planner used for determining global navigation achieving optimal field coverage. Ball et al. perform real field evaluation of the modified robot platform alongside 12 simulated robots used for covering a 55 hectacres field, containing three obstacles typically found on farms (i.e. a single human, static electricity pole, and a small utility vehicle). The real robot platform was used to investigate the real-time capabilities of the proposed navigation system in real field environments, while the simulated robots were used to investigate the performance of the multi-robot coverage planner. The proposed real robot platform was able to cover 6 hectacres in 1.8 hours and was able to successfully navigate 97.4% of its assigned 6 hectacres while properly avoiding obstacles. The work proposed by Ball et al. showed development and evaluation of a real autonomous agricultural robotic platform capable of safe and reliable navigation within real field environments using a GPS path given by their proposed multi-robot path planner, however its application is restricted to multi-robotic systems used for broad coverage for field maintenance and does not address its capabilities for improvement of agriculture in other areas of research.

Mueller-Sim et al.[43] propose a novel ground-based agricultural robot, called the Robotanist, which was used to autonomously collect plant phenotype measurements of Sorghum bicolor, which enables geneticists to improve important crop genetics resulting in improvements to crop yields. The proposed work was developed in order to support a variety of different possible phenotyping tasks. In order to achieve this, Mueller-Sim et al. proposed the design of the Robotanist platform showing autonomous navigation capabilities for row crops (i.e. sorghum and corn), as well as under crop-canopy situations. Additionally, the Robotanist is equipped with a novel manipulator and is designed to support a modular array of contact and non-contact sensors, all of which enable the gathering of vital data for various different phenotyping tasks. Mueller-Sim et al. proposed using the robot middleware ROS to develop the Robotanist software framework, which enables the use of both custom, and open-source, developed software packages for communicating with the sensor hardware, robot control, transmission of important system information, and data processing and visualization. Although the proposed Robotanist platform was shown to be capable of autonomous navigation, underneath the crop-canopy, within crop rows of sorghum using RTK GPS, its performance can be significantly reduced as the crop-canopy grows to be too tall and does not address safe and reliable GPS-denied autonomous navigation underneath the crop-canopy. Additionally, this work only shows the usefulness of the proposed robotic platform for phenotyping applications and does not address problems in other areas of agricultural and biological research.

For example, Bac et al.[4] review and analyze state-of-the-art, and possible future, perspectives relevant to the advancement of robotic technology for high-value crop harvesting, and highlights the growing importance for the use of autonomy in agricultural applications. Additionally, the work reveals that there is still a continued need for research and testing in order to realize an improvement in the performance of state-of-the-art robotic harvesting technologies. In the work, the following three sources of variation in the crop environment, which pose a challenge to the development of autonomous crop harvesting systems, identified: (1) "Objects in a crop" (i.e. the fruit, stem, leaves, etc.), was defined as a challenge due to crops often suffering from occlusions from leaves and branches, as well as undesirable positions, shapes, sizes, and colors, (2) "environment" is defined as the challenge of operation in crop environments where the visibility and accessibility of the target fruit can be difficult, and can result from various different factors (i.e. caused by varying lights conditions, different cultivation practices employed, etc.), and finally (3) "variation among crops" where the consideration of all high-value crops was identified as a challenge resulting from the difficulty of properly harvesting from many different crops, some of which require more complex robot-crop harvesting interaction and can be fundamentally different across different crops. In addition to identifying environmental challenges, Bac et al. performed a quantitative literature review on existing harvesting robots that have been developed and documented, in order to quantitatively assess the current state-of-the-art performance in harvesting robotic technology, and the quantitative analysis on the performance of 50 distinct projects, performed within a period of 30 years, is reported and compared. From the resulting analysis, Bac et al.. identified five different bottlenecks that were determined to have an influence in limiting the performance of the reviewed robotic harvesting methods. As a result, the following three challenges to future R&D areas that address the identified bottlenecks, are proposed: (1) "Simplifying the task" was identified as the R&D challenge related to the modification of the crop environment, and requires further investigation in order to help improve the performance of robotic harvesting technology, (2) "enhancing the robot" was identified as the R&D challenge related to the improvement of the physical robot harvesting platform design, and requires further investigation in order to realize an improvement in robotic harvesting technology performance by enabling harvesting robots to better deal with the complexities of their surrounding environments, and finally (3) "defining requirements and measuring performance" was identified as the R&D challenge related to identifying the goals and metrics used to benchmark and evaluate the performance of future projects, and is essential for fostering, and enabling the implementation in practice of, future research.

The work, of Bac et al., revealed three key challenges, caused by variations from sources within the harvesting environment, and even though they were proposed as obstacles that limited the performance of robotic harvesting technology, it could be argued that these challenges are fundamental obstacles to autonomy in most, if not all, agricultural applications. In addition to the environmental factors identified by Bac et al., there are many other factors (i.e. adverse weather, unavailable hardware, etc.) that can negatively influence the advancement of autonomy in agriculture, and most of them cannot be controlled. As result, there is a need for the use of simulations to help advance the development of state-of-the-art agricultural methods.

Edan et al.[14] attempt to solve the problem of developing an intelligent control system used for robotic harvesting of melons, and must be capable of operating reliably in highly unstructured and dynamic agricultural environments. In this work, Edan et al. propose the use of an intelligent control system which uses globally accessible databases, called Blackboards, where different processes are able to share data and results in order to work collaboratively to achieve a specified goal. Edan et al. proposed using two Blackboards, Planning and Control, to integrate with each other using a transfer controller and were responsible for interacting with the peripherals (i.e. sensors, actuators, etc.) and for planning appropriate trajectories and harvesting tasks. Additionally, the application of the "traveling salesman" algorithm was proposed in order to increase the system's autonomous harvesting throughput by determining the melon picking sequence, which resulted in reducing the melon harvesting time by 49%. In addition to the "traveling salesman" algorithm, an additional algorithm was used to vary the vehicle's forward speed, resulting in all of the ripe melons being successfully harvested. Although an intelligent control system capable of efficient autonomous melon harvesting was introduced, some of the vision processing methods used required delays, which had to be simulated for system operation, and were not developed for optimal real-time applications. In addition to non-optimal real-time vision, the developed melon harvesting methods investigated were implemented on a robot that was connected to a tractor. As a result, the methods proposed are not useful for applications where the robotic harvesting systems used are required to travel through narrow individual rows of crops. Additionally, field evaluation of the proposed methods required the presence of a human tractor operator to manually control the tractor speeds. Not only did Edan et al. show the use of an efficient autonomous melon harvesting system, but it was also shown that the use of 3D simulations played an important role in the development of the resulting intelligent control system. The 3D simulations were used to plan, model, and evaluate the robot and gripper throughout the development of the control system presented. Additionally, the simulations were useful for comparing the various motion control algorithms and choosing the most efficient approach, which showed a resulting improvement in performance, however, the simulation software used is outdated and most likely not useful for modern applications.

Zou et al. provide an example where the use of a simulation has facilitated in the development of an intelligent agricultural field robotic control method used for harvesting litchi trees[71]. In the work by Zou et al., a simulated virtual environment was utilized for visually realizing the kinematic behavior of the arm manipulator used for harvesting, and allowed behavioral knowledge to be extracted, in relation to how the arm manipulator interacts with the litchi fruits being harvested, which was essential in the development of the control method proposed. Zou et al.. employs the use of EON Studio 5.0 (a virtual reality environment development tool) and Microsoft Visual C++, in order to simulate the virtual environment (containing a single arm manipulator and a single litchi tree with some fruit) and controlling the arm manipulator (via a simple "point-n-click" method). The drawbacks are: (1) environment simulated does not entirely represent the real field environment and field conditions accurately, (2) the simulation environment used is not open-source, limited applications, and does not guarantee physics-based simulation for accurate analysis of the representation of the world.

Rodriguez and Nardi[53] provide an example where the use of a simulation environment was used to develop and evaluate a method for visually detecting the presence of downy mildew parasites in sunflower crop fields by estimating the plant height or leaf color. They proposed the development of a testing simulation environment used to generate images, from a simulated monocular camera attached to a simulated UAV, which was used for developing and evaluating the performance of a 3D reconstruction algorithm for estimating the average plant height of simulated crops, using a simulation environment based on the Unreal Engine 4[55]. The performance of the proposed height estimation algorithm was evaluated using the known ground truth heights of the 3D crops for comparing the accuracy of the height estimation algorithm at different UAV flying altitudes. Although it shows the usefulness of a simulation environment for developing a crop estimation algorithm for a UAV, the proposed approach did not account for the behavior of the real system, where the UAV moved perfectly along the given trajectory (i.e. no kinematics or dynamics considered) and the simulated camera was a perfect camera model (i.e. no realistic camera behaviors or noise considered). In addition, the proposed simulation environment is only applicable to a specific application in agriculture (e.g UAV monitoring of sunflower fields) and does not easily generalize to work in other areas of agricultural research topics.

Biber et al.[7] claim that the use of an autonomous, multi-purpose, agricultural robotic platform, called BoniRob, can be a competitive alternative to the standard practice of using single, expensive, and large farm equipment by showing the BoniRob's capabilities in autonomous agricultural applications (such as autonomous phenotyping, field maintenance tasks, etc.). In this work, a novel navigation system is presented which allows the BoniRob to autonomously navigate safely over row-based crop cultures and is shown to reliably navigate autonomously continuously for three days using a 3D lidar, an IMU, and an optional RTK GPS. The presented navigation system is composed of the following four layers: (1) the "Drivers" layer, which is the lowest level layer and is used for interfacing with the physical hardware components (i.e. actuators, sensors, etc.), (2) the "Reactive" layer is used to process, and reactively use, sensor data (such as row/ground detection, row/turning control, localization and mapping, etc.) and has no knowledge of any high-level states, (3) the "Semantic" layer extends the localization and mapping functionalities of the "reactive" layer allowing for high-level localization and mapping using agricultural specific semantic labels (i.e. crop row, open field, side of field, row gap, and beginning/end of row), and finally (4) the "Planning" layer is used to coordinate the data flow between the various elements in the lower layers, in addition to overall system management, and this layer contains the system configuration and state-machine of the given specific application. The proposed work, of Biber et al., shows an effective method for robust autonomous navigation of a multi-purpose agricultural robot, BoniRob, it is strictly limited to applications where the robotic platform can traverse over the crops (i.e. early stages of, and/or small, crops), and it does not address the difficult problem of navigating reliably underneath the crop-canopy. Although not discussed in depth, this work does provide an example where the use of a simulation environment, Gazebo[37], was crucial during the early stages of development of a novel autonomous field robotic navigation system when the physical robotic platform was not available.

The availability of skilled field workers, who are willing to perform tedious and manually intensive field labor, is very limited, resulting in an increased demand for autonomous harvesting mechanisms. The development of safe and reliable autonomous harvesting methods is challenging and can result in doing more harm than good especially in applications involved with delicate crops. For example, a robotic manipulator can easily harm or destroy vital crops if not controlled properly, thus negatively influencing the effective crop yields. Shamshiri et al. showed an example where a simulation environment, V-Rep, was used to develop and improve vision-based perception and control algorithms, such as visual servoing (which is a method of controlling a robot based on the input from a vision sensor), object recognition, etc., used for autonomous harvesting of sweet peppers, and was useful by enabling testing and debugging with zero risks of damage to vital crops and hardware[27]. In the proposed work, V-Rep, in combination with ROS, was used to evaluate four different vision-based control and sensing strategies implemented using various commonly used manipulator platforms. Additionally, experiments were performed showing the simulation environment, as well as both simulated and real-world visual data, being used for the development and evaluation of the following: (1) a fruit detection and tracking algorithm which utilizes different camera views, (2) manual and automated fruit and plant scanning in four different camera orientations, (3) fruit and plant scanning using 3D sensors (i.e. Microsoft Kinect, lidars, etc.), and (4) a visual servoing control law used for a single, as well as multiple, end-effector(s). Not only were various experiments performed, but the proposed simulation platform was shown to be capable of simulating, relatively, realistic 3D sweet pepper orchards (i.e. stems, leaves, fruits, etc.). In addition to relatively realistic sweet pepper orchards, the simulation platform was able to simulate a wide range of different sensors, as well as a few different pairing configurations of various types of arm/end-effector 3D models. The proposed simulation environment is an effective development and evaluation tool for autonomy in agriculture, however, it is only useful for indoor applications and does not consider outdoor agricultural environments which are a more challenging environment to simulate (i.e. adverse, cluttered, and many unknown environmental factors). In addition to indoor environments, the proposed work only focuses on the simulation of sweet pepper orchards and arm manipulators and is not useful for other areas of agricultural field robotic research (such as autonomous ground vehicles and common crops like corn). Additionally, Shamshiri et al. showed the use of a simulation environment capable of facilitating in the development and evaluation of various different control and sensing strategies, however, they did not consider the integration of the physical hardware components for algorithm evaluation.

Emmi et al.[15] showed how a simulation environment and a set of tools developed to be useful for the design and evaluation of fleet-based Precision Agriculture (PA) techniques. The proposed set of tools, called SEARFS, enables users the ability to easily configure various aspects related to the fleet-based PA systems, such as the individual robots that make up a given fleet (i.e. sensors, actuation, team role, etc.), the characteristics of the 3D simulated field environment (i.e. crop/weed spatial distribution/density, field obstacles, etc.), as well as aspects related to the high-level PA fleet-management strategies (i.e. field chemical treatment methods, weeding mechanisms, etc.). Additionally, SEARFS allows users the ability to integrate their own custom elements, such as individual robot control algorithms, custom robot models, as well as custom 3D crop and/or weed models. The proposed SEARFS provides a good example of a simulation environment that can be useful for many different applications in precision agriculture, by providing many user-configurable features for their specific needs, however, the proposed simulation environment focuses primarily on fleet management development and evaluation. Although SEAFRS claims to be open-source, it is based on Webots[42] and utilizes Matlab functionality (both of which are not free and available for everyone), which makes it difficult to use for some users. In addition to focusing on fleet management, the proposed simulation environment is strictly simulation/software based and does not consider physical hardware integration and support.

Linz et al.[40] use a combination of ROS and Gazebo to help develop a navigation controller for field service robot navigating through vineyards. Since outdoor trials are only feasible during limited conditions (i.e. spring and summer), Linz et al. use Gazebo to perform preliminary development and analysis of the intended navigation controller as it is used with a simulated robot model in its intended environment. After preliminary development, the navigation controller is used on the physical indoor robot platform and an artificial indoor environment, both of which are modeled with Gazebo and are determined to be sufficient surrogates for the intended outdoor use cases. Linz et al. show successful usage of a simulation environment to perform preliminary tests for a specific robot platform for a specific agricultural need prior to its application in the field.

As the need for more autonomy in agricultural applications increases, so to will there be an increased need for collaboration among the many different areas within agricultural field robotics, in order to facilitate in the improvement of autonomous agricultural field robotic methods. Even though developed methods are published and made available to the academic community, the amount of open-source field robotic software that has already been developed and well-documented is limited, which can make collaboration difficult. A solution to this could be an open-source software architecture, or framework, tailored to the field robotic academic community which can help to cultivate collaboration within the community and provide the tools enabling rapid development and testing of field robotic methods.

The work presented by Wang et al.[67] shows how ROS can be used to facilitate, and improve the development efficiency, of a developed algorithm allowing rapid deployment of the algorithm on an actual robot platform. Specifically, they present of the framework in which ROS was used to aid in the deployment of developed algorithms used on a humanoid agricultural robot platform, BUGABOO, used for the task of autonomously harvesting tomatoes. It was shown that the ROS tools that are available, particularly the robot modeling, visualization, motion planning, vision processing, and serial communication tools, can effectively reduce and benefit in the development and evaluation of robotic algorithms, especially when many different developers collaborate. Unfortunately, the proposed work only provides an example use case where the application of ROS is capable of being beneficial for collaboration among different research groups within agriculture, and there are no quantitative analysis or results presented in this work. Although the proposed work presents the successful use of ROS to enable the development of an agricultural robotic platform, its use was only investigated for the autonomous harvesting within an indoor tomato environment, and does not address the challenges of for other areas of agricultural research, as well as the difficulty of safe and reliable operation within outdoor agricultural environments.

Nebot et al. investigate the application of multiple agricultural robotic systems used for solving various tasks associated with orange groves (such as collecting oranges, grove weed, and tree maintenance, etc.)[44]. In this work, a novel control architecture, called Agriture, is presented, which combines the utilities of three separate systems resulting in a control architecture better suited for developing cooperative robotic systems, utilizing both physical and simulated hardware. The proposed architecture is based on the following three systems: (1) High-Level Architecture (HLA), (2) JADE, and (3) Player. HLA is the system which allows the possibility for both physical and simulated entities to be used at the same time, or individually. Entities can be anything in the system that can be simulated (i.e. robots, sensors, environments, etc.). Additionally, the HLA is used to handle the implicit communication between different elements within the system, as well as handling the exchange and synchronization of data that is exchanged between various elements. The JADE system is Java-based software framework used for agent-based application development, and it simplifies the implementation of multi-agent systems through a middleware. JADE is used to handle the explicit communication between system elements and is used to determine what information goes to which agent, as well as which tasks are assigned to which physical elements. The Player system is a network-oriented device server used to access the actuators and sensors of a robot, via TCP-based client/server interactions. Player is used to define the abstraction of physical external devices, either real or simulated (i.e. robot platforms). The Player system is accompanied with a lightweight and configurable multi-robot simulator, called Stage, which allows the control of simulated robots in virtual environments. Although no quantitative experiments were performed, Nebot et al. propose potential applications where the proposed architecture could be used. One of the proposed applications mentioned, that is of interest, is the use of Agriture allowing hybrid development applications, where both real and simulated elements are combined, and used, together. The proposed hybrid application where the Agriture architecture could be useful for agricultural robot development is the use of real robots operating in an empty warehouse, where real video and GPS data, collected elsewhere (i.e. real orange groves), can be overlaid such that the robot thinks that it is the real field. Unfortunately, the presented control architecture only provides a framework that is potentially useful for the development of multi-agent agricultural systems utilizing real, simulated, and hybrid data, and no experimental results are presented. Additionally, the Agriture is only an interface for various components and has not been shown to be useful for evaluating field robotic algorithms.

It is a challenge in the general robotic community to reliably benchmark different multi-robot control algorithms because they are all evaluated differently (i.e. physical robotic setups, the method of simulation, environmental conditions, etc.). Since the specific methods used for benchmarking can differ widely within the academic community, the results published in academic papers can be quantitatively incomparable. Yan et al.[70] proposed an open-source simulation test bed to help solve this problem. The proposed approach was developed to provide the academic community with a reproducible framework capable of allowing many different multi-robot control algorithms to be benchmarked, and report results, easily. The proposed testbed is based on ROS, and utilizes a set of ROS nodes used to contain the following three developed components: (1) MORSE, which is a 3D realistic simulation platform used for simulating the multi-robot system, (2) various different robot controllers which are used for individual robot control and interaction, as well as team coordination, and (3) a monitor used to supervise all of the running experimental processes. Additionally, the work proposed the distribution of the robot controllers among a computer cluster, which acts to simulate multiple individual robots interacting together in real-time. The distributed robot controllers were deployed among a cluster of 70 high-performance computers. Yan et al. proposed the use of an open-source virtual machine in order to maximize the re-usability of the proposed testbed by enabling each computer in the cluster to support the use of ROS (i.e. computer operating systems varied and may not support ROS). Yan et al. showed the capabilities of a simulation testbed proposed developed to support the benchmarking of, and collaboration within the academic community of, multi-robot coordination, however, their work does not consider its application for agricultural specific applications. Additionally, the proposed simulation testbed is only interested in distributed approaches and requires individual robot controllers to provide their own ROS nodes for enabling centralized approaches. In addition to utilizing an open-source virtual machine, the proposed test bed requires the support of ROS, which limits its ability to be used for realistic benchmarking of physical multi-robot systems where the hardware for some of the individual robots may not be able to support ROS and/or a virtual machine.

Jensen et al.[32] proposed an open-source software platform, called FroboMind, which was tailored specifically for field robotic applications in precision agriculture (PA). FroboMind aims to support the advancement of field robotic systems utilized in agriculture by cultivating collaboration within the academic community, facilitate in software reuse at an application level, and support quick development and evaluation of investigated methods across a wide range of agricultural projects. FroboMind is based on the commonly used robotic middleware ROS, and its architecture is composed of the following three component layers: (1) Perception is the layer of components that are used to perceive the surrounding environment using an established abstraction of the physical sensors to collect data, in addition to processing the collected raw data into data more useful for other system functionalities (i.e. navigation, obstacle avoidance, etc.), (2) Decision Making (DM) is the layer of cognitive components responsible for keeping track of the information received from the sensors and outputting the required robotic system behaviors, necessary for achieving a user-defined task, and finally (3) Action is the layer of components responsible for the abstraction and low-level control of the robotic system's hardware components necessary to achieve the behaviors generated from the DM layer. The aforementioned layers, and their associated components are represented in the form of individual ROS packages, all of which are located in the FroboMind's "architecture layer" directory. Experimental results were obtained with the application of FroboMind for the following: (1) computational resource (i.e. CPU and RAM) loads were collected, at 5Hz, showing the near-real-time operation of differentially-driven robotic platform executing autonomous waypoint following, (2) during a 5 day development workshop, a breakdown on the amount of time and resources spent on different development tasks was documented, in order to quantitatively evaluate the effect of using FroboMind to quickly enable any custom developed robotic platform to solve a typical PA application (i.e. autonomous navigation rows of an apple tree orchard), and finally (3) the number of lines of code (C++, Python, etc.) contained in all of the custom developed files for running a software-based system, or Source Lines Of Code (SLOC), that was used within a span of 3 years throughout various PA projects (i.e. GPS-based navigation of row crops, control of both passive and active autonomous farming implements, etc.), was compared in order to quantitatively evaluate FroboMind's software reuse capabilities. FroboMind was shown to facilitate in the integration of new software implementations onto a physical robotic platform, enabling real field evaluations relatively quickly, however, the presence of the necessary robotic hardware was required and does not address the application of a simulation environment to enable hardware-dependent development. Additionally, it was shown that complications resulting from undesirable field conditions (i.e. changing weather) had an impact on the amount of time spent towards the successful implementation and evaluation of the autonomous robotic platform in real field trials, which highlights one of the fundamental obstacles that can hinder the development and evaluation of most, if not all, autonomous field robotic systems used to solve many problems faced in agricultural applications. This fundamental obstacle is that both the physical hardware elements essential for robotic platform operation, as well as desirable environmental and target field conditions, must be present in order for proper and efficient field trials to be performed. This provides a glimpse of the potential importance that simulation environments can have for enabling the efficient development and evaluation of novel autonomous agricultural field robotic systems and algorithms in a wide range of agricultural applications, by allowing the simulation of both of these essential elements (i.e. robot platform and field environment conditions).

2.2 Robot Operating System (ROS)

The Robot Operating System (ROS) [50] is a widely used open-source, multilingual (C++, Python, Java, etc.), peer-to-peer based software framework that supports many different types of robotic platforms, manipulators, sensors, and algorithms. Due to the vast amount of open-source developed packages, drivers, and algorithms, the very active and wide community support, and the wide range of supported robotic and sensor hardware, ROS can be very useful tool that can simplify the development and evaluation of robotic algorithms for a wide range of robotic applications, both in simulation and on hardware. Since it provides many useful features that help to improve the development of general robotic applications, it was chosen in this thesis to utilize these features of ROS to develop a development environment aimed, specifically, at facilitating in the development of field robotic methods for use in agricultural environments.

This section will first provide an overview of some the fundamental elements of ROS that will be helpful for understanding some of the, later discussed, components developed in this work. Finally, this section will give a brief introduction to the standard utilities, provided by ROS, that can be useful for troubleshooting and debugging of developed robotic systems.

2.2.1 Fundamental Components

When any work being developed uses ROS in some fashion, it is necessary to first discuss some of the basic components that make up ROS, in order to have a general understanding of how ROS operates. For the work developed in this thesis, the following components, and how they are used within ROS, will be discussed: (1) Nodes, (2) topics, (3) messages, (4) publishers and subscribers, and finally (5) coordinate frames.

ROS **nodes** can be thought of as modular "software modules", or processes using some type of supported programming language that performs some type of computation, or function, such as processing sensor data or controlling a robot's motors. When multiple nodes exist, either on a single computer or distributed among several computers, what is known as a "ROS network" is formed. Any node on a "ROS network" is able to interact, and share data, with other ROS nodes on the same "ROS network", an example can be seen **Figure 2.8**.

ROS **topics** can be thought as data sources, or a relay for a specific type of information, such as the acceleration data received by an IMU. ROS topics allow the data, that is available from the various sources of information over a ROS network, to be passed between, and used by, multiple ROS nodes. Since there can be many ROS topics available at any one time, ROS topics should be identified by a unique string identifier, such as `imu_1/acceleration_data`.

ROS **messages** can be thought of as the container for specific information that is available in a ROS network. A ROS message, or *msg*, is a strictly typed data structure that is used to store and pass data between nodes, via topics, and is defined in a *.msg* file, an example is shown in **Listing 2.1**. A *msg* can be made up of any basic primitive data type (integer, floating points, booleans, etc) and can even be made up of other ROS messages.

```
# This expresses a position and orientation on a 2D manifold.
float64 x
float64 y
float64 theta
```

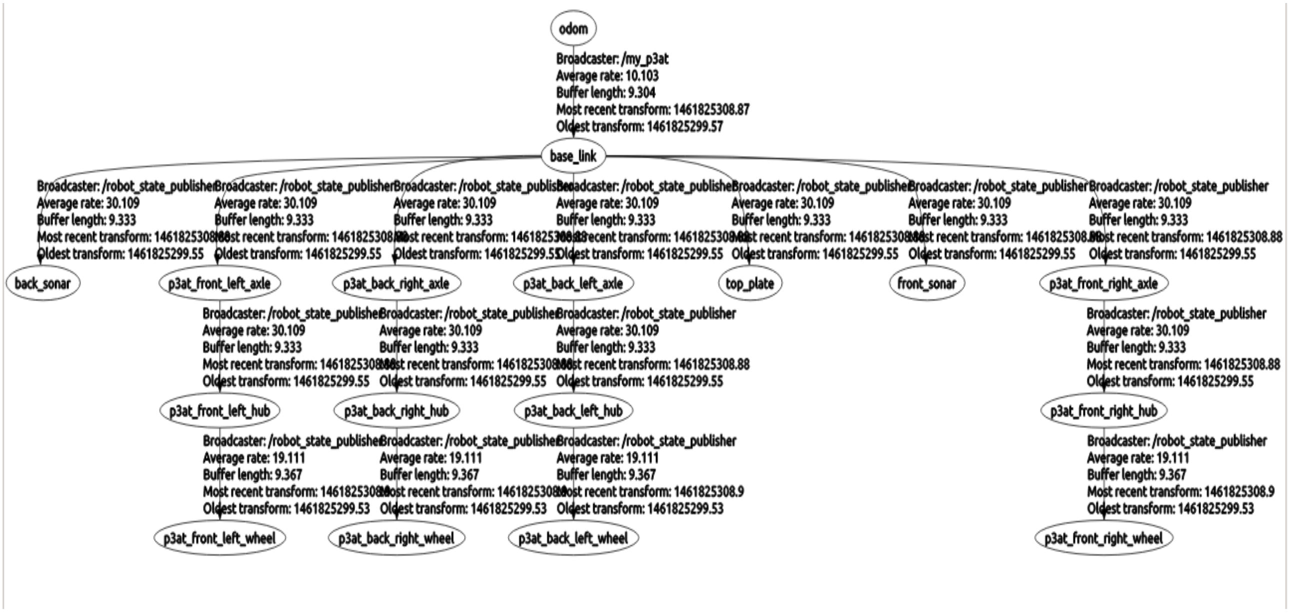
Listing 2.1: Example of a ROS message defined in the "Pose2D.msg" file.

ROS **publishers** can be thought of as the entities in charge of making information available on a ROS network. Publishers are the components that allow data to be available, and used, by other nodes on a ROS network. Publishers broadcast information, stored in the form of a ROS message, to a specific ROS topic, in order for other ROS nodes, can use that information for their own purposes.

ROS **subscribers** can be thought of as the ROS publisher counterparts. Specifically, subscribers are the components that listen to a specific ROS topic and receive the data of that topic as soon as any ROS messages become available. When the ROS topic being "subscribed" to has ROS messages available, the subscriber can be used by a ROS node to retrieve the available data, of a ROS topic, in order to be used internally.

Although not a fundamental component of ROS, coordinate transformations are the origins to the various objects in a robotic system that interact with each other, and they are fundamental for properly characterizing the geometric relations between these objects. Most, if not all, robotic systems can be described as a collection of 3D coordinate frames, such as a camera frame, robot base frame, world frame, etc., that change over time. Each of these 3D coordinate frames can be related to each other using a series of coordinate transformations. These coordinate transformations can easily be represented in ROS, using the ROS package called *tf*, which can automatically keep track of the all the defined coordinate frames as they change with time while maintaining the relationship between the frames in a tree structure. For example, shown in **Figure 2.7**, the Pioneer 3AT robot would have the resulting *tf* tree structure, which is used to keep track of the 3D coordinate frame's pose for each sub-component of the 3AT robot.

(a) Figure 2.7 (continued)



(b)

Figure 2.7: (a) Shows the Pioneer 3AT robot platform. (b) Shows the resulting `tf` tree of the Pioneer 3AT platform as represented in ROS and all of the associations between its various sub-components.

2.2.2 Standard ROS Utilities

ROS has a variety of standard utilities that can be very useful debugging and visualization tools, which make it easier to interact with and troubleshoot, the many various elements operating within a ROS network. Most of these tools can be easily used, and are interact-able, from a ROS application, called RQT, which is based on the Qt framework[49] for developing Graphical User Interfaces (GUI). The RQT package can be used to easily interact with the various elements of, and tools available with, ROS from a GUI.

The **topic monitor** is a utility that displays all of the ROS topics that exist on a given ROS network, in addition to being able to view the information, when available, from each topic. Specifically, the topic monitor can be used to quickly see which ROS topics, associated with the robot platform (i.e sensor data, motion commands, etc.), are available, the frequency in which each available topic is being published, as well as the data (i.e sensor data, data timestamps, etc.) associated with each available ROS topic.

The **message publisher** is a utility used to publish any type of ROS message to any ROS topic that exists on a ROS network. This is a particularly useful utility when it is desired to quickly test the behavior of a ROS node using a user-defined ROS message sent to a specific topic. For example, the message publisher could be used to send commands to a developed state machine, in order to test its behavior for worst-case scenarios or scenarios that may not occur frequently.

The **node graph** is a utility used to show all of the ROS nodes and topics that exist on a ROS network at any given moment in time, as well as how the nodes and topics interact and communicate with each other, in the form of an undirected graph, an example of which can be seen in **Figure 2.8**. This is a useful tool for verifying that various ROS nodes are properly interacting with the other ROS nodes they should be interacting with, as well as ensuring that the various ROS nodes are publishing to, and receiving from, the proper ROS topics. For example, suppose that a joystick, handled by the `/joy_node`, is supposed to be used to control the motion of a robot platform, via the `/cmd_vel` topic, the nominal node graph can be shown in **Figure 2.8**. In the event that the robot platform does not move when inputs are given to the joystick, the node graph could be used to see if the various ROS elements exist and are properly connected.

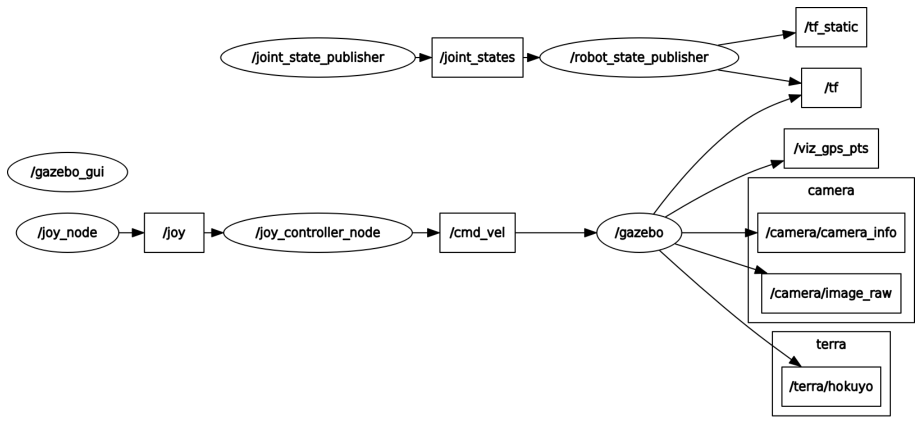
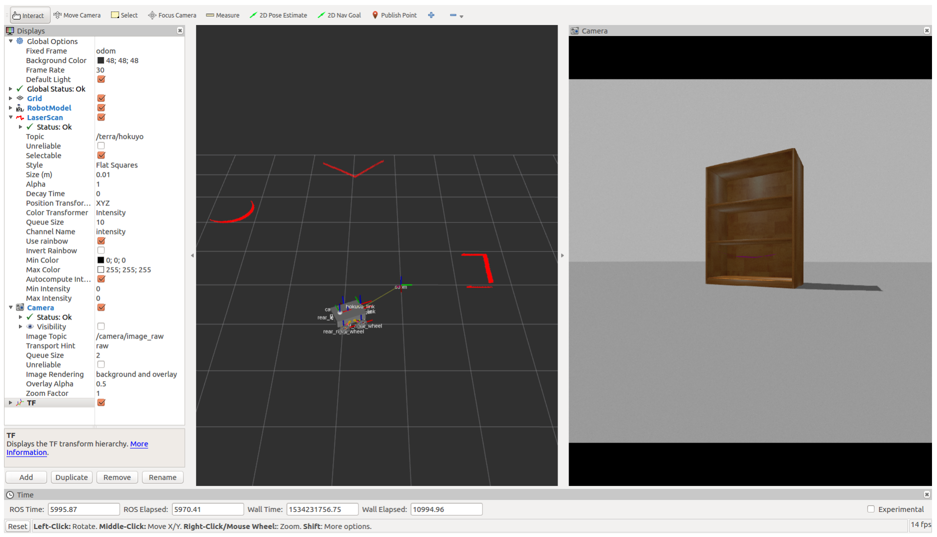


Figure 2.8: Shows the node graph with the available ROS topics and nodes on the ROS network in a scenario where a joystick is used to control the motion of robot model simulated in Gazebo. ROS nodes are represented by the elements contained in an ellipse, ROS topics are represented by the elements contained in a rectangle, and the ROS topic information that is shared in the form of ROS messages is represented by the arrows connecting the different components.

The ***tf* tree** is a utility plugin that visualizes the *tf* tree, or the tree-like representation of all the relationships between the various coordinate frames associated with the components used to compose any specified robotic platform, or model (i.e humanoid arm manipulator). The *tf* tree can be visualized either using a command-line-utility or with the available RQT plugin, associated with the ***tf*** package. Visualizing a robotic platform's *tf* tree can be useful when it is necessary to verify that the various sub-components of a robotic platform are properly modeled and connected to each other, in order to ensure that any pose or sensor information is correctly generated.

The `rosbag` is a very useful tool that can be used to record specific, or all of the ROS topics available on a ROS network during the time of recording, as well as saving the recorded topics into a ROS bag, or a `.bag` file. In addition to being recorded, the recorded ROS topics, of a particular `.bag` file can be played back later. ROS bags are especially useful when it is desired to replay the various ROS information available during a specific scenario, or when it is desired to post-process, using external software such as Matlab or a *python* script, the various recorded ROS topics.

Rviz is a GUI utility that can be used to visualize various forms of information that is available over a ROS network. Some of the forms of information that can be visualized include, but is not limited to, the following: robot models, 3D poses of various different bodies, markers useful for visualizing a trajectory or landmarks, Lidar data, video camera streams, and IMU data. Being able to visualize data, in itself, can be helpful to understand a given situation by presenting the information in a different manner. Rviz is a useful tool for visualizing the various types of data available, especially when it is necessary for viewing the information as seen from various perspectives from ROS. For example, suppose that the output of a vision-based control algorithm, being used by a robot platform, does not exhibit the correct behavior that should be witnessed at a known location. Rviz could be useful for evaluating the validity of the camera's input image stream, by visualizing the raw or processed, image stream, an example of this can be seen in **Figure 2.9**.



Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
Grid	✓
RobotModel	✓
LaserScan	✓
✓ Status: Ok	
Topic	/terra/hokuyo
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Min Color	■ 0; 0; 0
Max Color	☐ 255; 255; 255
Autocompute Int...	✓
Min Intensity	0
Max Intensity	0
Camera	✓
✓ Status: Ok	
Visibility	☐
Image Topic	/camera/image_raw
Transport Hint	raw
Queue Size	2
Unreliable	☐
Image Rendering	background and overlay
Overlay Alpha	0.5
Zoom Factor	1
TF	✓

TF
Displays the TF transform hierarchy. More Information.

ROS Time: 5995.87 ROS Elapsed: 5970.41 Wall Time: 1534231756.75 Wall Elapsed: 10994.96 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift:** More options. 14 fps

Figure 2.9: Shows the rviz GUI showing the image stream (right window), the lidar data (red lines), and the coordinate frames of the various sub-components of a given robot model simulated in an example Gazebo world.

2.3 Gazebo

Although there have been many different multi-robot 3D simulation platforms available, previously discussed, that offer many of the same capabilities and features, Gazebo was chosen as the most suitable simulation environment for use in this work, because it had the most features and capabilities suitable for facilitating in the development, evaluation, and application of algorithms intended for field robotics research for agricultural applications. These features that were deemed as the most suitable for field robotic algorithm testing were the following: (1) it is fully open-source, enabling potential collaborating developers the ability to use, as well as develop their own modified version of Gazebo easily, (2) wider and more active community support, (3) well maintained and better documentation of code and API, (4) easiest interaction with an interface of ROS, (5) easier modification and programmatic generation of custom simulated objects and worlds, and finally (6) best trade-off between computational overhead of accurate simulation of complex environments and available features.

This section discusses some the fundamental aspects of Gazebo necessary for the creation of the development environment presented in this work. The following section discusses the following: (1) the various elements used to compose any given simulated object in Gazebo, as well as how the various objects and worlds can be represented, (2) the method in which various types of sensor measurements can be simulated in Gazebo, as well as how various Gazebo simulated objects can be interacted with and controlled, and finally (3) the structure of a Gazebo Model Directory enabling quick, and easy, selection, as well as loading, of various custom develop models within Gazebo.

2.3.1 Representing Simulated Objects

Gazebo is capable of simulating the physical composition for a very wide range of custom objects (i.e robots, sensors, worlds, etc.), or models, and the physical composition for any given object can be represented using a combination of the following two primary elements: **links**, and **joints**. The following section discusses the properties of these two primary elements, as well as discusses how they can be assembled and used, in order for the physical body of any given object to be simulated in Gazebo.

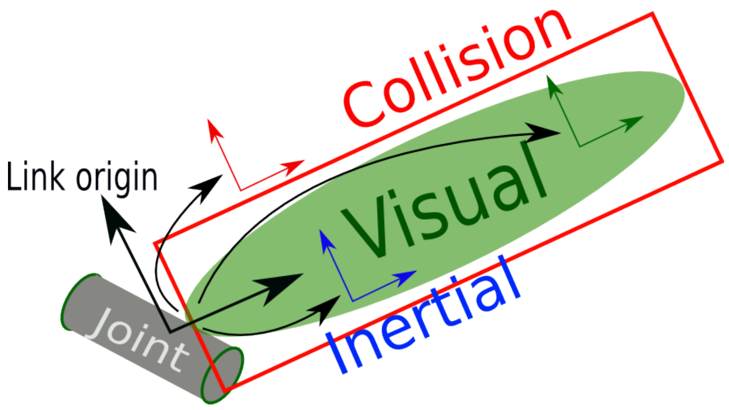


Figure 2.10: Shows the the different different representational (i.e. collision, visual, and inertial) bodies that are used for defining the construction of a simulated object, or link.

Links, an example is shown in **Figure 2.10**, are the individual rigid-body components of a simulated model. Any simulated model, simple or complex, can be composed of a single link, such as a wheel, or as multiple links, such as an arm-type manipulator (e.g. base, arm, and manipulator). Although many other elements can be used to create any imaginable custom body, the essential elements used to define any link are the following:

- **Origin:** The 3D pose of the coordinate frame used as the origin of the defined link. This coordinate frame is used as the reference origin used for relating the coordinate frame origins of the visual, collisional, and inertial bodies respective to the associated link.

- **Inertial:** The *inertial* body of a link, used to define all of its inertial properties, such as mass and rotational inertias.

- **Visual:** The *visual* body of a link, used to define a link's various visual properties such as the visual geometry, color, and texture, which are used to form the visual rendering of the link as seen in Gazebo. A link can have multiple *visual* bodies defined which can be combined to form a composite visual body.

- **Collision:** The *collision* body of a link, used to define a link's collisional boundaries, or geometry. The collisional body of a link is the geometry that is used to simulate the contact, and dynamic collisions, with other simulated objects in the surrounding environment. Similar to the *visual* body, a link can have multiple *collision* bodies defined which can be combined to form the composite *collision* body of a link, however, the computational resources required by the physics engine increases as the complexity of a link's composite collisional body increases.

The geometrical bodies used to define both the *visual* and *collision* bodies of a link can be represented using either simple geometrical shapes (i.e cube, cylinder, sphere) or a supported 3D surface mesh file. Gazebo supports the following 3D surface mesh file formats: (1) Stereolithography (STL)[54], or **.stl* files, (2) Collada[1], or **.dae* files, and (3) Wavefront[52], or **.obj* files.

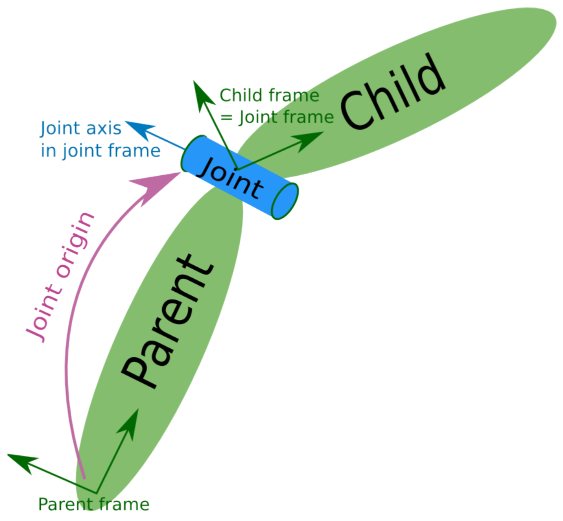


Figure 2.11: Shows how a joint is used to attach two different links, and how the joint's origin and the links' origin coordinate frames are related relative to each other.

Joints are the dynamic connections between two different bodies, or links, and enable all of a simulated model's associated links to be attached, as shown in **Figure 2.11**. For example, a joint can be used to connect a simulated motor output shaft to a simulate wheel enabling it to be controlled. Although there are many kinematic and dynamic properties that can be defined by it, the most basic and fundamental elements used to define any joint are the following:

- **Type:** is the specific type of joint (such as fixed, revolute, prismatic, and more) used to connect the attached links.
- **Origin:** is the 3D transformation from the parent link's coordinate frame origin into the child link's coordinate frame origin.
- **Parent:** is the name of the parent link.
- **Child:** is the name of the child link to be connected to the parent link.
- **Axis:** is the axis about which the translational/rotational movement, if allowed, between the two links is allowed, in reference to the joint reference coordinate system.

Now that it is possible to properly represent the individual elements that could be used to construct any simulated object, it is necessary to represent the entire composition (i.e links, joints, plugins, etc.) of a custom simulated object, such that it can be represented properly within ROS or simulated using Gazebo. Two different XML-style file formats, the Unified Robot Description Format (URDF) [65] and the Simulation Description Format (SDF) [56], both can be used for defining all of the links, joints, and miscellaneous characteristics (i.e attached Gazebo plugins, physics engine parameters, rendered lighting, etc.) associated with any given simulated object in specially named files, **.urdf* files for the URDF-style and **.sdf* files for the SDF-style. Both of these formats are supported by ROS and Gazebo, and they share some common elements and can be used to represent the same exact object. For example, the SDF and URDF files, defined in **Listing A.1** and **Listing A.2** respectively, both result in the same Gazebo-simulated simple GPS antenna, shown in **Figure 2.12**.

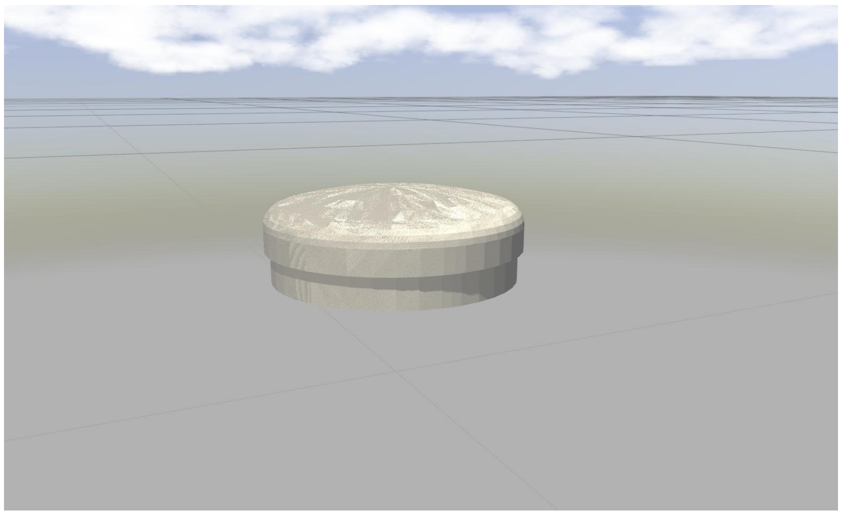


Figure 2.12: A simple GPS model simulated in Gazebo as a result of using both the SDF and URDF model representation approaches, defined in **Listing A.1** and **Listing A.2**, respectively.

Although both of these file formats can be used to represent and simulate the same object, they do not share the same potential capabilities. The SDF format is the only format that is supported for simulating a Gazebo world and is capable of defining many characteristics other than those required to simulate a robot model. For example, the SDF format, used within an example Gazebo world description file (i.e **.world*), can be used to define characteristics pertinent to the Gazebo simulated world including, but is not limited to, the following characteristics associated with the world: any simulated models (i.e obstacles, robots, etc.), environmental conditions (i.e lighting, atmospheric conditions, etc.), physics simulation characteristics (i.e world physics, physics engine and solver parameters used, etc.), and world terrain.

Contrary to the SDF format, the URDF format is only capable of defining elements associated with a single robot or model. Although its capabilities may seem limited to the SDF format, the URDF format is capable of implementing the use of the XML Macros (xacro) language [68], which is not supported by the SDF format. The xacro language (denoted by **.xacro* files) within URDF files (denoted by **.urdf.xacro* files) enables the following functionalities, all of which make the development and configuration of URDF models much easier: (1) modular design/configuration of simulated objects, and easy re-use of previously developed components, by enabling the use of elements defined within multiple other **.urdf.xacro* files, (2) support for parameterization of URDF/*xacro* components, using variables and function-like methods, enables quicker modification, configuration, and simplified use of large portions of code.

2.3.2 Gazebo Plugins

Not only can Gazebo properly simulate the physical composition of any given object, but it can also simulate any custom behavior associated with that object through the use of "Gazebo Plugins". Gazebo plugins, or **Plugins**, are Gazebo-standard C++ classes that have direct access to all of Gazebo's core functionalities. Plugins are extremely useful tools which enable developers to have custom control over any aspect related to the Gazebo simulation environment, such as simulating a sensor measurement, controlling the behavior of a specific simulated object, and even programmatic control of, and interaction with, the Gazebo system, GUI, and the world. Not only do Gazebo plugins give users control over the Gazebo environment, but they also automatically support the integration of, and interaction with, ROS. This provides an additional layer of ease enabling even quicker development, and easier use, of Gazebo for the development, evaluation, and extension of autonomous robotic methods.

There are six main categories of Gazebo plugin types: (1) World plugins are plugins that can be attached to a simulated world and give users the ability to control various world properties (i.e physics engine, ambient lighting, etc.), (2) Model plugins can be attached to a specific model and enable users the control over that model's behaviors, (3) Sensor plugins can be attached to a specific sensor and give users the ability to have control over that sensor's properties and how it acquires information, (4) System plugins are executed before the Gazebo environment is loaded and gives users the control over the Gazebo startup process, (5) Visual plugins can be attached to specific models and gives users control over the object's visual properties and how it is visually rendered, and (6) GUI plugins can be attached to the Gazebo GUI and give users the ability to add-on to, modify, and have additional control over the Gazebo GUI window.

As previously mentioned, one of the benefits of using Gazebo is that it has high-quality community support and crowd-sourced content, which can enable it to support the rapid development and testing for autonomous field robotic methods developed specifically for agricultural applications. These benefits can be witnessed in this thesis from the freely available and ready-to-use Gazebo sensor and model plugins that have already been developed by Team Hector[38] and the Gazebo development team [20]. From these open-source plugins, some were used as is for the developed simulated sensor models (i.e IMU, GPS, Lidar, camera, etc.) and the control of the simulated robot models (i.e differential and skid-steering drive controllers), and some of them had been taken and modified in order to enable custom sensor behavior more useful for rapid agricultural field robot development and evaluation.

There are various methods in order for any open-source or custom developed Gazebo plugin to be properly applied to and used by the Gazebo environment, all of which depend on the specific plugin. In this thesis, the model and sensor plugins were primarily interacted with and were therefore easily attached to and used by the various developed simulated models by simply including them in the respectively **.urdf.xacro* files of the associated attached models.

2.3.3 Gazebo Model Database

As previously mentioned, Gazebo supports the use of the SDF format to define custom simulated worlds. In addition to defining custom worlds, Gazebo supports the use of the SDF format to create statically defined simulated objects, or objects that exhibit modifications very rarely, which can be found, and kept track of, by Gazebo allowing easy loading and simulation, using either a GUI based "drag-n-drop" technique or including into a **.world* file. This quick and easy usage of SDF defined models is enabled with the application of the Gazebo Model Database (GMDB), which is the collection of all the available models that can be simulated using Gazebo. The GMDB not only keeps track of all the community-developed available models, but it also can keep track of any local custom-developed models. In order for a simulated model to be recognized by, and usable through, the GMDB, a simulated model has to be contained within a Gazebo Model Directory (GMD) which has a specific directory structure. For example, given a crop model, to be named `test_crop_model`, that is to be used to create an example simulated agricultural field, the crop model must be contained within a GMD, root directory named `test_crop_model`, which should have the following directory structure and files:

- ***model.config*:** Meta-data about `test_crop_model`. Each model must have a *model.config* file in its root directory, which contains meta information about the model (such as the author, description, etc.).
- ***model.sdf*:** SDF file describing all the links, joints, plugins, and other properties describing the model, `test_crop_model`.
- ***model.sdf.erb*:** Standard SDF file which can contain Ruby code embedded. This option is used to programmatically generate SDF files using Embedded Ruby code templates. Please note that the ruby conversion should be done manually (erb model.sdf.erb ¿ model.sdf) and the final model.sdf file must be uploaded together with the model.sdf.erb.
- ***meshes*:** A directory for all COLLADA and STL files. This is an optional directory that contains all of the COLLADA and/or STL files for the model.
- ***plugins*:** A directory for plugin source and header files
- ***materials*:** A directory which should only contain the textures and scripts subdirectories. This is an optional directory that contains all of the textures, images, and OGRE scripts for the model. Texture images must be placed in the textures subdirectory, and OGRE script files in the scripts directory.
  - ***textures*:** A directory for image files (jpg, png, etc).
  - ***scripts*:** A directory for OGRE material scripts

2.4 Summary

This chapter first discussed previous work that has been done in the area of field robotics in agricultural applications, in addition to some of the simulation platforms that are available and could potentially be useful for supporting field robotic development. A brief overview of the Robot Operating System (ROS) middleware is then presented, where some of its key components and utilities are discussed. Finally, this chapter discusses the Gazebo simulation platform that was chosen for use in the following work that is presented.

Chapter 3

System Overview

This chapter provides an overview of the novel development environment presented in this work to enable a more efficient approach to development and field evaluation for general agricultural robotic applications, by enabling development to take place even when unfavorable field conditions may exist or essential hardware is unavailable. Next, this chapter discusses the physical robotic platform that was used in order to evaluate the proposed development environments capability to enable the development and evaluation of various different autonomous field robotic algorithms intended for operation on a custom target robotic platform in real field environments.

3.1 Novel Development Environment

The primary contribution presented in this work is the novel environment named ARDEE, or Agricultural Robotic Development and Evaluation Environment. The ARDEE architecture is based on the ROS middleware, previously discussed, enabling quick development, deployment, and evaluation of AFR by utilizing the benefits of having many different open-source hardware driver interfaces and general robotic algorithms already developed and well-maintained. This section will first highlight some of the core components that make up ARDEE and how they help to fulfill the objectives defined in this work. Finally, this section will discuss how ARDEE can be utilized to help facilitate AFR development and evaluation for a few different scenarios in agricultural applications.

3.1.1 Core Components

ARDEE consists of the following three primary components: **(1)** Realistic crop and field simulation, **(2)** realistic field robotic platform (i.e sensors, robot base dynamics, etc.) simulation allowing development and testing regardless of any physical resource limitations (i.e no crops, hardware, etc.), **(3)** UDP-based communication mechanisms for extending ARDEE utilization across multiple different hardware platforms and devices, and **(4)** user-friendly functionalities and utility functions enabling setup, customization, and utilization of ARDEE for supporting advancement of autonomy in agriculture (i.e enable usage and collaboration for both expert and beginning developers/users), all of which will be discussed later in **Chapter 4**.

Realistic Crop and Field Simulation

Many environments in agriculture are constantly changing (i.e. crop growth, lighting conditions throughout the day, etc.) and will occasionally experience drastic changes caused by weather (i.e. tornadoes, heavy rainstorms, etc.). These changes in the environment could result in a field robot experiencing abnormal behaviors, which could possibly lead to damage of vital crops and/or robotic hardware. As a result, when developing field robotic methods whose performance is dependent upon its environment, it is important that these methods be tested in their target environments, in order to properly evaluate performance and develop sufficiently robust robotic methods.

This is particularly true in the case of field robotics in agricultural applications, where safe and reliable operation is required to prevent any harm done to farmers, field equipment, or high-value crops. When developing safe and robust robotic methods that may be required to operate under vastly different field environments (i.e crop, field, and lighting conditions) it is ideal to perform evaluation and testing within the real field environment using the target robotic hardware.

Unfortunately, real field testing can be often times be prevented for long periods of time, which can be a major hindrance when time-critical field robotic development and testing is required. For example, in some areas in Illinois corn can be out of season five months out of every year. Therefore, in an attempt to create a development environment intended to be beneficial for a broad range of agricultural applications and research, the following simulated crop models, shown in **Figure 3.1**, have been developed in order to be properly simulated in Gazebo, and were chosen due to their significance in the TERRA-MEPP and RIPE projects: (1) 22 variations of Corn, or Zea Mays, (2) 9 variations of sorghum, or Sorghum Bicolor, and (3) 9 variations of tobacco, or Nicotiana tabacum.

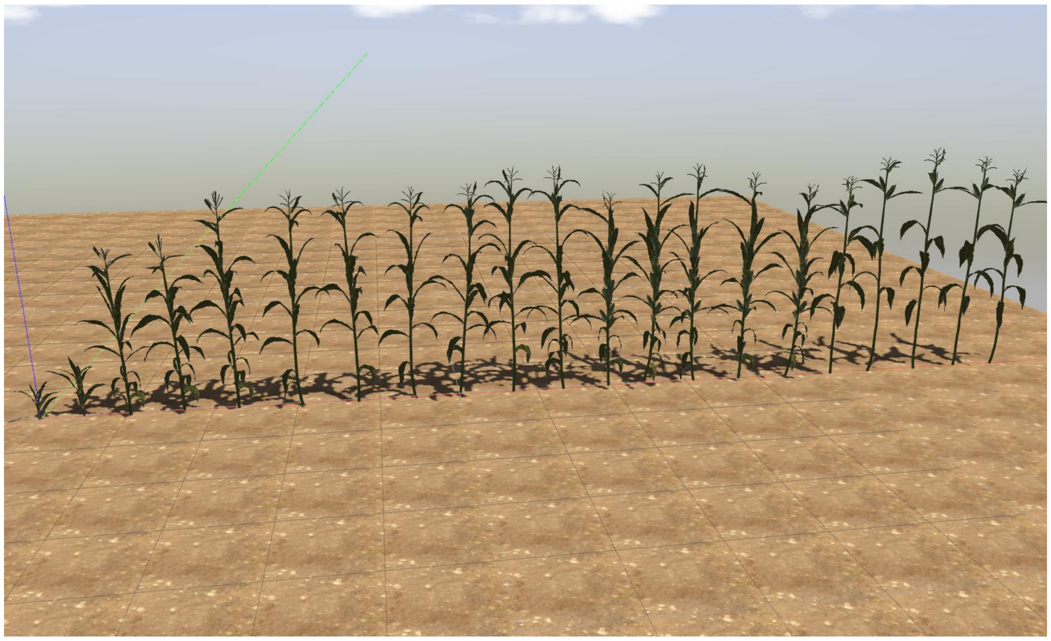


Figure 3.1 (cont.): Shows all of the available simulated corn model variations that have been extracted for Gazebo simulation.

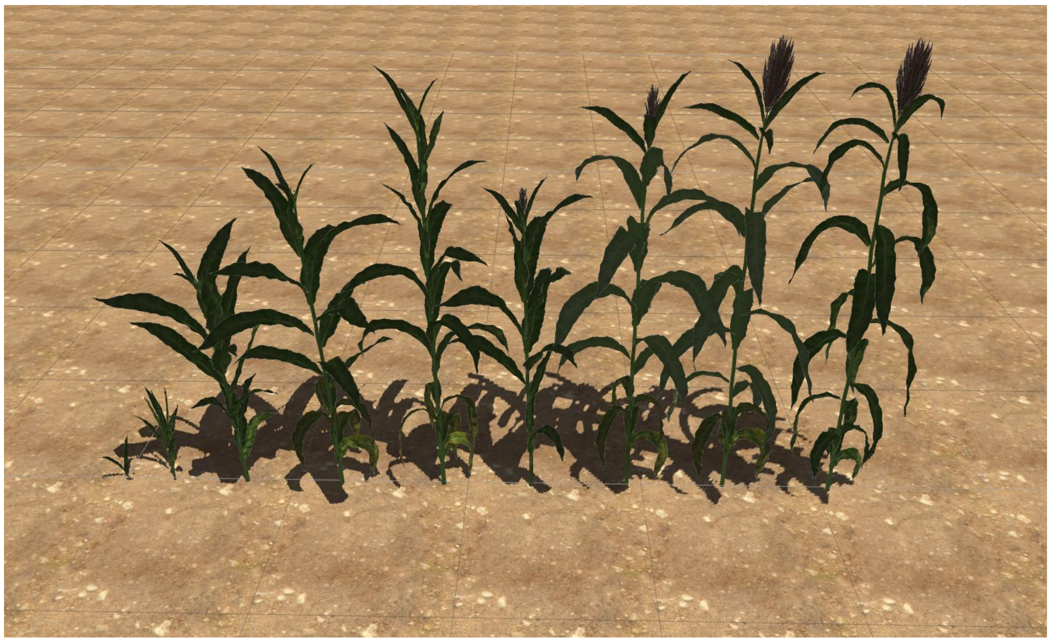


Figure 3.1 (cont.): Shows all of the available simulated sorghum model variations that have been extracted for Gazebo simulation.

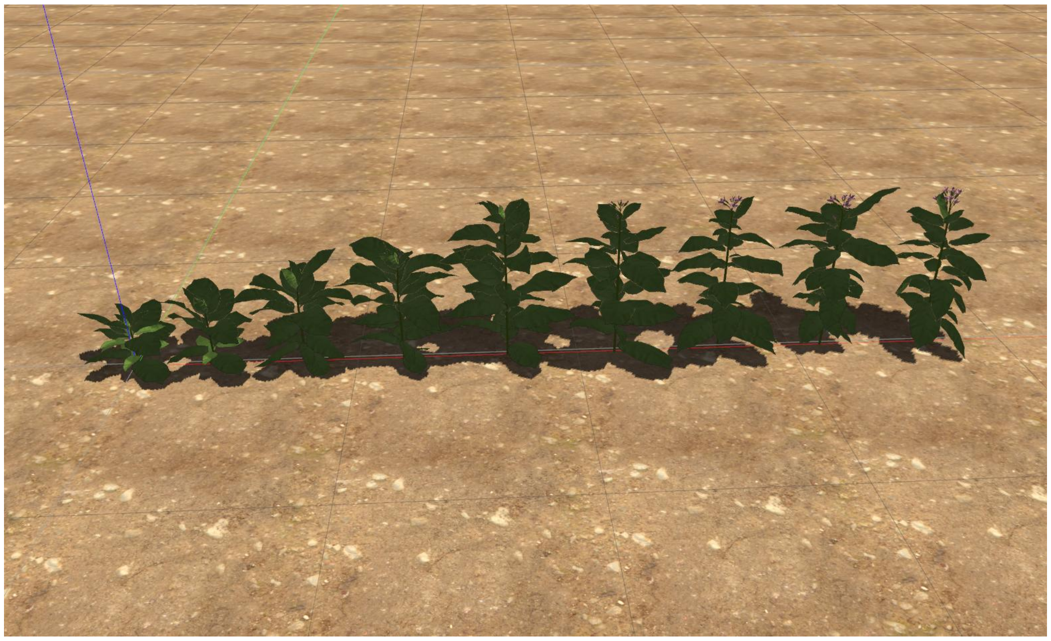


Figure 3.1 (cont.): Shows all of the available simulated tobacco model variations that have been extracted for Gazebo simulation.

Figure 3.1: Shows an overview of all the available individual crop models that were extracted and processed for Gazebo simulation. The crop models presented are intended to simulate several different variations in the early and late stages of crop development for corn, sorghum, and tobacco.

Although, the crop models presented are not useful for every possible agricultural and biological application and research topic (i.e robotic harvesting applications), the process that was used to develop the Gazebo-simulatable crop models is discussed in **Section 4.2.1**, such that many more specific crop species can be added to the collection of available simulated crop models.

Field Robot and Sensors Simulation

Similar to environmental factors, there are many factors that commonly occur during field robotic development and testing that can prevent the target robotic system from being used, such as adverse weather conditions, limited energy source, and damaged components resulting from extended operation in potentially harsh field environments. Unlike environmental factors, these resource factors can be prevented and fixed, however, they can still occur resulting in effective development and field evaluation being halted and precious development time being wasted.

A collection of commonly used sensor models ready for Gazebo simulation have been made available, shown in **Table 3.1**, and can easily be configured and attached for any user-specific needs. These simulated sensors can be used to supplement time-critical development and evaluation of developed field robotic methods when essential resources (i.e. robot hardware, sensors, etc.) are unavailable.

Sensor Type	Sensor Name	xacro macro call-name
GPS	Standard GPS	`sensor_gps`
	GPS w/ variable accuracy	`sensor_gps_with_dropout`
	Standard GPS on mast	`sensor_gps_with_mast`
IMU	Hector IMU	`sensor_imu_hector`
	Modified Hector IMU	`sensor_imu_terra`
2D Lidar	Hokuyo URG-04LX	`sensor_hokuyo_urg04lx`
	Hokuyo UST-10LX	`sensor_hokuyo_ust10lx`
	Hokuyo UTM-30LX	`sensor_hokuyo_utm30lx`
	Sick S300	`sensor_sick_s300`
	Sick S3000	`sensor_sick_s3000`
	Sick TIM 571	`sensor_sick_tim571`
	RPLidar	`sensor_rplidar`
3D Lidar	Hokuyo 3D	`sensor_hokuyo3d`
RGB Camera	Axis M5013	`sensor_axis_m5013`
	Axis P5512	`sensor_axis`
	GigE uEye CP	`sensor_ueye_cp_gige`
RGBD Camera	Asus Xtion Pro	`sensor_asus_xtion_pro`
	Kinect	`sensor_kinect`
	Kinect V2	`sensor_kinectv2`
	Fotonic E Series	`sensor_fotonic`
	OrbBec Astra	`sensor_orbbec_astra`
Misc	Sonar	`sensor_sonar`
	Approximate Battery	`sensor_approx_battery`
	Rotary Encoder	`sensor_encoder`

Table 3.1: List of all the available sensors that can be simulated and attached to any simulated robot model.

When evaluating field robotic platforms designed to be useful for more than just one specific agricultural application, each specific application requiring a different suite of attached sensors, being able to switch between these different sensor suites can get tedious if files have to be manually modified. In order to enable more efficient use of, and swapping between, different pre-defined sensor suite configurations, as well as robot platform base configurations, ARDEE utilizes the modularity of the URDF/*xacro* to enable the configuration of many different specific simulated robotic platform setups through the use of linking of separately defined configuration files into the main URDF/*xacro* file used for defining the specific robot model. In order to provide an example of how this could be achieved several different versions of the Terrasentia robotic platform, discussed later, were imported for Gazebo simulation.

UDP-Based Communication Mechanisms

Simulating a relatively realistic environment provides the capability of developing and evaluating the performance of field robotic algorithms whose performance is environment-dependent, which can sometimes be difficult using real field conditions, such as when the specific real field conditions are not present. Not only is it important to evaluate a field robotic algorithm's performance in its intended field environment, but it is also important for it to be tested, and executed, using its intended target hardware, or Hardware-In-the-Loop (HIL). It is important to support HIL testing, because it enables the field robotic algorithm, or technique, to be integrated with its target hardware platform, which is important for evaluating the hardware-constrained performance. The integration of the hardware components for a robotic system enables one to determine if their developed field robotic algorithm can be implemented, feasibly, on a real robotic system for real field applications. Unfortunately, there are many problems that can occur during the integration of hardware which may require debugging and troubleshooting processes that waste precious development and field testing time. In addition to hardware integration, supporting HIL development and testing among potentially many different robot platforms can be difficult to support due to specific device capabilities.

In order to support HIL development and evaluation for, potentially, different field robotic algorithms and platforms, the development environment presented in this work incorporates UDP-based communication mechanisms which enable the interaction of the Gazebo simulation environment (i.e simulated robot-world interactions, simulated sensor measurements, and simulated robot controls) with the physical hardware components across, potentially, many different robotic platforms, whose hardware capabilities may not support the use of ROS. In order to support HIL development and testing across different robotic platforms, and collaborating groups, the developed communication mechanisms enable client/server interactions between the simulation environment and the target hardware systems, using a developed standardized UDP packet data structure. This combination of client/server-based interactions and standardized UDP packet data structures enables the de-coupled development and evaluation of many different field robotic algorithms, executed on their target hardware systems, an example of this can be seen in **Figure 3.2**.

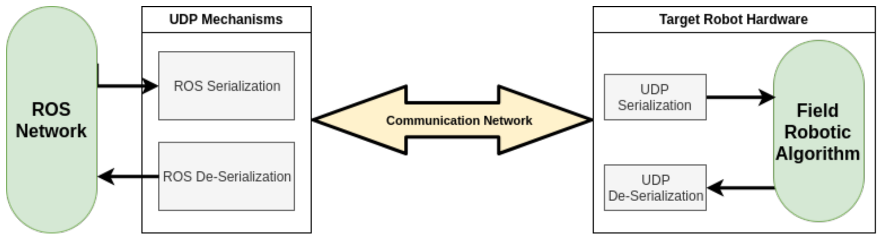


Figure 3.2: Shows the interaction between the developed ARDEE UDP-based communication mechanisms and a de-coupled field robotic algorithm that is being executed on the target hardware of a robotic platform.

As long as they are known to both the client and the server systems, the UDP packet data structures enable the following two scenarios: (1) the various Gazebo-simulated sensor measurements generated from the interaction with a Gazebo-simulated field environment can be broadcasted to, received by, and used for evaluating, the field robotic algorithm that is being investigated, and executed, on its target robotic platform, and (2) the algorithm-generated commands from the robotic platform can be broadcasted to, received by, and use for controlling the simulated robot, thus driving the simulated robot-field interactions.

By allowing the Gazebo simulated robot and sensor measurements to be used for evaluating any algorithm of interest as it is being executed on the target hardware of its intended robotic platform, the development environment can be used for partial, or entire, system hardware integration, and will ultimately enable quicker deployment of newly developed field robotic methods onto target robotic platforms for real field testing.

3.1.2 Applications

ARDEE enables one of the two following approaches which can be used for field-ready development, deployment, and evaluation of any field robotic algorithm: (1) Simulation-based development, and (2) Simulation Hybrid development. By supporting the use of these two approaches, ARDEE enables the following benefits to be realized by field robotic algorithm developers:

- Field robotic algorithms can be developed when necessary field conditions, as well as essential hardware components (i.e sensors, actuators, etc.), are not present.
- The time spent on integrating hardware, troubleshooting algorithmic logic, and debugging miscellaneous aspects during the early stages of a newly developed field robotic platform, and/or algorithm, can be used more efficiently.
- The limited time available for real field testing and evaluation can be used more efficiently by enabling the development of a field-ready robotic system throughout the entire year.

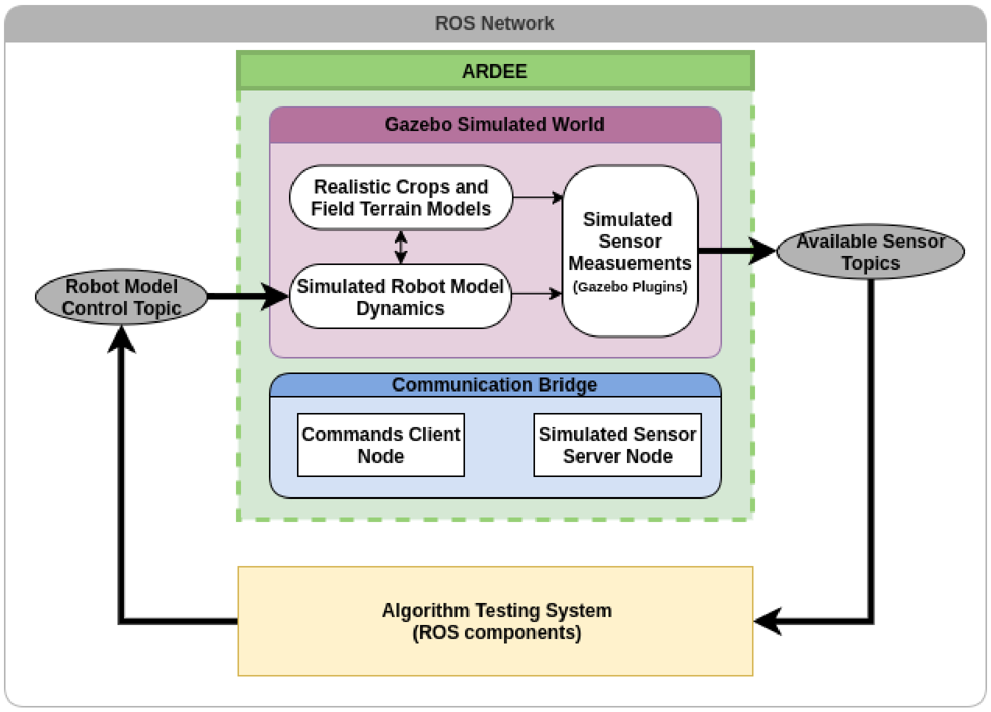

ROS Network
ARDEE
Gazebo Simulated World
Realistic Crops and Field Terrain Models
Simulated Robot Model Dynamics
Simulated Sensor Measurements (Gazebo Plugins)
Robot Model Control Topic
Available Sensor Topics
Communication Bridge
Commands Client Node
Simulated Sensor Server Node
Algorithm Testing System (ROS components)

Figure 3.3: Shows how ARDEE can be used for simulation-based development and evaluation of a robotic algorithm and/or methods implemented using ROS components (i.e nodes, topics, etc.).

Simulation-Independent is the approach that only uses the simulation environment, and does not consider the integration of hardware, and the interaction between the various elements can be seen in Figure 3.3. In this approach, ARDEE can be used to generate and simulate the specific field environment and robotic platform described for the hypothetical scenario, and for evaluating the field robotic algorithm, mentioned, it could be developed, in a variety of ways, as a ROS node that subscribes to the topics of all of the available simulated sensors, and publishes the algorithm-generated robot controls to the specified ROS topic that is used to control the simulated robot. This ROS node could be executed on the computer dedicated to running the Gazebo simulation, or it could be executed from computational hardware, separate from the Gazebo-dedicated computer, that can support ROS and is connected to the same ROS network that is running the Gazebo simulation. By enabling this approach, ARDEE allows the hypothetical field robotic algorithm to be extracted from its publication and developed in software such that its feasibility for operation in various different field environments and field conditions, can be evaluated, regardless of whether the target field conditions and/or the target hardware is present and available.

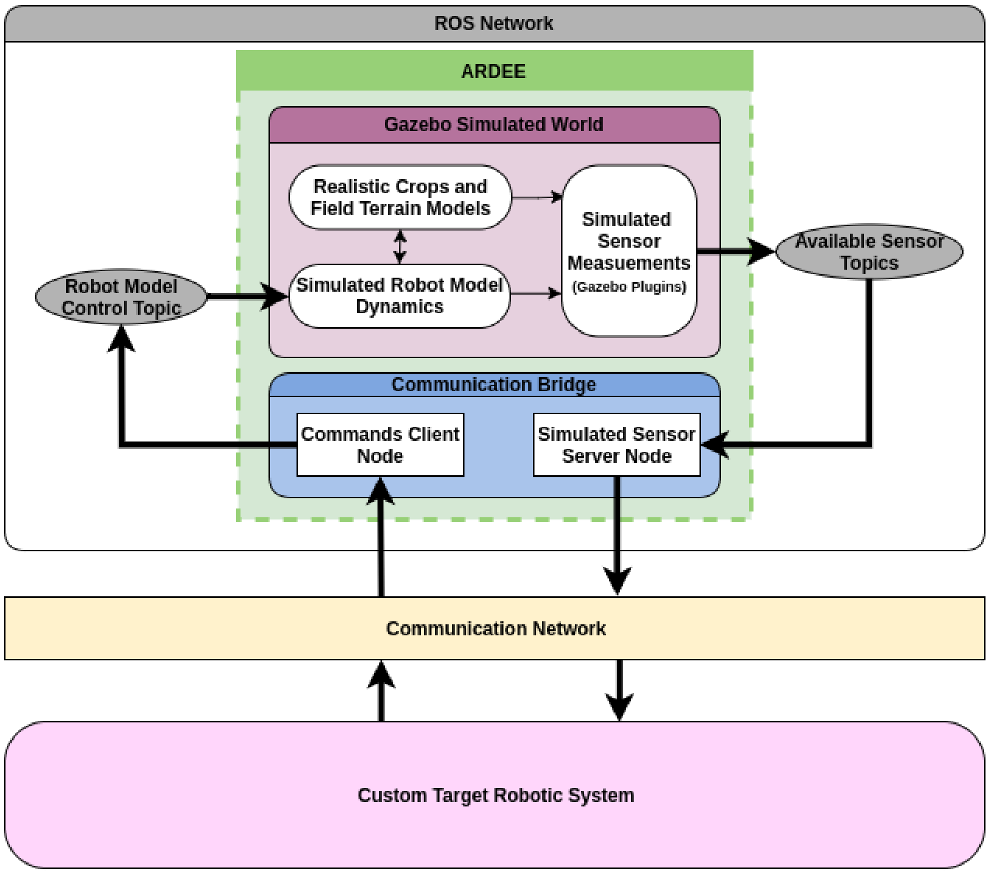

ROS Network
ARDEE
Gazebo Simulated World
Realistic Crops and Field Terrain Models
Simulated Robot Model Dynamics
Simulated Sensor Measuements (Gazebo Plugins)
Robot Model Control Topic
Available Sensor Topics
Communication Bridge
Commands Client Node
Simulated Sensor Server Node
Communication Network
Custom Target Robotic System

Figure 3.4: Shows the interaction with ARDEE and a custom robotic platform (running a robotic algorithm) for HIL development and evaluation.

Simulation Hybrid, or HIL, is the approach that integrates the target hardware components of either the partial, or the entire, robotic system, and uses the simulation environment as a surrogate for the real physical robot-field interaction dynamics, and the interaction between the various elements can be seen in **Figure 3.4**. In this approach, ARDEE is used to generate and simulate the specific field environment and robotic platform described for the hypothetical scenario. Additionally, the ARDEE communication mechanisms allow the hypothetical autonomous field robot navigation algorithm to developed, deployed, and evaluated for real-time operation on the target hardware system, which does not support ROS, by allowing it to interact with and utilize the information available from the ROS-based simulated sensors and robot platform that is simulated in Gazebo.

Using the following process, ARDEE can be used to evaluate the real-time behavior of the hypothetical navigation algorithm, executing on a separate hardware system, as it interacts with a ARDEE-simulated field of corn: (1) all the available simulated sensors, attached to the simulated robotic platform, and their measurements are serialized into UDP packets, which are then sent out to the network from ARDEE and made available for use by the target robotic system using the **Simulated Sensors Server** (discussed in **Section 4.5.2**), (2) the target robotic system, listening to the simulated sensors server, chooses which simulated sensors measurement UDP packets to receive, de-serialize, and store for later algorithm use, with a user-developed simulated sensor client using the documented and supported standardized UDP packet data structures, (3) the stored sensor measurements are then processed and used as inputs for the next computational step of the implemented algorithm, as if they were received from the on-board physical sensors, (4) the algorithm generates new data and controls that can be saved to file, or sent out to for controlling the robot's motion, (5) the algorithm-generated motion commands are then sent to a user-developed controls server used to serialize the algorithm-generated commands into a standardized UDP data packet and broadcasted back to the simulation computer to control the simulated robot's motion, and can optionally be used by the on-board motion controllers to control the on-board actuators of the physical robot platform, (6) all available newly generated algorithm-outputted controls are received by the **Controls Client** (discussed in **Section 4.5.3**) operating on the simulation computer, which de-serializes the desired UDP packets received, converts the de-serialized command data into a corresponding ROS message, and then publishes the converted algorithm-generated commands to the ROS topic corresponding to the simulated robot model's motion controller, (7) the simulated robot's behavior is updated in the Gazebo simulation environment, using either the algorithm-generated controls or manually-generated controls (i.e joystick, etc.), and new simulated sensor measurements of the surrounding Gazebo-simulated field environment are generated. The aforementioned process is repeated for the entire length of time specified by the user.

3.2 Robotic Platform

A high-throughput phenotyping field robotic platform currently being used in real field experiments was used for evaluating the usefulness of the presented development environment for developing and evaluating AFR methods and algorithms intended for operation with a physical robotic platform and its associated hardware components. The mobile robot platform used, is a low-cost, ultra light and compact (0.35 m wide, 6.6 kg), 3D-printed agricultural field robot, called Terrasentia [35], shown in **Figure 3.5**. It is currently being used for automating the process of collecting field data necessary to improve the identification of key plant traits for efficient phenotyping.

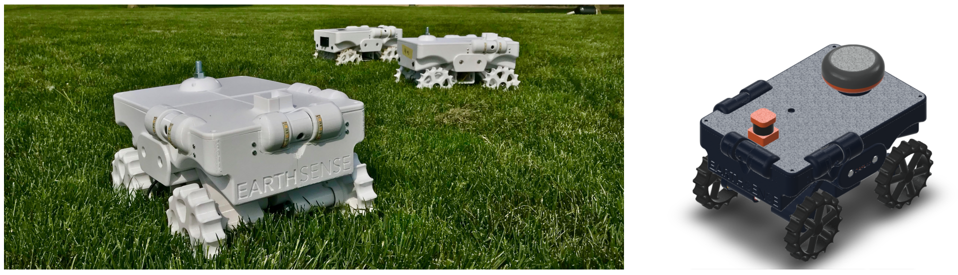


Figure 3.5: (1) Shows the physical Terrasentia robot platform, and (2) shows the isometric view of the Terrasentia robot platform as modeled with a 3D CAD program.

The Terrasentia robot is equipped with four 12V DC motors, each having a 71.165:1 planetary gearbox (69 kgf-cm, 20A max stall torque and current respectively) allowing robot movement over rough terrain, and a quadrature encoder (3,416 pulses/rev) providing speed measurements and incremental positioning. Two dual-channel Roboclaw motor controllers are used for reliable speed control of the four DC motors independently, and they are used to drive the Terrasentia robot's motion using a skid-steering drivetrain. Each Roboclaw motor controller is able to supply each motor channel with 30A of continuous current, as well as controlling each motor's speed using a PID controller that can be automatically tuned.

An RTK GPS unit (Septrino Altus NR3) is mounted on top of the Terrasentia robot giving the robot centimeter-level accurate global positions at a rate up to 20Hz. A 9 Degree of Freedom (DoF) Razor IMU, capable of outputting raw accelerometer, gyroscope, and magnetometer readings at 100Hz, is mounted directly in the center of the Terrasentia robot facilitating in the estimation of robot pose and motion. A Hokuyo UST-10LX Lidar is equipped providing a 2D scan, with 40mm accurate distance measurements with a max range of 30m, at 40Hz, which can be used for both perception and navigation of the surrounding environment.

The Terrasentia robot is also equipped with three camera gimbal units mounted on the front, left, and right sides giving the Terrasentia robot three different stabilized camera views, which can be used for visual navigation, crop phenotyping, environment perception, and more. Each camera gimbal unit has the following installed: (1) a 2 megapixel RGB USB camera able to feed color images at a rate of 120FPS (640x480 pixels), or 30 FPS (1920x1080 pixels), (2) a 6DoF IMU (BNO-055) providing absolute camera orientation feedback, and (3) a brushless gimbal motor used to stabilize the camera angle given IMU feedback.

The main control unit of the Terrasentia robot is a Raspberry Pi 3 B+ (RPi3), having a 1.4GHz 64-bit quad-core processor, dual-band wireless LAN, Bluetooth 4.2/BLE, Ethernet, and four USB 2.0 ports. The RPi3 is used for interfacing with all of the onboard hardware (i.e actuators, sensors, etc.), in addition to running all of the autonomous navigation algorithms. In addition to the RPi3, the Terrasentia robot is equipped with an Intel NUC mini-computer, with an i7 CPU and 8Gb RAM, which is used for camera streaming, field data collection, and optionally running any machine vision algorithms.

The Terrasentia robot has a Linksys E1200 router installed. The router is not only used for allowing communication between the NUC and RPi3, but it is also used as a wireless access point (AP) enabling the use of an in-house developed, Android app for manually controlling the Terrasentia robot, as well as controlling and configuring many other aspects of the Terrasentia robot (i.e controlling camera angles, recording data, etc.). **Figure 3.6** shows all of the hardware components and their interaction with each other.

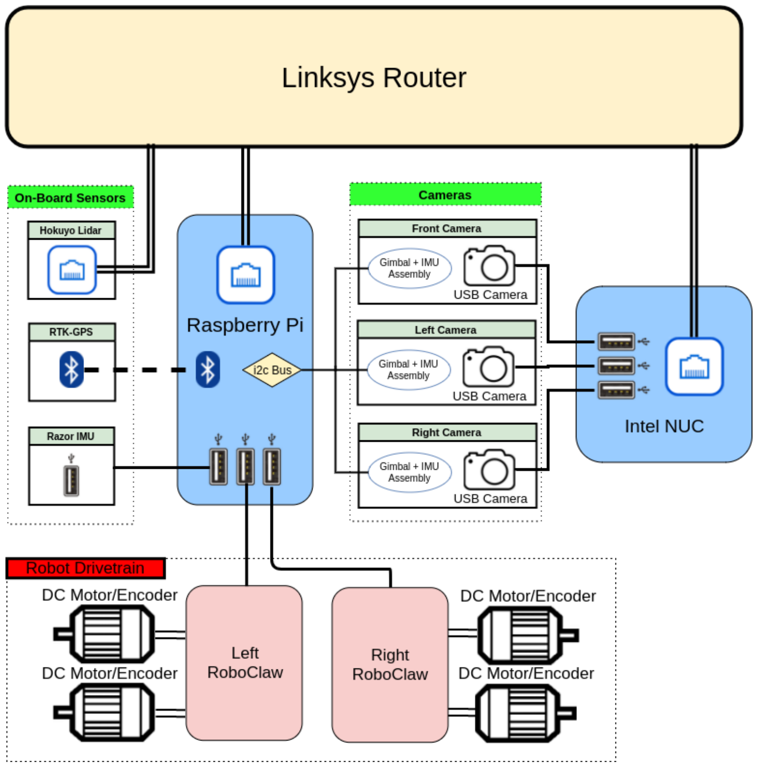


Figure 3.6: Shows a diagram of all of the hardware components on-board a Terrasentia robot platform which allow for autonomous operation within agricultural environments.

Chapter 4

Development of Simulation Environment

4.1 Introduction

The following chapter discusses the creation of the various components that were used by the novel development environment presented in this work, ARDEE. Specifically, this chapter will discuss the following topics:

Section 4.2 discusses the development of the various elements used to simulate field environments (i.e. crop models, plot generation utilities, terrain, etc.), **Section 4.3** discusses the development and utilization of the various simulated sensors in the available collection of sensor models, **Section 4.4** discusses the development of the simulated Terrasentia robot platform for Gazebo simulation, **Section 4.5** discusses the development of the UDP-based communication methods, and finally, **Section 4.6** discusses the user-friendly utilities developed for easy configuration and loading of any custom simulated field environment (i.e. robot platform and field environment).

4.2 Agricultural Field Simulation

When developing field robotic algorithms intended to be deployed and evaluated for real agricultural field environments, being able to simulate the different field conditions (i.e. crops and terrain) with sufficient accuracy for various agricultural fields can be an important asset, especially when the necessary physical field conditions are not currently present. For example, the corn grown in some areas of Illinois is limited to being available for approximately ∼8 months out of every year, which can cause a considerable setback to the effective development of any field robotic algorithm intended for operation in corn fields. By enabling various different field conditions to be simulated relatively realistically, field robotic methods can be developed more efficiently all year long, as opposed to only a portion of the year.

This section discusses the development of the different elements used for enabling various different sufficiently accurate, or realistic, field environments to be simulated, for a wide range of possible field robotic research. More specifically, this section discusses the following topics: (1) the development of the Gazebo simulatable models for different species, and variations, of crops potentially useful for a wide range of possible agricultural research topics, (2) the simulated field generation script used to generate a more realistic user-defined crop field model, and (3) the two different simulated terrain models that are available and how they were developed.

4.2.1 Realistic Crop Simulation

The creation of high-quality 3D crop models from scratch can be a painstakingly tedious and long process and can require 3D graphics modeling expertise, which many field robotic developers don't want to spend the time doing. As a result, high-quality 3D crop models are not a widely available resource.

The approach used in this thesis was to utilize already created 3D graphical models, obtainable from websites such as Thingiverse[61], GrabCAD[23], and Turbosquid[62], and adapt these models for simulation in Gazebo, however, the usefulness of this approach is entirely dependent upon the existence of realistic crop models being available.

In order for ARDEE to be a useful field robotic development tool for, potentially, many different areas of agricultural and biological research the following three crop species, which appeal to a majority of agricultural field robotics research, were made into Gazebo-simulatable crop models and made available for usage: (1) 22 different variations of corn, or Zea mays, (2) 9 variations of sorghum, or Sorghum bicolor, and (3) 9 variations of tobacco, or Nicotiana tabacum. The following section discusses the process used for enabling the collection of available crops models, shown in **Figure 3.1**, to be simulated in Gazebo, thus facilitating in agricultural field robotic algorithm development and evaluation.

3D Mesh Generation

When creating Gazebo models, the following components are required, in order to develop a high-quality 3D simulation of a crop: (1) 3D surface mesh file, or a file containing the 3D construction of the simulated object's surface, (2) mesh textures, or images of the object's visual textures, and (3) any other visual features (important lighting, shadowing, etc.). All of these components can be created, modified, and extracted using any 3D computer graphics modeling software, such as Blender[8] which was used in this work.

High-quality 3D models for corn, sorghum, and tobacco, had already been designed by the xFrog team[30], in addition to many other agricultural and biological foliage and plant models. The 3D models designed by the xFrog team were not free, however, the 3D meshes purchased were of high-quality and were designed with an acceptable degree of accuracy. Additionally, the 3D models provided included different crop model variations, as well as 3D surface mesh file formats. As a result, the amount of time required to create all of the Gazebo-simulated crop models, shown in **Figure 3.1**, was greatly reduced. All of these Gazebo-simulated models were derived from three different xFrog plant libraries (i.e. corn, sorghum, and tobacco), each containing nine different model variants, which are capable of representing various stages of the crops' life cycle (i.e. early, mid, and late season). Most of these model variants contained individual crop stalks, however, some consisted of a grouping of crop stalks.

Gazebo Model Preparation

Some 3D graphics model preparation was necessary (using Blender), in order to properly simulate all of the reported crop models in Gazebo.

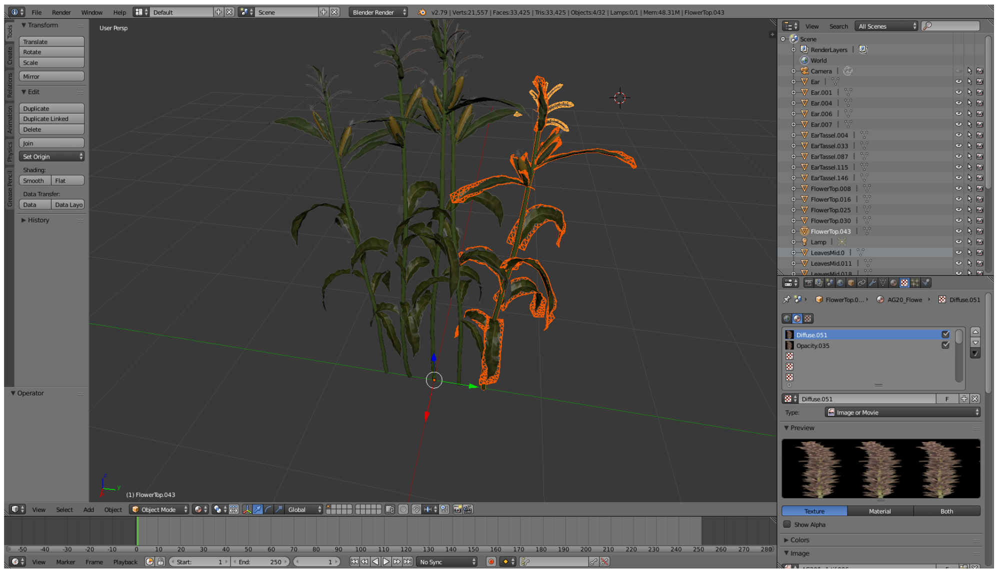


Figure 4.1: Shows one of the xFrog corn meshes consisting of multiple individual corn stalks in Blender. Blender was used to separate each individual crop stalks and extract them as their own separate crop model meshes in order to increase the number of crop models variants.

First, it was necessary to essentially "crop" out, from the xFrog crop meshes containing multiple crop model meshes, individual crop stalks into their own separate meshes, an example is shown in **Figure 4.1**, in order to increase the total number of individual simulated crop stalk variations available. After individual crop stalk mesh extraction, the resulting number of available crop model meshes to be usable in Gazebo were the following: corn (22 variants), sorghum (9 variants), and tobacco (9 variants).

Next, it was necessary to export the components from each individually extracted 3D crop model, in order to be properly visualized and simulated in Gazebo. The following specific components needed to be exported: (1) a Gazebo supported 3D geometry mesh file, and (2) all the images necessary to recreate a graphics model's visual textures. Exporting the necessary texture images, used to visualize the various color mappings and features, for a 3D graphic model is fairly straightforward because Gazebo can support many types of image formats. In order for a model's geometrical bodies to be properly rendered and visualized, a 3D graphic model's surface geometries need to be exported, and represented, into one of the following Gazebo-supported file formats: (1) the Stereolithography format (STL)[54], as a *.stl* file, (2) a Wavefront geometry definition format (OBJ)[52], as a *.obj* file, or (3) the COLLAborative Design Activity (COLLADA) format[1], as a *.dae* file. The COLLADA file format was chosen as the format to represent the crop models in Gazebo, due to its ability to capture the visual, as well as 3D surface geometry, attributes of a graphic model. Blender was used to export the *.dae* mesh file and the texture images for each of the graphical crop models extracted from the original xFrog crop models.

After all of the necessary components have been extracted for every individual crop model, the final step required for simulation in Gazebo was the creation of the GMD, discussed in **Section 2.3.3**, for all of the extracted crop models. The following process was executed for each crop model, in order to create and configure the GMD for visualization and simulation in Gazebo:

(Step i) Create a uniquely named empty directory, consisting of the same structure and subdirectories as a GMD.

(Step ii) Place all of the Blender exported components (i.e. 3D surface mesh file and texture images) into the GMD for their respective crop models.

(Step iii) Create an OGRE material script which was necessary to ensure proper visual rendering in Gazebo of the simulated crop models, discussed in more detail later.

(Step iv) Create the *model.sdf* file used to defined the Gazebo simulated model and its associated body elements (i.e. visual, inertial, collision).

Gazebo Model Modifications

OGRE is used for rendering any simulated, in Gazebo, object's visualizable components. Specifically, OGRE defines a "material" as the property which determines a surface mesh's visual properties to be rendered, such as colors, textures, and lighting properties, and is defined using a "material script". Although Gazebo automatically handles the OGRE interfacing for most of the primary OGRE material property configurations (i.e. colors and textures), "material scripts" offer users the ability to define complex materials not easily defined with Gazebo native features.

It should be noted that the OGRE material script mentioned in **(Step iii)**, is not normally needed when creating any normal simulated, specifically volumetric body types, object model, such as a cube or cylinder, using Gazebo native features. This is because an OGRE material, by default, is only visible from one side (i.e. the inside of the model is not visible). This can be problematic for flat-surfaces, or in the case of simulated crop models the leaves, flowers, and fruits. To visualize these flat surfaces, it was found that it was necessary to disable this default OGRE property, by enabling the "both-side visibility" property[45]. This can be done for each crop model by creating an OGRE material script, similar to the one shown in **Listing A.14**, which is referenced, and linked, to each crop model, via the *model.sdf* file similar to the one shown in **Listing A.24**.

It can be seen in **Figure 4.2** that the use of the OGRE script greatly improves the crop model's visual quality, and thus the simulated camera RGB data.

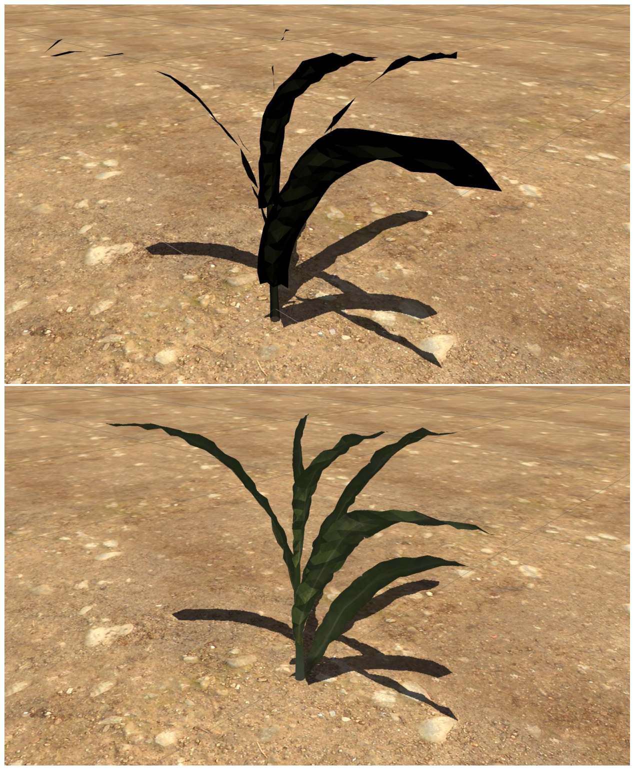


Figure 4.2: Shows the effects before and after effects, top and bottom respectively, of using an OGRE material script to fix the "both-side visibility" issue for an early season corn crop model.

In addition to the OGRE material scripts that had to be developed for proper visual rendering, it was necessary to determine each crop model's collisional body. Ideally, a simulated crop model should be modeled with soft-body dynamics, or as deformable objects, such that it would be possible to accurately simulate the behaviors of a real crop, such as swaying of the leaves and branches caused by strong winds. At the time of writing, feasibly modeling soft-body dynamics for simulated objects in Gazebo is not easily possible. This is largely a result from the limited availability of physics-based simulation software that can support soft-body dynamics. Additionally, simulating soft-body dynamics adds an additional level of complexity that increases the computational demands required for simulation.

Therefore, it is necessary to determine the rigid-body geometric approximation for the `<collision>` body, for each of the extracted crop models, to ensure that external objects properly interact with simulated crop models. This `<collision>`, or collisional, body acts as a "hard-stop" which is like the collision with a solid wall, and the geometry of this collisional body is used to determine when other simulated objects come into contact with each. The most rigid structural member for common agricultural crops is the main central stalk, and for most row-based crops (i.e. corn, sorghum, tobacco) this can be roughly approximated as a cylinder. Therefore, the geometric approximation used for the `<collision>` bodies of the all of the Gazebo crop models was a cylinder that was approximately the same width and height for each of the respective crop models.

Not only does this help keep the simulation's computational demands at reasonable levels, but this also allows for more accurate simulation of dynamical interactions between the simulated crops and external objects essential for developing field robotic methods (i.e. obstacle avoidance, autonomous navigation, etc.). For example, assume the mesh file containing 3D surface geometry was used, instead of a simple cylinder, for a crop model's collisional body. This makes not just the crop's main stalk, but also the crop's leaves, act as a "hard-stop", which results in collisional behavior between a simulated robot model and the crop model that is not realistic. This results in simulated robots colliding with crop leaves, like they were colliding with a solid wall, thus preventing the simulated robot model from passing through rows of simulated crops. Additionally, this does not accurately simulate the interaction between a real robot and crop, where the robot is able to move through rows of crops by "pushing aside" crop leaves.

The method used for determining the width and height measurements for the crop models' collisional body consisted of visually matching the collision body displayed in Gazebo to that of the perceived visual body of the crop's main stalk, shown in **Figure 4.3**. The accuracy of the collisional body for the various crop models', although rough approximates currently, could be later modified to reflect these dimensions more accurately.

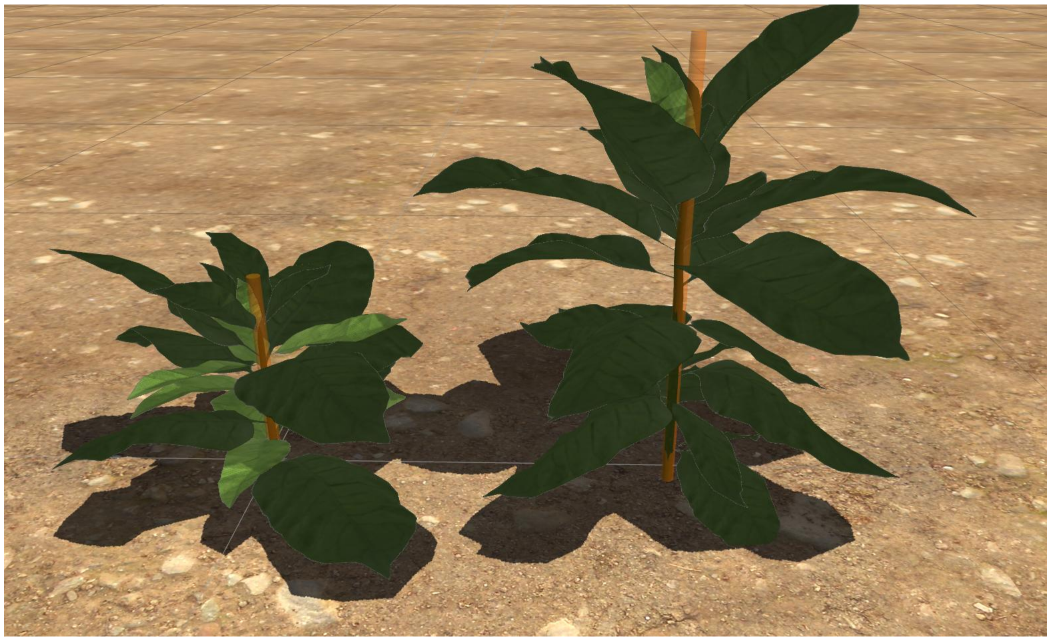


Figure 4.3: Roughly shows how the dimensions of the various crop models' `<collision>` bodies was derived. The parameters of the collision bodies visualized in Gazebo, depicted as the transparent orange cylinders, were modified until the cylinders were roughly the size of the crop's center stalk.

4.2.2 User-Friendly Field Model Generation

When using a simulation for developing field robotic methods in agriculture, simulating individual realistic crop models, rather than using simple geometry (i.e. cube, cylinder, etc.), within any field environment is important for getting a better approximation of the true performance for any developed algorithm, especially those whose performance depends on a plant-by-plant basis (i.e. robotic harvesting, weed maintenance, etc.). In addition to individual crops, simulating realistic field models, which is defined by all of the individual uniquely placed crops models in an entire field, is also important when evaluating the performance and behaviors of developed field robotic algorithms, especially those whose performance depend on the many different crop models in the surrounding environment (i.e. obstacle avoidance, autonomous navigation, etc.).

The Gazebo simulated crops models, discussed in the previous section, enable sufficiently realistic simulation for a collection of visually diverse crops models. In addition to realistic individual crops models, realistic field models can be procedurally generated, for any user-specific needs, by manually placing all of the crop models within a simulated field model. Even though it enables user's to custom-generate any simulated realistic field model they desire, this procedural generation approach requires each individual crop model to be manually placed and arranged, for all of the crops in an entire field, which can be a time-consuming task that does not generalize well for large fields containing a vast amount of crop models.

Therefore, a custom field generation Python script[48], shown in **Figure 4.4**, was developed in order to speed up, and automate, the process of procedurally generating any custom field model and its associated files for simulation in Gazebo. The script was designed to accomplish two main objectives: realistic simulation of actual field environments, and user-friendly usage and customization.

```
Input                  : $pool_{CM}$ Gazebo crop model selection pool
Input (Optional)       : $centers_{user}$ user-custom array of ideal mean stalk center locations for entire
                         field
Arguments (Optional):  $n_{rows}$ rows per plot,
                         $N_{plots}$ number of plots in X/Y axes,
                         $R_{noise}$ Gaussian noise covariance diagonals,
                         $l_{row}$ length of plots (m),
                         $\rho_{emerge}$ min/max stalk emergence probability,
                         $h_{stalk}$ min/max mean stalk height (m),
                         $d_{stalks}$ spacing b/w rows and stalks in rows(m),
                         $d_{plots}$ spacing between plots in X/Y axes(m),
                         $o_{field}$ GPS origin of field (latitude/longitude)
Result                 : Script-generated *.urdf.xacro file used by Gazebo to simulate the user-specified,
                         randomized, crop field model.

/* Generate mean stalk coordinates if not already provided by user */
if $centers_{user}$ not provided then
    $X_{field} = []$
    $O_{plots}$ = GenerateMeanPlotOrigins($N_{plots}$, $d_{plots}$, $l_{row}$, $o_{field}$)
    for $o_i$ in $O_{plots}$ do
        $X_{plot}$ = GeneratePlotIdealMeanStalkCenters($n_{rows}$, $d_{stalks}$, $l_{row}$,
          $N_{plots}$, $o_i$)
        $X_{field}$.append($X_{plot}$)
    end
else
    $X_{field} = centers_{user}$
end
/* Generate the relatively realistic simulated field model XML string */
for $x_{stalk}$ in $X_{field}$ do
    $\epsilon_i$ = GenerateStalkEmergenceThreshold($\rho_{emerge}$)
    if $randn() > \epsilon_i$ then
        $\hat{x}_{stalk}$ = RandomizeIdealMeanStalkCenter($x_{stalk}$, $R_{noise}$)
        $pose_i$ = GenerateRandomStalkPose()
        $h_{biomass}$ = RandomizedBiomassHeight($h_{stalk}$)
        $str_{model}$ = GrabRandomGazeboModelSample($pool_{CM}$)
        $str_{tmp}$ = GenerateModelXmlString($\hat{x}_{stalk}$, $pose_i$, $h_{biomass}$, $str_{model}$)
        $str_{xml} = str_{xml} + str_{tmp}$
    else
        continue
    end
    SaveFieldXmlFile($str_{xml}$) /* Generate field model file */

end
```

Figure 4.4: Shows the workflow of, and the logic used in, the developed Python script which enables custom generation of a relatively realistic simulated crop model for any user's specific needs.

Simulating Realistic Crop Fields

When designing the field generation script, the primary objective was for the generated custom field model to be capable of more realistically simulating physical field environments, and in particular the collection of all the crops within any field and their specific arrangement. It would be possible to simulate a field of crops (e.g. corn) using a collection of perfectly placed cylinders, however, this would not be sufficient for simulating all of the noise and variability inherent in agricultural environments, such as the one shown in **Figure 4.5**.

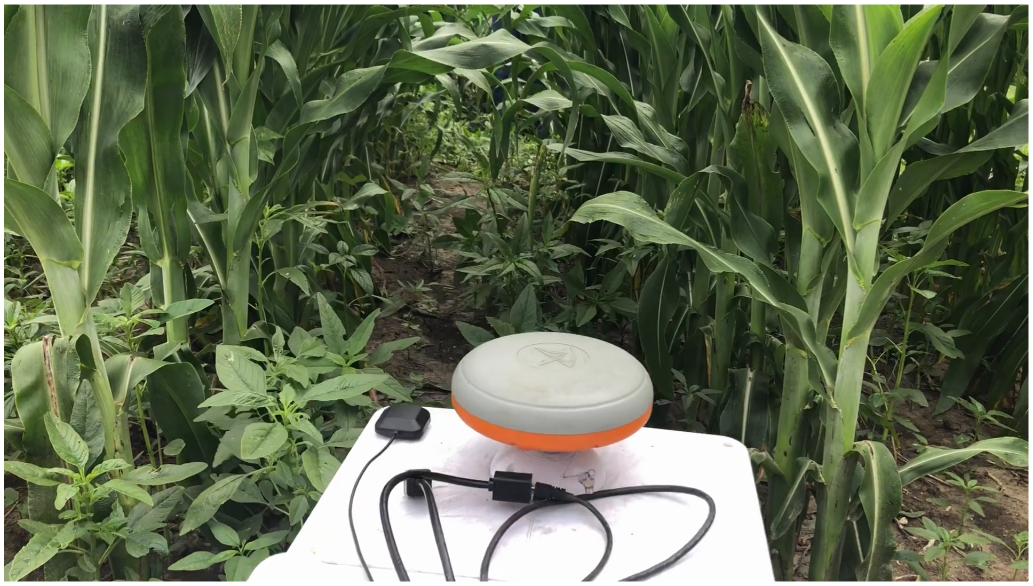


Figure 4.5: Shows the Terrasentia robot platform about to enter a row of corn in an example agricultural field that is not properly maintained for weeds.

Therefore, in order to create more realistic simulated field models the developed script attempts to mimic the noise and variability in natural growth of crops in agricultural field environments, by incorporating the following four different sources of environmental variations into the simulated field model: **(1)** probabilistic crop model emergence, **(2)** additive Gaussian noise on the individual crop stalks' mean geometric center, **(3)** random sampling from the pool of available Gazebo crop models, and finally **(4)** randomization of each selected Gazebo crop models pose (i.e. angular orientation).

Modern Precision Agriculture (PA) technologies that are readily available to farmers can be used for enabling more precise planting of crops, and allow farmers to formulate possible planting strategies for, or the exact placement for all the crop seeds in, an entire field. Similarly, the developed script uses a planting strategy, defined as an array of Cartesian coordinates that describe the center locations for all of the crops planted within any given field, is used to programmatically-generate a realistic simulated field model. The field generation script allows users to define their own custom planting strategies, either programmatically or as input arguments to the developed script. In the case where a user-custom planting strategy is not defined, the developed script uses a simple default strategy where crops are planted in plots consisting of a user-defined number of straight rows. It is assumed that for this default strategy, the mean crop center locations can be derived using a combination of different geometric field measurements, such as spacing between rows of crops, spacing between individual stalks within a row, and the length of the rows, all of which can be easily configured for any simulated field model users may desire, via input arguments passed at script runtime.

State-of-the-art PA technologies allow for these planting strategies to be accomplished with sufficient accuracy, however, due to environmental factors, such as wind, rain, and animals, the seeds that are planted, and eventually emerge, rarely ever grow in perfectly aligned rows in agricultural fields. For example, often times planted seeds may, occasionally, grow slightly offset from the intended planted location. As a result, the developed script implements the use of additive Gaussian noise in an attempt to more accurately simulate, and represent, this source of variation in the center locations of all the crops in a field, like the one shown in **Figure 4.6**. More specifically, a zero-mean multi-variate Gaussian noise distribution, with user-definable covariances, is used to generate a random "offset" Cartesian coordinate for each crop model. The randomly sampled "offset" coordinate is added to its corresponding crop's mean location, defined by the planting strategy used, to replicate the noisy placement for all of the crops planted in real fields.

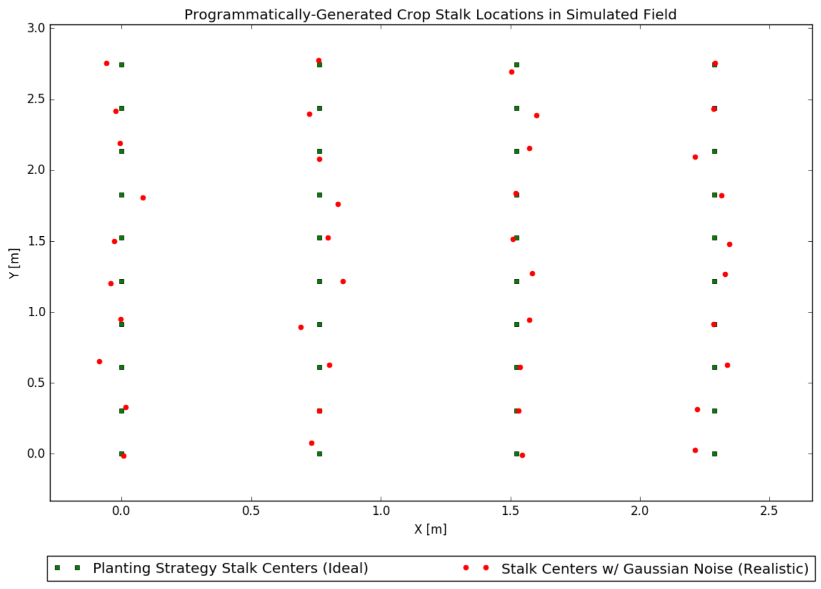
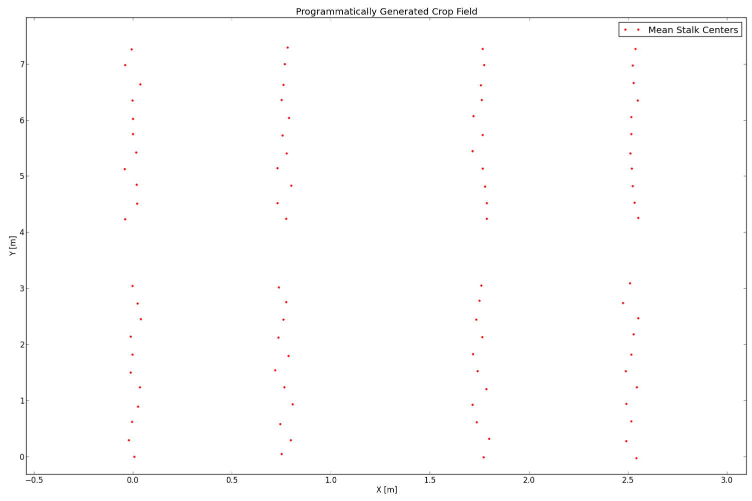


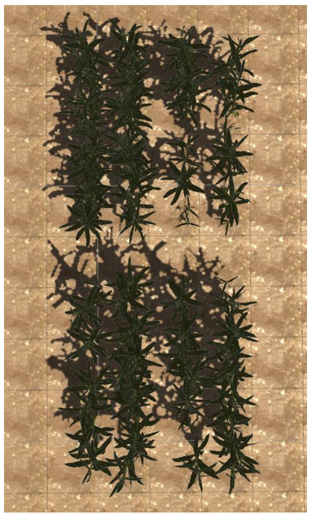
Figure 4.6: Shows the difference between the perfectly placed crop stalk centers generated using the default planting strategy (Green) and the more realistic crop stalk centers generated by the script using additive Gaussian noise (Red).

In addition to imperfect crop locations, often times in agriculture some of the seeds that are planted may never grow due to the various different environment and soil conditions, such as insufficient soil nutrients or extreme weather conditions. The successful growth, or emergence, of the planted crops, is largely dependent on the probability that a crop receives adequate growth conditions. In order to simulate this probabilistic emergence, a random number is generated for each simulated crop model that is used to determine whether the simulated crop model emerges, thus being used in the simulation. The successful emergence of a particular simulated crop model is determined by comparing the randomly generated number, associated with it, against a probabilistic threshold defined for successful emergence.

Lastly, all crops experience growth differently which results in every crop grown in agriculture being a unique specimen. For example, the growth in stalks of corn often results in the key features of two different corn stalks being far from similar, such as placement of leaves, stem width, and corn husk size. In respect to agricultural field robotics, these differences, in the different crop features, directly impact the performance of robotic control and perception algorithms which rely on extrinsic sensory inputs, such as vision, Lidar, and even GPS when the upper layer of crop leaves (or crop canopy) becomes dense and reduces satellite signal accuracy. As a result, when a simulation of an agricultural field environment is used to evaluate developed field robotic algorithms, it is important that the virtual field environment used is capable of representing this variation among different crops sufficiently so that the algorithm's resulting performance is a better approximate to how it would perform in real field environments. Representing this variation among crops can be achieved using the last two elements mentioned, random Gazebo crop model sampling and Gazebo crop model orientation randomization. By incorporating these two elements, it is possible to represent the variation among crops in the simulated field sufficiently (i.e. less uniform) using the limited available selection of within the collection of simulated crop models.



(a) Figure 4.7 (continued)



(b) Figure 4.7 (continued)

Figure 4.7: (a) Shows a figure of the noisy crop stalk locations generated using the generation script. (b) Shows the corresponding resulting simulated field model automatically generated using the Python script discussed.

By combining any defined planting strategy and all of these user-configurable sources of variation, the developed field generation script is able to generate a custom-defined simulated field model, like the one shown in **Figure 4.7**, capable of realistically replicating the inherent noise and variability found in common agricultural field environments.

User-Friendly Features

When developing a simulation environment for facilitating with field robotic development within the academic community, the generation of realistic simulated field models is essential to allow proper evaluation of an algorithm's performance. In order for a simulation environment, intended to assist in field robotic development within the academic community, to fully realize its potential, it should be relatively easy for its intended users to use and apply its functionality for their own specific needs.

Therefore, when developing the custom field generation script, the second objective was that is should be user-friendly, and be relatively easy for users to generate simulated field models for their specific needs. This objective was chosen to cultivate collaboration, as well as the use of ARDEE, within the academic community, by allowing custom field models to be quickly created and easily modified for any user's specific needs. As a result, the developed field generation script incorporates the following two design features: it can easily be used among users with different programming skills, and it supports quick and simple configuration of the parameters used for custom simulated field model generation.

The first user-friendly feature incorporated is that the field generation script was developed using Python, which enables quick application and usage of the developed script's functionalities, as well as allows users to more easily configure parameters, and adapt script logic, to fit their own purposes, regardless of programming experience. In addition to being developed in Python, the developed script automatically generates the files necessary for simulating a user-defined field model in Gazebo, which makes it easier for users to create their own customized field models, regardless of their experience using ROS/Gazebo (i.e. ROS/Gazebo knowledge is not required).

The second feature incorporated to promote the user-friendly application of the developed script is that it enables any simulated field model to be quickly configured and modified for custom field simulation in Gazebo. This can be achieved through the use of the script's supported command-line arguments, that allow users to modify the parameters that the script uses for generating the custom field model, at runtime (i.e. manual modification of script code not required). Not only can these command-line arguments be used for configuring the parameters used for the default planting strategy (crop centers array) generation, but they also enable to specify the various probabilistic parameters used for generating the randomized sources of environmental variations, discussed previously.

4.2.3 Simulated Terrain Generation

Not only is simulating realistic 3D crop models important, but it is equally important to be capable of simulating the terrain and soil characteristic for any given user-specific field environment. This is an important capability, because the terrain and soil of a field can, not only influence the dynamics resulting from the robot-field interaction, but it also influences the measurements received from the robot platform's onboard sensors.

Any given field's terrain can roughly be characterized using the following three properties: (1) the heightmap, or the elevation of the ground, which defines a field's bumpiness/flatness, (2) the textures, or the visual details, which defines how the terrain is visualized, and (3) the dynamic parameters, such as coefficient of friction, coefficient of restitution, etc., which define the dynamic interactions between the terrain and external objects.

Soil can be roughly characterized using the following two properties: (1) the soil compactness, or how tightly packed the particles in the soil are packed, can be used to define how easily the soil can be displaced, which also defines the dynamics for soil-wheel interactions, and (2) moisture content, or how dry/muddy the soil is, can be used to define, among other things, how the particles of soil group, or clump, together, as well as how they can clump to other objects (i.e. tire tread).

Accurately modeling not just the dynamic properties of the terrain, such as the coefficient of friction in dynamic environments, but also the soil properties in constantly changing field environments is difficult and is an active area of field robotic research. Additionally, properly simulating soil (i.e. a collection of many small particles), as well as the dynamics of wheel-soil interactions, with modern simulation platforms is either impossible or is computationally inefficient, where real-time evaluation in simulation is not possible. Therefore, it was determined that accurate simulation of soil composition and variable terrain/soil dynamics and soil was out of the scope for this project.

As a result, the dynamic properties of the field terrains simulated using ARDEE was assumed to be constant and uniform throughout the terrain. Additionally, it was assumed that a single rigid-body model will be used to represent both the simulated soil and terrain in Gazebo. These were determined to be sufficient for simulating agricultural field environments, because they can easily allow the use of ARDEE for developing, and evaluating in real-time, field robotic methods to a point where it is possible for entire robotic systems and methods to operate robustly, or well enough to acquire useful data for further improvements, in their target field environments.

Since it is now modeled as a single rigid-body, sufficiently realistic soil/terrain for a simulated field can be characterized as simulated terrain models that are visually similar to real field terrains, and cause more realistic (i.e. noisy) simulated robot dynamic behaviors resulting in the generation of more realistic simulated sensor measurements.

The main objective of this section is to discuss the following two approaches used for generating custom simulated field terrains for any user-specific needs: (1) image-based, and (2) 3D mesh-based.

Image-Based Generation

The first approach that can be used to generate a custom field's terrain is an image-based approach, which combines the use of a custom grayscale image used for defining the terrain's elevation heightmap, and any custom combination of texture images which can be used to represent the terrain's visual rendering.

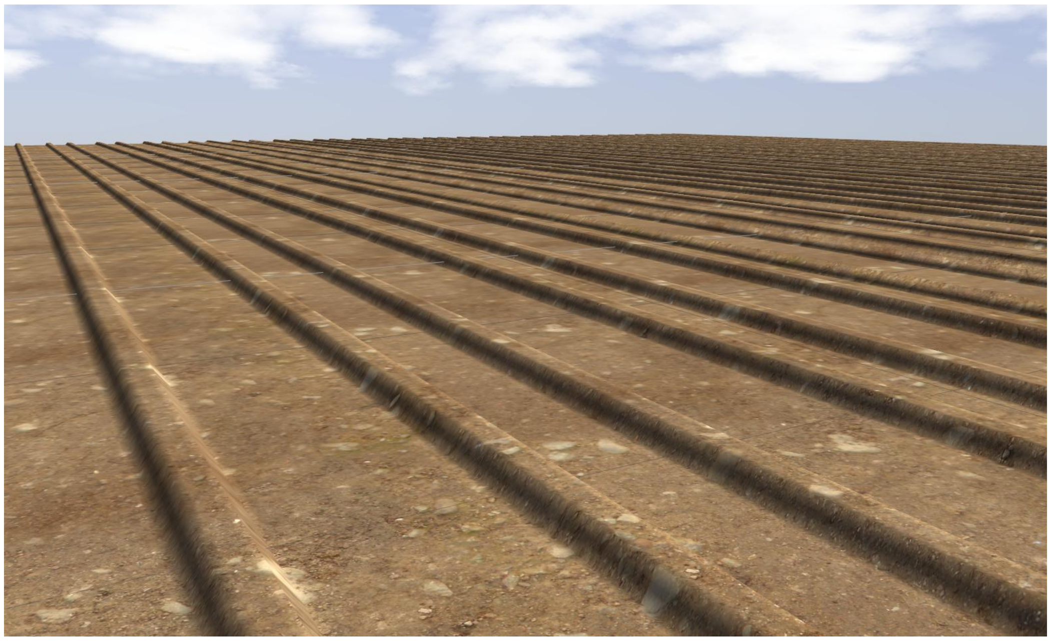


Figure 4.8: Shows an example of a programmatically generated terrain model, by using a programmatically created custom grayscale image used for the terrain's `<heightmap>` element, in combination with some standard Gazebo available `texture` images.

The custom simulated field terrain, shown in **Figure 4.8**, was generated by combining the following components into the *model.sdf* located in the specific GMD, named for this specific field terrain: (1) a programmatically generated grayscale heightmap image, shown in **Figure 4.9**, and some generic texture images, shown in **Figure 4.10**, that was already available with Gazebo.

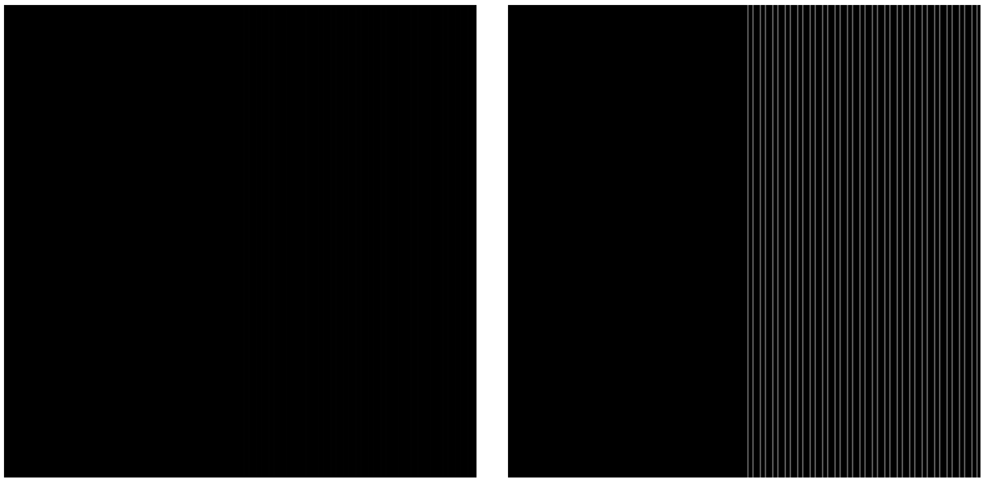


Figure 4.9: (Left) Shows the whole grayscale image generated by a custom python script. Although very difficult to see, the lighter regions, or rows, of the image correspond with pixels having values of 1, out of 255. (Right) Shows the same grayscale image, but with pixel values of 100 in order to exaggerate the generated rows. This image also emphasizes the difficulty in creating the same grayscale image manually.

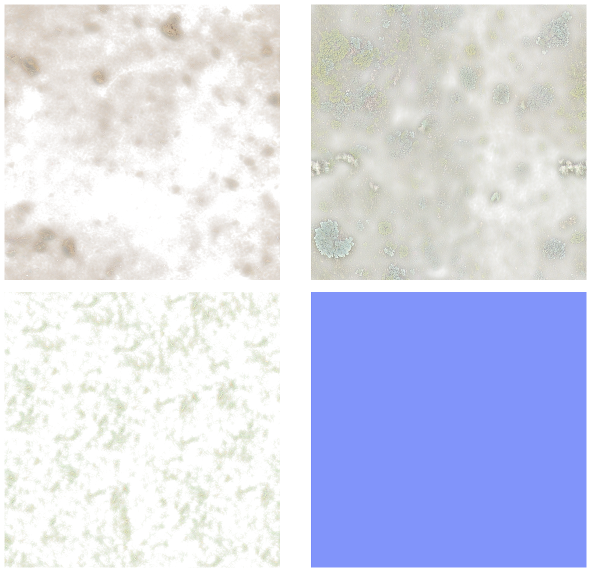


Figure 4.10: Shows all of the Gazebo standard `texture` images that were used to visualize the color of the terrain model, as seen in **Figure 4.8**.

The grayscale image, shown in **Figure 4.9**, is used to represent the terrain's heightmap, or the collision 3d surface geometry, and is specified in the `<heightmap>` SDF tag-block (Lines 26 - 55) of the *model.sdf* file, shown in **Listing A.23**. In Gazebo, the heightmap grayscale image's pixel values (0-255) are used to represent the elevation values for each of the areas in the scaled-up simulated model. Although the grayscale image shown was generated programmatically, the grayscale image used for the heightmap can be created using alternative methods such as the following: (1) heightmap grayscale image can be created manually using image editing software, such as the GNU Image Manipulation Program (GIMP) [22], and (2) available websites, such as terrain.party [60], can be used to download real-world terrain data, not just the grayscale image, for various locations in the world.

It should be noted that it can be extremely difficult to capture the fine-grained elevation details, such as the ruts from a tractor or the small clods of soil, of a field when creating a grayscale heightmap image manually. For example, the small "dirt mounds", represented by the slightly raised areas in **Figure 4.8**, would have had to be created, in GIMP, using an extremely small ($\sim$1 - 2 pixel) paint tool. While using this size of a paint tool to modify a very small area of pixels at a time is possible, it is very easy to make many small mistakes that can quickly add up to, potentially, be a very time-consuming task.

Mesh-Based Generation

The second method for generating the customized field terrain was done by using an already created 3D graphic model to represent the field's terrain. The 3D graphics model, created by [2] was found, again, using TurboSquid. In order for the graphics model to be used as Gazebo model to simulate a field's terrain, the process, previously discussed in **Section 4.2.1**, was used to create the simulated terrain model, shown in **Figure 4.11**.

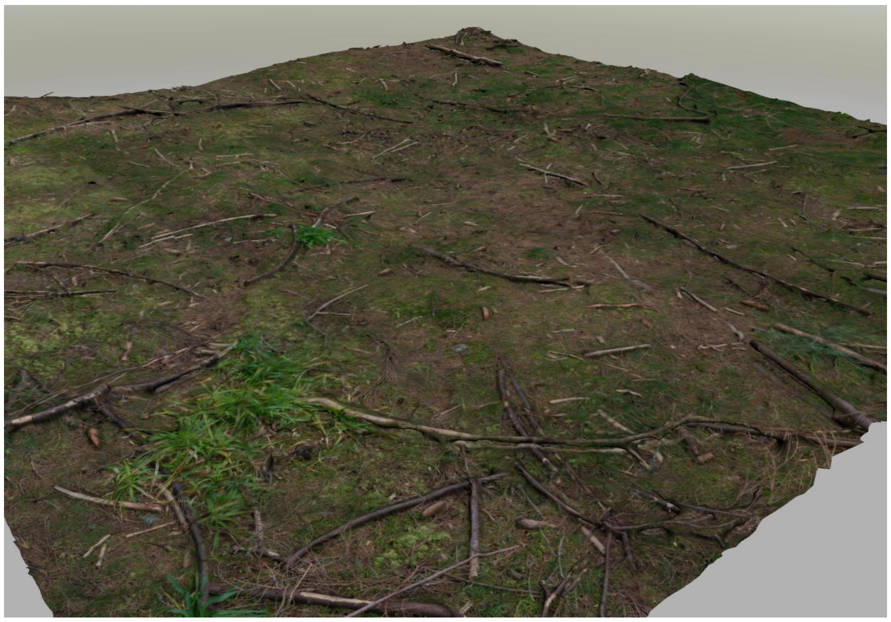


Figure 4.11: Shows an example of a terrain model that was created using the same approach discussed in **Section 4.2.1**, however the meshes used to create the surface geometry is different.

Although only two specific methods were discussed it should be noted that there may exist, potentially, better methods in which a custom field's terrain can be created. Ideally, only a few terrain heightmaps would have to be created, in combination with many different specific terrain texture images, in order to provide users a quick and easy way to simulate their specific field's terrain, since the majority of agricultural fields have similar elevation characteristics.

4.3 Simulated Sensors

Another important feature that ARDEE should be capable of, is the ability to simulate various types of sensors that have potential use for any number of agricultural applications. Not only should the various sensors be physically simulated (i.e. visual, collision, etc.), but it is equally important to be able to simulate the respective sensors' measurement behaviors. Although not as important, quick and easy modification of the properties (such as specific noise parameters, the location on the robot the sensor is attached to, and the sensor resolution) associated with the various simulated sensor models that may be attached to any simulated robotic platform is an important feature ARDEE should be capable of allowing in order to further reduce the amount of time spent on customizing simulated sensor behaviors.

The following section discusses the following: first custom Gazebo "sensor" plugins required for properly simulating sensor measurements useful for field robotic development that were not already available were developed, then the Gazebo sensor models used for simulating and attaching any of the available sensors, in the collection shown in **Table 3.1**, to any simulated robotic platforms were developed, and finally an investigation of the simulated Lidar measurement behaviors is discussed.

4.3.1 Custom Gazebo Plugins

Similar to ROS, Gazebo's open-source community allows rapid simulation and evaluation using already developed Gazebo plugins, simulated robotic models, and even simulated sensor models. Gazebo plugins allow users to easily configure and simulate most, if not all, of the necessary sensors they may wish to use for their specific purposes. The following section discusses some of the custom sensor plugins developed in order to simulate some of the specific sensors models, and behaviors, that can be useful for agricultural field robotic development.

Rotary Encoders

Rotary encoders are used for many common robotic platforms and a wide range of other applications. In agricultural field robotic specific applications, such as in [35], encoder readings are necessary for autonomous navigation throughout a field. As a result, it is should be possible to attach, and simulate, a modifiable rotary encoder. Although simulating rotary encoders is not an overly difficult task, it is surprisingly, not widely available as an already developed standalone Gazebo plugin. Therefore, it was necessary to develop a plugin which allows users to simulate a rotary encoder, and attach to any arbitrary object in Gazebo such as the rotary encoder model shown in **Figure 4.12**.

The developed plugin is able to simulate essentially any rotary encoder by easily allowing users to configure a specific simulated encoder by changing the core rotary encoder parameters for the associated simulated encoder element, an example is shown in **Listing A.5**.

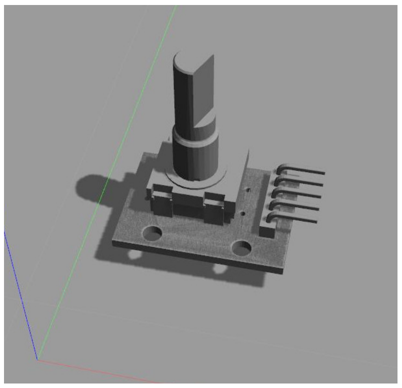


Figure 4.12: Shows the Gazebo simulated model associated with the developed rotary encoder plugin.

Using the developed encoder plugin, users now have access to the simulated encoder sensor readings associated with the attached simulated dynamic objects. The simulated encoder readings are generated by Gazebo which keeps track of all of the various joints' dynamic information related to each of the joints as they change over the course of the simulation. The developed plugin uses the attached joint's angular position and velocity information, kept track by Gazebo, in order to create the encoder readings based on the user-defined parameters and publishes the simulated encoder readings to the user-specified ROS topic for use across the ROS network.

As a matter of convenience, the skid-steering plugin, already created by [20], was slightly modified incorporating the developed simulated encoder plugin code for publishing the simulated encoder readings for each of the independent driving motors.

Variable-Accuracy GPS

The use of GPS and RTK-GPS is a very commonly used sensor, and in precision agriculture it is essential. Therefore, it is necessary that have a simulated GPS sensor that is able to accurately represent the true GPS behavior while operating in an agricultural environment. Although there is an already developed plugin to simulate a GPS sensor [38], the simulated GPS measurements that this plugin generates assumes the noise parameters are fixed, however, in real-world agricultural environments, the noise is not always constant. For example, in a real-world agricultural field, the GPS accuracy is optimal when it is in open areas, however when under heavy occlusion from late-season crop canopies the

GPS accuracy fluctuates very frequently. As a result, it is necessary to develop a method in which the GPS behavior, caused by multi-path errors and occlusions, is properly simulated.

The developed plugin was based on the already developed, and open-source, GPS plugin [38], however, it was modified such that GPS measurement accuracy could programmatically update the GPS noise parameters in order to simulate the dynamic GPS accuracy based on environmental conditions, such as multi-path errors and reduced GPS accuracy while under crop canopies.

The, potentially, many sources for GPS error can be defined by two different, configurable, XML-formatted files. The first file is used to define the different GPS noise coefficients and the range of conditions in which that specific noise is applied to the GPS measurements, an example can be seen in **Listing A.6**. Specifically for a small compact robot, like the one discussed in **Section 3.2**, which is intended to be traveling between rows of crops, and under the crop canopy, the range of crop heights were used as the conditions for each noise coefficient.

In order for the various user-defined GPS noise coefficients to be correlated to specific areas of a simulated field, the second XML-formatted file is used to represent the bounding regions in which the GPS measurements should be affected by the different GPS noise coefficients defined, an example can be seen in **Listing A.7**. In order for them to use a specific GPS noise coefficient, the defined bounding regions have an associated average crop height, which is used to determine which GPS noise coefficient is applied to it, an example can be seen in **Figure 4.13**.

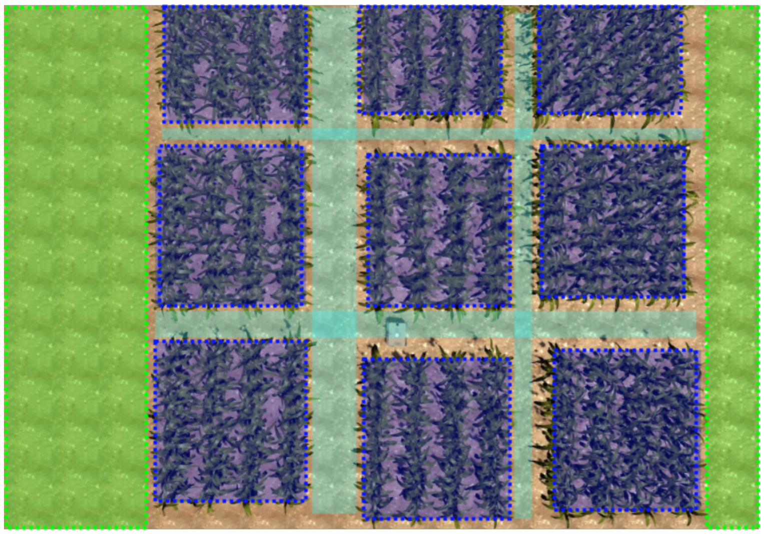


Figure 4.13: Shows an example field environment containing three primary regions used to dynamically modify the measurement accuracy of the simulated GPS.

Approximate Battery Usage

When developing health-critical robotic algorithms it is essential that the remaining battery life can be measured. Specifically being able to simulate a battery is useful for developing and evaluating any algorithms requiring battery-dependent functionality, such as a state machine which gives a robot a different directive once remaining battery life has reached a certain limit. For example, for a team of autonomous robots used for agricultural field maintenance (i.e. weeding, watering, etc.) it is important to know each robot's health so that it is possible to know when a robot should be brought back to recharge. As a result, it is necessary to incorporate a simulated battery plugin that users can use for developing any battery-dependent functionality.

It should be noted that there exists an already available and developed battery plugin [31] which provides a more in-depth usage, and more accurate simulation, of a battery depending on the user-defined components which consume the battery health. Additionally, since the plugin provides a more accurate simulation of a battery it could take some time in order for the simulated battery health to reach a certain level. In cases where a user may want to do a very quick test of some developed battery-dependent functionality, or algorithm behavior, using this plugin could be an unfavorable option.

Therefore, a user-friendly battery plugin was developed in order to provide a simulated battery plugin which allows quick evaluation of battery-dependent functionality and can easily be used by users who may have little to no experience with ROS/Gazebo. It should also be noted that if users wish to evaluate any battery-dependent functionality, or algorithm behavior, where the simulated battery is represented accurately, it is recommended that users use the plugin available in [31], which the developed plugin is modeled off of.

In order to configure, and use, the simulated battery for a given simulated robot, users only have to add the battery plugin XML block, similar to the one shown in **Listing A.8**, to the robot in which the battery is associated with.

4.3.2 Sensor Model Collection

Now that it is possible to generate the various sensor measurements, potentially useful for agricultural field robotic applications, in Gazebo, it is necessary to develop the simulated sensor models, in order to define the "physical" body of the simulated sensor and attach a Gazebo "sensor" plugin to for generating simulated sensor measurements. All of the available simulated sensor models, shown in **Table 3.1**, are packaged as individual URDF/*xacro* files (i.e. **.urdf.xacro*) containing all of the essential elements required for the specific sensor to be properly simulated in Gazebo, such as associated 3D surface mesh files, Gazebo plugins (either open-source from [38][20] or custom developed from the above sections), and more.

In addition to neatly containing all of the necessary files and definitions required for simulation, each of the specific simulated sensor models have been defined within their own individual *xacro:macro* blocks, which greatly simplifies the configuration of sensor parameters (i.e. sensor noise, etc.) and the attachment to any robotic platform, via the passing of input arguments into the *xacro:macro* call XML-block. In the case of the specific off-the-shelf sensors that have been made available, whose specific characteristics have been defined in a provided datasheet and will not change, the use of the *xacro:macro* enables sensors to be very easily attached to any given robotic platform with a simple *xacro:macro* call. For example, the Hokuyo UST-10LX Lidar can easily be attached to any given robot by using only 3 lines of XML code, shown in **Listing 4.1**, which also enable easier reading of a custom robots URDF file.

```
<xacro:sensor_hokuyo_ust10lx name="hokuyo" parent="base_footprint"
    color="black">
    <origin xyz="0.22 0 0.173" rpy="0 0 0"/>
</xacro:sensor_hokuyo_ust10lx>
```

Listing 4.1: Example URDF xacro snippet (lines 8-10) used to add a Hokuyo UST-10LX Lidar to a robot. Lines 1-6 are optional comments.

Contrary to specific off-the-shelf sensors, the generic sensor models that have been provided (i.e. IMU, GPS, etc.), are the sensors where the fundamental sensing behaviors are the same, but the specific properties can be different. For these types of sensors, the developed *xacro:macro*s give users much more control over the simulated sensor model that would be attached to any given robot model. For example, the code is shown in **Listing A.9** is an example of a generic IMU model that has been custom configured and attached to a robot.

4.3.3 Lidar Behavior Investigation

During the process of developing the various components for simulating the available sensor models, there were some important details, related to the simulated Lidar sensor data, that was discovered. These details are important to discuss, because it was noticed that the simulated Lidar sensor had different fundamental behaviors, depending on the type of plugin used with the simulated Lidar sensor model. The natural behavior for the sensor measurements of a real Lidar is correlated with the strength of the returned laser light which is reflected off of the various surfaces within the surrounding environment. Normally the surfaces of the environment are the ones which are seen visually. As a result, in order to correctly generate the measurements of a simulated Lidar, in certain cases, it matters which plugin is used. Therefore, this section discusses the discovered behaviors and their impact on the generated measurements for a simulated Lidar sensor.

The particular behaviors noticed were discovered as a result of using the following two Gazebo model plugins, created and available [20], both of which are used to simulate 2D Lidar sensor measurements: (1) the `gazebo_ros_laser`, or the collision-based, plugin, and (2) the `gazebo_ros_gpu_laser`, or the visual-based, plugin. The only fundamental difference between these two plugins is in the methods used to represent, and generate simulated sensor readings from, the surrounding environment. For example, the collision-based plugin interacts with the collisional bodies, defined by the `<collision>` element block, whereas the visual-based plugin interacts with the visual bodies, defined by the `<visual>` element block, of the various objects in the surrounding environment.

In cases where the simulated objects in the surrounding environment have relatively simple geometries and the `<collision>` and `<visual>` element blocks both reference the same geometric bodies, there is no difference between the simulated readings returned from both plugins, however, this can not be said for cases where the `<collision>` and `<visual>` element blocks do not reference the same geometric bodies. For example, due to the complex surface geometry, the developed simulated crop models, discussed in **Section 4.2.1**, use the complex geometric body for the `<visual>` element block, and for the `<collision>` element block a simple cylindrical body is used, for reasons discussed in **Section 4.2.1**. In this case, using the collision-based plugin will produce Lidar measurements which do not accurately simulate how a real Lidar sensor would sense the surrounding environment, as seen in **Figure 4.14**. As a result, for these cases, the correct plugin to use is the visual-based plugin, which generates an accurate representation of the sensed objects in the surrounding environment, as seen in **Figure 4.14**.

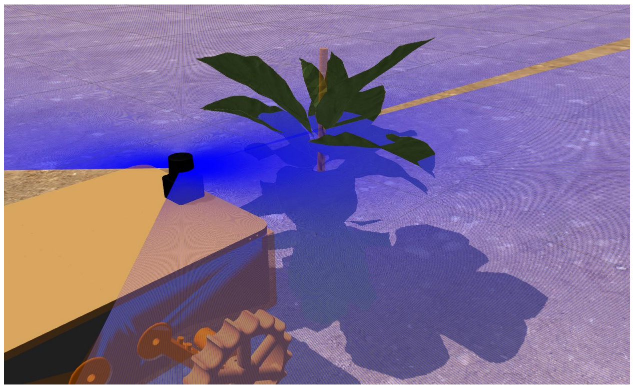


Figure 4.14 (continued)

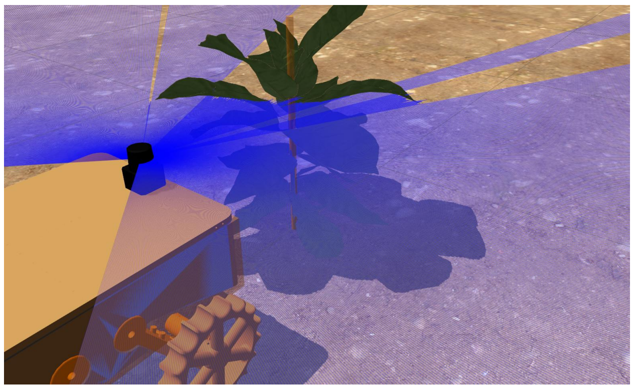


Figure 4.14 (continued)

Figure 4.14: Shows the simulated Lidar readings of a simulated tobacco crop model for the following: (Top) CPU-based simulated Lidar measurements are generated based on the `<collision>` bodies defined for the simulated crop models, represented by the transparent orange cylinder, and (Bottom) GPU-based simulated Lidar measurements are generated based on the `<visual>` bodies defined for a simulated crop model, and results in a more accurate reflection of what an actual Lidar scan would return.

In addition to the methods used to represent and interact with the surrounding environment, the computational resources (i.e. CPU, RAM, GPU) that each plugin uses was discovered to be different. For example, it was noticed that the collision-based plugin was CPU expensive, whereas the visual-based plugin was GPU expensive and used hardly any CPU.

4.4 Simulated Robot Model

The Terrasentia robotic platform, discussed previously, is a novel robotic platform that has been developed in-house and is currently being used for a variety of different agricultural and biological field robotic applications. In addition to being used to evaluate the effectiveness of ARDEE, three different versions of the Terrasentia robotic platform, shown in **Figure 4.15**, were imported for Gazebo simulation to show the following: **(1)** the process that can be used to enable other field robotic researchers' custom robotic platforms to be evaluated in Gazebo, and **(2)** the benefit of the robot configuration approach developed which utilizes the modularity of the separate URDF/*xacro* configuration files to enable different custom robot platform bases (i.e. core body, or no sensors) and custom configured arrangement of simulated sensors can be easily and quickly swapped to enable any custom robot model configuration to be simulated in Gazebo for any user-specific needs, with little effort.

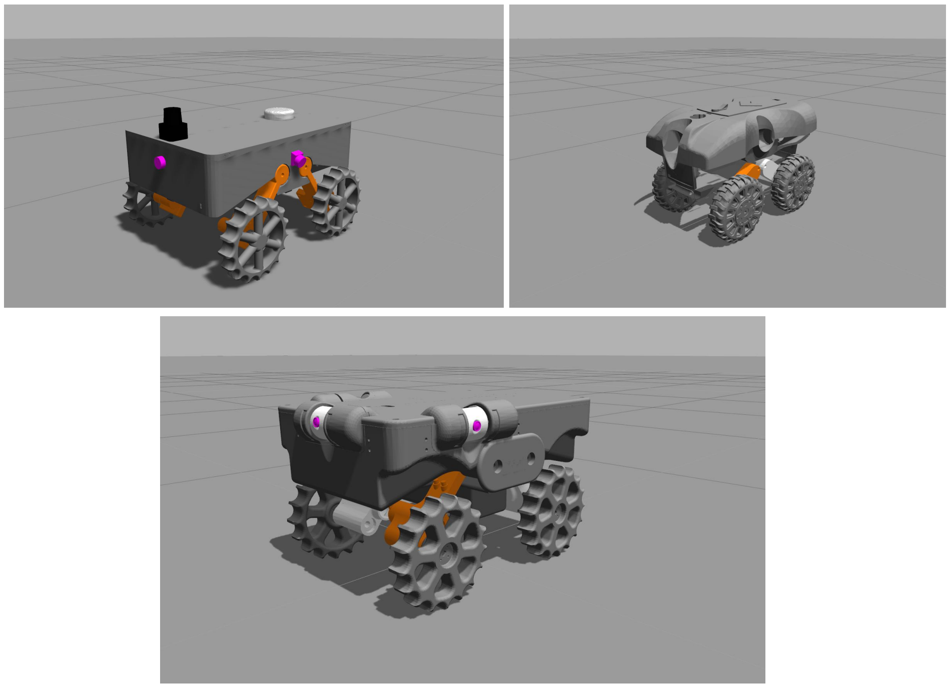


Figure 4.15: Shows the various Terrasentia platforms from different stages of development in Gazebo: (Top Left) first generation design, (Top Right) next-generation, or 2019, design, and (Bottom Center) currently used version, or 2018 design.

4.4.1 Robot Model Configuration

Since there are various research groups and individuals who are using the same robot platform, each using their own specific combination of sensors necessary for their own research needs, the URDF file format is the more suitable file format to be used to create and represent the simulated robot model. Since using the URDF/*xacro* format allows representing a simulated robot model as a collection of *xacro* macros, or modular sections of URDF elements (potentially defined across multiple different files), it is possible to modularize the simulated robot model. There can be many different ways to construct a single model, however, in this thesis, the simulated robot model is represented, and created, using a combination of elements defined by the following three primary file categories, which are defined by separately named URDF/*xacro* files:

Type i **Robotic Component Files** - is the category of files used to contain the individual components that can be used to construct any custom simulated robot model (i.e. links, joints, Gazebo plugins, etc.).

Type ii **Configuration Files** - is the category of files used for containing all of the URDF/*xacro* parameters/components (i.e. dynamic coefficients, attached sensors, etc.) that can be used for specific simulated robot model customization, or enabling a specific arrangement of simulated sensors required for a particular application (such as crop phenotyping) to be quickly and easily equipped by many different custom simulated robot models.

Type iii **Robot Assembly Files** - is the category of files used to assemble all of the components (defined in **Type i** files) and specific configuration files (defined in **Type ii**) required to create a specific custom simulated robot model.

Configuration files have been separated into the following two categories, both of which should be located in `terrasentia_description/config`: (1) **Sensors** is the category of configuration files, denoted by the `*_sensors_config` file naming convention, that contains all the simulated sensor model links that are to be attached to a given object, as well as all of their respective configurations, and (2) **Robot Base** is the category of configuration files, denoted by the `*_config.urdf.xacro` file naming convention used to define all of the parameters that could be used modify any aspect associated with the core body, or base, of a simulated robot model (i.e. 3D surface meshes used for simulated bodies, model plugin parameters, dynamic coefficients, etc.). An example of a sensors and a robot base configuration file categories can be seen in **Listing A.12** and **Listing A.11**, respectively. By separating configuration files in this way, all of the specifically configured sensors defined in a sensor configuration file (which may be modified frequently) can easily be used across many different robotic platforms, while limiting the modification of a specific simulated robot model's base configuration.

4.4.2 Extracting Custom Robot Components

The first step that is required in order to be able to simulate any custom robotic platform (in this case the Terrasentia platform) in Gazebo, the target robot's simulated model (i.e. links, joints, etc.) has to be created and defined, in addition to the generation of each individual component's 3D surface mesh into a Gazebo-supported file format. In other words, all of the individual body elements that make up the entire physical robot's body have to be created, defined, and attached together using the URDF/*xacro* format. This can be a tedious task if done from scratch, without the help of a Computer-Aided Design (CAD) program, such as Autodesk Fusion360[3] or Solidworks[18]. For the Terrasentia robot platform, the various available versions, shown in **Figure 4.15**, have already developed using a CAD software, either Autodesk Fusion360 or Solidworks. This makes it easy to extract each individual component's 3D surface mesh (as an STL file), however, extracting the information required for the various URDF/*xacro* elements (i.e. inertial/visual/collision body information, 3D coordinate transforms of joints, etc.) associated with each individual component is still a tedious process.

Solidworks has a very useful add-on[59], or plugin, that is able to automatically export either an individual Solidworks part, or an assembly, into a comprehensive URDF model directory containing all the necessary files and elements (i.e. all necessary URDF elements, 3D geometry mesh files, 3D coordinate frame poses for links and joints, etc.). The resulting directory, and subdirectories, are automatically structured and named such that the generated URDF is ready for simulation in Gazebo, in addition to `roslaunch` files used to start the Gazebo simulation with the generated URDF model in it.

This Solidworks plugin can greatly expedite the process of importing any custom robotic platform design into Gazebo enabling quick development and testing with ROS, however, it does not generate the URDF/*xacro* format which is desired for modular configuration and parameterized components. Therefore, each plugin-generated **.urdf* element (i.e. wheel, leg, chassis, etc.) was extracted and re-formatted into its own parameterized *xacro:macro* enabling easy reuse of previously developed individual components for potential future designs, an example can be seen in **Listing A.10**. In order to easily identify and use them later, the individually extracted components for the different Terrasentia platform versions (i.e. TSv1, TSv2, and TSv3) were separated into the following three **Type i** files (located in the `terraentia_description/urdf` directory), each representing the respective Terrasentia models: **(1)** `links_v1.urdf.xacro`, **(2)** `links_v2.urdf.xacro`, and **(3)** `links_v3.urdf.xacro`.

4.4.3 Torsional Spring Joint Plugin

Now that the robot model can be simulated in Gazebo using the URDF/*xacro* format, it is necessary to verify that the simulated robot model, not only looks but also, behaves dynamically the same as the real robot would, such as the legs and wheels moving similarly.

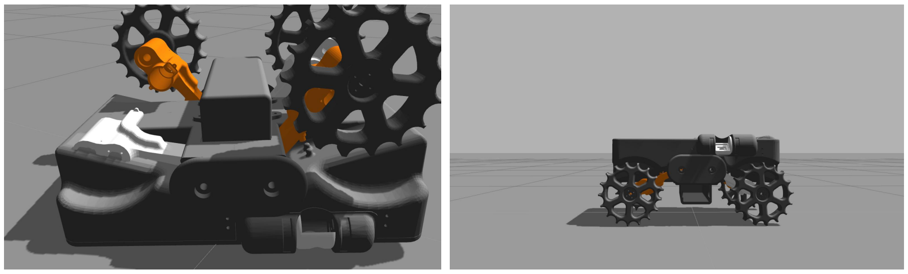


Figure 4.16: (Left) Shows how the simulated robot legs fall inside of the robot chassis due to the spring force not being properly simulated, as well as the defined angular limits not being enforced, which is clearly incorrect. (Right) Shows how the simulated robot legs do not produce any restoring spring force necessary for supporting the weight of the simulated robot model, which is incorrect spring behavior.

While evaluating the robot model's behavior in the simulation environment, it was found that some aspects of the physical robot's leg joints, or the joints connecting the robot's movable legs to the robot chassis, were not properly simulated. The first behavioral aspect found to be incorrect was that the angle limits, although defined, for the leg joints were not properly being simulated, or enforced, causing the legs to move into the robot's chassis, shown in **Figure 4.16**, which is a physically impossible scenario. The second behavioral aspect found was that the spring force, from the robot's suspension system, used to push the legs to their natural home location (fully extended) was not being simulated at all, as seen in **Figure 4.16**. It was found that both of these problems were caused by the same issue, which was that the URDF properties relating to the joint limits and the joint dynamics were not being properly translated by Gazebo which resulted in Gazebo not simulating those aspects. In order for Gazebo to simulate these behaviors, it is necessary to either represent the robot model using the SDF format or create a plugin that tells Gazebo how to simulate the joint behavior. In order to properly simulate the leg joint dynamics, while keeping the configurability and modularity inherent of the URDF/*xacro* format, a simple Gazebo "model plugin" was created to ensure that the spring force used to push out against the leg is generated while enforcing the angular limits set.

In order to simulate the spring force created by the linear springs in the leg suspension system of the physical robot, the spring force can be modeled as a torsional harmonic oscillator, and is calculated using the following equation:

$$F_{suspension} = -b\dot{\theta} - k(\theta - \theta_{ref}) \tag{4.1}$$

where b, k, $\dot{\theta}$, θ, and θ_{ref} represent the damping coefficient, spring stiffness, joint angular velocity, current joint angle, and the reference joint angle, respectively. In addition to the configurable coefficients for spring stiffness and damping, the plugin allows the users to define the angular limits that need to be enforced. Since all of the leg joints should behave in the same fashion, only one Gazebo "model" plugin, which allows the user to configure the parameters related to the specific joint the plugin is attached to, was created in C++, and is shown in Appendix **Listing A.26** and **Listing A.27**. In order to be able to use any custom Gazebo plugin with any Gazebo simulated object, it has to first be successfully compiled and built from source (i.e. CMake/Make, ROS's `catkin_make`, or similar tools).

Once built, the custom plugin can then be easily attached and configured for any number of simulated joints by adding the code snippet, shown in **Listing A.4**, to an *xacro* macro block for attaching to a simulated leg link which should have a spring suspension element. Since it is parameterized, the suspension plugin parameters can easily be modified by linking and passing a specified robot base configuration file to the desired "robot model assembly" file.

4.4.4 Controlling the Simulated Robot

A simulated robot model is worthless if it can not be controlled or moved when given target commands. Typically in mobile-robot navigational control algorithms, the generated target commands, used to control the general motion of the robot's body with respect to the world, are the target linear and angular velocities of the robot's body. The specific behaviors of the robot's individual actuators necessary to achieve these target commands can vary greatly depending on the defined motion model (i.e. differential drive, tricycle drive, etc.) of the robot. Additionally, regardless of the robot's specific motion model, it is essential that the actuators (either physical hardware or simulated) need to be able to be controlled properly according to the target commands received. In the case of physical hardware it is a simple matter of sending the necessary signals needed to control the various hardware elements, however for the simulated robot model a Gazebo “model” plugin must be created in order to receive that target robot commands (over ROS topic) and then tell Gazebo to move all the necessary actuators (joints) according to the robot’s desired assigned motion model. The source of the generated target robot commands matters as well. For example, if the motion controller is a ROS node which outputs the target commands via a ROS topic then all that is needed is to send the commands to the same ROS topic that the simulated robot’s motion controller plugin listens to. If, however, the target robot commands are generated by a motion controller running on a computer, or machine, separate from the computer used to run the simulation environment, then the target commands need to be communicated somehow between machines and then once received, be reformatted, and published to the specific ROS topic that is used by the Gazebo plugin to move the respective simulated actuators, more discussion on this in **Section 4.5**.

The specific motion model that defines the simulated Terrasentia model’s kinematic and dynamic behavior is the skid-steering motion model. In order to make the simulated robot model move, and be controlled, with a skid-steering motion behavior, it is necessary to create a plugin in order to receive high-level robot body motion target commands and control each of the simulated robot models motors such that the desired target robot body motion is achieved. Luckily, there is already a skid-steer Gazebo plugin controller available, [20], which simply needs to be attached to the simulated robot model, in a similar fashion as shown in **Listing A.3**, in order for the simulated robot model to be controlled. This plugin was used as the simulated robot model’s low-level motion controller throughout this work.

4.4.5 Final Model Assembly

The final step necessary for simulating any custom robotic platform in Gazebo (in this case the Terrasentia platform), is the creation of the specific robotic platform’s ***robot model assembly*** file, which is used to combine all of the necessary components and configuration files, previously discussed, into a single uniquely-named file (located in the `terrasentia_description/robots` directory), as well as the only file referenced and used to spawn the custom robot into Gazebo. For example, the ***robot model assembly*** file, shown in Appendix **Listing A.13**, was used for simulating the first-generation Terrasentia robot platform in Gazebo, shown in **Figure 4.17**, by referencing the configuration files (using the ***xacro:include*** call in **Lines 15-17**) `tsv1_config.urdf.xacro` and `tsv1_sensors_config.urdf.xacro` for the robot base (shown in Appendix **Listing A.11**) and the attached sensors (shown in Appendix **Listing A.12**), respectively.

It is assumed that the core compositional elements used for constructing the robot model's base will very rarely ever have to be changed. Therefore, the ***robot model assembly*** file should only have to be created, and modified, once in order to define the individual body components used to represent the robot model's base, and how they need to be connected, shown in **Lines 23-101** of **Listing A.13**.

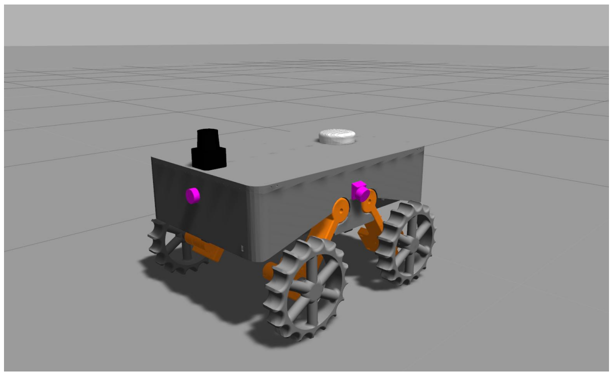


Figure 4.17: Shows the first-generation Terrasentia robot platform simulated in Gazebo using the robot model assembly file created in **Listing A.13**.

4.5 Distributed Data Communication

The primary approach, of using ARDEE, for HIL development and evaluation, of any field robotic algorithm being implemented on custom robotic hardware, is to transfer any essential information, needed to evaluate the performance of a field robotic method, between the custom robotic hardware and a de-coupled computer system used for simulating any user-specific test environment (i.e. simulated robot, sensors, and field), using ARDEE. Therefore, some communication method needs to be developed in order to enable this essential information to be easily transferred between many different devices.

One of the main benefits of using ARDEE for general agricultural field robotic development is that it enables the behavior and performance of a field robotic algorithm to be evaluated in as if it were in a real field, via the Gazebo simulated robot and field. Unfortunately, one of the drawbacks to using anything that is based on ROS is that it can only be utilized by ROS-supported hardware, which could limit the effectiveness of ARDEE for general agricultural field robotic research. As a result, some methodology for converting communicated information from system agnostic to ROS-specific data, and vice versa, should be utilized by ARDEE in order to enable HIL development and testing to a wider range of agricultural robotic systems.

This section discusses the custom methods that were developed in order to address the aforementioned issues related to distributed HIL development and evaluation of field robotic methods among many different devices. Specifically, the following topics are discussed: **(1)** the method used for communicating information between devices using ARDEE and various external devices, **(2)** the custom data server mechanism developed for communicating sensor information to ARDEE-external devices, and **(2)** the custom control client mechanism developed for receiving ARDEE control information generated from ARDEE-external devices.

4.5.1 Transporting Data

The standard transport protocol used for communicating the necessary information for developing and evaluating field robotic algorithms (i.e. sensor measurements, robot commands, etc.) between many different devices was chosen to be the User Datagram Protocol (UDP). UDP was chosen as the basis for data communication because it lacks any delays caused by data retransmission, is suitable for real-time data transmission (e.g live video streaming and sensor data broadcasting), and can be used by any network-able devices (i.e. support Ethernet or wireless communication).

When using UDP for communication between many different devices, information is contained, and communicated, in packets, known as “datagrams”, and is composed of a “header” and “data”, as shown in **Figure 4.18**.

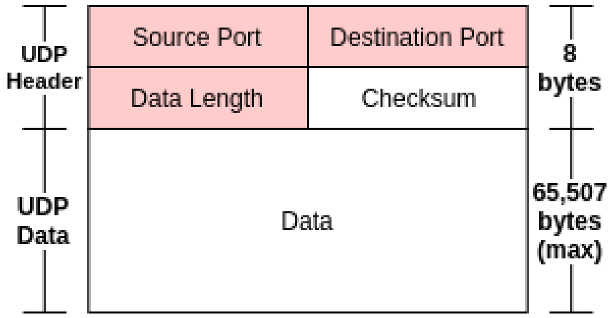


Figure 4.18: Shows the structure of a basic UDP datagram which is used to communicate information across many different devices.

The UDP header contains the following information: (1) source port number, which is the number of the sender, (2) destination port number, or the port the datagram is addressed to, (3) data length, or the length in bytes of the UDP header and any encapsulated data, and (4) checksum, which is optional and used in error checking. Additionally, it is necessary for each device to know each others port and ip address, in order for any UDP datagram to be sent from a source process (server) and successfully received by an end-user process (client).

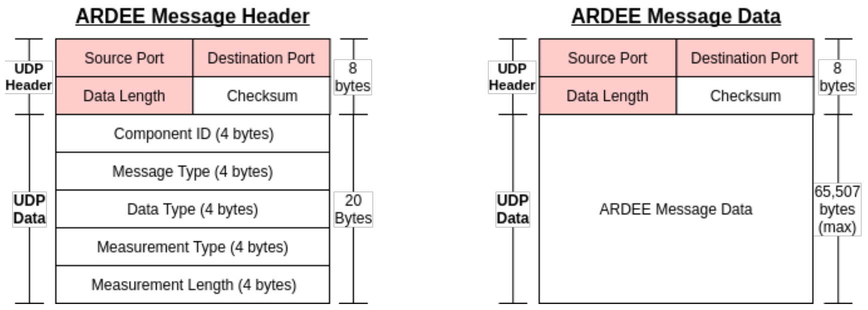


Figure 4.19: Shows the two custom UDP packets (*ARDEE Message Header* and *ARDEE Message Data*) used to compose an *ARDEE Message* packet and their respective data structures.

Although a UDP datagram by itself is well equipped to achieve many different low-latency data transmission applications among many different devices, it was necessary to develop a customized UDP packet combined structure, termed an "*ARDEE Message Packet*", in order to enable the various types of information, specific to field robotic HIL development and testing, to be communicated between a computer running a Gazebo simulation and many different devices.

An *ARDEE Message Packet* is composed of the following two UDP packets, shown in **Figure 4.19**: (1) an *ARDEE Message Header* and (2) an *ARDEE Message Data*.

The ***ARDEE Message Header*** is a UDP packet containing all of the following meta-data, defined in Appendix **Listing A.16**, associated with the proceeding *ARDEE Message Data* packet:

1. ***Component Id*** - is used to help identify which component the following data corresponds to, and is useful when dealing with multiples of the same data (i.e. two encoders).

2. ***Message Type*** - is used to identify the type of message that follows (i.e. control message, sensor data, etc.).

3. ***Data Type*** - if the following message is sensor data, then this is used to specify what kind of sensor type the data is associated with (i.e. IMU, GPS, Lidar, etc.).

4. ***Measurement Type*** - if the following message is sensor data, then this is used to specify if the following sensor data has only one measurement (i.e. one IMU reading), or multiple measurements (i.e. multiple range readings from a single Lidar scan), associated with it.

5. ***Measurement Length*** - if the following message is sensor data with multiple measurements, then this used to specify how many measurement values should be read in.

The *ARDEE Message Header* is used to help specify how the incoming *ARDEE Message Data* should be read in, as well as enabling more efficient communication of data between devices. For example, the *ARDEE Message Header* can be used to filter *ARDEE Messages* for the ones containing the data that is relevant to the user's specific needs, instead of receiving every incoming message. The ***ARDEE Message Data*** is the UDP packet containing all of the essential data associated with the *ARDEE Message* that is being sent, all of which is defined in Appendix **Listing A.17**.

By sending these two custom standardized UDP packets in succession, it is possible to communicate any essential information required for HIL development using UDP-based client/server interactions between a simulation computer and any target robotic platforms, an example is shown in **Figure 4.20**.

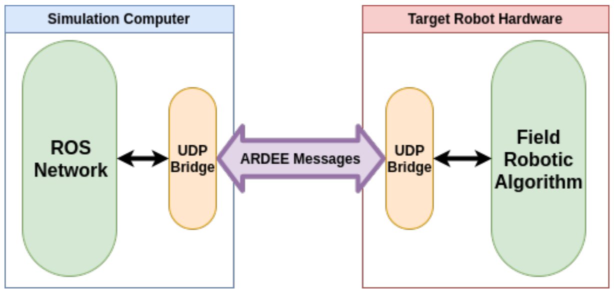


Figure 4.20: Shows a simplistic overview of the method used for communicating data between a device using ARDEE and an external robotic platform, via *ARDEE Message*'s.

4.5.2 Simulated Sensor Server

One of the benefits that ARDEE provides is the ability to simulate relatively realistic agricultural field environments, different variations of common crop species, as well as simulated sensor measurements, all of which can be useful for field robotic development and testing for agricultural and biological applications. For example, when developing a vision-based algorithm being developed for autonomous navigation underneath crop canopies, or for autonomous harvesting, ARDEE can be used to test the performance of this algorithm using simulated camera data received during interaction with a simulated target field environment, which could be useful for preventing any damage to high-value crops during early developmental stages. Unfortunately, ARDEE is based on ROS, which means that any field robotic platform, which cannot support ROS, will not be able to utilize any useful simulated sensor measurements.

As a result, ARDEE utilizes a UDP server mechanism, called the "Simulated Sensor Server" (SSS), that was developed to enable the various, ROS-specific, simulated sensor measurements to be utilized by a wider range of field robotic platforms, that may or may not be able to support ROS. In order for the SSS to accomplish this, the following components, shown in **Figure 4.21**, were used: **(1)** a custom ROS node, **(2)** a UDP Serializer, and finally **(3)** an ***ARDEE Message*** **Packet Broadcaster**, or AMPB for brevity.

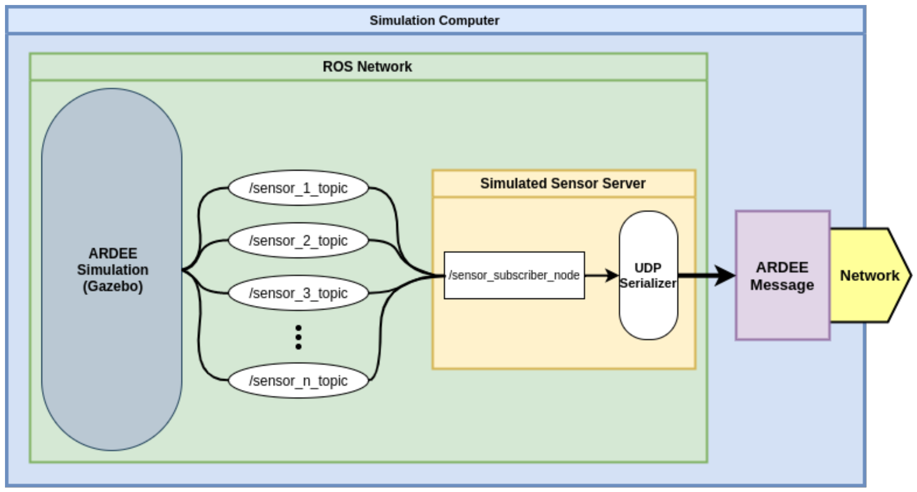


Figure 4.21: Shows the interaction between the different components of ARDEE and the developed *Simulated Sensor Server*, as well as how they interact together to broadcast the simulated sensor measurements for distributed HIL development and testing.

The **Custom ROS Node** is the component that interfaces with ROS, via ROS Subscribers, to retrieve any available simulated sensor measurements, via their respective ROS topics. The ROS node's primary purpose is for receiving and extracting ROS-specific data as soon as they become available.

As soon as a simulated sensor's measurement is available, and extracted, from its associated ROS topic, the **UDP Serializer** serializes the extracted simulated sensor's information such that it can be packed into an *ARDEE Message* to be sent out later. The UDP Serializer uses ARDEE-standardized sensor data structures, defined in Appendix **Listing A.17**, for containing various types of sensor measurements into their corresponding serialize-able data structure, which can be later packed into a UDP datagram, broadcast, and then unpacked by any capable external device. Currently, the following types of simulated sensors are supported, however, additional types of sensors can easily be supported later: **(1)** Rotary encoders, **(2)** battery voltage, **(3)** IMU (accelerometers, gyroscope, and compass), **(4)** GPS, **(5)** RGB/RGBD cameras, and **(6)** 2D/3D Lidars.

Finally, the ***AMPB*** is used to load all of the serialized data, as well as metadata (which can vary depending on the combination of sensor type and specific sensor source), associated with any extracted simulated sensor, into its respective *ARDEE Message* packet or *ARDEE Message Header* and *ARDEE Message*

Data packets. As soon as a complete *ARDEE Message* packet is prepared and ready, the *AMPB* then broadcasts, first the *ARDEE Message Header* packet then the *ARDEE Message Data* packet, out to a specified port, at a specified ip address.

All of these components are contained in a C++ class in order to allow for the immediate broadcasting of ROS-extracted data as soon as it is becomes available for minimal delay, in addition to the ability to easily configure the various ROS sensor topics to listen to and what port and ip address to broadcast to, like in the example shown in Appendix **Listing A.18**.

Since it is the intent for ARDEE to enable HIL development among, potentially, many different custom robotic platforms, it is left up to the developers of a specific robotic platform to implement their own ARDEE sensor client to receive and extract the data of interest into any format, or programming language, they desire for seamless integration and use within their specific system. In order to successfully receive any broadcast simulated sensor measurements, all that a user's specific client needs to know is the following: **(1)** the ip address that the simulated sensor measurements are being broadcast from (i.e. the simulation computer using ARDEE), **(2)** the port that the simulated sensor measurements are being broadcast to, and finally **(3)** all of the ARDEE data types, defined in Appendix **Listing A.16** and **Listing A.17**.

The benefit to this is that any custom robotic platform will be able to use any broadcast simulated sensor's measurement, once received and extracted, as if it came from its physical sensor, which is vital for allowing field robotic methods to be developed and tested for field-readiness without the need to go out into the real field (which may or may not have desired field testing conditions). Additionally, this approach makes it such that any field robotic algorithm being investigated does not need to be developed any differently for HIL development and testing, and the only thing that is different between HIL and real field development and testing is where the field robotic algorithm's input comes from.

4.5.3 Robot Controls Client

When developing any kind of field robotic control algorithm, such as low-level motion control of an individual field robot, distributed control of multi-agent teams of field maintenance robots, or autonomous field robotic harvesting, it is important that the various robotic commands generated from the developed algorithm, as executed from its target hardware, are able to control the respective elements being simulated in the Gazebo environment (i.e. robot velocity commands, multi-agent team commands, or robotic harvesting manipulator control), in order to properly evaluate real-time performance on target hardware. Additionally, this allows a proper end-to-end evaluation of algorithm performance, by allowing the simulated system dynamics to be driven, and sensor measurements accordingly, as they would have been if they were in a real field environment.

As a result, ARDEE utilizes a UDP client mechanism, called the "Controls Client" (CC), that was developed to enable any Gazebo simulated object to be controlled by a wide range of external robotic platforms, that may or may not support ROS. The CC accomplishes this with the use of the following components, which are similar to the *SSS*, shown in **Figure 4.22**: **(1)** an ***ARDEE Message*** **Packet Receiver**, or AMPR for brevity, **(2)** a UDP De-Serializer, and finally **(3)** a custom ROS node.

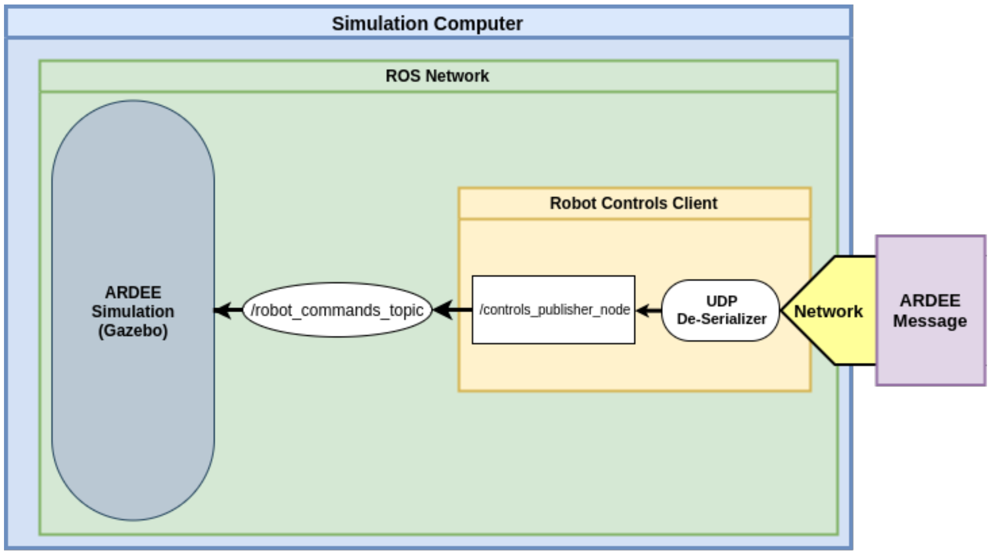


Figure 4.22: Shows the interaction between the different components of ARDEE and the developed *Controls Client*, as well as how they interact together to control a Gazebo simulated object for distributed HIL development and testing.

Before discussing the different components developed for the CC, it should be noted that, similar to SSS, it is left up to the developers of a specific robotic platform to implement their own UDP mechanism for packing any desired command outputs, generated from either a specific field robotic algorithm or any other specific methods used controlling the robotic system (i.e. joystick, fail-safes, etc.) and any other relevant data, into the necessary *ARDEE Message Header* and *ARDEE Message Data* packets, using all of the ARDEE standard data types defined in Appendix **Listing A.16** and **Listing A.17**, and broadcasting the resulting *ARDEE Message* packet out through a specified port, sending it to the ip address corresponding to that of the simulation computer running the ARDEE CC. As a result, de-coupled development and testing is possible across many different robotic systems with the use of a pre-determined standardized data structures, which prevent the need for any interacting end-devices (i.e. other robotic platforms, other simulation computers, etc.) having to know the various, potentially, custom-developed internal mechanisms of the other devices.

Similar to how the SSS uses the *AMPB*, the CC uses the ***AMPR*** to listen for any UDP packets which are being broadcast to a specified port, from a robot-specific ip address. Since the broadcast UDP packets should consist of the *ARDEE Message* structure, the CC uses the first UDP datagram (i.e. a *ARDEE Message Header*), from a pair of received packets, and uses that to listen for any *ARDEE Message*'s containing robot control data. Once an *ARDEE Message Header* is received for any robot control data, the AMPR sends all of the data from the retrieved *ARDEE Message* packet to the UDP De-Serializer to be extracted.

The **UDP De-Serializer** de-serializes all of the relevant information from the received, serialized, *ARDEE Message* packet into the standardized data structures, defined in Appendix **Listing A.16** and **Listing A.17**, corresponding to the specific robot control data received, and is sent to the CC custom ROS node. Currently, the only control data types that are supported is the motion control commands (specified in **Lines 83 - 87** of **Listing A.17**), however, additional forms of control data types, such as Gazebo simulation play/pause, multi-agent robot team macro commands, camera angle control, robotic harvestor controls, and much more, could be easily supported later.

The CC **Custom ROS Node** uses the de-serialized, extracted, and standardized control data, and re-formats it into the necessary ROS message format, specific to the control data type received so that the control data can be communicated over the ROS network. Finally, the CC ROS node publishes, via ROS Publishers, the re-formatted ROS message data to control the intended ROS components by publishing the ROS message data to the specified ROS topic.

All of these components are contained in a C++ class, shown in Appendix

Listing A.19, in order to allow for immediate publishing of the extracted control data to the ROS network as soon as any externally-broadcast control data becomes available, which allows for a minimal delay in control.

4.6 User-Friendly Launch Utilities

Now that all of ARDEE components have been developed to enable not only relatively realistic simulation of agricultural field environments and field robotic platform, but also platform-independent HIL development and evaluation of field robotic algorithms implemented on their target hardware, the final ARDEE element was the development of user-friendly utilities, such that, both expert and novice, users are able to easily and quickly configure, simulate, and utilize all of the, previously discussed, ARDEE components developed.

Not only do these utilities enable user-defined, relatively, realistic field robotic testing environments (i.e. field and robotic platform) to be simulated, but they also help to further reduce development times by using more efficient troubleshooting, data visualization, and robotic development methods, all of which are possible with the help from the open-source community and standard packages that with using ROS.

This section will discuss some of the user-friendly utilities that were developed making it easier for, both novice and expert, users to configure, load, and run various ARDEE components to fit their specific needs. Specifically, this section discusses the following topics: **(1)** using the `rosrun` command to run individual ROS nodes, **(2)** using the `roslaunch` command for running an easily configurable collection of ROS nodes, and finally **(3)** executing the custom bash/shell scripts developed enabling a compacted and runtime configurable execution of many different elements.

4.6.1 Using `rosrun`

Since most, if not every, aspect of the simulation implements ROS functionality, users are able to run the various aspects of the simulation environment using ROS standard command-line utilities. The most basic command-line utility available is the `rosrun` command. The `rosrun` command is used for running a single ROS node, which can be used for printing the terminal output statements from an individual node, which can be useful when troubleshooting its operation. Using the `rosrun` command can be useful for viewing a specific ROS node's terminal printouts (i.e. debugging, verbose, warning, etc. statements), as opposed to having, potentially, many different ROS nodes outputting print statements in the same terminal which can make it difficult to decipher certain ROS node operations. For example, the ARDEE *Controls Client*, discussed in **Section 4.5.3**, can be run individually by executing the following command in a terminal:

```
rosrun ardee_bridge controls_client_node
```

After executing this command, the *Controls Client* begins listening for any incoming *ARDEE Message* packets that may be broadcast from an external robotic platform, in addition to printing out the robot controls it is receiving. This functionality can be particularly useful for establishing whether robot controls, generated from an external robotic platform, are being received successfully or not.

Additionally, ROS nodes can sometimes fail, for various reasons, and as a result, it may sometimes be necessary to kill the failed node and restart it, which can be problematic when running multiple ROS nodes at the same time using the `roslaunch` command, discussed later. When this happens, it is necessary to kill all of the ROS nodes, not just the one, which can result in time wasted due to waiting for ROS nodes to be restarted, especially when some of the ROS nodes take a while to start. Users are notified by the ROS system, via command terminal statements, when a ROS node fails. Well-developed ROS nodes capable of robustly handling fail-case scenarios should not typically fail, however, it is more likely for ROS nodes when they are in early, or proof-of-concept, stages of development. By executing these ROS nodes that are more likely to fail using the `rosrun` command, it is possible to more efficiently test and debug developed ROS nodes by isolating them from a collection of ROS nodes that only need to be executed once.

For example, this was useful during the initial development of the ARDEE UDP mechanisms, where the ROS nodes, used within the UDP mechanisms, would fail due to improper UDP communication initialization. As a result of using `rosrun`, it was possible to prevent having to restart the Gazebo simulation (i.e. field with multiple crops and simulated robot model), which could sometimes take quite a while to load, depending on the number of resources that had to be rendered and spawned.

4.6.2 Using `roslaunch`

Although it is possible to run all of the necessary ROS nodes required for using the ARDEE simulation environment for testing any developed field robotic methods by executing each node singularly using the **`rosrun`** command, however, this requires the use of a separate terminal window for the lifespan of each ROS node executed, which can lead many terminal windows being opened at one time. As a result, it can be tedious when navigating between all the various terminal windows, and can even cause viewing the Gazebo simulation to be difficult. Luckily, there is another ROS standard command-line utility, called **`roslaunch`**, which is able to execute multiple ROS nodes defined in ROS launch files, or XML-formatted **.launch* files. Additionally, it is possible to pass command-line arguments giving users the ability to dynamically modify various ROS node input parameters on-the-fly.

In this work, the **`roslaunch`** command was used for loading any custom simulation environment in two main portions, both of which are defined in their own separate *.launch* files, in the following order of execution: (1) the simulated world, and (2) the simulated robot model and any other utilities.

Simulated World Launching: The simulated world can be composed of many different elements which describe the environment in which simulated robots operate in, and interact with, such as physics, lighting conditions, environmental conditions, and the simulated objects associated with the world (i.e. terrain, crops, etc.), all of which is defined SDF-formatted **.world* files. For agricultural field robot development, it can be expected that there will be many crop models being simulated in Gazebo at any given moment, which can result in the simulated world taking a while to load properly in Gazebo, due to the potentially resource-expensive rendering operations on Gazebo start-up. Therefore, it was chosen to launch, before anything else, all of the world-related components for any given simulated field environment, in a single **.launch* file. Not only does this ensure that the simulated world is properly initialized and simulated before running any un-related ROS nodes but it also helps to reduce the likelihood for a ROS node to fail due to any potential over-usage of available computational resources.

Furthermore, a handful of various, user-configurable, custom **.launch* files have been created (located in the `ardee_world/launch`), one example can be seen in **Listing A.20**, such that users can easily choose which simulated world, and any other configuration parameters, they want to load at runtime, by simply passing the command-line arguments after the launch file to be loaded, an example is shown in **Listing 4.2**, which results in the simulated world shown in **Figure 4.23**. Additionally, it is possible to easily add-on, and switch between, many different user-custom worlds since all of the worlds available for simulating should be located in the `ardee_world/worlds` directory.

```
roslaunch ardee_world custom_world.launch world:=farm.world
```

Listing 4.2: Shows the terminal command used to simulate the user-specified world shown in **Figure 4.23**.

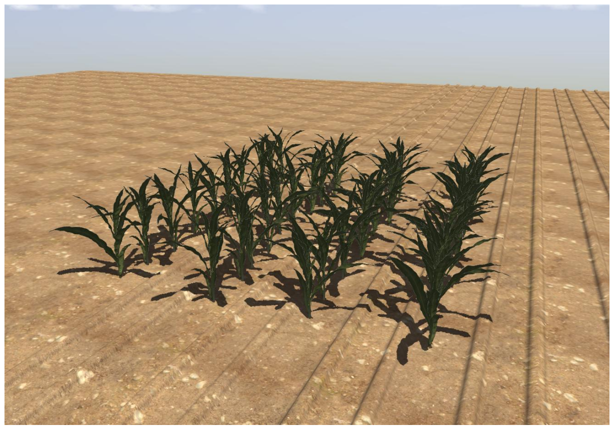


Figure 4.23: Shows the simulated world resulting from the execution of **Listing 4.2**.

Simulated Robot Model Launching: Once a simulated world is successfully loaded in Gazebo, the second `.launch` file that is executed is responsible for loading a user-specified simulated robot model in Gazebo, as well as any additional ROS nodes required by the user for developing and evaluating any field robotic algorithm, such as ROS nodes for controlling and interacting with the loaded simulated robot model, ROS nodes for visualization and debugging (i.e. rviz, rqt, etc.), any custom user-developed ROS nodes (i.e. nodes running any developed algorithms, or methods), and more. This second *launch* file should give users the ability to quickly, and easily, configure the specific simulated robot, as well as evaluation, ROS nodes they wish to use, in addition to any input node parameters used for runtime configuration. For example, the *launch* file shown in **Listing A.21** which does the following: (1) spawns a user-defined robot model configuration with a user-defined pose, (2) configures and loads the ROS nodes for the user-defined method of simulated robot control, (3) starts any additional debugging and visualization tools (i.e. rviz, rqt, etc.), and (4) executes any other ROS nodes necessary for proper interaction with the simulated robot model and/or hardware.

Additionally, the *launch* file, in **Listing A.21**, allows many of the specific ROS node parameters to be modified on-the-fly by passing the necessary command-line arguments for a specific ROS node. For example, the following terminal command can be used to execute the same launch file, in **Listing 4.3**, but instead loads a different simulated robot model, as shown in **Figure 4.24**, from the specified default robot:

```
roslaunch terrasentia_description launch_default_tsv1.launch
bot_model:=tsv2_default.urdf.xacro use_joy:=true dev_joy:=/dev/
input/js0
```

Listing 4.3: Shows the terminal command used to simulate the user-specified robot model shown in **Figure 4.24** and configures the robot model to be control using the user-specified joystick interface.

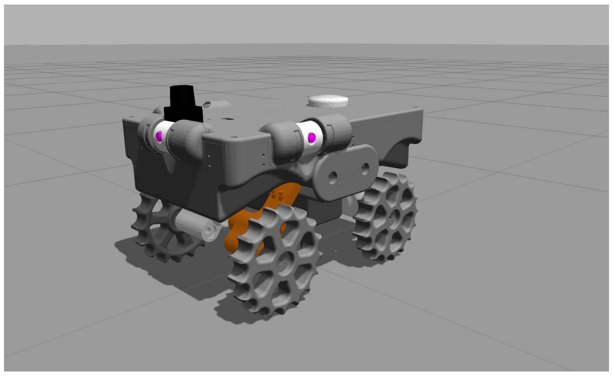


Figure 4.24: Shows the simulated robot model simulated in Gazebo as a result from the execution of **Listing 4.3**.

4.6.3 Using bash/shell scripts

Using the `roslaunch` command helps to execute, and dynamically configure, multiple ROS nodes in a compact form, however, it lacks the capability to define, and execute, custom terminal commands on its own which may be necessary for some users who want to execute any non-ROS utilities, such as executing the custom python script, discussed in **Section 4.2.2**, used to generate a custom and randomized simulated crop field. Additionally, in order to allow users who are not familiar with, or have no desire to learn ROS, using a bash/shell script to perform all the necessary environmental setup and configuration of the various ROS elements can make using the ARDEE simulation environment seem less daunting a task to those who may be new to using ROS. Additionally, using bash/shell scripts allows for a user-friendly, streamlined, user-interface that can be useful for configuring and loading everything required to run a custom simulation environment, for both beginners and experts alike. Therefore, a handful of bash/shell scripts, such as the one shown in **Listing A.22**, were developed in order to perform all of the necessary setup procedures, and customization required, for launching a specific simulation scenario, in addition to providing a, relatively, user-friendly command-line user-interface for executing various ROS components. For example, the simulation environment, shown in **Figure 4.25**, can be simulated by executing the terminal command, shown in **Listing 4.4**. The bash/shell first uses the first two shell script arguments which are used to select an available Gazebo terrain model and crop field model which are combined into a custom generated **.world* file which is then passed to the *roslaunch* command, as the "world" input argument, executed for loading the specified Gazebo. Next, the shell script tells the user (via terminal output) that it is waiting for the user to tell it when it should begin loading the simulated robot model into Gazebo (i.e. wait until the user sees the successful initialization of Gazebo). Finally, the shell script, at any point where the user wants to stop the simulation, executes the shell commands that properly shut down all of the system services used for simulation, so that the entire simulation is shut down quickly and properly.

```
./run_custom_world ardee_world/urdf/corn_plot.urdf.xacro
ardee_models/heightmap_ground
```

Listing 4.4: Shows the terminal command used to execute the custom bash/shell script which allows a user to hand-pick what terrain and field they wish to load into Gazebo, and then eventually loads a default robot model and the Rviz GUI used to show all the ARDEE simulated measurements. The result of the terminal command can be seen in **Figure 4.25**.

Not only does utilization of the custom bash/shell scripts show a quick, easy, and user-friendly method for loading a user-specified simulated world, robot model, and Rviz GUI, as shown in **Figure 4.25**, but the resulting simulated

Terrasentia robot's sensor measurement visualized in the Rviz GUI provides a sneak-peek into the potential realistic simulated sensor capabilities that ARDEE could be very useful for HIL development and evaluation for may different field robotic algorithms.

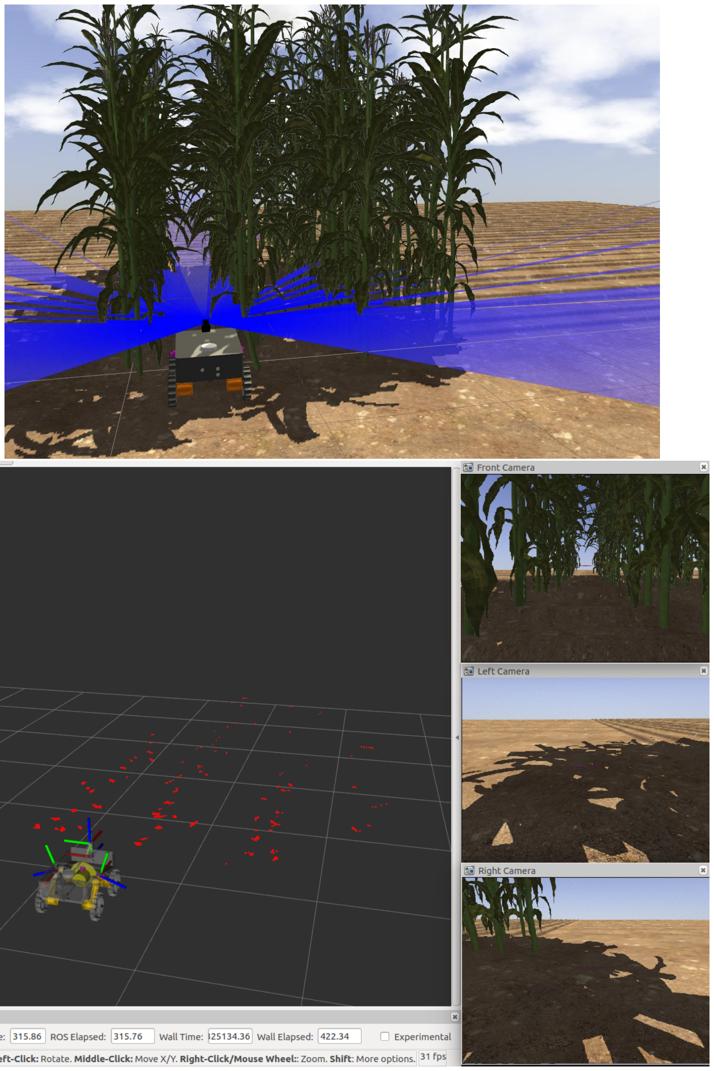

Front Camera
Left Camera
Right Camera
Time
ROS Time: 315.86 ROS Elapsed: 315.76 Wall Time: 125134.36 Wall Elapsed: 422.34 Experimental
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options. 31 fps

Figure 4.25: Shows the result from executing the terminal command, shown in Listing 4.4. Additionally, the Rviz GUI (Bottom) shows a preview of ARDEE's usefulness in its ability to generate seemingly realistic simulated Lidar (red points), as well as the simulated front, left, and right RGB camera video streams of the Terrasentia robot platform (leftmost three small windows stacked on each other).

Chapter 5

Lidar-Based Navigation

One desirable capability of the developed simulation environment tools, discussed in this work, is the ability to evaluate environment-dependent field robotic methods as they perform using simulated sensor measurements, and in simulated environments, that can not be easily replicated using conventional methods, such as Matlab. For example, it would be extremely difficult to simulate the various Lidar measurements, and evaluate the performance of a Lidar-based navigation method, using Matlab.

As a result, the Lidar-Based In-Row Navigation Technique (LBIRNT), discussed in detail in [28], was used to investigate the feasibility of using the developed simulation tools for the development of a Lidar-based navigation algorithm capable of robustly, and reliably, autonomously navigating the Terrasentia robot platform between individual rows of crops, throughout a variety of late-season field conditions.

5.1 Introduction

The LBIRNT uses the range measurements obtained from 2D Lidar scan to extract, and navigate about, the estimated center of the crop row using a set of heuristics of the Lidar data.

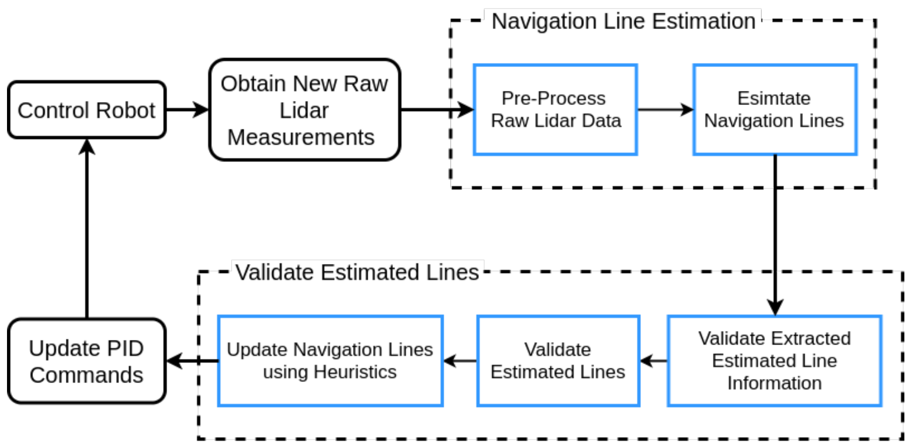

Navigation Line Estimation
Control Robot
Obtain New Raw Lidar Measurements
Pre-Process Raw Lidar Data
Esimtate Navigation Lines
Validate Estimated Lines
Update PID Commands
Update Navigation Lines using Heuristics
Validate Estimated Lines
Validate Extracted Estimated Line Information

Figure 5.1: Shows the workflow of the method developed in [66].

The primary elements of the Lidar-based navigation technique used can be described in the following three steps, and the workflow of these steps is shown in **Figure 5.1**: (1) the estimation of the navigation lines, (2) the validation of the estimated lines, and (3) controlling the robot's motion. For sake of completeness, the following section will briefly describe the keys ideas for these major components.

5.1.1 Line Estimation

The first primary LBIRNT step deals with the estimating of the linear model describing the rows of crops on both sides of the robot (left and right) used for navigating about the center of the crop row.

Before the crop row lines can be estimated, the raw Lidar measurements need to be pre-processed so that they can be useful for estimating the crop row lines, as well as reducing the computational costs by removing unnecessary measurements. For pre-processing, the LBIRNT first converts the raw Lidar measurements from polar coordinates into Cartesian coordinates, then separates the measurements into either the left or right, side. After being separated between left and right sides, the Lidar measurements are then filtered, only keeping measurements that are found within a rectangular area (for both the left and right sides) where the crop rows are expected to be in. The last pre-processing step, before the crop row lines are estimated, consists of choosing a subset of the filtered Lidar measurements that will be used for estimating the left/right crop lines, and should have a high probability of containing the most essential Lidar measurements that are characteristic of the crop rows. This subset of data is chosen using a simple histogram-based method to determine the left/right bin index from which the Lidar measurements should be pulled from.

Now that the raw Lidar measurements have been sufficiently pre-processed and filtered, the obtained Lidar subset data is used for estimating the linear coefficients for both the left and right crop rows using linear least squares regression for fitting the Lidar data to the slope and y-intercept of a linear model, using **(5.1)** and **(5.2)** respectively.

$$m = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{5.1}$$

$$b = \frac{\sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2} \tag{5.2}$$

After both the left and right linear models have been estimated, the estimated linear models are used to extract some additional information about the estimated lines which will be used later for validating how reliable the estimated lines are, and they consist of the following: line length, lateral distances to obtained lines, estimated line angle, standard deviation from the original values, and the change in the lateral distance from the previously found lateral distance.

5.1.2 Estimated Line Validation

The second LBIRNT step primarily deals with validating the reliability of the estimated crop row lines by using a series of "validity checks". This step determines the estimated crop row lines that are actually used to navigating within the row of crops, depending on the validity of the current estimated lines.

The first "validity checks" performed check validity of the extracted line information from the individual (left and right) estimated lines looking at whether the estimated lateral distances are within an acceptable bounds and if the change in the current estimate line angle from previously found line angle is within an acceptable threshold, for both left/right sides.

The second series of "validity checks" check to make sure that the individual estimated lines as a whole are valid. For these checks, two derived values, Lane Width LW and "Line Grade" are used for determining left and right line validity. For example, the "Line Grade" for the left side of the robot is calculated using

(5.3),

$$LineGrade[L] = \frac{np[L]}{np[L] + np[R]} + \frac{ly[L]}{ly[L] + ly[R]} + \left(1 - \frac{sd[L]}{sd[L] + sd[R]}\right) + \left(1 - \frac{dd[L]}{dd[L] + dd[R]}\right) \tag{5.3}$$

where $np[S]$ represents the number of data points used for the line fitting in (5.1) and (5.2), $ly[S]$ represents the length of the fitted lines in the y-axis, $sd[S]$ represents the standard deviation of the difference between the original input y-values and the resulting fitted line's y-values, and $dd[S]$ represents the difference in the orthogonal distance to the previously used and current estimated line. The $[S]$ in all of the previously mentioned variables is a placeholder used to represent the line in which the variable is associated with, either the left side $[L]$ or the right side $[R]$.

Finally, a set of heuristics are used for determining how all of the necessary estimated, extracted, and derived values are updated for the next update iteration.

5.1.3 Robot Control

The final LBIRNT step uses the estimated crop row lines resulting from the previous steps to derive the center line of the crop lane. This center line is used as the target for a classic PID controller which is used to control the robot's turn-rate in order to reduce the error between the robot's position and angle relative to the derived centerline.

5.2 Experimental Field Scenarios

In order to investigate, and evaluate, how accurately the developed simulation tools are able to replicate the Lidar measurements that would be acquired from real agricultural field environments, experiments were performed investigating the performance of the LBIRNT in some of its target field environments. Currently, the LBIRNT is successfully being used to autonomously navigate within a variety of corn and sorghum field environments. The LBIRNT has not been tested in real fields of tobacco as extensively as it has been in fields of corn and sorghum, thus there is an insufficient amount of useful real field data needed for comparison with the ARDEE simulated data. As a result, the evaluation of simulated tobacco fields is not discussed in this work. The three experiments that were performed investigated the LBIRNT's performance in three different, both simulated and real, field environments: a controlled indoor testing environment, a corn field, and a sorghum field.

In this section, the experimental setup and the testing procedures used for all the experiments performed will first be discussed. Finally, the specific field environments used for each experiment will be discussed.

5.2.1 Experimental Methodology

Regardless of the specific field environment and conditions being used during testing, the LBIRNT algorithm is developed in C++, and operates on the Terrasentia robot's hardware (i.e Raspberry Pi). This was done in order to investigate the usefulness of the developed simulation tools in their ability to effectively facilitate in the development of field robotic algorithms on their intended target hardware. As a result, the field robotic algorithm being developed, in this case, the LBIRNT, will exhibit the same behavior whether it is interacting with a physical robot in a physical environment or a simulated robot in a simulated environment.

The only difference between a physical-based, and a simulation-based, interaction, with respect to the LBIRNT, is where the required sensor measurements come from, and what the algorithm-generated command outputs control. In the typical physical-based, or normal real-world, interaction case, the LBIRNT simply communicates with the robot's physical sensors (i.e Hokuyo Lidar) to retrieve necessary measurements to generate PID target commands used to control the physical DC motors.

Alternatively, in the simulation-based interaction case, the simulated sensor measurements are serialized in a standardized UDP packet structure and broadcasted from the computer running the simulation environment, using the previously discussed sensor relay module. A simple UDP client operating on the target robot's hardware receives any available simulated sensor measurements being broadcasted and de-serializes the simulated sensor measurements into a standard format such that the measurements are indiscernible from the physical sensors. Using the received, and unpacked, simulated sensor measurements, the LBIRNT generates PID target commands. Instead of controlling the robot's physical motors, a simple UDP server (operating on the target robot's hardware) serializes the algorithm-generated target commands into a standardized UDP data packet structure and then broadcasted. These broadcasted commands are then received by the simulation computer, using the previously discussed controls relay module, and the unpacked into the ROS topic used to control the simulated robot model.

For investigating the LBIRNT's performance in both real and simulated agricultural field environments, the same experimental procedure is used regardless of the specific field environment used for testing.

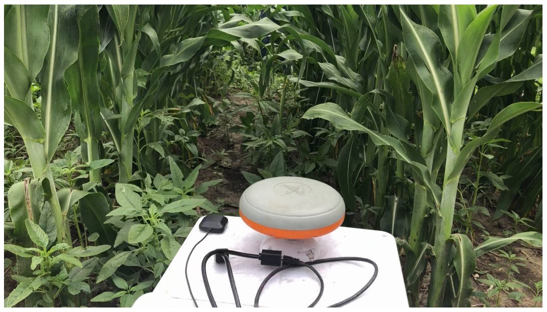


Figure 5.2: Shows an example experimental setup where the Terrasentia robot is placed close to the beginning of a row of corn and near the center, before beginning an autonomous run.

The experimental procedure used for each experiment consists of the following steps:

1. Before initiating autonomous row-following mode, the robot should be located relatively close to the entrance of any given row of crops, as well as relatively near the center of the crop row, similar to what is shown in **Figure 5.2**.

2. The following steps should be repeated (throughout the entire length of any given row of crops):

   (a) **Update Sensor Measurements:** New Lidar measurements are received, whether being received from the simulation environment or the physical robot's Lidar and pre-processed for LBIRNT usage.

   (b) **Estimate/Extract Line Information:** Pre-processed updated Lidar point-cloud data is used to update estimated row lines and the resulting estimated line information is extracted for later validation steps.

(c) **Update Robot Motion:** New robot motion control commands are generated based on the updated line estimates for left and right sides. PID controller is updated to achieve updated robot motion commands.

3. Once both, left and right sides, rows of the crop are no longer viewable by the Lidar, robot motion should cease, thus ending a single experimental run.

5.2.2 Scenario 1: Artificial Corn Row

The first scenario performed took place in a controlled testing environment where artificial, but relatively realistic, corn stalks, 23 evenly spaced 6 in apart, was set up to form an ideal row of corn with a 30 in spacing, shown in **Figure 5.3**. The purpose of this experiment was to investigate the LBIRNT's performance in a controlled setting where the environment remains constant and environmental variables can easily be identified. In using this controlled environment, a baseline assessment of the developed simulation tools' capabilities for simulating Lidar measurements can be presented. Additionally, this testing environment was used because it can be representative of a testing environment that would typically be used, if the simulation tools developed in this work did not exist, during the early stages of an algorithm's development, in the sense that it allows an algorithm to be tested, on the target hardware, and developed further without worrying about damaging high-value assets (i.e crops, etc.).

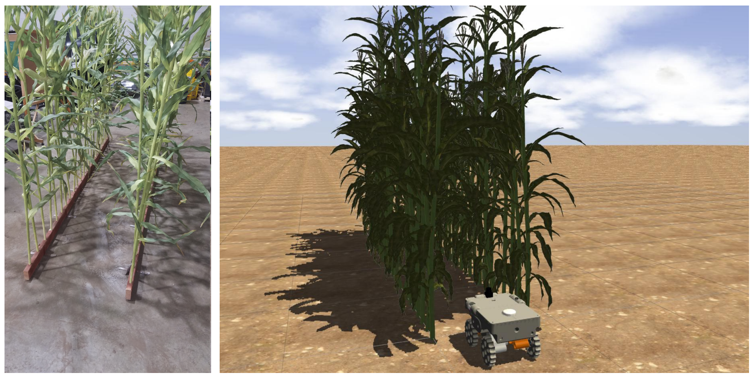


Figure 5.3: (Left) Shows the indoor controlled testing setup previously being used for LBIRNT development, during times when crops have been harvested, using realistic artificial corn stalks. (Right) Shows the simulated replication of the physical testing environment using the developed simulation utilities.

Since the terrain of the controlled environment was flat, the simulated field terrain was created by using a simple black grayscale image representing a flat heightmap. The field generation script, previously discussed in **Section 4.2.2**, was used to create the simulated corn rows, simply by passing the necessary command-line arguments such that the following real field characteristics are replicated: a single plot of corn containing two rows of crops, the spacing between the rows is 30in, each row contains 23 stalks (randomly chosen simulated corn stalk models) spaced 6in evenly apart. When creating the simulated corn field model, the script-generated locations for each corn stalk did not include add any noise to each stalk's mean center, because the corn stalk locations of the real controlled testing environment are perfectly aligned. By combining both the generated terrain model and the generated field model, the simulated replica of the real controlled testing environment is created, as shown in **Figure 5.3**.

5.2.3 Scenario 2: Real Corn Field

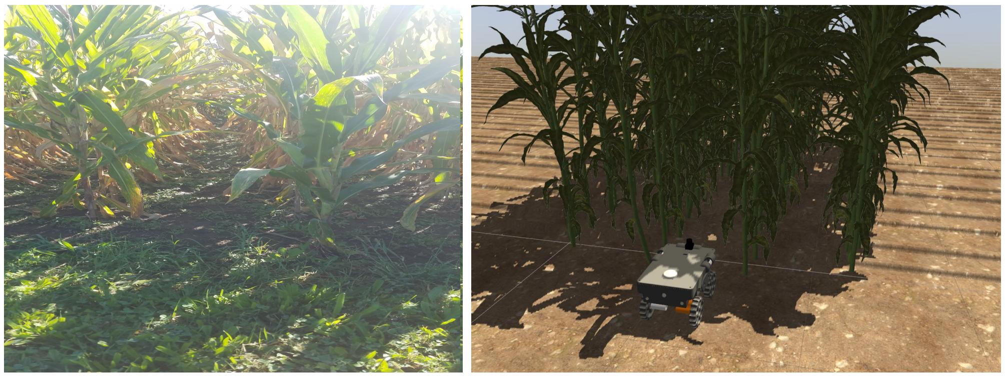


Figure 5.4: (Left) Shows an example portion of what is typically seen in actual corn fields that the Terrasentia robot normally navigates. (Right) Shows the simulated replication of a small portion of a typical corn field that would be encountered using the python script-generated field model and field terrain model using the presented simulation utilities.

The second scenario takes place in a typical field of corn, similar to what is shown in **Figure 5.4**. In contrast to the controlled environment, this field environment does not have easily identifiable environmental parameters. This field environment was used to investigate how capable the presented simulation tools are able to simulate, with some degree of accuracy, the inherent randomness of a typical corn field. Additionally, this experiment was performed to investigate how effective the simulated corn models are able to simulate physical corn stalks, with some degree of accuracy.

The accuracy of simulated field terrain was assumed to be of secondary concern, since the primary source of information required by the LBIRNT lies in the interaction with the physical, and simulated, crops. As a result, the simulated field's terrain was programmatically generated using a custom python script that created a grayscale image, used for the Gazebo heightmap, containing rows of pixels simulating periodically uneven terrain. This was done in order to capture, with a low degree of accuracy, the effect on the resulting Lidar measurements that can be expected from a typically rough and uneven terrain.

Since the creation of the simulated counterpart of a typical corn field is not as straight-forward as the controlled field environment, a few assumptions had to be made on the configuration of the corn stalks before the simulated corn field could be created. The first assumption is that there are four rows of corn with a mean spacing between each row of corn is approximately 30in. Secondly, it was assumed that each row of corn was approximately 5m long, with an average of 50 mature corn stalks per row. With these assumptions in mind, the previously discussed field generation script was used to generate a resulting plot of corn rows capable of simulating with a sufficient degree the inherent randomness that could be encountered in a typical corn field. Using both the programmatically generated terrain and corn field model, the resulting simulated corn field, used to replicate the targeted real corn field, can be seen in **Figure 5.4**.

5.2.4 Scenario 3: Real Sorghum Field

The third scenario takes place in a typical field of sorghum, similar to **Figure 5.5**. This field environment has many similarities to that of a typical corn field in regards to the behavior of the received Lidar measurements due to the similarities in the leaves encountered at Lidar-level, however in sorghum fields the Lidar measurements contains a considerably higher amount in "noise" resulting from the highly cluttered, with frequent sensor occlusions, present in sorghum rows. This field environment was used to investigate how capable the presented simulation tools are able to simulate, with some degree of accuracy, the inherent adverse operating conditions of a typical field of sorghum. Additionally, this experiment was performed to investigate how effective the simulated sorghum models are able to simulate physical sorghum stalks, with some degree of accuracy.

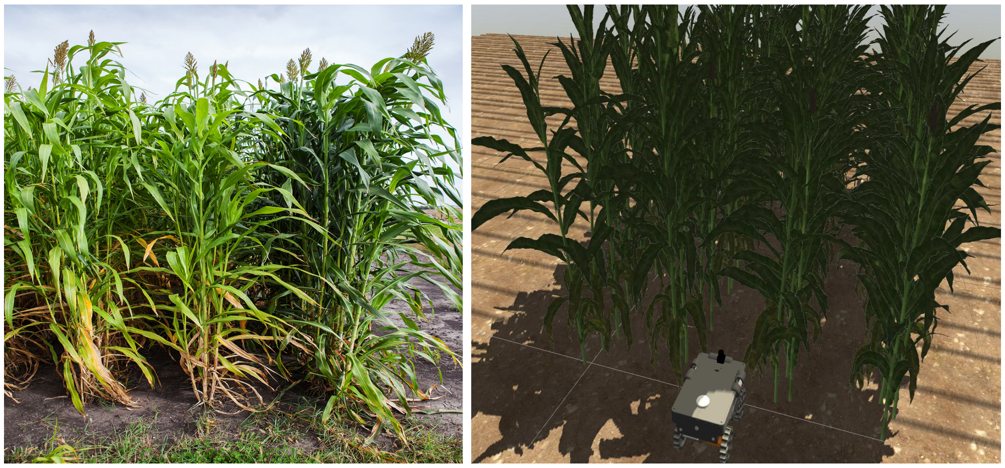


Figure 5.5: (Left) Shows an example portion of what is typically seen in actual sorghum fields that the Terrasentia robot normally navigates. (Right) Shows the simulated replication of a small portion of a typical corn field that would be encountered using the python script-generated field model and field terrain model using the presented simulation utilities.

In the case of a field of sorghum, the terrain was very similar to that of the average field of corn. As a result, the same programmatically-generated heightmap discussed in the previous section was used for simulating the sorghum field's terrain. The field of sorghum was generated similar to the field of corn, where there are four 5m-long rows of mature sorghum, spaced approximately 30in apart, each containing an average of 50 mature sorghum stalks; however when using the field generation script, there was a slight increase in the Gaussian noise that was added to each of the mean sorghum stalk centers. The resulting simulated sorghum field can be seen in **Figure 5.5**.

5.3 Results

Using the experimental procedure previously discussed, the performance of the LBIRNT deployed on the Terrasentia robot's hardware, as well as the raw Lidar measurements, obtained in both the real, and Gazebo-simulated, field environments in order to investigate the presented development environment's ability to accurately represent, and simulate, the real field conditions necessary for Lidar-based field robotic algorithm development and evaluation. In this section, the obtained raw Lidar measurements, as well as the LBIRNT's crop-row line detection performance, obtained, in real-time, from the three experimental scenarios defined are presented in this section.

Figure 5.6 presents the results obtained from the first experimental scenario, shown in **Figure 5.3**. By using a controlled testing environment, in which the exact positions of the clutter-free artificial corn stalks in a physically-simulated corn row, it is possible to establish a baseline comparison of the results obtained, from both the physical and Gazebo-simulated environments.

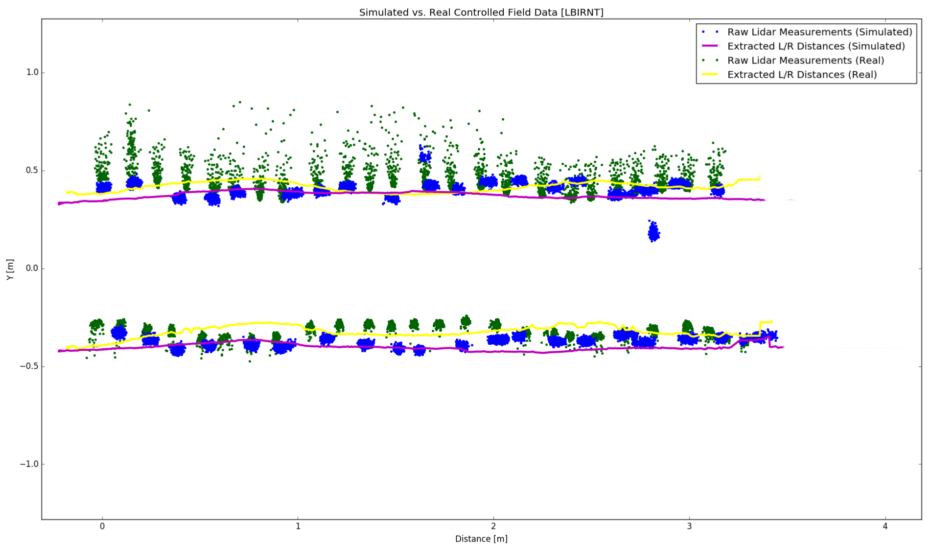


Figure 5.6: Shows both the raw Lidar measurements and the LBIRNT generated perceived crop row distances obtained from both the real-world (dark green and light green, respectively) and Gazebo-simulated (dark blue and light blue, respectively) controlled testing environments shown in **Figure 5.3**.

In **Figure 5.6**, it can be seen that the simulated raw Lidar measurements (displayed in blue) and the real raw Lidar measurements (displayed in green) obtained are nearly identical. Not just raw Lidar measurements, but even the behavior of the LBIRNT resulting from both the real and simulated environments are a close match.

Unfortunately, the first experimental scenario is not a representative substitute for actual corn field environments, and could even be mathematically simulated. As a result, the second experimental field environment scenario, shown in **Figure 5.4**, investigates the presented development environment's ability to simulate the Lidar measurements, necessary for developing and evaluating Lidar-based field robotic algorithms, in actual corn field environments, which are highly cluttered, random, and are nearly impossible to mathematically simulate.

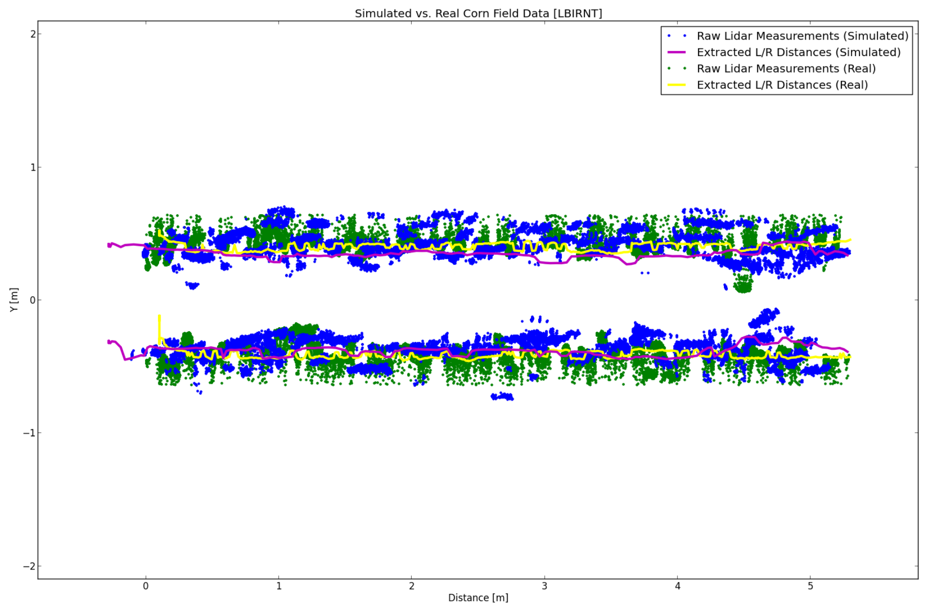


Figure 5.7: Shows both the raw Lidar measurements and the LBIRNT generated perceived crop row distances obtained from both the real-world (dark green and light green, respectively) and Gazebo-simulated (dark blue and light blue, respectively) corn field environments shown in **Figure 5.4**.

In **Figure 5.7**, the results obtained from the second experimental scenario (corn field) are presented. Although they are not as easy to compare opposed to those obtained in the first scenario, it can roughly be seen that the raw Lidar measurements obtained from the simulated replica of the real corn field relatively accurately replicates the essence of the real Lidar measurements that were obtained from the real corn field. Additionally, it can be seen that the behavior of the LBIRNT resulting from the simulated Lidar measurements closely matches the LBIRNT's behavior resulting from the real Lidar measurements, which further validates the presented development environment's ability to relatively accurately simulate the Lidar measurements that would be obtained from a real corn field.

Similar to the second scenario, the third experimental scenario (sorghum field) investigates the presented development's ability to accurately simulate the Lidar measurements, necessary for developing and evaluating Lidar-based field robotic algorithms, in sorghum field environments, which are increasingly more dense and cluttered compared to corn fields.

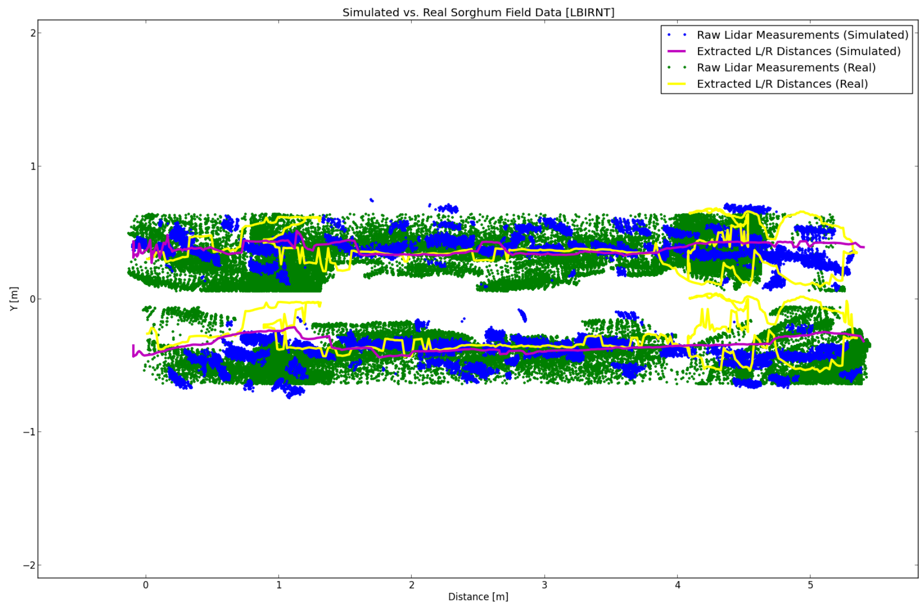


Figure 5.8: Shows both the raw Lidar measurements and the LBIRNT generated perceived crop row distances obtained from both the real-world (dark green and light green, respectively) and Gazebo-simulated (dark blue and light blue, respectively) sorghum field environments shown in **Figure 5.5**.

Figure 5.8 presents the results obtained from the third experimental scenario. Unlike the second scenario, the Lidar measurements obtained, from the real sorghum field, contain a significantly higher degree of noise and clutter caused by the frequently encountered leaf occlusions from the sorghum field's dense foliage. Although it does not properly simulate this high degree of noise and clutter in some areas of the row, it can be seen from the figure that resulting simulated Lidar measurements, for the most part, effectively simulate the noisy Lidar measurements obtained from the real sorghum field. The resulting LBIRNT's behavior supports this, where the estimated distances to the center of the left and right sides of the crop row from the simulated Lidar measurements closely match those resulting from the real Lidar measurements.

5.4 Discussion

In this section, the capabilities and effectiveness of the development environment, presented in this thesis, used as a field robotic algorithm development and evaluation tool for Lidar-based methods are investigated.

In addition, the behavior of an Lidar-based field robotic method developed for under-canopy autonomous navigation of a novel ultra-lightweight, and compact, robotic platform, and the raw Lidar measurements used by it, are presented, and compared, in three different environments, both simulated and real, which are representative of the typical agricultural field environments that the robotic platform normally operates within.

The results presented show that the development environment presented in this work is able to relatively accurately simulate the Lidar measurements that would be obtained from a real field environment.

Chapter 6

Conclusion

6.1 Future Work

In this work, ARDEE's ability to effectively allow the development and evaluation of a Lidar-based field robotic algorithm on real hardware for both real and simulated field environments, using the presented simulated corn and sorghum crop models, was presented. Furthermore, the accuracy of the simulated vision data to the real vision data needs to be investigated for all three crop models using different field environments.

In addition to comparing the raw Lidar measurements, the experimental results presented a Lidar-based navigation algorithm that was already developed and is used, for autonomous under-canopy navigation. Therefore, by combing the need to verify the visual accuracy of the simulated crop models, future work should quantitatively evaluate the effectiveness of ARDEE as it is used to develop a novel vision-based full season navigation method from a proof-of-concept stage to a field-ready deployment stage.

The novel vision-based approach has shown promise for real-time operation (i.e 30FPS) during early stages of corn using real camera data collected and has been shown to be capable of working in a variety of corn and sorghum fields during later stages in crop development, examples shown in **Figure 6.1**. Furthermore, ARDEE has already been used for preliminary development of the vision-based navigation approach, also shown in **Figure 6.1**.

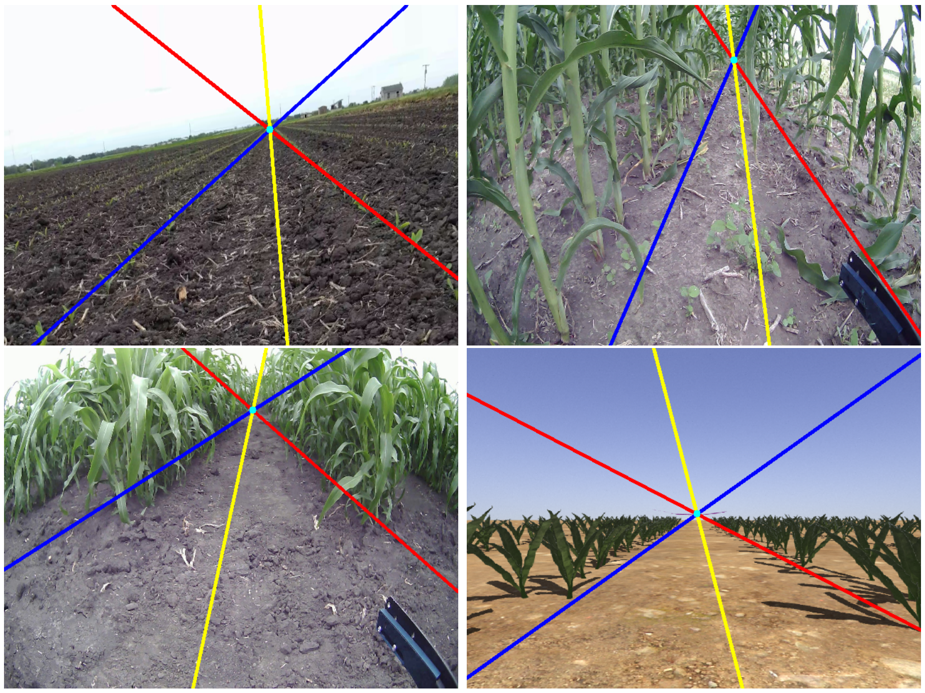


Figure 6.1: Shows the preliminary results of the experimental vision-based full-season navigation method for different field environments. **(Top Left)** Shows reliable performance for an early-season corn field, **(Top Right)** Shows sufficient performance for a fairly cluttered late-season corn field, **(Bottom Left)** Shows sufficient performance for a mid-season corn field, **(Bottom Right)** Shows good performance in the ARDEE simulated early season corn field.

Unfortunately, more development and evaluation is required in order for it to be reliable enough for full-season autonomous navigation, an example of non-reliable results can be seen in **Figure 6.2**, and needs to be robust to the following cases: varying field and lighting conditions, various developmental stages of different crop species, to frequent camera occlusions resulting from crop leaves.

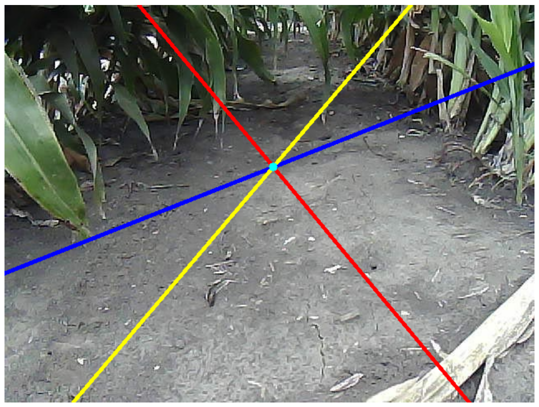


Figure 6.2: Shows an example field environment (grain sorghum) where the vision-based navigation approach does not work well and requires further investigation.

6.2 Summary

In summary, this thesis presents a novel Agricultural Robotic Development and Evaluation Environment (ARDEE) used for a more efficient approach to the development and evaluation of any field robotic method intended for operation on, potentially, many different physical robotic platforms being used for real field applications.

ARDEE incorporates the use of the Gazebo simulation environment, in combination with a collection of relatively realistic agricultural crop models and simulated sensor models, to enable field robotic methods to be developed and tested for a variety of different field environments all-year long, which is difficult, or impossible, to do using traditional field development and testing methods. Through the use of the custom developed UDP-based communication mechanisms employed, ARDEE allows partial, as well as entire, system development and hardware integration, potentially across many different custom field robotic platforms. Not only does the standardized UDP mechanisms allow the development of field robotic systems ready for real field testing, in spite of any hardware limitations, but they also support the use of the Gazebo simulated world (i.e simulated sensor measurements, simulated robot control, custom field-robot interaction dynamics, etc.) to evaluate the real-time operation of custom-developed field robotic algorithms executed on target robotic hardware, which may or may not support the use of ROS, as if they were in a real physical field.

Experimental results were presented comparing the raw Lidar measurements obtained for three different pairs (i.e real and corresponding simulated counterpart) of agricultural field environments. The results showed that the simulated raw Lidar measurements (resulting from the ARDEE simulation) very closely matched real raw Lidar measurements (resulting from a physical Hokuyo UST-10LX Lidar) for all three field environment variations.

In addition to raw Lidar measurements, the experimental results presented were used to compare the real-time performance of a Lidar-based under-canopy crop row navigation technique, executed on the target hardware of a novel field robotic platform (Terrasentia), from operation in the same three different pairs of field environments. The results showed that ARDEE, and the developed UDP-based communication mechanisms, successfully enables the development and evaluation of a Lidar-based field robotic algorithm's real-time performance on its target hardware using the Gazebo simulated environment as a surrogate for the sensor measurements input and the control of the robotic platform.

Bibliography

[1] R. Arnaud and M.C. Barnes. *Collada: Sailing the Gulf of 3d Digital Content Creation*. Ak Peters Series. A K Peters, 2006. ISBN: 9781568812878. URL: https://books.google.com/books?id=laxQAAAAMAAJ.

[2] *Asset Scan 3D Terrain Models*. URL: https://www.turbosquid.com/Search/Artists/Asset-Scan-3d.

[3] *Autodesk Fusion360*. URL: https://www.autodesk.com/products/fusion-360/overview.

[4] C. Wouter Bac et al. "Harvesting Robots for High-value Crops: State-of-the-art Review and Challenges Ahead". In: *Journal of Field Robotics* 31.6 (Dec. 2014), pp. 888–911. ISSN: 1556-4967. DOI: 10.1002/rob.21525. URL: http:https://doi.org/10.1002/rob.21525.

[5] Benjamin Balaguer et al. "USARSim: a validated simulator for research in robotics and automation". In: *Workshop on Robot Simulators: Available Software, Scientific Applications, and Future Trends at IEEE/RSJ*. Citeseer. 2008.

[6] David Ball et al. "Robotics for Sustainable Broad-Acre Agriculture". In: *Field and Service Robotics: Results of the 9th International Conference*. Ed. by Luis Mejias, Peter Corke, and Jonathan Roberts. Cham: Springer International Publishing, 2015, pp. 439–453. ISBN: 978-3-319-07488-7. DOI: 10.1007/978-3-319-07488-7_30. URL: https://doi.org/10.1007/978-3-319-07488-7_30.

[7] Peter Biber et al. "Navigation System of the Autonomous Agricultural Robot " BoniRob " *". In:

[8] *Blender 3D Graphics Editor*. URL: https://www.blender.org/.

[9] T. Boge and O. Ma. “Using advanced industrial robotics for spacecraft Rendezvous and Docking simulation”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980583.

[10] Erwin Coumans. “Bullet Physics Simulation”. In: *ACM SIGGRAPH 2015 Courses*. SIGGRAPH ’15. Los Angeles, California: ACM, 2015. ISBN: 978-1-4503-3634-5. DOI: 10.1145/2776880.2792704. URL: http://doi.acm.org/10.1145/2776880.2792704.

[11] M. Freese E. Rohmer S. P. N. Singh. “V-REP: a Versatile and Scalable Robot Simulation Framework”. In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. 2013.

[12] Peter Eastman and Michael Sherman. *Simbody: Multibody Physics API*. URL: https://www.simtk.org/projects/simbody.

[13] Gilberto Echeverria et al. “Simulating Complex Robotic Scenarios with MORSE”. In: *SIMPAR*. 2012, pp. 197–208. URL: http://morse.openrobots.org.

[14] Y. Edan, B. A. Engel, and G. E. Miles. “Intelligent control system simulation of an agricultural robot”. In: *Journal of Intelligent and Robotic Systems* 8.2 (1993), pp. 267–284. ISSN: 1573-0409. DOI: 10.1007/BF01257998. URL: https://doi.org/10.1007/BF01257998.

[15] Luis Emmi et al. “Fleets of robots for precision agriculture: a simulation environment”. In: *Industrial Robot: An International Journal* 40.1 (2013), pp. 41–58. DOI: 10.1108/01439911311294246. eprint: https://doi.org/10.1108/01439911311294246. URL: https://doi.org/10.1108/01439911311294246.

[16] Noah Fahlgren, Malia A Gehan, and Ivan Baxter. “Lights, camera, action: high-throughput plant phenotyping is ready for a close-up”. In: *Current Opinion in Plant Biology* 24 (2015), pp. 93 –99. ISSN: 1369-5266. DOI: https://doi.org/10.1016/j.pbi.2015.02.006. URL: http://www.sciencedirect.com/science/article/pii/S1369526615000266.

[17] FAO. *The State of the World’s land and water resources for Food and Agriculture (SOLAW) Managing systems at risk*. London: Earthscan, 2011.

[18] Fred Fulkerson. *SolidWorks Basics: A Project Based Approach*. New York, NY, USA: Industrial Press, Inc., 2015. ISBN: 083113593X, 9780831135935.

[19] Robert T. Furbank and Mark Tester. “Phenomics – technologies to relieve the phenotyping bottleneck”. In: *Trends in Plant Science* 16.12 (2011), pp. 635 –644. ISSN: 1360-1385. DOI: https://doi.org/10.1016/j.tplants.2011.09.005. URL: http://www.sciencedirect.com/science/article/pii/S1360138511002093.

[20] *Gazebo ROS Plugins*. URL: https://github.com/ros-simulation/gazebo_ros_pkgs.

[21] Olaf Gietelink et al. “Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations”. In: *Vehicle System Dynamics* 44.7 (2006), pp. 569–590. DOI: 10.1080/00423110600563338. eprint: https://doi.org/10.1080/00423110600563338. URL: https://doi.org/10.1080/00423110600563338.

[22] *GNU Image Manipulation Program*. URL: https://www.gimp.org/.

[23] *GrabCAD*. URL: https://grabcad.com/.

[24] L. Grimstad et al. “On the design of a low-cost, light-weight, and highly versatile agricultural robot”. In: *2015 IEEE International Workshop on Advanced Robotics and its Social Impacts (ARSO)*. 2015, pp. 1–6. DOI: 10.1109/ARSO.2015.7428210.

[25] Lin Haibo et al. “Study and Experiment on a Wheat Precision Seeding Robot”. In: *J. Robot.* 2015 (Jan. 2015), 12:12–12:12. ISSN: 1687-9600. DOI: 10.1155/2015/696301. URL: https://doi.org/10.1155/2015/696301.

[26] David Hall et al. “Towards Unsupervised Weed Scouting for Agricultural Robotics”. In: *CoRR* abs/1702.01247 (2017). arXiv: 1702.01247. URL: http://arxiv.org/abs/1702.01247.

[27] Ibrahim A. Hameed et al. “Robotic Harvesting of Fruiting Vegetables: A Simulation Approach in V-REP, ROS and MATLAB”. In: *Automation in Agriculture*. Ed. by Stephan Hussmann. Rijeka: InTech, 2018. Chap. 5. DOI: 10.5772/intechopen.73861. URL: https://doi.org/10.5772/intechopen.73861.

[28] V. A. H. Higuti et al. “Under canopy LiDAR-based autonomous navigation”. In: *second revision submitted to the Journal of Field Robotics on September 9th, 2018* ().

[29] Victor IC Hofstede, Bachelor Opleiding Kunstmatige Intelligentie, and A Visser. “The importance and purpose of simulation in robotics”. In: (2015).

[30] Xfrog Inc. *xFrog Agricultural Models*. http://xfrog.com/product/X-55.html.

[31] Pooyan Jamshidi. *Gazebo Simulated Battery*. URL: https://github.com/pooyanjamshidi/brass_gazebo_battery.

[32] Kjeld Jensen et al. "Towards an Open Software Platform for Field Robots in Precision Agriculture". In: *Robotics* 3.2 (2014), pp. 207–234. ISSN: 2218-6581. DOI: 10.3390/robotics3020207. URL: http://www.mdpi.com/2218-6581/3/2/207.

[33] Suero Alain Jerez Julio and Manfred Manik. *Newton Game Dynamics*. URL: https://github.com/MADEAPPS/newton-dynamics.

[34] Kyle Johns and Trevor Taylor. *Professional Microsoft Robotics Developer Studio*. Birmingham, UK, UK: Wrox Press Ltd., 2008. ISBN: 0470141077, 9780470141076.

[35] Thompson B. Young H. Kayacan E. and G. Chowdhary. "High precision of an ultra-compact 3d printed field robot in the presence of slip." In: *In International Conference on Robotics and Automation, Brisbane, Australia. IEEE. Submitted.* (2018).

[36] Felix Kerger. *OGRE 3D 1.7 Beginner's Guide*. Packt Publishing, 2010. ISBN: 1849512485, 9781849512480.

[37] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[38] Stefan Kohlbrecher and Johannes Meyer. *Hector Gazebo Plugins*. URL: http://wiki.ros.org/hector_gazebo_plugins.

[39] Jeongseok Lee et al. "DART: Dynamic Animation and Robotics Toolkit". In: *The Journal of Open Source Software* 3.22 (2018), p. 500. DOI: 10.21105/joss.00500. URL: https://doi.org/10.21105/joss.00500.

[40] Andreas Linz and Arno Ruckelshausen. "Autonomous Service Robots for Orchards and Vineyards : 3 D Simulation Environment of Multi Sensor-based Navigation and Applications". In:

[41] O. Ma. "Contact dynamics modelling for the simulation of the Space Station manipulators handling payloads". In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 2. 1995, 1252–1258 vol.2. DOI: 10.1109/ROBOT.1995.525453.

[42] O. Michel. “Webots: Professional Mobile Robot Simulation”. In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf.

[43] Tim Mueller-Sim et al. “The Robotanist: a ground-based agricultural robot for high-throughput crop phenotyping”. MA thesis. 2017, pp. 3634–3639.

[44] Patricio Nebot, Joaquín Torres-Sospedra, and Rafael J. Martínez. “A New HLA-Based Distributed Control Architecture for Agricultural Teams of Robots in Hybrid Applications with Real and Simulated Devices or Environments”. In: *Sensors* 11.4 (2011), pp. 4385–4400. ISSN: 1424-8220. DOI: 10.3390/s110404385. URL: http://www.mdpi.com/1424-8220/11/4/4385.

[45] *OGRE Material*. URL: http://wiki.ogre3d.org/-Material.

[46] R. Pepy, A. Lambert, and H. Mounier. “Path Planning using a Dynamic Vehicle Model”. In: *2006 2nd International Conference on Information Communication Technologies*. Vol. 1. 2006, pp. 781–786. DOI: 10.1109/ICTTA.2006.1684472.

[47] Carlo Pinciroli et al. “ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems”. In: *Swarm Intelligence* 6.4 (2012), pp. 271–295.

[48] *Python*. URL: https://www.python.org/.

[49] *Qt: UI Development Tool*. URL: https://www.qt.io/.

[50] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009.

[51] Md. Matiur Rahaman et al. “Advanced phenotyping and phenotype data analysis for the study of plant growth and development”. In: *Frontiers in Plant Science* 6 (2015), p. 619. ISSN: 1664-462X. DOI: 10.3389/fpls.2015.00619. URL: https://www.frontiersin.org/article/10.3389/fpls.2015.00619.

[52] Martin Reddy. *Wavefront object (obj) file format*. http://www.martinreddy.net/gfx/3d/OBJ.spec.

[53] J Rodriguez and D Nardi. “Simulation environment for the deployment of robots in precision agriculture”. In: ().

[54] LE Roscoe et al. “Stereolithography interface specification”. In: *America-3D Systems Inc* 27 (1988).

[55] Andrew Sanders. *An Introduction to Unreal Engine 4*. Natick, MA, USA: A. K. Peters, Ltd., 2016. ISBN: 1498765092, 9781498765091.

[56] *SDFormat*. URL: http://sdformat.org/spec.

[57] CM Labs Simulations. *Vortex Physics Engine*. URL: https://www.cm-labs.com/vortex-studio/.

[58] Russell Smith et al. “Open dynamics engine”. In: (2005).

[59] *SolidWorks to URDF Exporter*. URL: http://wiki.ros.org/sw_urdf_exporter.

[60] *terrain.party*. URL: https://terrain.party/.

[61] *Thingiverse*. URL: https://www.thingiverse.com/.

[62] *TurboSquid: 3D Models for Professionals*. URL: https://www.turbosquid.com/.

[63] Population Division United Nations Department of Economic and Social Affairs. *World population prospects: The 2015 Revision, Key Findings and Advance Tables*. Working Paper No. ESA/P/WP.241. 2015.

[64] K. Uno and K. Kashiyama. “Development of simulation system for the disaster evacuation based on multi-agent model using GIS”. In: *Tsinghua Science and Technology* 13.S1 (2008), pp. 348–353. ISSN: 1007-0214. DOI: 10.1016/S1007-0214(08)70173-1.

[65] *URDF Format*. URL: http://wiki.ros.org/urdf.

[66] AE Velasquez et al. “Helvis-A Small-scale Agricultural Mobile Robot Prototype For Precision Agriculture”. In: *Proceedings of the 13th International Conference on Precision Agriculture (ICPA2016), St. Louis, MO, USA*. Vol. 31. 2016, pp. 1–18.

[67] Zhenyu Wang et al. “Rapid Developing the Simulation and Control Systems for a Multifunctional Autonomous Agricultural Robot with ROS”. In: *Intelligent Robotics and Applications*. Ed. by Naoyuki Kubota et al. Cham: Springer International Publishing, 2016, pp. 26–39. ISBN: 978-3-319-43506-0.

[68] *xacro*. URL: http://wiki.ros.org/xacro.

[69] Quan-Zhong Yan, John M. Williams, and Jim Li. “Chassis Control System Development Using Simulation: Software in the Loop, Rapid Prototyping, and Hardware in the Loop”. In: *SAE Technical Paper*. SAE International, May 2002. DOI: 10.4271/2002-01-1565. URL: https://doi.org/10.4271/2002-01-1565.

[70] Zhi Yan et al. “Building a ROS-Based Testbed for Realistic Multi-Robot Simulation: Taking the Exploration as an Example”. In: *Robotics* 6 (2017), p. 21.

[71] Xiangjun Zou et al. “Extracting Behavior Knowledge and Modeling Based on Virtual Agricultural Mobile Robot”. In: *Advances in Artificial Reality and Tele-Existence*. Ed. by Zhigeng Pan et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 28–37. ISBN: 978-3-540-49779-0.

[72] Leon Žlajpah. “Simulation in robotics”. In: *Mathematics and Computers in Simulation* 79.4 (2008). 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects, pp. 879 – 897. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2008.02.017. URL: http://www.sciencedirect.com/science/article/pii/S0378475408001183.

Appendix A: Code Listings

A.1 Background

A.1.1 Example SDF Model Representation

```
<?xml version='1.0'?>
<sdf version="1.4">
  <model name="simple_gps">
    <pose>0 0 0 0 0 0</pose>
    <static>false</static>

    <!-- ==============================================================
           Physical Body Component Definitions
    ============================================================== -->
    <link name="gps_link">
      <!-- Inertial Body (Used by Physics Engine) Parameters -->
      <inertial>
        <mass>0.5</mass>
        <pose>0 0 0 0 0 0</pose>
        <inertia>
          <ixx>0.05</ixx>
          <ixy>0.0</ixy>
          <ixz>0.0</ixz>
          <iyy>0.05</iyy>
          <iyz>0.0</iyz>
          <izz>0.05</izz>
        </inertia>
      </inertial>

      <!-- Collisional Body (will hit other objects) Parameters -->
      <collision name="collision">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
```

```
          <mesh>
            <uri>model://simple_gps_model/meshes/antenna_3GO16.stl</uri>
          </mesh>
        </geometry>
      </collision>

      <!-- Visual Body (the one seen in Gazebo) Parameters -->
      <visual name="visual">
        <pose>0 0 0 0 0 0</pose>
        <geometry>
          <mesh>
            <uri>model://simple_gps_model/meshes/antenna_3GO16.stl</uri>
          </mesh>
        </geometry>
      </visual>
    </link>

    <!-- ==================================================
         Gazebo Sensor Plugin: Generates measurements
    ================================================== -->
    <plugin name="${name}_controller" filename="libgazebo_ros_gps.so">
      <bodyName>gps_link</bodyName>
      <frameId>gps_link</frameId>
      <topicName>gps</topicName>
      <velocityTopicName>gps/fix_velocity</velocityTopicName>
      <updateRate>5</updateRate>
      <gaussianNoise>0.05 0.05 0.05</gaussianNoise>
      <alwaysOn>1</alwaysOn>

      <!-- Use GPS coordinates of Urbana, IL for origin of reference -->
      <!-- Ref. Altitude is height above WGS84 ellipsoid. NOT SEA LEVEL -->
      <referenceAltitude>-32.593</referenceAltitude>
      <referenceLatitude>40.1105875</referenceLatitude>
      <referenceLongitude>-88.2072697</referenceLongitude>
    </plugin>

  </model>
</sdf>
```

Listing A.1: Shows the SDF code that can be used to create a simple GPS sensor like the one shown in **Figure 2.12**.

A.1.2 Example URDF Model Representation

```
<?xml version="1.0"?>
<robot name="sensor_gps" xmlns:xacro="http://www.ros.org/wiki/xacro
    ">

  <xacro:include filename="$(find ardee_sensors)/urdf/includes/
    materials.urdf.xacro"/>

    <xacro:macro name="sensor_gps" params="name parent *origin
    color ref_lat ref_long ref_head drift vel_drift noise vel_noise
     update_rate">


    <link name="${name}_link">
      <inertial>
        <mass value="0.5"/>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <inertia ixx="0.05" ixy="0" ixz="0" iyy="0.05" iyz="0" izz=
    "0.05"/>
      </inertial>
      <visual>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
          <mesh filename="package://ardee_sensors/meshes/
    antenna_3GO16.stl"/>
        </geometry>
        <xacro:color_material color="${color}"/>
      </visual>
      <collision>
        <origin rpy="0 0 0" xyz="0 0 0"/>
        <geometry>
          <mesh filename="package://ardee_sensors/meshes/
    antenna_3GO16.stl" scale="1.0 1.0 1.0"/>
        </geometry>
      </collision>
    </link>


    <joint name="${name}_joint" type="fixed">
      <axis xyz="0 1 0"/>
      <xacro:insert_block name="origin"/>
      <parent link="${parent}"/>
      <child link="${name}_link"/>
    </joint>

    <xacro:color_gazebo parent="${name}_link" color="${color}"/>


    <!-- ==========================================================
          Gazebo Sensor Plugin: Generates measurements
    ========================================================== -->
    <gazebo>
      <plugin name="${name}_controller" filename="libgazebo_ros_gps
    .so">
        <bodyName>gps_link</bodyName>
        <frameId>gps_link</frameId>
        <topicName>gps</topicName>
        <velocityTopicName>gps/fix_velocity</velocityTopicName>
        <updateRate>5</updateRate>
        <gaussianNoise>0.05 0.05 0.05</gaussianNoise>
        <alwaysOn>1</alwaysOn>

        <!-- Use GPS coordinates of Urbana, IL for origin of
    reference -->
        <!-- Ref. Altitude is height above WGS84 ellipsoid. NOT SEA
     LEVEL -->
        <referenceAltitude>-32.593</referenceAltitude>
        <referenceLatitude>40.1105875</referenceLatitude>
        <referenceLongitude>-88.2072697</referenceLongitude>
      </plugin>
    </gazebo>

  </xacro:macro>
</robot>
```

Listing A.2: Shows the URDF/*xacro* code that can be used to create a simple GPS sensor like the one shown in **Figure 2.12**.

A.2 Environment Development

A.2.1 Skid-Steering Plugin

```
<gazebo>
     <plugin name="skid_steer_drive_controller" filename="
    libgazebo_ros_skid_steer_drive.so">
          <updateRate>50.0</updateRate>
          <robotNamespace></robotNamespace>
```

```
            <leftFrontJoint>front_left_wheel_rev</leftFrontJoint>
            <rightFrontJoint>front_right_wheel_rev</rightFrontJoint>
            <leftRearJoint>rear_left_wheel_rev</leftRearJoint>
            <rightRearJoint>rear_right_wheel_rev</rightRearJoint>
            <wheelSeparation>0.4318</wheelSeparation>
            <topicName>cmd_vel</topicName>
            <robotBaseFrame>base_footprint</robotBaseFrame>
            <torque>10</torque>
            <broadcastTF>true</broadcastTF>
            <commandTopic>cmd_vel</commandTopic>
            <odometryTopic>odom</odometryTopic>
            <odometryFrame>/odom</odometryFrame>
            <covariance_x>0.0001</covariance_x>
            <covariance_y>0.0001</covariance_y>
            <covariance_yaw>0.01</covariance_yaw>
      </plugin>
</gazebo>
```

Listing A.3: XML snippet showing how the already developed skid-steering controller plugin can be attached to a simulated robot model in order to control that model using high-level motion commands.

A.2.2 Torsional Spring Plugin

```
<gazebo>
  <plugin name="spring_plugin_${prefix}_${suffix}" filename="
     libTorsionalSpringPlugin.so" >
    <joint_handle>${prefix}_${suffix}_leg_joint</joint_handle>
    <spring_stiffness>${stiffness}</spring_stiffness>
    <spring_damping>${damping}</spring_damping>
    <spring_reference>${spring_reference}</spring_reference>
    <upper_limit>${upper_limit}</upper_limit>
    <lower_limit>${lower_limit}</lower_limit>
    <verbose>${verbose}</verbose>
  </plugin>
</gazebo>
```

Listing A.4: Snippet of the XML code used to attach the developed torsional spring plugin to any simulated robot model's leg link using the custom *xacro* macro input parameters for modifying the associated dynamic spring variables.

A.2.3 Rotary Encoder Plugin

```
<xacro:macro name="sensor_encoder" params="ns:=/ parent joint:=
    front_left_wheel_rev d:=0.15 topic:=/encoder update_rate:=100.0
     ppr:=150">
  <link name="${parent}_encoder_link">
    <collision>
      <origin rpy="-0.02 0.03 -0.02" xyz="0 0 0"/>
      <geometry>
        <box size=".03 .04 .04"/>
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <mesh filename="package://ardee_sensors/meshes/
    rotary_encoder.dae"/>
      </geometry>
    </visual>

    <inertial>
      <mass value="1e-5" />
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1
    e-6" />
    </inertial>
  </link>

  <joint name="${parent}_encoder_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <parent link="${parent}"/>
    <child link="${parent}_encoder_link"/>
  </joint>


  <gazebo>
    <plugin name="${parent}_encoder_plugin" filename="
    libardee_ros_encoder.so">
      <robotNamespace>${ns}</robotNamespace>
      <joint>${joint}</joint>
      <wheelDiameter>${d}</wheelDiameter>
      <pulsesPR>${ppr}</pulsesPR>
      <encoderTopic>${topic}</encoderTopic>
      <updateRate>${update_rate}</updateRate>
    </plugin>
  </gazebo>
</xacro:macro>
```

Listing A.5: XML snippet used to add the developed simulated rotary encoder with custom parameters to a simulated object's joint.

A.2.4 Variable-Accuracy GPS Plugin

```
<?xml version="1.0" ?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <field_gps_noise_characteristics>
    <biomass_height_noise_coefficient>
      <r1>0</r1>
      <r2>.2</r2>
      <mean>1</mean>
      <variance>1</variance>
    </biomass_height_noise_coefficient>
    <biomass_height_noise_coefficient>
      <r1>.2</r1>
      <r2>.5</r2>
      <mean>1</mean>
      <variance>1.2</variance>
    </biomass_height_noise_coefficient>
    <biomass_height_noise_coefficient>
      <r1>.5</r1>
      <r2>1</r2>
      <mean>1</mean>
      <variance>1.5</variance>
    </biomass_height_noise_coefficient>
    <biomass_height_noise_coefficient>
      <r1>1</r1>
      <r2>2</r2>
      <mean>1</mean>
      <variance>4</variance>
    </biomass_height_noise_coefficient>
    <biomass_height_noise_coefficient>
      <r1>2</r1>
      <r2>-1</r2>
      <mean>1</mean>
      <variance>5</variance>
    </biomass_height_noise_coefficient>
  </field_gps_noise_characteristics>
```

```
</robot>
```

Listing A.6: Example XML file used to define the different GPS noise coefficients and the ranges in which they are effective.

A.2.5 Variable-Accuracy GPS Dropouts Config

```
<?xml version="1.0" ?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <plot>
    <plot_id>${plot_id}</plot_id>
    <plot_range_index>${plot_range_index}</plot_range_index>
    <plot_crossrange_index>${plot_crossrange_index}</
    plot_crossrange_index>
    <x>${x}</x>
    <y>${y}</y>
    <z>${z}</z>
    <rotation>${rotation}</rotation>
    <row_count>${row_count}</row_count>
    <row_width>${row_width}</row_width>
    <row_length>${row_length}</row_length>
    <average_height>${average_height}</average_height>
    <average_density>${plot_id}</average_density>
  </plot>
</robot>
```

Listing A.7: Example XML file used to define the different bounding regions where GPS dropouts, or accuracy reductions, occur, including the parameters associated with each region.

A.2.6 Simulated Battery Plugin

```
<gazebo>
  <plugin name="simulated_battery" filename="libardee_ros_battery.
    so">
    <updateRate>100.0</updateRate>
    <robotBaseFrame>base_footprint</robotBaseFrame>
    <batteryTopic>ardee/battery</batteryTopic>
    <batteryLink>main_battery</batteryLink>
    <batteryName>5200mAh_3S_LiPo</batteryName>
    <capacity>5.2</capacity>
    <initialCharge>5.2</initialCharge>
    <vMax>16.0</vMax>
    <vMin>2.0</vMin>
    <vInitial>14.0</vInitial>
    <deltaVoltage>0.01</deltaVoltage>
  </plugin>
</gazebo>
```

Listing A.8: Example XML snippet used to add the developed simulated battery plugin with custom parameters to a simulated robot.

A.2.7 Generic IMU Plugin

```
<xacro:sensor_imu name="imu" parent="base_footprint" color="red"
      update_rate="50.0"
      noise="0.005"
      yaw_offset="0.0"
      yaw_drift="0.02"
      yaw_noise="0.01"
      rate_offset="0.0 0.0 0.0"
      rate_drift="0.002 0.002 0.002"
      rate_noise="0.001 0.001 0.001"
      accel_offset="0.0 0.0 0.0"
      accel_drift="0.005 0.005 0.005"
      accel_noise="0.005 0.005 0.005">
  <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:sensor_imu>
```

Listing A.9: Example URDF xacro snippet used to add an IMU with custom parameters to a robot.

A.2.8 Example Specific Robot Base

```
<!-- Macro for creating the 2nd generation chassis used to attach
    all the other components -->
<xacro:macro name="chassis_v2" params="name stl_chassis stl_lid
    color">

     <!-- Create the chassis body link -->
     <link name="${name}">
          <!-- Configure the inertial body -->
```

```
        <inertial>
            <origin xyz="0 -0.01723 0.04522" rpy="0 0 0" />
            <mass value="7.227" />
            <inertia
                ixx="0.09243"
                ixy="0.00023"
                ixz="0"
                iyy="0.05913"
                iyz="0.00445"
                izz="0.146" />
        </inertial>

        <!-- Configure the visual body -->
        <visual>
            <origin xyz="-0.2325 0.264 -0.07325" rpy="1.5708 0 0
" />
            <geometry>
                <mesh filename="${stl_chassis}" scale="0.001
0.001 0.001"/>
            </geometry>
            <xacro:color_material color="${color}"/>
        </visual>

        <!-- Configure the collisional body -->
        <collision>
            <origin xyz="-0.2325 0.264 -0.07325" rpy="1.5708 0 0
" />
            <geometry>
                <mesh filename="${stl_chassis}" scale="0.001
0.001 0.001" />
            </geometry>
        </collision>
    </link>


    <!-- Create the lid body link to attach to the chassis body --
>
    <link name="lid_link">
        <!-- Configure the inertial body -->
        <inertial>
            <origin xyz="-0.00997 0.12337 0.07276" rpy="0 0 0" /
>
            <mass value="0.01" />
            <inertia
                ixx="0.0242"
                ixy="0"
```

```
                    ixz="0.00011"
                    iyy="0.03556"
                    iyz="0"
                    izz="0.01136" />
        </inertial>

        <!-- Configure the visual body -->
        <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
                <mesh filename="${stl_lid}" />
            </geometry>
            <xacro:color_material color="${color}"/>
        </visual>

        <!-- Configure the collisional body -->
        <collision>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
                <mesh filename="${stl_lid}" />
            </geometry>
        </collision>
    </link>

    <!-- Attach the lid link to the chassis with a fixed joint -->
    <joint name="lid_joint" type="fixed">
        <origin xyz="0.00974 0.08522 0.00018" rpy="1.5708 0 0" />
        <parent link="${name}" />
        <child link="lid_link" />
        <axis xyz="0 0 0" />
    </joint>

    <!-- Add friction to the chassis to keep it from sliding
   around in Gazebo -->
    <xacro:add_friction_gazebo parent="${name}" mu1="1.0" mu2="1.0
   " fdir1="0 0 0" />

    <!-- Customize the color of both the chassis and the lid as
   seen in Gazebo -->
    <xacro:color_gazebo parent="${name}" color="${color}"/>
    <xacro:color_gazebo parent="lid_link" color="${color}"/>

```

```
</xacro:macro>
```

Listing A.10: Shows an example implementation that users could use for the creation and later usage, of a custom robot chassis. Particularly, this shows the creation of the second-generation Terrasentia robot's base as a Gazebo simulated model.

A.2.9 Example Robot Base Configuration File

```
<?xml version="1.0" ?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">

    <!-- ====================================================================
                            STL Filepaths
    ==================================================================== -->
    <xacro:property name="chassis_stl" value="package://terrasentia_description/meshes/v1/base_link.STL"/>
    <xacro:property name="leg1_stl"    value="package://terrasentia_description/meshes/v1/leg_type_1.STL"/>
    <xacro:property name="leg2_stl"    value="package://terrasentia_description/meshes/v1/leg_type_2.STL"/>
    <xacro:property name="lid_stl"     value="package://terrasentia_description/meshes/v1/lid.STL"/>
    <xacro:property name="wheel_stl"   value="package://terrasentia_description/meshes/v1/wheel.STL"/>

    <!-- ====================================================================
                            Define Variables
    ==================================================================== -->
    <xacro:property name="WHEEL_RADIUS" value="0.097"/>
    <xacro:property name="WHEEL_DEPTH" value="0.03838"/>
    <xacro:property name="CHASSIS_SIZE" value="0.35322 0.508 0.13136"/>
    <xacro:property name="CHASSIS_OFFSET" value="0.04545"/>
    <xacro:property name="LID_SIZE" value="0.35322 0.508 0.015"/>
    <xacro:property name="LID_OFFSET" value="0.0"/>
    <xacro:property name="FOOTPRINT_OFFSET" value="0.175"/>

    <!-- ====================================================================
                            Suspension Properties
    ==================================================================== -->
    <xacro:property name="susp_stiffness" value="100"/>
    <xacro:property name="susp_damping" value="1"/>
    <xacro:property name="susp_reference" value="0"/>

    <!-- ===============================================================
                           Wheel Properties
    =============================================================== -->
    <xacro:property name="FR_WHEEL_FRICTION" value="0.75"/>
    <xacro:property name="FL_WHEEL_FRICTION" value="0.75"/>
    <xacro:property name="RR_WHEEL_FRICTION" value="0.75"/>
    <xacro:property name="RL_WHEEL_FRICTION" value="0.75"/>

    <!-- ===============================================================
                             Constants
    =============================================================== -->
    <xacro:property name="M_PI" value="3.1415926535897931" />

</robot>
```

Listing A.11: An example base configuration file, named `tsv1_config.urdf.xacro`, for the first-generation Terrasentia simulated robot base parameters.

A.2.10 Example Sensor Attachment Configuration File

```
<?xml version="1.0" ?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">

    <!-- ===============================================================
         Include Sensor Database for Available Sensors
    =============================================================== -->
    <xacro:include filename="$(find ardee_sensors)/urdf/
   all_sensors.urdf.xacro" />

    <!-- ===============================================================
                     Attach and Configure Sensors
    =============================================================== -->

    <!-- LiDAR -->
    <xacro:sensor_hokuyo_ust10lx name="hokuyo" parent="
   base_footprint" color="black">
         <origin xyz="0.22 0 0.173" rpy="0 0 0"/>
    </xacro:sensor_hokuyo_ust10lx>

    <!-- Front Camera -->
    <xacro:sensor_ueye_cp_gige name="camera" parent="
base_footprint" color="purple">
        <origin xyz="0.23 0 0.06" rpy="0 0 0"/>
    </xacro:sensor_ueye_cp_gige>

    <!-- Left Side Camera -->
    <xacro:sensor_ueye_cp_gige name="camera_left" parent="
base_footprint" color="purple">
        <origin xyz="0 0.16 0.06" rpy="0 0.349066 1.5708"/>
    </xacro:sensor_ueye_cp_gige>

    <!-- IMU -->
    <xacro:sensor_imu_ardee name="imu" parent="base_footprint"
color="red"
                update_rate="50.0"
                noise="0.005"
                yaw_offset="0.0"
                yaw_drift="0.02"
                yaw_noise="0.01"
                rate_offset="0.0 0.0 0.0"
                rate_drift="0.002 0.002 0.002"
                rate_noise="0.001 0.001 0.001"
                accel_offset="0.0 0.0 0.0"
                accel_drift="0.005 0.005 0.005"
                accel_noise="0.005 0.005 0.005">
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:sensor_imu_ardee>

    <!-- GPS w/ Variable Dropouts-->
    <xacro:sensor_gps_with_dropout name="gps" parent="
base_footprint" color="white"
                ref_lat="40.1021496545"
                ref_long="-88.2267974168"
                ref_head="90.0"
                update_rate="10.0"
                drift="0.0"
                noise="0.001"
                vel_drift="0.00001"
                vel_noise="0.00001">
        <origin xyz="-0.14 0 0.12" rpy="0 0 0"/>
    </xacro:sensor_gps_with_dropout>

    <!-- Simple Simulated Battery -->
    <gazebo>
        <plugin name="simulated_battery" filename="
libardee_ros_battery.so">
```

```
                <updateRate>100.0</updateRate>
                <robotNamespace></robotNamespace>
                <robotBaseFrame>base_footprint</robotBaseFrame>
                <commandTopic>cmd_vel</commandTopic>
                <batteryTopic>ardee/battery</batteryTopic>
                <batteryLink>main_battery</batteryLink>
                <batteryName>5200mAh_3S_LiPo</batteryName>
                <constantCoef>12.694</constantCoef>
                <linearCoef>-100.1424</linearCoef>
                <initialCharge>5.2</initialCharge>
                <capacity>5.2</capacity>
                <internalResistance>0.061523</internalResistance>
                <smoothCurrentTau>1.9499</smoothCurrentTau>
                <vMax>16.0</vMax>
                <vMin>2.0</vMin>
                <vInitial>14.0</vInitial>
                <deltaVoltage>0.01</deltaVoltage>
            </plugin>
    </gazebo>

</robot>
```

Listing A.12: An example configuration file, named `tsv1_sensors_config.urdf.xacro`, for all the simulated sensors attached to the first-generation Terrasentia simulated robot base.

A.2.11 Example Specific Robot Model Assembly URDF

```
<?xml version="1.0"?>
<robot name="TSv1" xmlns:xacro="http://ros.org/wiki/xacro">

    <!-- ==============================================================
                            Includes for Components
    ============================================================== -->
    <xacro:include filename="$(find terrasentia_description)/urdf/links_v1.urdf.xacro"/>
    <xacro:include filename="$(find terrasentia_description)/urdf/joints_v1.urdf.xacro"/>

  <!-- ==============================================================
       Reference the Robot Configuration Files
  ============================================================== -->

  <!-- Robot base configuration -->
```

```
<xacro:include filename="$(find terrasentia_description)/config/
  tsv1_config.urdf.xacro"/>
<!-- Attached sensors configuration -->
   <xacro:include filename="$(find terrasentia_description)/
  config/tsv1_sensors_config.urdf.xacro"/>

   <!-- ================================================
        Create each individual body link component
   ================================================== -->

<!-- Create the chassis which will act as the single main body
  link to attach everything else -->
<xacro:chassis_v0 name="base_footprint" stl="${CHASSIS_STL}" size
  ="${CHASSIS_SIZE}" z_offset="${CHASSIS_OFFSET}" color="grey"/>

<!-- Creates the lid body -->
<xacro:lid stl="${LID_STL}" size="${LID_SIZE}" z_offset="${
  LID_OFFSET}" color="grey"/>

<!-- Creates each leg body -->
   <xacro:leg_type_1 prefix="front" suffix="right" stl="${
  leg1_stl}" color="orange"/>
   <xacro:leg_type_1 prefix="rear" suffix="left" stl="${leg1_stl}
  " color="orange"/>
   <xacro:leg_type_2 prefix="front" suffix="left" stl="${leg2_stl
  }" color="orange"/>
   <xacro:leg_type_2 prefix="rear" suffix="right" stl="${leg2_stl
  }" color="orange"/>

<!-- Create each wheel body -->
   <xacro:wheel prefix="front" suffix="right" stl="${wheel_stl}"
  color="grey"
        radius="${WHEEL_RADIUS}" depth="${WHEEL_DEPTH}" reflect="
  1" friction="${FR_WHEEL_FRICTION}" scale="0.0254">
        <origin xyz="0 -${WHEEL_DEPTH} 0" rpy="0 0 0" />
   </xacro:wheel>

   <xacro:wheel prefix="front" suffix="left" stl="${wheel_stl}"
  color="grey"
        radius="${WHEEL_RADIUS}" depth="${WHEEL_DEPTH}" reflect="
  -1" friction="${FL_WHEEL_FRICTION}" scale="0.0254">
        <origin xyz="0 ${WHEEL_DEPTH} 0" rpy="0 0 -3.154" />
   </xacro:wheel>

   <xacro:wheel prefix="rear" suffix="right" stl="${wheel_stl}"
  color="grey"
```

```
        radius="${WHEEL_RADIUS}" depth="${WHEEL_DEPTH}" reflect="
    1" friction="${RR_WHEEL_FRICTION}" scale="0.0254">
        <origin xyz="0 -${WHEEL_DEPTH} 0" rpy="0 0 0" />
    </xacro:wheel>

    <xacro:wheel prefix="rear" suffix="left" stl="${wheel_stl}"
    color="grey"
        radius="${WHEEL_RADIUS}" depth="${WHEEL_DEPTH}" reflect="
    -1" friction="${RL_WHEEL_FRICTION}" scale="0.0254">
        <origin xyz="0 ${WHEEL_DEPTH} 0" rpy="0 0 -3.154" />
    </xacro:wheel>

    <!-- ===================================================
        Attach each individual link to a single link
    ========================================================== -->

  <!-- Attach the lid to the chassis -->
  <xacro:attach_lid parent="base_footprint">
        <origin xyz="0.23525 0.0214 -0.2085" rpy="0 0 0"/>
    </xacro:attach_lid>

  <!-- Attach the legs to the chassis -->
    <xacro:attach_leg_type_1 parent="base_footprint" link="
    front_right_leg" reflect="-1" min="-0.682" max="0" stiffness="
    ${susp_stiffness}" damping="${susp_damping}" reference="${
    susp_reference}">
        <axis xyz="0 0 -1"/>
    </xacro:attach_leg_type_1>

    <xacro:attach_leg_type_1 parent="base_footprint" link="
    rear_left_leg" reflect="1" min="-0.682" max="0" stiffness="${
    susp_stiffness}" damping="${susp_damping}" reference="${
    susp_reference}">
        <axis xyz="0 0 -1"/>
    </xacro:attach_leg_type_1>

    <xacro:attach_leg_type_2 parent="base_footprint" link="
    front_left_leg" reflect="1" min="0" max="0.682" stiffness="${
    susp_stiffness}" damping="${susp_damping}" reference="${
    susp_reference}">
        <axis xyz="0 0 -1"/>
    </xacro:attach_leg_type_2>

    <xacro:attach_leg_type_2 parent="base_footprint" link="
    rear_right_leg" reflect="-1" min="-0.682" max="0" stiffness="${
    susp_stiffness}" damping="${susp_damping}" reference="${
```

```
    susp_reference}">
        <axis xyz="0 0 1"/>
    </xacro:attach_leg_type_2>

  <!-- Attach the wheels to each leg -->
    <xacro:attach_wheel_leg prefix="front" suffix="right">
        <origin xyz="0.1601 0.023246 0.0254775" rpy="-1.5708 0
    0.95382" />
        <axis xyz="0 1 0"/>
    </xacro:attach_wheel_leg>

    <xacro:attach_wheel_leg prefix="front" suffix="left">
        <origin xyz="-0.1601 0.023246 0.0254775" rpy="1.5708 0
    2.18777" />
        <axis xyz="0 1 0"/>
    </xacro:attach_wheel_leg>

    <xacro:attach_wheel_leg prefix="rear" suffix="right">
        <origin xyz="-0.1601 0.023246 0.0254775" rpy="-1.5708 0
    -0.95382" />
        <axis xyz="0 1 0"/>
    </xacro:attach_wheel_leg>

    <xacro:attach_wheel_leg prefix="rear" suffix="left">
        <origin xyz="0.1601 0.023246 0.0254775" rpy="1.5708 0
    4.09542" />
        <axis xyz="0 1 0"/>
    </xacro:attach_wheel_leg>

    <!-- ============================================================
      Skid-Steering Controller used for Robot Motion Control
    ================================================================
    -->
    <gazebo>
        <plugin name="skid_steer_drive_controller" filename="
    libgazebo_ros_skid_steer_drive.so">
            <updateRate>50.0</updateRate>
            <robotNamespace></robotNamespace>
            <leftFrontJoint>front_left_wheel_rev</leftFrontJoint
    >
            <rightFrontJoint>front_right_wheel_rev</
    rightFrontJoint>
            <leftRearJoint>rear_left_wheel_rev</leftRearJoint>
            <rightRearJoint>rear_right_wheel_rev</rightRearJoint
    >
            <wheelSeparation>0.4318</wheelSeparation>
```

```
                <wheelDiameter>0.194</wheelDiameter>
                <topicName>cmd_vel</topicName>
                <robotBaseFrame>base_footprint</robotBaseFrame>
                <torque>10</torque>
                <broadcastTF>true</broadcastTF>
                <commandTopic>cmd_vel</commandTopic>
                <odometryTopic>odom</odometryTopic>
                <odometryFrame>/odom</odometryFrame>
                <covariance_x>0.0001</covariance_x>
                <covariance_y>0.0001</covariance_y>
                <covariance_yaw>0.01</covariance_yaw>
            </plugin>
        </gazebo>
</robot>
```

Listing A.13: An example robot model assembly file, named `tsv1.urdf.xacro`, used to assemble all of the components and sensors necessary to represent, configure, and simulate the first-generation Terrasentia platform shown in **Figure 4.17**.

A.2.12 OGRE Material Script

```
material Corn/LeavesBase
{
  technique
  {
    pass
    {
      alpha_rejection greater 128

      texture_unit
      {
        texture AG20lef3.tif
      }
      cull_hardware none
      cull_software none
    }
  }
}
```

Listing A.14: A snippet from the material script used to fix rendering issues for the model shown in **Figure 4.2**.

A.2.13 Farm Gazebo World File

```
<?xml version="1.0" ?>
<sdf version='1.5'>
     <world name='default'>

    <!-- When Gazebo is loaded zoom into where the robot spawns -->
          <gui>
               <camera name="user_camera">
                    <pose>1.953330 -2.960521 2.117045 0 0.411456
    1.892190</pose>
               </camera>
          </gui>

    <!-- Initialize some lighting condtions -->
          <light name='sundir' type='directional'>
               <cast_shadows>1</cast_shadows>
               <pose>0 0 10 0 -0 0</pose>
               <diffuse>1.0 0.95 0.8 1</diffuse>
               <specular>0.7 0.7 0.7 1</specular>
               <attenuation>
                    <range>1000</range>
                    <constant>0.9</constant>
                    <linear>0.01</linear>
                    <quadratic>0.001</quadratic>
               </attenuation>
               <direction>-0.3 0.4 -1.0</direction>
          </light>

    <!-- Configure ODE Physics engine parameters -->
          <physics type="ode">
               <max_step_size>0.005</max_step_size>
               <real_time_factor>1.0</real_time_factor>
               <real_time_update_rate>0.0</real_time_update_rate>
               <gravity>0 0 -9.8</gravity>
          </physics>

    <!-- Configure all of the scene parameters -->
          <scene>
      <!-- Use a blue sky with moving clouds  -->
               <sky>
                    <clouds>
                         <speed>12</speed>
                    </clouds>
               </sky>
    <!-- Show the grid on the floor -->
            <grid>true</grid>
        </scene>

    <!-- Use the python-script generated heightmap model -->
        <include>
            <uri>model://heightmap_ground</uri>
            <pose>0 0 0 0 0 0</pose>
        </include>

    <!-- Use an early-season crop plot model generated using python
    -script -->
        <include>
            <uri>model://small_corn_stalk_plot</uri>
            <pose>0 0 0 0 0 -1.57</pose>
        </include>
    </world>
</sdf>
```

Listing A.15: An example world assembly file, named **`farm.world`**, used to assemble all of the components necessary to represent and configure the simulated world shown in **Figure 4.23**.

A.3 UDP Communication

A.3.1 ARDEE Message Header

```
#pragma once

typedef void ArdeeMessage;

typedef struct Ardee_Msg_Data_Header{
    union{
        int32_t component_id;
        int32_t header;
    };
    int32_t msg_type;
    int32_t data_type;
    int32_t measurement_type;
    int32_t measurement_length;
}Ardee_Msg_Data_Header;

typedef enum ArdeeMessageType{
  ARDEE_MESSAGE_SET_PORT = 0,
  ARDEE_MESSAGE_CONTROL = 1,
  ARDEE_MESSAGE_DATA_GPS = 2,
  ARDEE_MESSAGE_DATA_GYRO = 3,
  ARDEE_MESSAGE_DATA_ACCELERATION = 4,
  ARDEE_MESSAGE_DATA_RCCOMMANDS = 5,
  ARDEE_MESSAGE_DATA_LIDAR = 6,
  ARDEE_MESSAGE_DATA_ORIENTATION = 7,
  ARDEE_MESSAGE_DATA_ENCODER = 8,
  ARDEE_MESSAGE_DATA_FRONT_CAM = 9,
  ARDEE_MESSAGE_DATA_IMU = 9,
  ARDEE_MESSAGE_DATA_BATTERY = 10,
}ArdeeMessageType;

typedef enum ArdeeDataType{
  ARDEE_DATA_NONE = 0,
  ARDEE_DATA_ALL = 1,
  ARDEE_DATA_GPS = 2,
  ARDEE_DATA_GYRO = 3,
  ARDEE_DATA_ACCELERATION = 4,
  ARDEE_DATA_LIDAR = 5,
  ARDEE_DATA_ORIENTATION = 6,
  ARDEE_DATA_ENCODER = 7,
  ARDEE_DATA_FRONT_CAM = 8,
  ARDEE_DATA_IMU = 9,
  ARDEE_DATA_BATTERY = 10,
  num_ArdeeDataTypes = 11,
}ArdeeDataType;

typedef enum ArdeeMeasurementType{
      ARDEE_MEASUREMENT_SINGLE       = 0,
      ARDEE_MEASUREMENT_MULTIPLE     = 1,
}ArdeeMeasurementType;

typedef enum ArdeeControlCode{
  ARDEE_CONTROL_STOP = 0,
  ARDEE_CONTROL_START = 1,
  ARDEE_CONTROL_PAUSE = 2,
  ARDEE_CONTROL_RESUME = 3,
}ArdeeControlCode;
```

Listing A.16: Shows the ARDEE standardized data structures defined in C++ for an *ARDEE Message Header* packet.

A.3.2 ARDEE Message Data

```
#pragma once

/** SECTION:
      BASIC DATA
*/
typedef struct XYZ_INT{
  int32_t x;
  int32_t y;
  int32_t z;
} XYZ_BASE_INT;

typedef struct ORIENTATION_QUATERNION_BASE_INT{
  int32_t x;
  int32_t y;
  int32_t z;
  int32_t w;
} ORIENTATION_QUATERNION_BASE_INT;

typedef struct ORIENTATION_EULER_BASE_INT{
  int32_t roll;
  int32_t pitch;
  int32_t yaw;
} ORIENTATION_EULER_BASE_INT;

/** SECTION:
      INTERMEDIATE DATA
*/
typedef struct MOTION_DATA_INT{
  int32_t x;
  int32_t y;
  int32_t z;
  XYZ_INT covariance;
  XYZ_INT bias;
  XYZ_INT offset;
} XYZ_DATA_INT;

typedef struct Ardee_Msg_Ack{
  int32_t sequence;
}Ardee_Msg_Ack;

typedef struct Ardee_Msg_OrientationData{
  int32_t x;
  int32_t y;
  int32_t z;
  int32_t w;
  ORIENTATION_EULER_BASE_INT covariance;
  ORIENTATION_QUATERNION_BASE_INT bias;
} Ardee_Msg_OrientationData;

/** SECTION:
     SENSOR DATA TYPES
*/
typedef struct Ardee_Msg_IMUData{
  MOTION_DATA_INT accel;
  MOTION_DATA_INT gyro;
  Ardee_Msg_OrientationData orientation;
}Ardee_Msg_IMUData;

typedef struct Ardee_Msg_GPSData{
  int32_t time;
  int32_t latitude;
  int32_t longitude;
  int32_t altitude;
  XYZ_INT velocity;
  XYZ_INT covariance;
  uint8_t covariance_type;
  uint16_t service;
  int8_t status;
}Ardee_Msg_GPSData;

typedef struct Ardee_Msg_GyroData{
  int32_t x;
  int32_t y;
  int32_t z;
}Ardee_Msg_GyroData;

typedef struct Ardee_Msg_AccelerationData{
  int32_t x;
  int32_t y;
  int32_t z;
}Ardee_Msg_AccelerationData;

typedef struct Ardee_Msg_MotionCommands{
  int32_t normalized_speed;
  int32_t normalized_yaw_rate;
  int32_t multiplier;
}Ardee_Msg_MotionCommands;

typedef struct Ardee_Msg_LidarData{
  int32_t angle_min;
  int32_t angle_max;
  int32_t dAngle;
  int32_t scan_time;
  int32_t dTime;
  int32_t range_min;
  int32_t range_max;
  int32_t ranges[1081];
  int32_t intensities[1081];
}Ardee_Msg_LidarData;

typedef struct Ardee_Msg_EncoderData{
  int32_t id;
  int32_t position;
  int32_t speed;
  int32_t qpps;
}Ardee_Msg_EncoderData;

typedef struct Ardee_Msg_BatteryData{
  int32_t voltage;
  int32_t vmax;
  int32_t vmin;
}Ardee_Msg_BatteryData;
```

Listing A.17: Shows the ARDEE standardized data structures defined in C++ for an *ARDEE Message Data* packet.

A.3.3 Sensor Server Node

```
#include <ros/ros.h>
#include "../include/ardee_sensor_relay.h"

using namespace std;
using namespace ardee;

int main (int argc, char** argv){

  // Initialize the ROS node
  ros::init(argc, argv, "ardee_sensor_relay_node");
  ros::NodeHandle nh;

  // Configure UDP-specific parameters
  char* ip = "192.168.1.135";
  int recPort = 35555;
```

```

// Create the SensorRelay object class
TerraSensorRelay relay(nh, 30000,ip);

// Configure the simulated sensor ROS topics to listen for and
  send over UDP
const char* imuTopic  = "/ardee/imu/custom";  // Simulated IMU
const char* gpsTopic  = "/ardee/gps";   // Simulated GPS
const char* lidarTopic  = "/ardee/hokuyo";    // Simulated Lidar
const char* batTopic  = "/ardee/battery";   // Simulated Battery
const char* encFrTopic  = "/ardee/encoder/fr";  // Simulated
  Front-Right Encoder
const char* encFlTopic  = "/ardee/encoder/fl";  // Simulated
  Front-Left Encoder
const char* encRrTopic  = "/ardee/encoder/rr";  // Simulated Rear
  -Right Encoder
const char* encRlTopic  = "/ardee/encoder/rl";  // Simulated Rear
  -Left Encoder

// Add the simulated IMU to the sensor relay
relay.addRosSensor("Hector IMU",imuTopic, SIMULATOR_DATA_IMU);

// Add the simulated GPS to the sensor relay
relay.addRosSensor("GPS", gpsTopic, SIMULATOR_DATA_GPS);

// Add the simulated Lidar to the sensor relay
relay.addRosSensor("Hokyu Lidar", lidarTopic,
  SIMULATOR_DATA_LIDAR);

// Add the simulated Battery to the sensor relay
relay.addRosSensor("Main Battery", batTopic,
  SIMULATOR_DATA_BATTERY);

// Add the simulated motor encoders to the sensor relay
relay.addRosSensor("Front Right Encoder", encFrTopic,
  SIMULATOR_DATA_ENCODER);
relay.addRosSensor("Front Left Encoder", encFlTopic,
  SIMULATOR_DATA_ENCODER);
relay.addRosSensor("Rear Right Encoder", encRrTopic,
  SIMULATOR_DATA_ENCODER);
relay.addRosSensor("Rear Left Encoder", encRlTopic,
  SIMULATOR_DATA_ENCODER);

// Begin the SensorRelay node
relay.run();
```

```
}
```

Listing A.18: Shows the code, created in `ardee_sensor_server_node.cpp`, for configuring various aspects associated with the developed "sensor relay" module, such as the specific UDP port and IP address to broadcast UDP packets to and what simulated sensor measurements to broadcast.

A.3.4 Command Client Node

```
#include "../include/control/ros_rc_controller.h"

using namespace std;

int main (int argc, char** argv){
  // Initialize the ROS node
  ros::init(argc, argv, "ardee_rc_controller_node");
  ros::NodeHandle nh;
  ros::NodeHandle _nh("~");

  // Configure the limits of the simulated robot controls (should
    be based on physical robot's limits)
  float max_vel = 1.2;        // Max Linear Velocity [m/s]
  float max_turn_radius = 0.381;  // Max Turn Radius [m]

  // Create the controls relay class object and start it
  RosRcController ss(nh,_nh,max_vel,max_turn_radius);
  ss.run();
}
```

Listing A.19: Shows the code, created in `ardee_controls_client_node.cpp`, which configures the controls relay node in charge of receiving any broadcasted robot commands over UDP generated by any developed component (i.e developed algorithm or anything else) running on an external device, and re-packages into a ROS format to control the simulated robot model in Gazebo.

A.4 Launch Utilities

A.4.1 Simple roslaunch file

```
<?xml version="1.0"?>
<launch>

    <!-- Launch configuration settings -->
    <arg name="world" default="custom.world"/>
    <arg name="gui" default="false" />

    <!-- Set parameters to ROS parameter server -->
    <param name="use_gui" value="$(arg gui)" />

    <!-- Load Gazebo world -->
    <include file="$(find ardee_world)/launch/world_template.
  launch">
        <arg name="world_name" value="$(find ardee_world)/worlds
  /$(arg world)"/>
        <arg name="paused" value="true"/>
        <arg name="gui" value="true"/>
    </include>

</launch>
```

Listing A.20: Shows the launch file, `custom_world.launch`, used to load a custom world in Gazebo, defined by the `custom.world` file. It should be noted that the `world_template.launch` file is just a copy of the open-source launch file `empty_world.launch` available from the `gazebo_ros` ROS package [20]. Additionally, the *.world* file can be changed on-the-fly by passing a command-line argument "world" with a value of the *.world* file a user wants to use.

A.4.2 Dynamically Configurable roslaunch Example

```
<?xml version="1.0"?>
<launch>

    <!-- ==============================
           Parameter Configuration
    ============================== -->

    <!-- Flags -->
    <arg name="use_joy" default="true"/>
    <arg name="use_xbox" default="true"/>
    <arg name="use_rviz" default="true"/>

    <!-- Robot pose-->
```

```
<arg name="x" default="0"/>
<arg name="y" default="0"/>
<arg name="z" default="0.175"/>
<arg name="roll" default="0"/>
<arg name="pitch" default="0"/>
<arg name="yaw" default="-3.14"/>

<!-- Robot Model configuration settings -->
<arg name="bot_model" default="default_bot.urdf.xacro" />
<param name="robot_description" command="$(find xacro)/xacro
$(find terrasentia_description)/robots/$(arg bot_model)" />

<!-- Robot Control Configuration Parameters -->
<arg name="dev_joy" default="/dev/input/js4"/>

<!-- ==============================
        Initialize ROS Nodes
==================================== -->

<!-- Necessary for keeping track of the simulated robot model'
s state -->
<node pkg="robot_state_publisher" type="state_publisher" name
="robot_state_publisher">
      <param name="publish_frequency" type="double" value
="200.0"/>
</node>

<!-- Necessary for using the simulated robot model's joint
states kept track by Gazebo -->
<node name="joint_state_publisher" pkg="joint_state_publisher"
 type="joint_state_publisher">
      <param name="rate" type="double" value="50.0"/>
</node>

<!-- Spawn the specified robot model in Gazebo with user-
defined pose -->
<node name="spawn_terrasentia" pkg="gazebo_ros" type="
spawn_model"
      args="-x $(arg x) -y $(arg y) -z $(arg z) -R $(arg roll)
     -P $(arg pitch) -Y $(arg yaw) -urdf -param
robot_description -model TerraSentia_0"
      output="screen"
/>

<!-- Load joystick nodes for manual control -->
<group if="$(arg use_joy)">
```

```

        <!-- Required for receiving joystick inputs -->
        <node pkg="joy" type="joy_node" name="joy_node">
            <param name="dev" value="$(arg dev_joy)"/>
        </node>

        <!-- Xbox Controller Configuration -->
        <group if="$(arg use_xbox)">
            <node pkg="teleop_twist_joy" type="teleop_node" name
="joy_controller_node">
                <param name="enable_button" value="7"/>
                <param name="axis_angular" value="2"/>
                <param name="scale_linear" value="1"/>
                <param name="scale_angular" value="1.5"/>
            </node>
        </group>

        <!-- PS3 Controller Configuration -->
        <group unless="$(arg use_xbox)">
            <node pkg="teleop_twist_joy" type="teleop_node" name
="joy_controller_node">
                <param name="enable_button" value="11"/>
                <param name="axis_angular" value="2"/>
                <param name="scale_linear" value="1"/>
                <param name="scale_angular" value="1.5"/>
            </node>
        </group>
    </group>

    <!-- Load UDP controls relay module for external device
control if not using manual control -->
    <group unless="$(arg use_joy)">
        <node pkg="ardee_bridge" type="ardee_rc_controller_node"
name="udp_control_node"/>
    </group>

    <!-- Start RViz for visualization -->
    <group if="$(arg use_rviz)">
        <node name="rviz" pkg="rviz" type="rviz"
              args="-d $(find terrasentia_description)/config/
rviz/waypoint_follower_paths.rviz" />
    </group>

    <!-- Broadcast simulated sensor measurements for external
devices -->
```

```
    <node pkg="ardee_sensors" type="ardee_sensor_relay_node" name=
   "sim_sensor_node"/>

</launch>
```

Listing A.21: Shows the launch file, `launch_default_bot.launch`, used for the following: (1) configure and load the default robot model, defined by the `default_bot.urdf.xacro` file, simulated in Gazebo, (2) configure and execute the necessary nodes for the user-defined method of controlling the simulated robot, (3) starting the sensor relay node for broadcasting the simulated sensor measurements over UDP, and (4) starting Rviz for visualization of simulated sensors and other important information (i.e estimated pose, etc.).

A.4.3 Example bash/shell script

```
#!/bin/bash
trap ctrl_c INT

function ctrl_c() {
        echo "------------ CTRL-C Triggered [EXITING] -------------"
     echo
     echo
     echo "      Shutting Down Gazebo Processes..."
     echo
     echo
     killall gzserver
     killall gzclient
     echo
     echo
     echo "*********** Simulation Exited **************"
     exit
}


#
   ################################################################################
# Kill any pre-existing Gazebo processes for proper re-
    initialization
#
   ################################################################################
```

```
killall gzserver
killall gzclient


#
    ##############################################################################
#       User input argument parsing
#
    ##############################################################################
if [ -z "$1" ]; then
  field=ardee_world/urdf/empty.urdf.xacro
else
  field=$1
fi
if [ -z "$2" ]; then
  heightmap=ardee_models/heightmap_flat
else
  heightmap=$2
fi
if [ -z "$3" ]; then
  gps_noise=ardee_geofence/default_gps_behavior.geofence
else
  gps_noise=$3
fi

echo "hmmm $field $heightmap $gps_noise"

here=$(pwd)

. /opt/ros/kinetic/setup.bash
. $HOME/catkin_ws/devel/setup.bash
. $HOME/theHuntingGround/devel/setup.bash

export GAZEBO_MODEL_PATH=${GAZEBO_MODEL_PATH}:"$here"/ardee_models:
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:"$here"/
    terrasentia_description/plugins/build:$HOME/catkin_ws/devel/lib
    :
export GAZEBO_RESOURCE_PATH=${GAZEBO_RESOURCE_PATH}:"$here"/
    ardee_models:"$here"/ardee_geofence:

#
    ##############################################################################
#   Generate Custom .world file with user-specified models
```

```
#
    ############################################################################
printf '<?xml version="1.0" ?>
<sdf version='1.5'>
     <world name='default'>

          <gui>
               <camera name="user_camera">
                    <pose>1.953330 -2.960521 2.117045 0 0.411456
    1.892190</pose>
               </camera>
          </gui>

          <light name='sundir' type='directional'>
               <cast_shadows>1</cast_shadows>
               <pose>0 0 10 0 -0 0</pose>
               <diffuse>1.0 0.95 0.8 1</diffuse>
               <specular>0.7 0.7 0.7 1</specular>
               <attenuation>
                    <range>1000</range>
                    <constant>0.9</constant>
                    <linear>0.01</linear>
                    <quadratic>0.001</quadratic>
               </attenuation>
               <direction>-0.3 0.4 -1.0</direction>
          </light>

          <physics type="ode">
               <max_step_size>0.005</max_step_size>
               <real_time_factor>1.0</real_time_factor>
               <real_time_update_rate>0.0</real_time_update_rate>
               <gravity>0 0 -9.8</gravity>
          </physics>

          <scene>
               <sky>
                    <clouds>
                         <speed>12</speed>
                    </clouds>
               </sky>

               <grid>true</grid>
          </scene>

          <include>
```

```
                <uri>model://custom_heightmap</uri>
                <pose>0 0 0 0 0 0</pose>
            </include>

            <include>
                <uri>model://custom_field</uri>
                <pose>0 0 0 0 0 0</pose>
            </include>

        </world>
</sdf>' > ardee_world/worlds/custom.world


# ######################################################################################
#
# ######################################################################################
rm -rf ardee_models/custom_heightmap
rm -rf ardee_models/custom_field

mkdir ardee_models/custom_heightmap
mkdir ardee_models/custom_field

cp -rf "$heightmap"/* ardee_models/custom_heightmap

cp "$field" temp_plots
cp "$field" temp_plants
end=$(grep -n "123456789end123456789" temp_plots | grep -Eo '^[^:]+')
line="$(($end+1))"
echo "el $end"
cmd=\'1,"$end"d\'
echo "$cmd"
cmd="sed -e $cmd < temp_plots > temp_plots2"
echo "$cmd"
eval "$cmd"

cmd=\'"$end",'$d'\'
echo "$cmd"
cmd="sed -e $cmd < temp_plants > temp_plants2"
echo "$cmd"
eval "$cmd"

```

```
cp temp_plants2 ardee_models/custom_field/model.sdf
cp temp_plots2 ardee_models/custom_field/plots.xacro
cp "$3" ardee_geofence/custom_gps_behavior.geofence

rm temp_plants
rm temp_plots
rm temp_plants2
rm temp_plots2


# ################################################################################
#       Generate Custom Field Model Directory
# ################################################################################
printf '<?xml version="1.0"?>

<model>
    <name>Custom Field</name>
    <version>1.0</version>
    <sdf version='1.5'>model.sdf</sdf>

    <author>
        <name>Hunter Young</name>
        <email>hunter.lw.young@gmail.com</email>
    </author>

    <description>
        Field of customized crop plots generated by run_world.sh
    </description>
</model>
' > ardee_models/custom_field/model.config

roslaunch ardee_world custom_world.launch &

sleep 10

read -n1 -p "Do you want to continue? Enter (y) or (n)" doit

if [[ $doit == "Y" || $doit == "y" ]]; then
    roslaunch terrasentia_description launch_default_ts0.launch
else
    killall gzserver
    killall gzclient
```

```
fi
sleep 5
echo
echo "Simulation Exited"
```

Listing A.22: Shows the bash script used, `run_demo_world.sh`, used for the following: (1) overlaying the necessary shell environment for proper integration of ROS system, (2) setting up the necessary environmental variables for proper usage of built custom plugins and proper loading and rendering of developed models (discussed in **Section 4.2**), (3) simplistic commandline UI elements for interacting with the user during the process of loading the pre-determined simulation environment, and (4) ensuring that the Gazebo-specific processes are properly terminated before loading of the simulation environment, as well as upon exiting simulation upon user request (via CTRL+C).

A.5 Miscellaneous

A.5.1

```
<sdf version="1.5">
    <model name="heightmap">
        <static>true</static>
        <link name="link">
            <collision name="collision">
                <geometry>
                    <heightmap>
                        <uri>file://heightmap_ground/meshes/
    heightmaps/corn_rows.png</uri>
                        <size>30 30 10</size>
                        <pos>0 0 0</pos>
                    </heightmap>
                </geometry>

                <surface>
                    <contact>
                        <ode>
                            <kp>100000.0</kp>
                            <kd>10000.0</kd>
                        </ode>
                    </contact>
                </surface>
```

```
      </collision>

      <visual name="visual">
        <geometry>
          <heightmap>
            <use_terrain_paging>false</use_terrain_paging>
            <texture>
              <diffuse>file://media/materials/textures/dirt_diffusespecular.png</diffuse>
              <normal>file://media/materials/textures/flat_normal.png</normal>
              <size>1</size>
            </texture>
            <texture>
              <diffuse>file://media/materials/textures/grass_diffusespecular.png</diffuse>
              <normal>file://media/materials/textures/flat_normal.png</normal>
              <size>1</size>
            </texture>
            <texture>
              <diffuse>file://media/materials/textures/fungus_diffusespecular.png</diffuse>
              <normal>file://media/materials/textures/flat_normal.png</normal>
              <size>1</size>
            </texture>
            <blend>
              <min_height>0.5</min_height>
              <fade_dist>1</fade_dist>
            </blend>
            <blend>
              <min_height>0.5</min_height>
              <fade_dist>1</fade_dist>
            </blend>
            <uri>file://heightmap_ground/meshes/heightmaps/corn_rows.png</uri>
            <size>30 30 10</size>
            <pos>0 0 0</pos>
          </heightmap>
        </geometry>
      </visual>
    </link>
 </model>
```

```
</sdf>
```

Listing A.23: The *model.sdf* file used for the simulated field terrain as seen in **Figure 4.8**.

A.5.2

```
<sdf version="1.5">
    <model name="corn_variant_1">
        <static>true</static>
        <link name="link">
            <inertial>
                <pose>0 0 0 0 0 0</pose>
                <mass>4</mass>
                <inertia>
                    <ixx>0.01</ixx>
                    <ixy>0</ixy>
                    <ixz>0</ixz>
                    <iyy>0.01</iyy>
                    <iyz>0</iyz>
                    <izz>0.01</izz>
                </inertia>
            </inertial>

            <collision name='collision'>
                <pose>0 0 0.25 0 0 0</pose>
                <geometry>
                    <cylinder>
                        <radius>0.0075</radius>
                        <length>0.5</length>
                    </cylinder>
                </geometry>
                <max_contacts>10</max_contacts>
            </collision>

            <visual name="stem">
                <geometry>
                    <mesh>
                        <uri>model://corn_variant_1/meshes/
    corn_variant_1.dae</uri>
                        <submesh>
                            <name>Stem</name>
                        </submesh>
                    </mesh>
                </geometry>
                <material>
                    <script>
                        <uri>model://corn_variant_1/materials
/scripts/</uri>
                        <uri>model://corn_variant_1/materials
/textures/</uri>
                        <name>Corn/Stem</name>
                    </script>
                </material>
            </visual>

            <visual name="leavesb">
                <geometry>
                    <mesh>
                        <uri>model://corn_variant_1/meshes/
corn_variant_1.dae</uri>
                        <submesh>
                            <name>LeavesBase</name>
                        </submesh>
                    </mesh>
                </geometry>
                <material>
                    <script>
                        <uri>model://corn_variant_1/materials
/scripts/</uri>
                        <uri>model://corn_variant_1/materials
/textures/</uri>
                        <name>Corn/LeavesBase</name>
                    </script>
                </material>
            </visual>

            <visual name="leavesm">
                <geometry>
                    <mesh>
                        <uri>model://corn_variant_1/meshes/
corn_variant_1.dae</uri>
                        <submesh>
                            <name>LeavesMid</name>
                        </submesh>
                    </mesh>
                </geometry>
                <material>
                    <script>
```

```
                        <uri>model://corn_variant_1/materials/scripts/</uri>
                        <uri>model://corn_variant_1/materials/textures/</uri>
                        <name>Corn/LeavesMid</name>
                    </script>
                </material>
            </visual>

        </link>
    </model>
</sdf>
```

Listing A.24: The *model.sdf* file used for one of the simulated corn models.

A.5.3

```
<?xml version="1.0" ?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">

    <!-- ================================================================
                    Include xacro:macro Utilities
    ================================================================ -->
    <xacro:include filename="$(find terrasentia_description)/urdf/includes/materials.urdf.xacro"/>
    <xacro:include filename="$(find terrasentia_description)/urdf/includes/utils.urdf.xacro"/>

  <!-- ================================================================
                    Leg Link Type 1: (FR and RL legs)
    ================================================================ -->
    <xacro:macro name="leg_type_1" params="prefix suffix stl min max damping stiffness spring_reference reflect color *origin_inertial verbose=false" >

        <!-- Create and configure the leg link body components -->
        <link name="${prefix}_${suffix}_leg_link">
            <!-- Configure the inertial body  -->
            <inertial>
      <mass value="0.735" />
                <xacro:insert_block name="origin_inertial"/>
                <inertia ixx="0.0037482" ixy="0.0000302" ixz="0.000768" iyy="0.0029930" iyz="-0.0000679" izz="0.0021259" />
```

```xml
            </inertial>

            <!-- Configure the visual body -->
            <visual>
                 <origin xyz="0 0 0" rpy="0 0 0" />
                 <geometry>
                      <mesh filename="${stl}" />
                 </geometry>
                 <!-- Defined in the "materials.urdf.xacro" supplemental file -->
                 <xacro:color_material color="${color}"/>
            </visual>

            <!-- Configure the collisional body -->
            <collision>
                 <origin xyz="0 0 0" rpy="0 0 0" />
                 <geometry>
                      <mesh filename="${stl}" />
                 </geometry>
            </collision>
      </link>

      <!-- Attach the leg to the main robot body link called "base_link"  -->
<joint name="${prefix}_${suffix}_leg_joint" type="revolute">
            <parent link="base_link" />
            <child link="${prefix}_${suffix}_leg_link" />
  <limit effort="10000" velocity="10000" lower="${min}" upper="${max}" />
  <dynamics damping="${damping}" spring_stiffness="${stiffness}" spring_reference="${spring_reference}" />

            <!-- Properly define the origin of the leg link to the base regardless of FR or RL -->
  <xacro:if value="${suffix == 'right'}">
    <axis xyz="0 -1 0" />
    <xacro:if value="${prefix == 'rear'}">
      <origin xyz="${9.5 * -0.01366} ${reflect * 0.04356} 0.03553" rpy="0 -1.12663 ${reflect * 1.5708}" />
    </xacro:if>
    <xacro:unless value="${prefix == 'rear'}">
      <origin xyz="-0.01366 ${reflect * 0.04356} 0.03553" rpy="0 -1.12663 ${reflect * 1.5708}" />
    </xacro:unless>
  </xacro:if>
  <xacro:unless value="${suffix == 'right'}">
```

```
        <axis xyz="0 1 0" />
        <xacro:if value="${prefix == 'rear'}">
          <origin xyz="${9.5 * 0.13031} ${reflect * 0.04356}
    0.03553" rpy="0 -1.12663 ${reflect * 1.5708}" />
        </xacro:if>
        <xacro:unless value="${prefix == 'rear'}">
          <origin xyz="0.13031 ${reflect * 0.04356} 0.03553" rpy="0
     -1.12663 ${reflect * 1.5708}" />
        </xacro:unless>
                </xacro:unless>
          </joint>

          <!-- Attach the Gazebo plugin developed for a torsional
    spring -->
          <gazebo>
                <plugin name="spring_plugin_${prefix}_${suffix}"
    filename="libTorsionalSpringPlugin.so" >
                      <joint_handle>${prefix}_${suffix}_leg_joint</
    joint_handle>
                      <spring_stiffness>${stiffness}</
    spring_stiffness>
                      <spring_damping>${damping}</spring_damping>
                      <spring_reference>${spring_reference}</
    spring_reference>
                      <verbose>${verbose}</verbose>
                </plugin>
          </gazebo>

          <!-- Customize the color of the leg component as seen in
    Gazebo -->
          <xacro:color_gazebo parent="${prefix}_${suffix}_leg_link"
     color="${color}"/>
     </xacro:macro>

</robot>
```

Listing A.25: Shows the code that is used to create and attach a leg link to a robot model base.

A.5.4

```
#ifndef __TORSIONAL_SPRING_PLUGIN_H__
#define __TORSIONAL_SPRING_PLUGIN_H__

```

```
#include <string>

#include "gazebo/common/common.hh"
#include "gazebo/common/Events.hh"

#include "gazebo/gazebo.hh"

#include "gazebo/physics/physics.hh"
#include "gazebo/physics/Model.hh"
#include "gazebo/physics/Joint.hh"

namespace gazebo{
  class TorsionalSpringPlugin : public ModelPlugin{
    private:
      void ExplicitUpdate();
      event::ConnectionPtr updateConnection;
      common::Time prevUpdateTime;
      physics::ModelPtr model;

      physics::JointPtr jointHandle;
      std::string jointName;
      double springStiffness;
      double springDamping;
      double springReference;
      bool verbose;
    public:

      TorsionalSpringPlugin();
      virtual void Load(physics::ModelPtr _model, sdf::ElementPtr
    _sdf);
      virtual void Init();
  };
}
#endif
```

Listing A.26: Shows the header file created for the torsional spring plugin developed in this work.

A.5.5

```
#include <ros/ros.h>
#include "gazebo/physics/physics.hh"
#include "TorsionalSpringPlugin.hh"
using namespace gazebo;
```

```

GZ_REGISTER_MODEL_PLUGIN(TorsionalSpringPlugin)

TorsionalSpringPlugin::TorsionalSpringPlugin(){}

void TorsionalSpringPlugin::Load(physics::ModelPtr _model, sdf::
    ElementPtr _sdf){

  this->model = _model;

  if(!_sdf->HasElement("joint_handle")){
    ROS_INFO_NAMED("libTorsionalSpringPlugin", "Plugin missing <
    joint_handle>, defaults to /base_link");
    this->jointName = "/base_link";
  } else this->jointName = _sdf->Get<std::string>("joint_handle");

  if(!_sdf->HasElement("spring_stiffness")){
    ROS_INFO_NAMED("libTorsionalSpringPlugin", "Plugin missing <
    spring_stiffness>, defaults to 100.0");
    this->springStiffness = 100.0;
  } else this->springStiffness = _sdf->Get<double>("
    spring_stiffness");

  if(!_sdf->HasElement("spring_damping")){
    ROS_INFO_NAMED("libTorsionalSpringPlugin", "Plugin missing <
    spring_damping>, defaults to 1.0");
    this->springDamping = 1.0;
  } else this->springDamping = _sdf->Get<double>("spring_damping");

  if(!_sdf->HasElement("spring_reference")){
    ROS_INFO_NAMED("libTorsionalSpringPlugin", "Plugin missing <
    spring_reference>, defaults to 1.0");
    this->springReference = 1.0;
  } else this->springReference = _sdf->Get<double>("
    spring_reference");

  if(!_sdf->HasElement("verbose")){
    ROS_INFO_NAMED("libTorsionalSpringPlugin", "Plugin missing <
    verbose>, defaults to false");
    this->verbose = false;
  } else this->verbose = _sdf->Get<bool>("verbose");

}

void TorsionalSpringPlugin::Init(){
  this->jointHandle = this->model->GetJoint(this->jointName);
  this->updateConnection = event::Events::ConnectWorldUpdateBegin(
    boost::bind(&TorsionalSpringPlugin::ExplicitUpdate, this));
}

void TorsionalSpringPlugin::ExplicitUpdate(){
  common::Time currTime = this->model->GetWorld()->GetSimTime();
  common::Time stepTime = currTime - this->prevUpdateTime;
  this->prevUpdateTime = currTime;

  double pos = this->jointHandle->GetAngle(0).Radian();
  double vel = this->jointHandle->GetVelocity(0);
  double force = -this->springStiffness * (pos - this->
    springReference) - this->springDamping * vel;

  if(this->verbose) gzdbg << "[Joint] ----- Pos, Vel, Force: " <<
    this->jointName << ",        " << pos << ",        " << vel << ",
        " << force << std::endl;
  // ROS_INFO_NAMED("TorsionalSpring", "Pos, Vel, Force: %f, %f, %f
     \r\n", pos, vel, force);

  this->jointHandle->SetForce(0, force);
}
```

Listing A.27: Shows the source file created for the torsional spring plugin developed in this work.