Hibachi: A ROS compatible skid-steer vehicle

Guido M. Sánchez*, Marina H. Murillo*, Jesús E. Benavidez*, Nestor N. Deniz* and Leonardo L. Giovanini*
*Research institute for signals, systems and computational intelligence, sinc(i)
UNL, CONICET, Ciudad Universitaria UNL, 4to piso FICH, (S3000) Santa Fe, Argentina
{gsanchez,mmurillo,ebena,ndeniz,lgiovanini}@sinc.unl.edu.ar

***Abstract*—Skid-steer vehicles are one of the most popular configuration of autonomous ground vehicles (AGVs). Their mechanical simplicity provides researchers and developers with an easy to build vehicle that can perform complicated tasks either in outdoor or indoor scenarios. However, the process of building an AGV usually requires to spend a lot of time solving problems that are already solved. This work focuses on the process required to make a working AGV –the Hibachi– compatible with Robot Operating System (ROS). We mainly describe and explain the tasks of developing the required software and device drivers to implement `ros_control` on the existing hardware, allowing easier sensor integration and providing a standarized testbed for AGV research.**

***Index Terms*—autonomous ground vehicle, robot operating system, robotics, Raspberry Pi, Arduino**

I. Introduction

In recent years, there has been a lot of development and construction of AGVs and the interest in this field has undergone tremendous improvement lately with a growing interest in precision agriculture [1]–[4]. However, while trying to accomplish the task to make an AGV work autonomously, developers and researchers usually spend a lot of time solving problems that are already solved, such as vehicle kinematics, PID control, among others. Since its conception in 2009, ROS [5], [6] has become the *de facto* standard framework for robotics software development. The core of ROS is licensed under the standard three-clause BSD license, which is a very permissive open license that allows for reuse in commercial and closed source products. It is a flexible framework for writing robot software, a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It has a wide variety of packages that are available and well tested. Many efforts have been made by researchers to create low-cost ROS compatible ground vehicles [7], [8], quadcopters [9], submarines [10] and even bring FPGA integration [11].

In this work, we describe the steps taken to make a working skid-steer vehicle –the Hibachi– compatible with ROS. The skid-steer configuration was chosen because of its mechanical simplicity and high maneuverability particularly in outdoor applications [12]. To make the Hibachi ROS compatible, we needed to develop certain software and drivers that allowed the AGV to behave as a ROS node. The main challenge was to implement `ros_control`, which is the software that glues the low-level embedded system software with the high-level ROS software running in the on-board computer. This is the reason why this work aims to focus on how to bridge the gap between the existing hardware and ROS, allowing the research group to start focusing on interesting problems specific to our AGV or application, instead of reinventing the wheel.

II. Hardware configuration

The main objective for the Hibachi was to make a fairly easy to build and easy to fix AGV. Because of that, the skid-steer configuration seemed to be quite obvious. The hardware election was based on existing parts on our laboratory. The Hibachi is made of the following components:

- **Raspberry Pi 3 model B+**: This is a single board computer that runs Raspbian and ROS. The Raspberry is connected to an Arduino Due to retrieve the wheel odometry information and the Navio2 to retrieve the IMU and GPS values.
- **Emlid Navio2**[1]: This is an autopilot HAT designed for the Raspberry Pi. This hardware device has a Global Navigation Satellite System (GNSS) module that is capable of tracking GPS, GLONASS, Beidou, Galileo and SBAS satellites. It also offers a dual Inertial Measurement Unit (IMU) setup, providing accelerometers, gyroscopes and magnetometers for orientation and motion sensing. Finally, it exposes various interfaces (ADC, I^{2}C and UART) for extra sensors and radios.
- **Arduino Due controller**: This Arduino board is based on the Atmel SAM3X8E ARM Cortex-M3 CPU. It provides 54 digital input/output pins, 12 analog inputs, 2 DAC and 2 CAN. This board is used to control the motors' speed and to save the odometry information. It receives each of the motors velocity setpoints and four embedded PID controllers ensure those setpoints are met. It also sends the encoder data back to the Raspberry Pi for odometry calculation.
- **DC gear motors with quadrature encoders**: Since our robot uses a *skid-steer* configuration, we have to drive four motors. Each motor is operated in 12 V. The motor shaft is attached to a quadrature encoder, which can deliver a maximum count of 3290 counts per revolution of the gearbox's output shaft. Motor encoders are one source of odometry of robot.
- **L298 dual H-Bridge motor drivers**: Each of the two H-Bridge installed allows full control of two DC motors. It receives PWM signals from the Arduino Due and outputs the DC motor voltage accordingly.

[1]https://navio2.emlid.com/

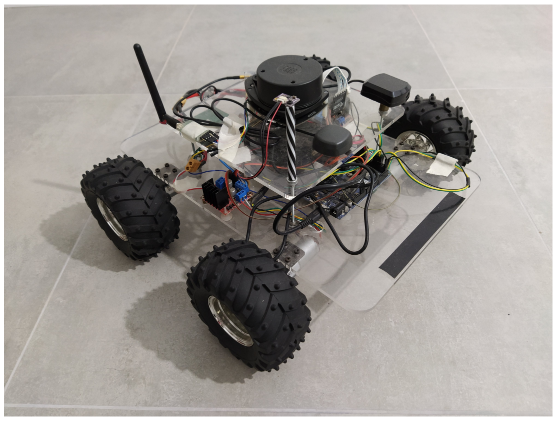

Fig. 1. Hibachi chassis with onboard computer in the middle and sensors mounted on top.

- **RPLidar A1**: is a low cost 2D LIDAR solution. It can scan 360° environment within 12 meter radius. The output of this sensor is very suitable to build map or do SLAM.

All these components are installed onto a 5 mm acrylic plate, allowing rapid prototyping and an easy mechanical setup (see Fig. 1). The Navio2 could be replaced with any GNSS module and IMU, since up to this day, these are the only parts being used.

III. Hibachi ROS packages

One of the main benefits of diving into the ROS ecosystem is that we can take advantage of the availability of a large number of widely tested packages, so developers and researchers do not have to do all the heavy lifting to get a robot up and running. In order to make ROS compatible robot, we need to develop the required software to turn it into a *ROS node* so it can interact and exchange information with other nodes and packages [8]. To achieve ROS compatibility, Hibachi must publish its geometric information using the Unified Robot Description Format (URDF), publish all the relevant sensor and odometry data as ROS topics and be capable of receive control actions from other nodes. Another requirement is to fullfill some of the ROS Enhancement Proposals (REP), such as REP-103 [13]: *"Standard Units of Measure and Coordinate Conventions"*, which provides a reference for the units and coordinate conventions used within ROS; and REP-105 [14]: *"Coordinate Frames for Mobile Platforms"*, which specifies naming conventions and semantic meaning for coordinate frames of mobile platforms used with ROS. The following list of ROS packages were developed for Hibachi, allowing the robot to interact with other ROS packages:

- `hibachi_description`: Hibachi URDF description which provides a code-independent, human-readable way to describe the geometry of the robot and its parts.
- `navio2_ros`: Provides a direct ROS interface for some of the Navio2 sensors. Currently it exposes the IMU and the GNSS module.
- `hibachi_firmware`: Provides the Arduino firmware for the Hibachi. It handles the communication with the Raspberry Pi, controls the velocity of the motors and sends the motor encoder information.
- `hibachi_base`: Hibachi hardware interface node, which holds the robot hardware abstraction. It has the definition of each of the Hibachi joints and its control interfaces.
- `hibachi_gazebo`: Hibachi simulation package. It setups a simulated robot in a virtual world. The robot definition is obtained from the `hibachi_description` URDF and then Gazebo is responsible for the physics simulation.

IV. Implementing `ros_control` on the Hibachi

The mission of `ros_control` is to lower the entry barrier for exposing hardware to ROS, and to promote the reuse of control code in the same way that ROS has allowed for higher-level code. The `ros_control` framework provides the capability to implement and manage robot controllers with a focus on both real-time performance and sharing of controllers in a robot-agnostic way [15]. Being a mature framework, `ros_control` is widely applied to not only to the production of robots but also into numerous research platform robots and comes with ready to use controllers. Controllers expose standard ROS interfaces for out-of-the box third party solutions to robotics problems like manipulation path planning and autonomous navigation [16]. This is the most important issue, because once the Hibachi is ROS compatible, there will be no need to write any additional code for many common robotics tasks, only configuration files.

As stated in "*ROS Control, An overview*" [17], to implement a custom robot using `ros_control`, one must inherit from the Robot Hardware Interface and Resource Manager class (`hardware_interface::RobotHW`). This class is meant to be used as a base class for abstracting custom robot hardware. After that, we need to setup the individual controllers each robot needs. In our case, we need to manage each of the four joints used to describe the four motors of the Hibachi. So we need the following:

- `hardware_interface::JointStateInterface` to support reading the state of the joints
- `hardware_interface::VelocityJointInterface` for commanding the velocity of the joints.

Listing 1 shows a snippet of C++ code that registers the four joints representing the Hibachi wheels. First, we define the same names as in the URDF and then we iterate over that list of names in order to create a `JointStateHandles` that holds each of the defined joints state (name, position, velocity, effort). After that, the `JointHandle` is used to read and command each joint.

Listing 1. Registering Hibachi joints in the hardware interface.

```
void HibachiHardware::
     registerControlInterfaces() {
```

```
ros::V_string joint_names =
boost::assign::list_of("front_left_wheel")
                      ("front_right_wheel")
                      ("rear_left_wheel")
                      ("rear_right_wheel");

for (unsigned int i = 0;
     i < joint_names.size();
     i++) {
  hardware_interface::JointStateHandle
  joint_state_handle(joint_names[i],
                     &joints_[i].position,
                     &joints_[i].velocity,
                     &joints_[i].effort);
  joint_state_interface_.
      registerHandle(joint_state_handle);
  hardware_interface::JointHandle joint_handle(
        joint_state_handle,
        &joints_[i].velocity_command);
  velocity_joint_interface_.
      registerHandle(joint_handle);
  }
registerInterface(&joint_state_interface_);
registerInterface(&velocity_joint_interface_);
}
```

The `joint_` variable is a Joint structure (see Listing 2) which allows us to easily access each joint state and command. This is hooked to `ros_control` InterfaceManager, to allow control via `diff_drive_controller`.

Listing 2. Joint struct.

```
struct Joint {
  double position;
  double position_offset;
  double velocity;
  double effort;
  double velocity_command;
  int16_t encoder_pulses;
  Joint() :
    position(0), velocity(0),
    effort(0), velocity_command(0),
    encoder_pulses(0)
  { }
}
```

After the robot setup we need to implement the `read()` and `write()` methods. These methods communicate the Raspberry Pi with the Arduino Due. In the Hibachi case, the `read()` method obtains the motor encoder position and it is responsible to calculate and populate the joint position in radians and velocity in radians per seconds (see Listing 3). The `write()` method obtains each of the wheel velocity setpoint from the ROS higher level controllers (mainly the `diff_drive_controller`) and sends them to the Arduino via serial port, which has to ensure each of the wheels reaches the desired velocity setpoint with a PID controller running on the board (see Listing 4).

Listing 3. Hibachi `read()` method.

```
void HibachiHardware::read(
  const ros::Duration &duration)
{
hibachi_base::GetFourWheelEncoder
  getFourWheelEncoder;
_serialPort.sendMessage(&getFourWheelEncoder);
auto fourWheelEncoderData =
    (FourWheelEncoderData*)_serialPort.waitMessage(
        FourWheelEncoderData::MESSAGE_TYPE, 1.0);

if (fourWheelEncoderData != NULL)
{
int16_t encoder_pulses_front_left_prev =
    joints_[FRONT_LEFT].encoder_pulses;
int16_t encoder_pulses_front_right_prev =
    joints_[FRONT_RIGHT].encoder_pulses;
int16_t encoder_pulses_rear_left_prev =
    joints_[REAR_LEFT].encoder_pulses;
int16_t encoder_pulses_rear_right_prev =
    joints_[REAR_RIGHT].encoder_pulses;

joints_[FRONT_LEFT].encoder_pulses =
    (int16_t)fourWheelEncoderData->getFrontLeftPulses();
joints_[FRONT_RIGHT].encoder_pulses =
    (int16_t)fourWheelEncoderData->getFrontRightPulses();
joints_[REAR_LEFT].encoder_pulses =
    (int16_t)fourWheelEncoderData->getRearLeftPulses();
joints_[REAR_RIGHT].encoder_pulses =
    (int16_t)fourWheelEncoderData->getRearRightPulses();

int16_t delta_front_left =
    joints_[FRONT_LEFT].encoder_pulses -
    encoder_pulses_front_left_prev;
int16_t delta_front_right =
    joints_[FRONT_RIGHT].encoder_pulses -
    encoder_pulses_front_right_prev;
int16_t delta_rear_left =
    joints_[REAR_LEFT].encoder_pulses -
    encoder_pulses_rear_left_prev;
int16_t delta_rear_right =
    joints_[REAR_RIGHT].encoder_pulses -
    encoder_pulses_rear_right_prev;

joints_[FRONT_LEFT].position +=
    ticksToAngle(delta_front_left);
joints_[FRONT_RIGHT].position +=
    ticksToAngle(delta_front_right);
joints_[REAR_LEFT].position +=
    ticksToAngle(delta_rear_left);
joints_[REAR_RIGHT].position +=
    ticksToAngle(delta_rear_right);

joints_[FRONT_LEFT].velocity =
    ticksToAngle(delta_front_left) / duration.toSec();
joints_[FRONT_RIGHT].velocity =
    ticksToAngle(delta_front_right) / duration.toSec();
joints_[REAR_LEFT].velocity =
    ticksToAngle(delta_rear_left) / duration.toSec();
joints_[REAR_RIGHT].velocity =
    ticksToAngle(delta_rear_right) / duration.toSec();

}
else
{
ROS_ERROR("No valid encoder data received");
}
}
```

Once the Hibachi `ros_control` stack is ready, we can see that the four joints information (position, speed, effort, control command) corresponding to each of the wheels is available for the higher level software to be employed.

Listing 4. Hibachi `write()` method.

```
void HibachiHardware::write()
{
double diff_speed_front_left =
     (joints_[FRONT_LEFT].velocity_command);
double diff_speed_front_right =
     joints_[FRONT_RIGHT].velocity_command;
double diff_speed_rear_left =
     joints_[REAR_LEFT].velocity_command;
double diff_speed_rear_right =
     joints_[REAR_RIGHT].velocity_command;

hibachi_base::SetSkidSteerMotorSpeed
```

G. Sanchez, M. Murillo, J. Benavídez, Nestor N. Deniz & L. Giovanini; "Hibachi: A ROS compatible skid-steer vehicle"

```
SetSkidSteerMotorSpeed(diff_speed_front_left,
                       diff_speed_front_right,
                       diff_speed_rear_left,
                       diff_speed_rear_right);
_serialPort.sendMessage(
                    &SetSkidSteerMotorSpeed);
}
```

V. Setting up a differential drive controller for Hibachi

As stated in the previous sections, we want to use ROS to save some time while designing and developing our robot. Once each of the Hibachi wheels are capable of receiving control commands, we can set up the ready to use ROS `diff_drive_controller`. This package works with wheel joints through a velocity interface. It takes *Twist* messages that contain the desired linear and angular velocity of the robot and converts the latter to rad/s for each wheel based on the configuration provided in the `hibachi_description` package. The robot odometry is computed from the hardware feedback, and published as an *Odometry* message. It is compatible with both differential drive and skid-steer vehicles. As stated in previous section, in order to use the `diff_drive_controller` package we do not have to write extra code, just a simple configuration file (see Listing 5). The configuration file requires us to define the list of wheel joint names (as we used in the URDF description file), the publish rate of the odometry information, the wheel separation and wheel radius (in meters) and allows to choose if we want to see the control command after limiters have been applied on the controller input (good for debugging purposes) and if we want the controller to publish the TF tree.

Listing 5. Hibachi **diff_drive_controller** configuration snippet.

```
hibachi_velocity_controller:
  type:
  "diff_drive_controller/DiffDriveController"
  left_wheel: ['front_left_wheel',
               'rear_left_wheel']
  right_wheel: ['front_right_wheel',
                'rear_right_wheel']
  publish_rate: 25

  cmd_vel_timeout: 0.25
  publish_cmd: True
  enable_odom_tf: True

  wheel_separation: 0.34
  wheel_radius: 0.06
```

VI. Publishing Hibachi state to ROS

To make other ROS packages aware of the current state of Hibachi, we use the `robot_state_publisher` package, which uses the URDF specified by `hibachi_description` and the joint positions from the topic `joint_states` published by the `JointStateInterface` to calculate the forward kinematics of the robot and publish the results to other ROS nodes. This is a glimpse of how ROS reutilizes software. This package takes the joint angles of the robot as input and publish the 3D poses of the robot links, using a kinematic tree model of the robot. After that, you can visualize the robot state (including every sensor data) in real-time using RViz, for example. Figure 2 shows the Hibachi model as described in the URDF and the axes of some of its links: `base_link`, which is located in the center of the acrylic plate; `front_left_wheel` and `front_right_wheel`, which are located in the center of the front left and right wheel, respectively, they are connected to the vehicle body through a joint and rotate on their $x-axis$ whenever the wheel moves; the `rplidar` and `gps` axis tell where each of these

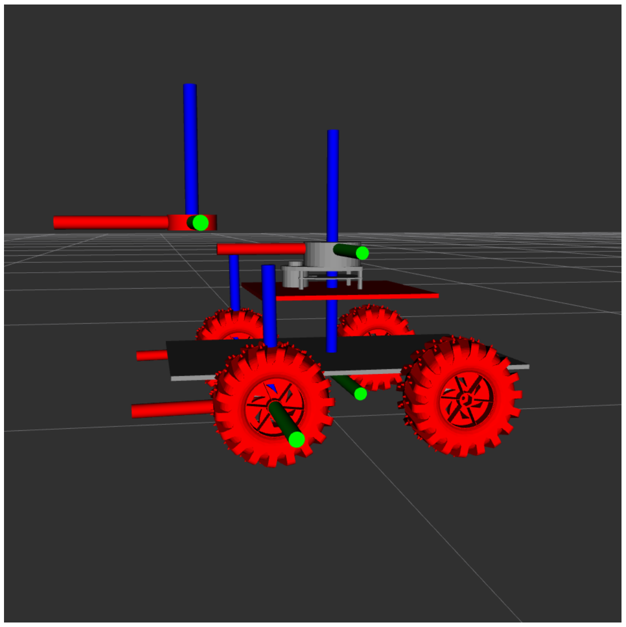
Fig. 2. Hibachi model showing various axes of the TF tree in RViz.

sensors are located and how they are orientated, allowing the system to calculate automatically the needed transformations from each one to the center of mass of the vehicle.

Simply launching the `robot_state_publisher` with the parameter `robot_description` specifying our URDF and without extra configuration is enough for the system to publish the current state: the 3D poses of all the robots links. Once the state gets published, it is available to all components in the system.

VII. Teleoperating the Hibachi

Once our robot is ROS compatible and accepts *Twist* messages to control the desired linear and angular velocities, we can simply configure and launch the already written tele-operating packages `teleop_twist_keyboard` or `teleop_twist_joy`, allowing our vehicle to start moving around. If our robot is correctly configured, the `teleop` packages will send *Twist* messages to the `diff_drive_controller`, which will send velocity commands to the robot wheels and it will update the odometry information accordingly. The approach that requires less configuration is to launch the `teleop_twist_keyboard` package, which will listen to a computer keyboard and will publish *Twist* messages. You can move the robot around by pressing the keys `[u]`, `[i]`, `[o]`, `[j]`, `[k]`, `[l]`, `[m]`, `[,]` and `[.]`.

VIII. Simulation with ROS and Gazebo

The ROS ecosystem has a high quality simulator in Gazebo[2], which offers the posibility to simulate and quickly test any ROS compatible robot through the `gazebo_ros_control` package. Once the `robot_description` is done, we can add simulated sensors choosing from a vast pre-existing library (such as IMU, GPS, LIDAR, etc) and we can launch our robot in a simulated world with a robust physics engine and high quality graphics. Each simulated sensor accepts different configuration parameters, such as frequency, noise variance, drift, bias, etc. This allows to simulate our vehicle and have realistic simulated sensors with noisy data. Moreover, each Gazebo world can be configured in many ways, giving a lot of flexibility. Figure 3 shows the Hibachi in a simulated world with

[2]http://gazebosim.org

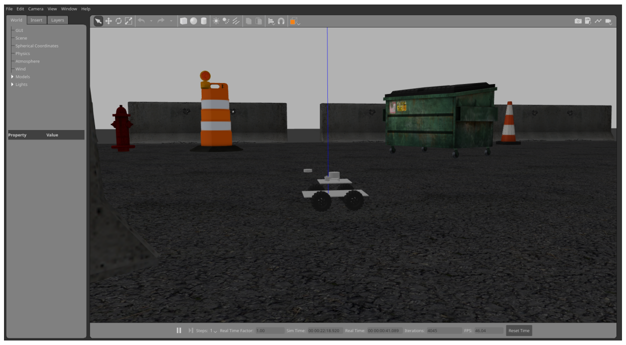

Fig. 3. Hibachi in a Gazebo simulation

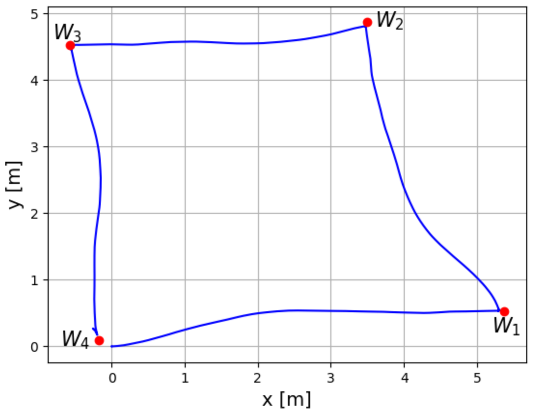

Fig. 4. Hibachi following outdoor waypoints.

obstacles around, thus allowing developers and researchers to test different algorithms even without access to the real hardware. The package `hibachi_gazebo` is responsible to load Hibachi URDF by launching the `hibachi_description` package and bring up the robot in a world run by Gazebo simulator. The simulated Hibachi is loaded with wheel encoders, accelerometer, gyroscope, magnetometer, GNSS sensor and a LIDAR, mimicking the on-board hardware on the real robot.

IX. Experimental results

In the following, we will show the experimental results obtained while the Hibachi follows a set of predefined outdoor waypoints arranged in a squared shape. For this experiment, we configured the `robot_localization` package to fuse information from the GPS, IMU and wheel encoders to obtain an estimation of the current position and orientation in the real world. The navigation uses ROS `navigation` stack. As Fig. 4 shows, the vehicle starts moving towards waypoint W_1. Once it reaches the waypoint, the Hibachi turns left and moves to W_2. This behaviour repeats until the Hibachi reaches W_4, which is the starting point.

These algorithms need some extra tuning, however this shows the importance of having an open-source framework for robotics, such as ROS that simplifies the transition to experimental robotics.

X. Conclusion and future work

Making a ROS compatible vehicle enables developers and researchers to stop spending time finding solutions to problems that have already been solved. It also facilitates sensor integration and exchange (for example, we could easily change an IMU as long as it has a ROS driver and it outputs its data as an `Odometry` message) and provides a standardized testbed for robotics research (thanks to many of the REPs, such as REP-103 and REP-105). We hope this work enlightens a little the path towards making a ROS compatible vehicle. Once the sensors and the vehicle hardware is properly configured, the user can choose what other ROS packages to configure and run according to the required necessities. For example, if the user needs indoor navigation with obstacle avoidance you should setup the `navigation` package, which will allow to fuse data from the IMU, the wheel encoders and some range sensor such as a LIDAR. For outdoor navigation you should setup the `robot_localization` package to fuse data from the GNSS, IMU and the wheel encoders to localize the robot in an outdoor environment. The use of a Raspberry Pi as an on-board computer allows the AGV to have a high degree of autonomy, requiring user intervention only to send mission plans or for emergency stops.

We are planning to migrate the Hibachi software to ROS2, since ROS is reaching its end of life. On the hardware side, we are intending to upgrade the motors in order to provide more power and be able to carry heavier payloads and add a higher quality IMU.

Acknowledgment

The authors would like to thank Emlid for the educational discount on their products, the *Universidad Nacional de Litoral* (with CAI+D 50620190100080LI and CATT 2019), the *Agencia Nacional de Promoción Científica y Tecnológica* and the *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET) from Argentina, for their support.

References

[1] H. Mousazadeh, "A technical review on navigation systems of agricultural autonomous off-road vehicles," *Journal of Terramechanics*, vol. 50, no. 3, pp. 211–232, 2013.

[2] S. Han, Y. HE, and H. Fang, "Recent development in automatic guidance and autonomous vehicle for agriculture: A review," *Journal of Zhejiang University (Agriculture and Life Sciences)*, vol. 44, no. 4, pp. 381–391, 2018.

[3] J. A. Thomasson, C. P. Baillie, D. L. Antille, C. R. Lobsey, C. L. McCarthy, *et al.*, *Autonomous technologies in agricultural equipment: A review of the state of the art*. American Society of Agricultural and Biological Engineers, 2019.

[4] A. Roshanianfard, N. Noguchi, H. Okamoto, and K. Ishii, "A review of autonomous agricultural vehicles (the experience of hokkaido university)," *Journal of Terramechanics*, vol. 91, pp. 155–183, 2020.

[5] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "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.

[6] *Robot Operating System (ROS)*, https://www.ros.org/, Accessed: 2021-08-01.

[7] A. Araújo, D. Portugal, M. S. Couceiro, J. Sales, and R. P. Rocha, “Desarrollo de un robot móvil compacto integrado en el middleware ros,” *Revista Iberoamericana de Automática e Informática industrial*, vol. 11, no. 3, pp. 315–326, 2014, ISSN: 1697-7920. DOI: 10.1016/j.riai.2014.02.009.

[8] S. S. H. Hajjaj and K. S. M. Sahari, “Bringing ros to agriculture automation: Hardware abstraction of agriculture machinery,” *International Journal of Applied Engineering Research*, vol. 12, no. 3, pp. 311–316, 2017.

[9] A. Majumdar, N. Gamez, P. Benavidez, and M. Jamshidi, “Development of robot operating system (ros) compatible open source quadcopter flight controller and interface,” in *2017 12th System of Systems Engineering Conference (SoSE)*, 2017, pp. 1–6. DOI: 10.1109/SYSOSE.2017.7994980.

[10] E. Gallimore, R. Stokey, and E. Terrill, “Robot operating system (ros) on the remus auv using recon,” in *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, 2018, pp. 1–6. DOI: 10.1109/AUV.2018.8729755.

[11] K. Hasegawa, K. Takasaki, M. Nishizawa, R. Ishikawa, K. Kawamura, and N. Togawa, “Implementation of a ros-based autonomous vehicle on an fpga board,” in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 457–460. DOI: 10.1109/ICFPT47387.2019.00092.

[12] R. Khan, F. Mumtaz, A. Raza, and N. Mazhar, “Comprehensive study of skid-steer wheeled mobile robots: Development and challenges,” *Industrial Robot*, vol. 48, pp. 142–156, Aug. 2020. DOI: 10.1108/IR-04-2020-0082.

[13] T. Foote and M. Purvis, *Standard units of measure and coordinate conventions*, https://www.ros.org/reps/rep-0103.html, Accessed: 2021-08-01, Oct. 2010.

[14] W. Meeussen, *Coordinate frames for mobile platforms*, https://www.ros.org/reps/rep-0105.html, Accessed: 2021-08-01, Oct. 2010.

[15] A. R. Tsouroukdissian, *[ROSCon 2014] ros_control: An overview*, https://vimeo.com/107507546, Accessed: 2021-08-01, Sep. 2014.

[16] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. R. Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. F. Perdomo, “ros_control: A generic and simple control framework for ROS,” *Journal of Open Source Software*, vol. 2, no. 20, p. 456, 2017. DOI: 10.21105/joss.00456.

[17] F. Pucher, *ROS Control, An overview*, https://fjp.at/posts/ros/ros-control/, Accessed: 2021-08-01, Mar. 2020.