Plugins Used:

Ray Sensor:

Used default plugin **"libgazebo_ros_laser"** (https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_laser.cpp)

This plugin reads Gazebo Ray sensor info, convert the gazebo ray scan message to ROS sensor msg: **sensor_msgs::LaserScan** and publishes to the ROS topic name specified in the SDF/URDF file. Below is the code that accomplishes this:

```
void GazeboRosLaser::OnScan(ConstLaserScanStampedPtr &_msg)
{
 sensor_msgs::LaserScan laser_msg;
 laser_msg.header.stamp = ros::Time(_msg->time().sec(), _msg->time().nsec());
 laser_msg.header.frame_id = this->frame_name_;
 laser_msg.angle_min = _msg->scan().angle_min();
 laser_msg.angle_max = _msg->scan().angle_max();
 laser_msg.angle_increment = _msg->scan().angle_step();
 laser_msg.time_increment = 0;  // instantaneous simulator scan
 laser_msg.scan_time = 0;  // not sure whether this is correct
 laser_msg.range_min = _msg->scan().range_min();
 laser_msg.range_max = _msg->scan().range_max();
 laser_msg.ranges.resize(_msg->scan().ranges_size());
 std::copy(_msg->scan().ranges().begin(),
        _msg->scan().ranges().end(),
        laser_msg.ranges.begin());
 laser_msg.intensities.resize(_msg->scan().intensities_size());
 std::copy(_msg->scan().intensities().begin(),
        _msg->scan().intensities().end(),
        laser_msg.intensities.begin());
 this->pub_queue_->push(laser_msg, this->pub_);
}
```

Following code creates a subscriber to ray node, for each ray element:

```
// Subscriber code
void GazeboRosLaser::LaserConnect()
{
 this->laser_connect_count_++;
 if (this->laser_connect_count_ == 1)
   this->laser_scan_sub_ =
     this->gazebo_node_->Subscribe(this->parent_ray_sensor_->Topic(),
                        &GazeboRosLaser::OnScan, this);
}
```

And finally following is the code that creates the publisher node, using ROS transport layer (Need to understand this more):

```
void GazeboRosLaser::LoadThread()
{
 this->gazebo_node_ = gazebo::transport::NodePtr(new gazebo::transport::Node());
```

```
 this->gazebo_node_->Init(this->world_name_);

 this->pmq.startServiceThread();

 this->rosnode_ = new ros::NodeHandle(this->robot_namespace_);

 this->tf_prefix_ = tf::getPrefixParam(*this->rosnode_);
 if(this->tf_prefix_.empty()) {
    this->tf_prefix_ = this->robot_namespace_;
    boost::trim_right_if(this->tf_prefix_,boost::is_any_of("/"));
 }
 ROS_INFO_NAMED("laser", "Laser Plugin (ns = %s)  <tf_prefix_>, set to \"%s\"",
          this->robot_namespace_.c_str(), this->tf_prefix_.c_str());

 // resolve tf prefix
 this->frame_name_ = tf::resolve(this->tf_prefix_, this->frame_name_);

 if (this->topic_name_ != "")
 {
  ros::AdvertiseOptions ao =
    ros::AdvertiseOptions::create<sensor_msgs::LaserScan>(
    this->topic_name_, 1,
    boost::bind(&GazeboRosLaser::LaserConnect, this),
    boost::bind(&GazeboRosLaser::LaserDisconnect, this),
    ros::VoidPtr(), NULL);
  this->pub_ = this->rosnode_->advertise(ao);
  this->pub_queue_ = this->pmq.addPub<sensor_msgs::LaserScan>();
 }

 // Initialize the controller

 // sensor generation off by default
 this->parent_ray_sensor_->SetActive(false);
}
```

Diff-Drive wheel controller (Slightly Complex):

Used default plugin **“libgazebo_ros_diff_drive”** (https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_diff_drive.cpp)

This plugin reads Twist messages (Linear + Angular velocities) and convert to wheel velocities based on Ackerman steering.

Following is the code that used Ackerman steering principle to extract wheel velocities:

```
void GazeboRosDiffDrive::getWheelVelocities()
{
   boost::mutex::scoped_lock scoped_lock ( lock );

   double vr = x_;
   double va = rot_;

   if(legacy_mode_)
   {
    wheel_speed_[LEFT] = vr + va * wheel_separation_ / 2.0;
    wheel_speed_[RIGHT] = vr - va * wheel_separation_ / 2.0;
```

```
  }
  else
  {
   wheel_speed_[LEFT] = vr - va * wheel_separation_ / 2.0;
   wheel_speed_[RIGHT] = vr + va * wheel_separation_ / 2.0;
  }
}
```

Following code publishes the wheel velocities to the wheel joints:

```
void GazeboRosDiffDrive::UpdateChild()
{
  for ( int i = 0; i < 2; i++ ) {
   if ( fabs(wheel_torque -joints_[i]->GetParam ( "fmax", 0 )) > 1e-6 ) {
    joints_[i]->SetParam ( "fmax", 0, wheel_torque );
   }
  }

  if ( odom_source_ == ENCODER ) UpdateOdometryEncoder();
#if GAZEBO_MAJOR_VERSION >= 8
  common::Time current_time = parent->GetWorld()->SimTime();
#else
  common::Time current_time = parent->GetWorld()->GetSimTime();
#endif
  double seconds_since_last_update = ( current_time - last_update_time_ ).Double();

  if ( seconds_since_last_update > update_period_ ) {
    if (this->publish_tf_) publishOdometry ( seconds_since_last_update );
    if ( publishWheelTF_ ) publishWheelTF();
    if ( publishWheelJointState_ ) publishWheelJointState();

    // Update robot in case new velocities have been requested
    getWheelVelocities();

    double current_speed[2];

    current_speed[LEFT] = joints_[LEFT]->GetVelocity ( 0 )   * ( wheel_diameter_ / 2.0 );
    current_speed[RIGHT] = joints_[RIGHT]->GetVelocity ( 0 ) * ( wheel_diameter_ / 2.0 );

    if ( wheel_accel == 0 ||
         ( fabs ( wheel_speed_[LEFT] - current_speed[LEFT] ) < 0.01 ) ||
         ( fabs ( wheel_speed_[RIGHT] - current_speed[RIGHT] ) < 0.01 ) ) {
      //if max_accel == 0, or target speed is reached
      joints_[LEFT]->SetParam ( "vel", 0, wheel_speed_[LEFT]/ ( wheel_diameter_ / 2.0 ) );
      joints_[RIGHT]->SetParam ( "vel", 0, wheel_speed_[RIGHT]/ ( wheel_diameter_ / 2.0 ) );
    } else {
      if ( wheel_speed_[LEFT]>=current_speed[LEFT] )
        wheel_speed_instr_[LEFT]+=fmin ( wheel_speed_[LEFT]-current_speed[LEFT],  wheel_accel *
seconds_since_last_update );
      else
        wheel_speed_instr_[LEFT]+=fmax ( wheel_speed_[LEFT]-current_speed[LEFT], -wheel_accel *
seconds_since_last_update );

      if ( wheel_speed_[RIGHT]>current_speed[RIGHT] )
        wheel_speed_instr_[RIGHT]+=fmin ( wheel_speed_[RIGHT]-current_speed[RIGHT], wheel_accel *
seconds_since_last_update );
      else
```

```
            wheel_speed_instr_[RIGHT]+=fmax ( wheel_speed_[RIGHT]-current_speed[RIGHT], -wheel_accel *
seconds_since_last_update );

          // ROS_INFO_NAMED("diff_drive", "actual wheel speed = %lf, issued wheel speed= %lf", current_speed[LEFT],
wheel_speed_[LEFT]);
          // ROS_INFO_NAMED("diff_drive", "actual wheel speed = %lf, issued wheel speed= %lf",
current_speed[RIGHT],wheel_speed_[RIGHT]);

          joints_[LEFT]->SetParam ( "vel", 0, wheel_speed_instr_[LEFT] / ( wheel_diameter_ / 2.0 ) );
          joints_[RIGHT]->SetParam ( "vel", 0, wheel_speed_instr_[RIGHT] / ( wheel_diameter_ / 2.0 ) );
        }
        last_update_time_+= common::Time ( update_period_ );
      }
}
```

Apart from the above node, other nodes are for publishing the Wheel joint states and Odometry messages.