

Real Robot & 3D sensor

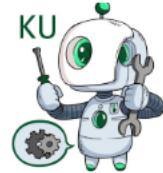
OVERVIEW

Real Robot

- Understanding robot hardware
- Setup Software Environment
- Control the Robot
 - With Controller
 - With Simple Script

Understanding 3D sensor

- Concept of 3D sensor
- Install and testing Kinect (3D sensor)
- Human Follower Experiment



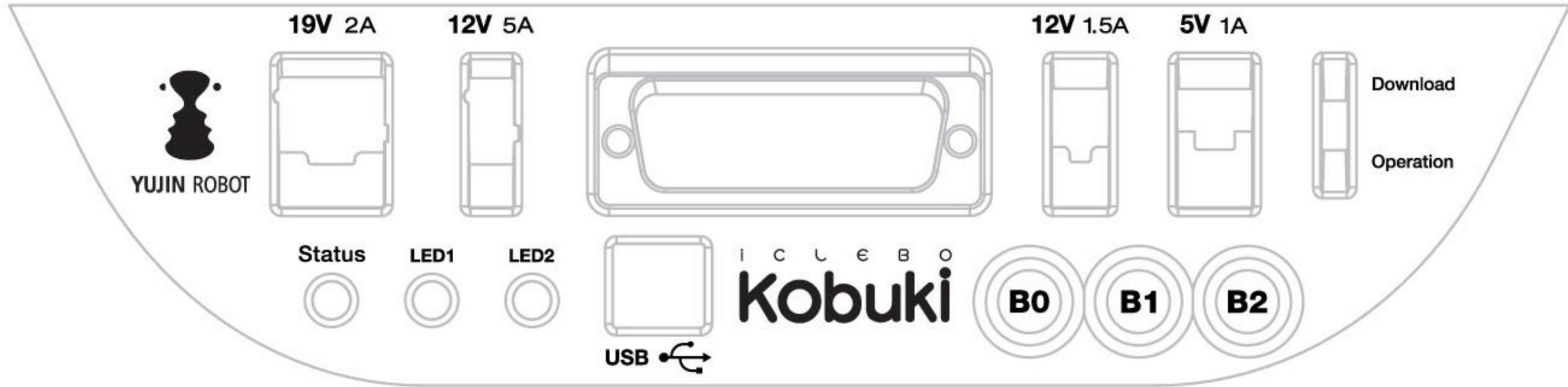


Robot Hardware



Turtlebot 2

Robot Hardware



19V/2A: Laptop power supply

12V/5A: Arm power supply

12V/1.5A: Microsoft Kinect power supply

5V/1A: General power supply

Status LED: Indicates Kobuki's status

Green: Kobuki is turned on and battery at high voltage level

Orange: On - Low battery voltage level (please charge soon)

Green blinking: On - Battery charging

Off: Kobuki is turned off.

LED1/2: Programmable LEDs

USB: Data connection

B0/1/2: Buttons

Firmware switch: Enable/disables the firmware update mode

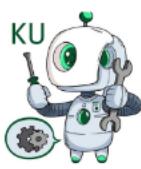


Robot Hardware

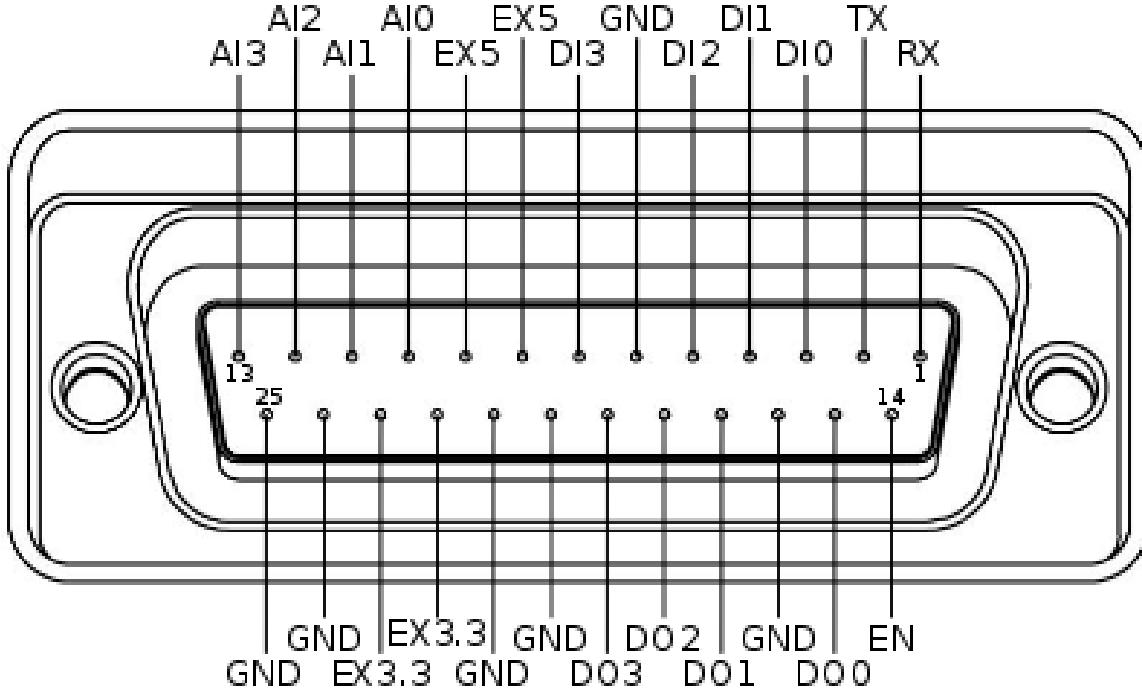


Power

- 5V@1A Molex PN : 43650-0218 – for custom boards
- 12V@1.5A : Molex PN : 43045-0224 – specially supporting the Kinect
- 12V@5A : Molex PN : 3929-9023 – for high powered accessories (e.g. robotic arm)
- 19V@2A : Molex PN : 3928-9068 – for recharging netbooks



Robot Hardware



I/O Port

- DB25 pin D-SUB Female connector

RX / TX: Serial data connection (RS232; used voltage level is 3.3V!)

EX3.3 / EX5: 3.3V/1A and 5V/1A power supply

DIO - 3: 4 x Digital input (high: 3.3 – 5V, low: 0V)

DOO - 3: 4 x Digital output (open-drain, pull-up resistor required)

AI0-3: 4 x Analog input (12bit ADC: 0 – 4095, 0 – 3.3V)

GND: Ground

EN: Used for detecting an external board (connect to external ground)

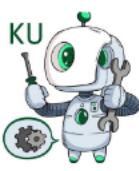


Robot Hardware



Battery

- 4S1P/4S2P Battery Pack Connector



Robot Hardware

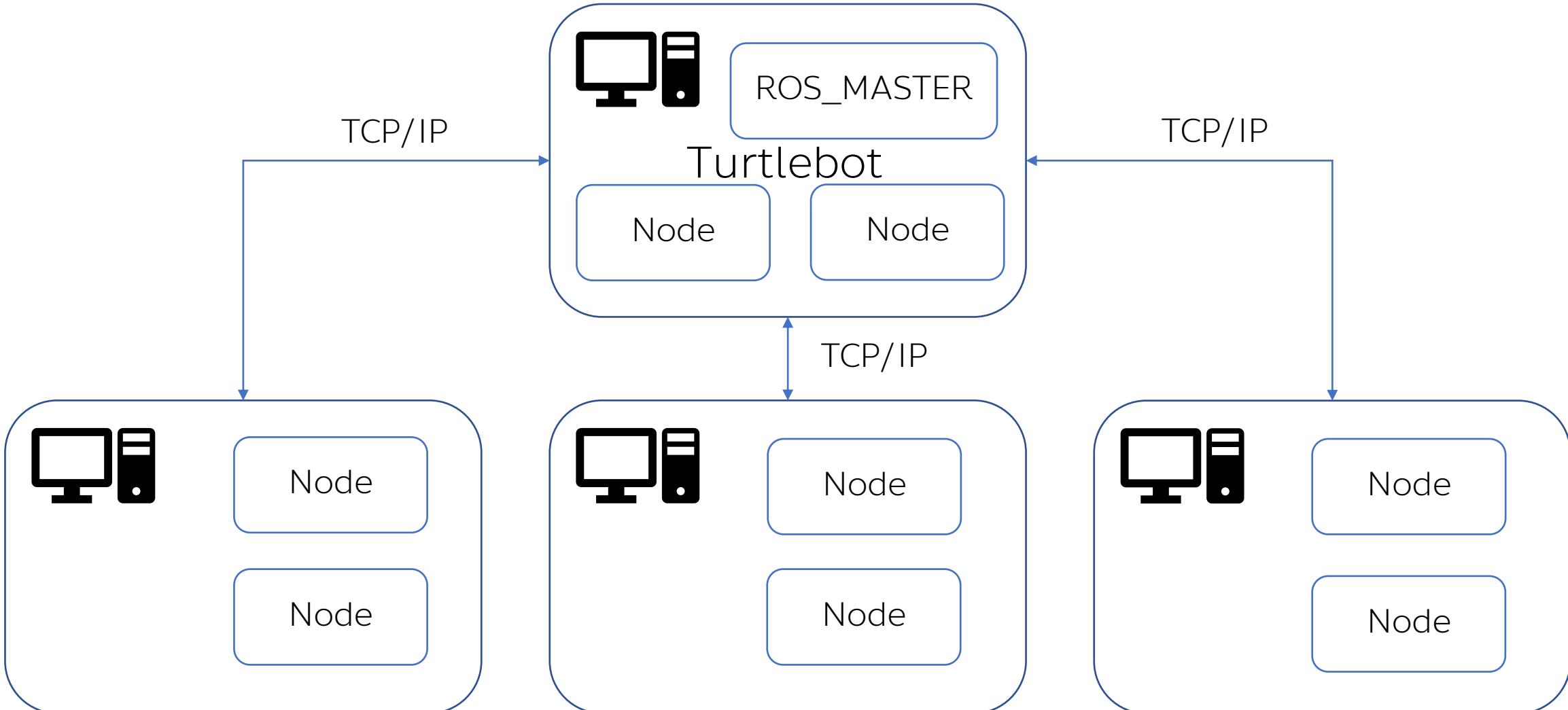


Cables

- 12V@1.5A : Molex PN : 43025-0200 – specially supporting the Kinect



Teleoperation Network config





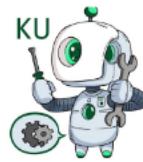
Teleoperation Networkconfig

Turtlebot - PC Configuration

ROS Master is running in TurtleBot
ROS_MASTER_URI=
http://localhost:11311
ROS_HOSTNAME=
<TURTLEBOT_IP>

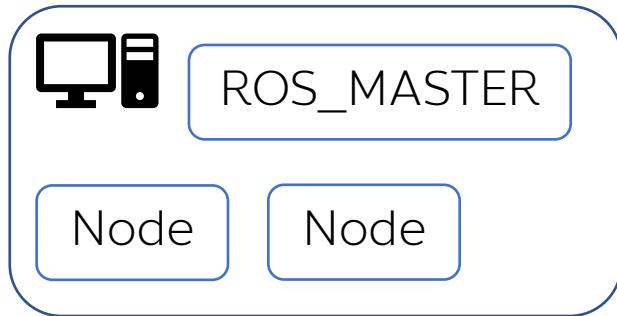


ROS_MASTER_URI=
http://<TURTLEBOT_IP>:
11311
ROS_HOSTNAME=<PC_IP>



Teleoperation Network config

On the Turtlebot (or ROS_MASTER)

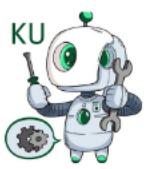


Setup SSH server

```
$ sudo apt-get install openssh-server
```

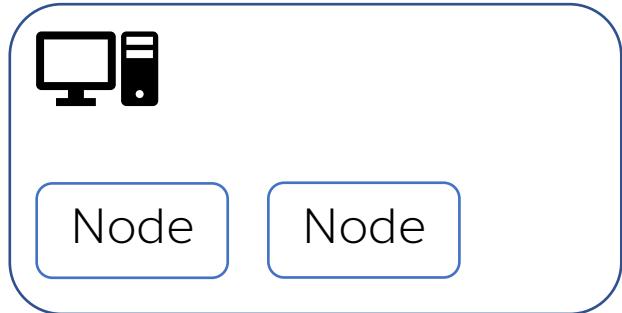
Setup Hostname and ROS_MASTER

```
$ echo export ROS_MASTER_URI=http://localhost:11311 >> ~/.bashrc
$ echo export ROS_HOSTNAME=IP_OF_TURTLEBOT >> ~/.bashrc
```



Teleoperation Network config

On the Remote PC

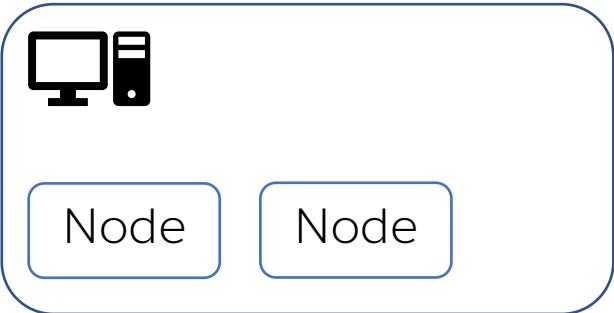


Setup Hostname and ROS_MASTER

```
$ echo export ROS_MASTER_URI=http://localhost:11311 >> ~/.bashrc
$ echo export ROS_HOSTNAME=IP_OF_PC >> ~/.bashrc
```

Teleoperation Network Config

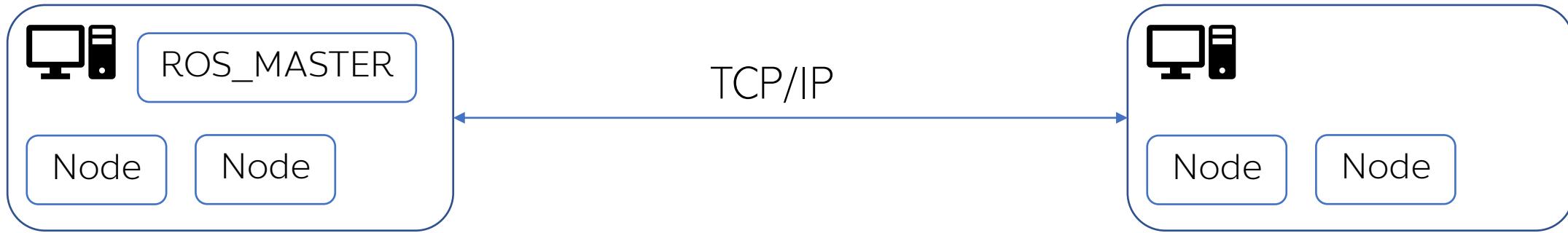
On the Remote PC



Setup Hostname and ROS_MASTER

```
$ echo export ROS_MASTER_URI=http://localhost:11311 >> ~/.bashrc
$ echo export ROS_HOSTNAME=IP_OF_PC >> ~/.bashrc
```

Teleoperation Network Config



On the ROS_MASTER

Verify from ROS_MASTER to Remote PC

```
$ roscore  
$ rosrun my_package talker.py
```

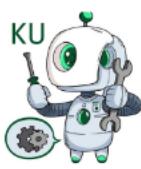
On the Remote PC

```
$ rostopic list  
$ rostopic echo /chatter
```

Verify from Remote PC to ROS_MASTER

```
$ roscore  
$ rostopic list  
$ rostopic echo /chatter
```

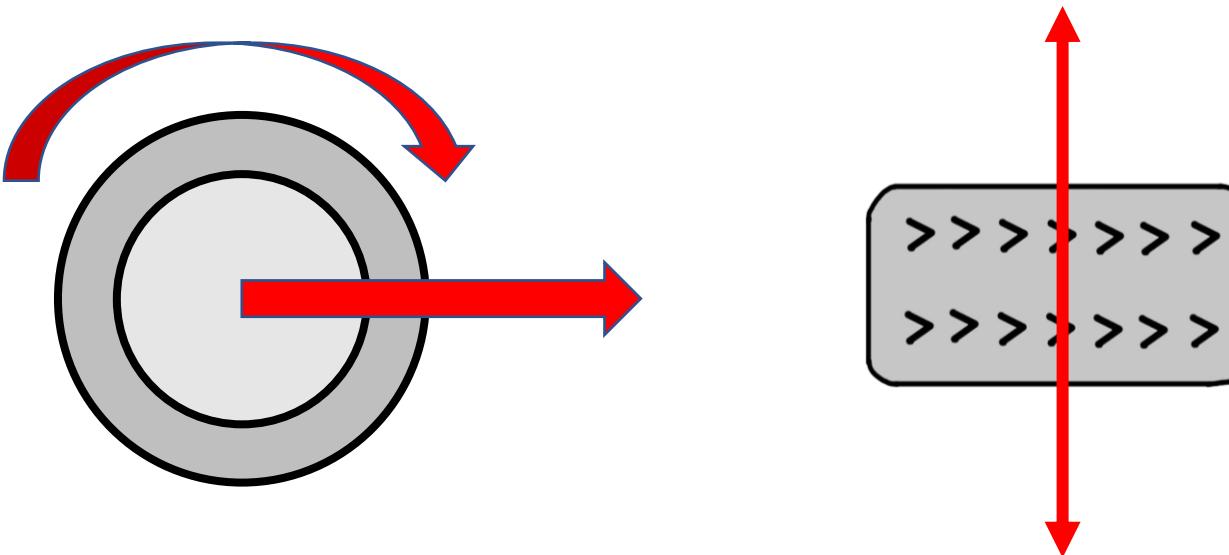
```
$ rostopic list  
$ rosrun my_package talker.py
```



Wheel Locomotion

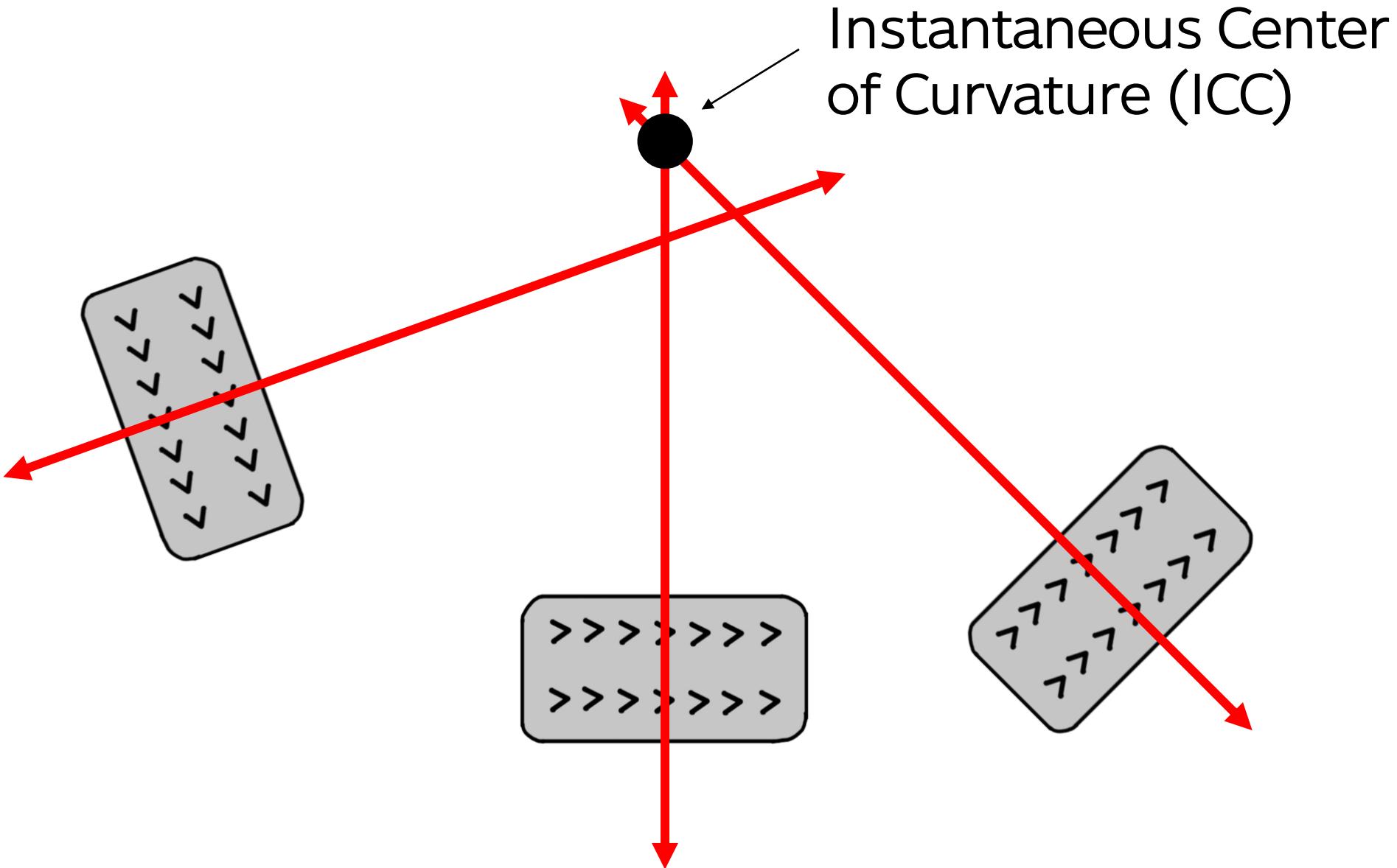
Locomotion : Power of motion from place to place

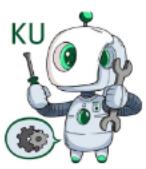
- Differential Drive (turtlebot)
- Car Drive (Ackerman steering)
- Mecanum Wheel Drive



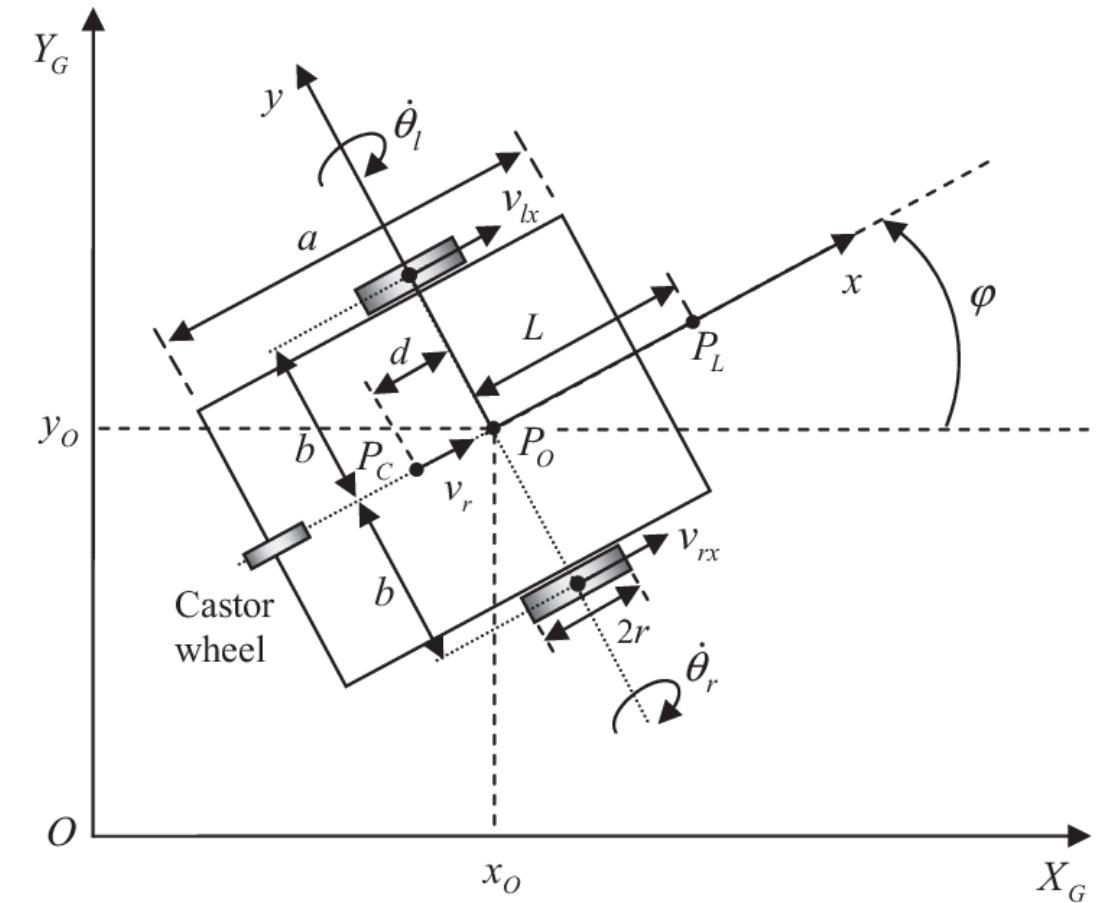
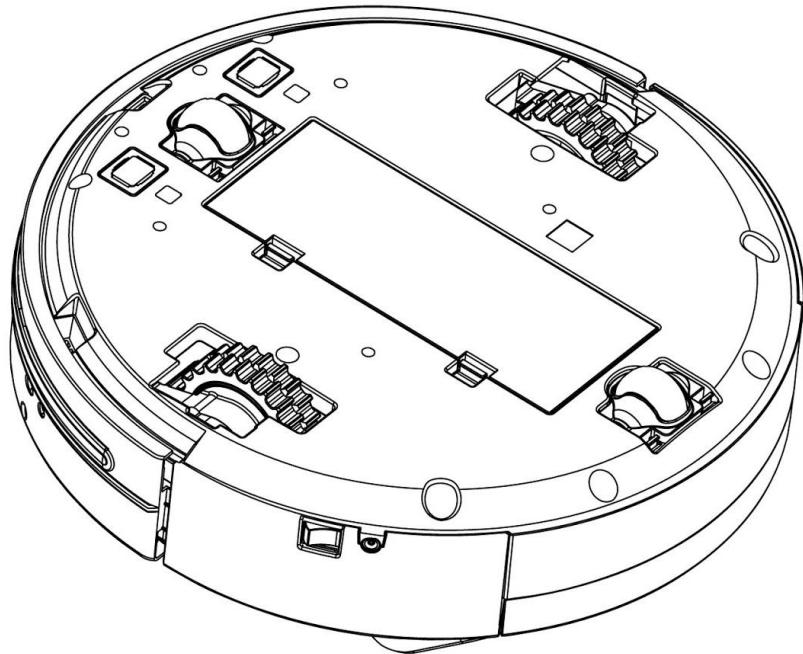


Wheel Locomotion



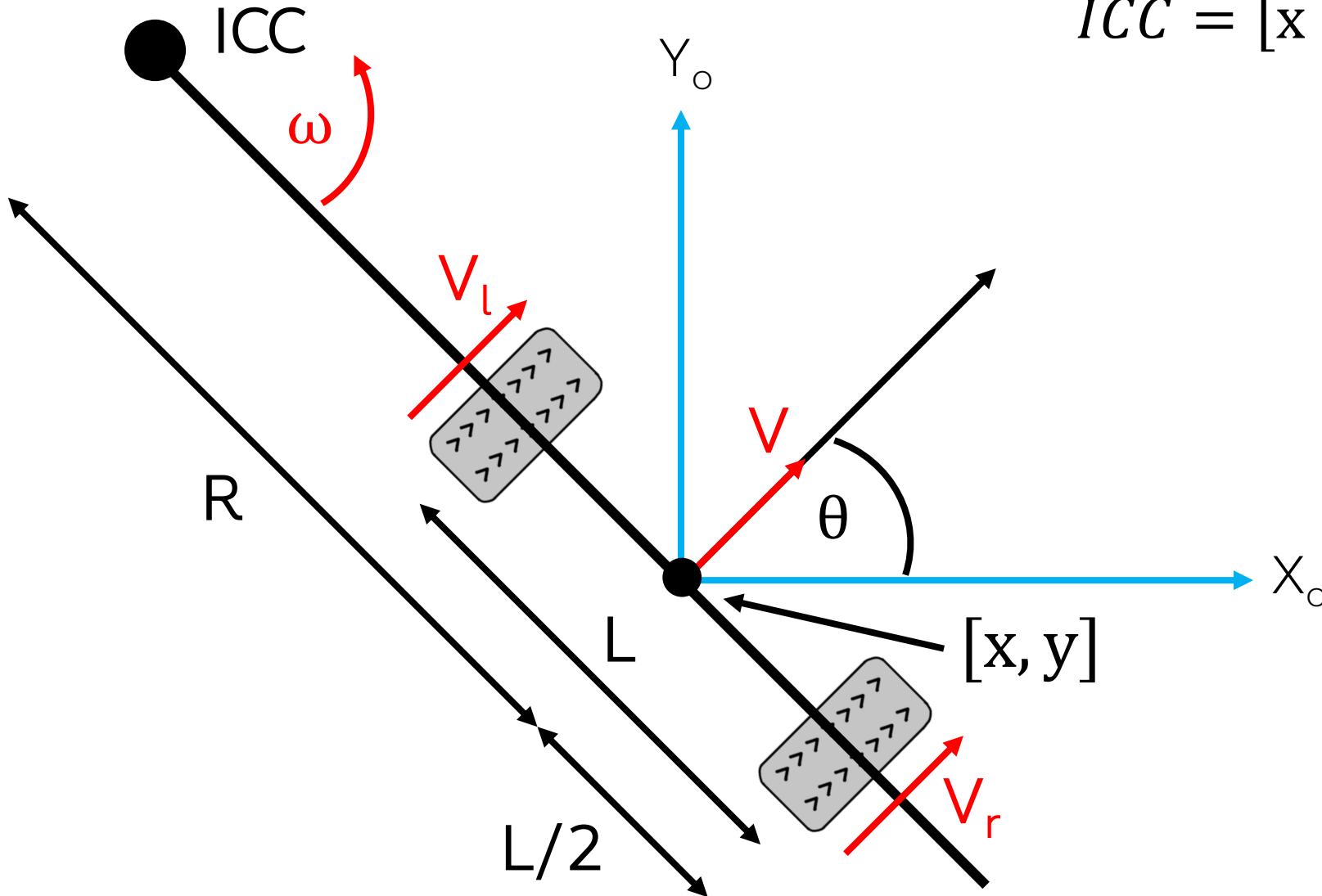


Wheel Locomotion (Differential Drive)



Differential Drive robot

Wheel Locomotion (Differential Drive)



$$ICC = [x - R\sin(\theta), y + R\cos(\theta)]$$

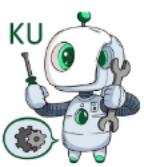
$$\omega \left(R + \frac{L}{2} \right) = V_r$$

$$\omega \left(R - \frac{L}{2} \right) = V_l$$

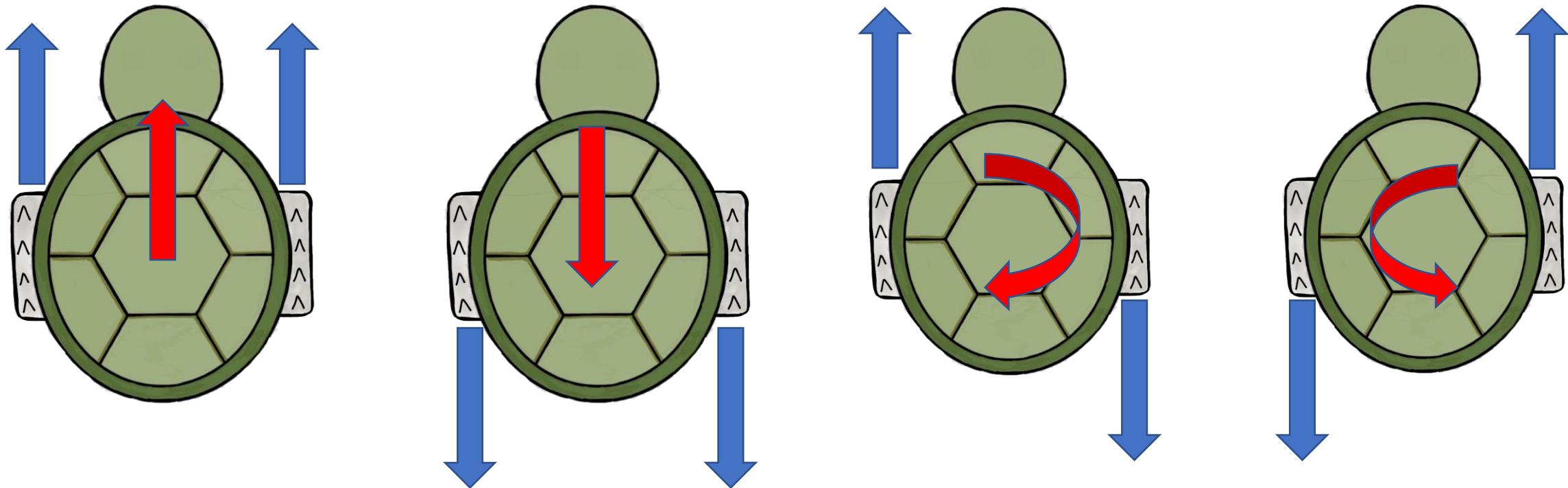
$$R = \frac{L(V_l + V_r)}{2(V_r - V_l)}$$

$$\omega = \frac{(V_r - V_l)}{L}$$

$$V = \frac{(V_r + V_l)}{2}$$

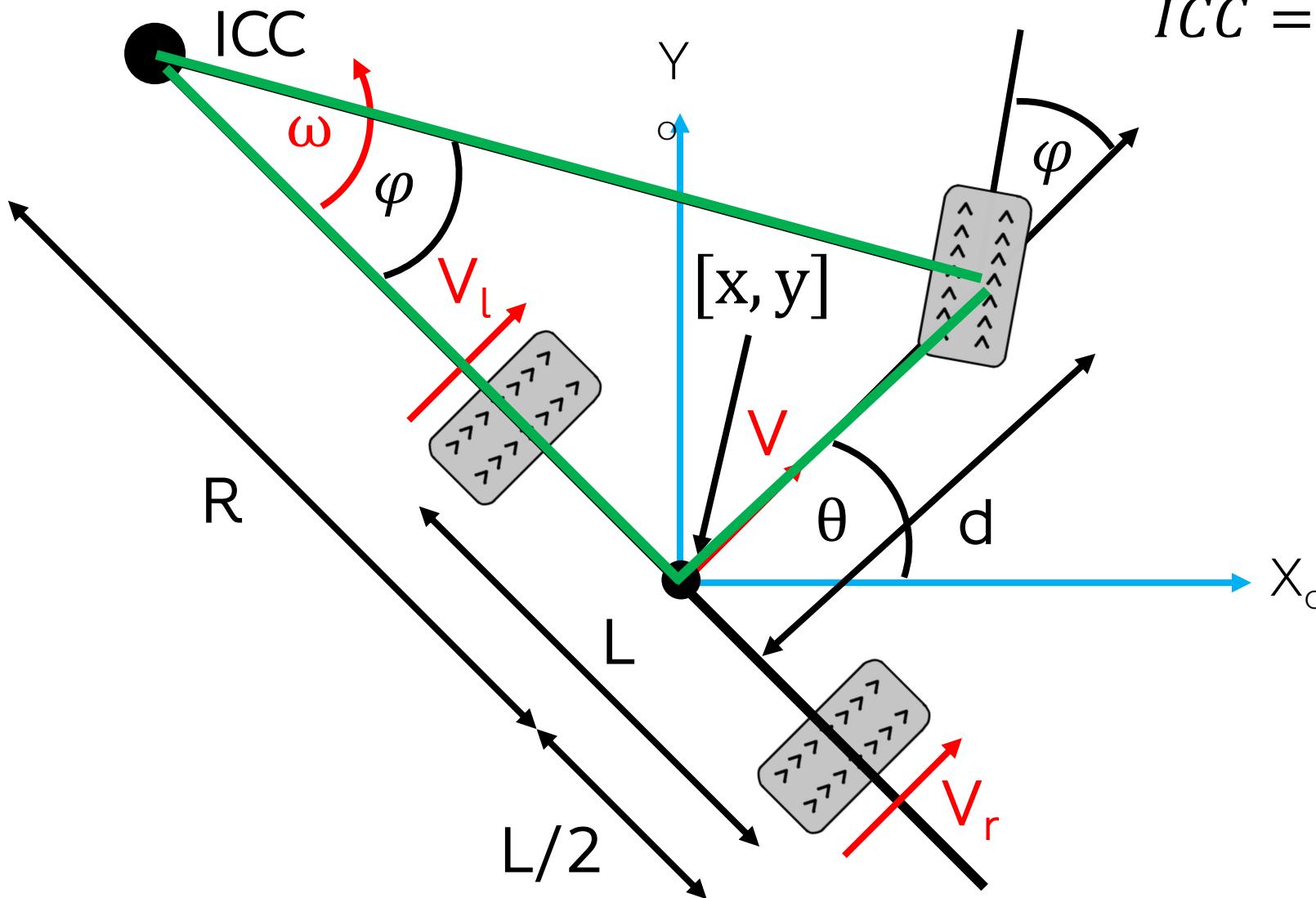


Wheel Locomotion



Differential Drive Robot Motion

Wheel Locomotion (Ackermann (car) Drive)



$$ICC = [x - R\sin(\theta), y + R\cos(\theta)]$$

$$R = \frac{d}{\tan(\varphi)}$$

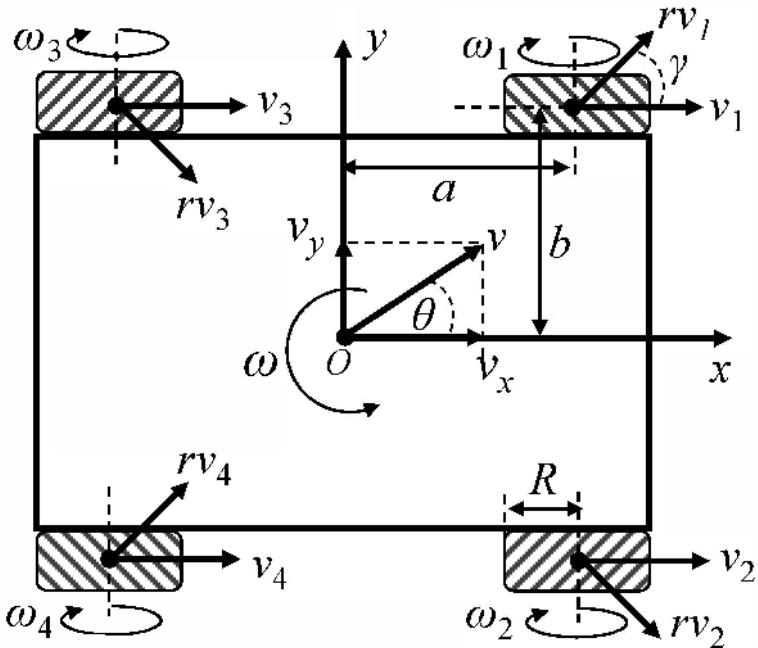
$$\omega \left(R + \frac{L}{2} \right) = V_r$$

$$\omega \left(R - \frac{L}{2} \right) = V_l$$

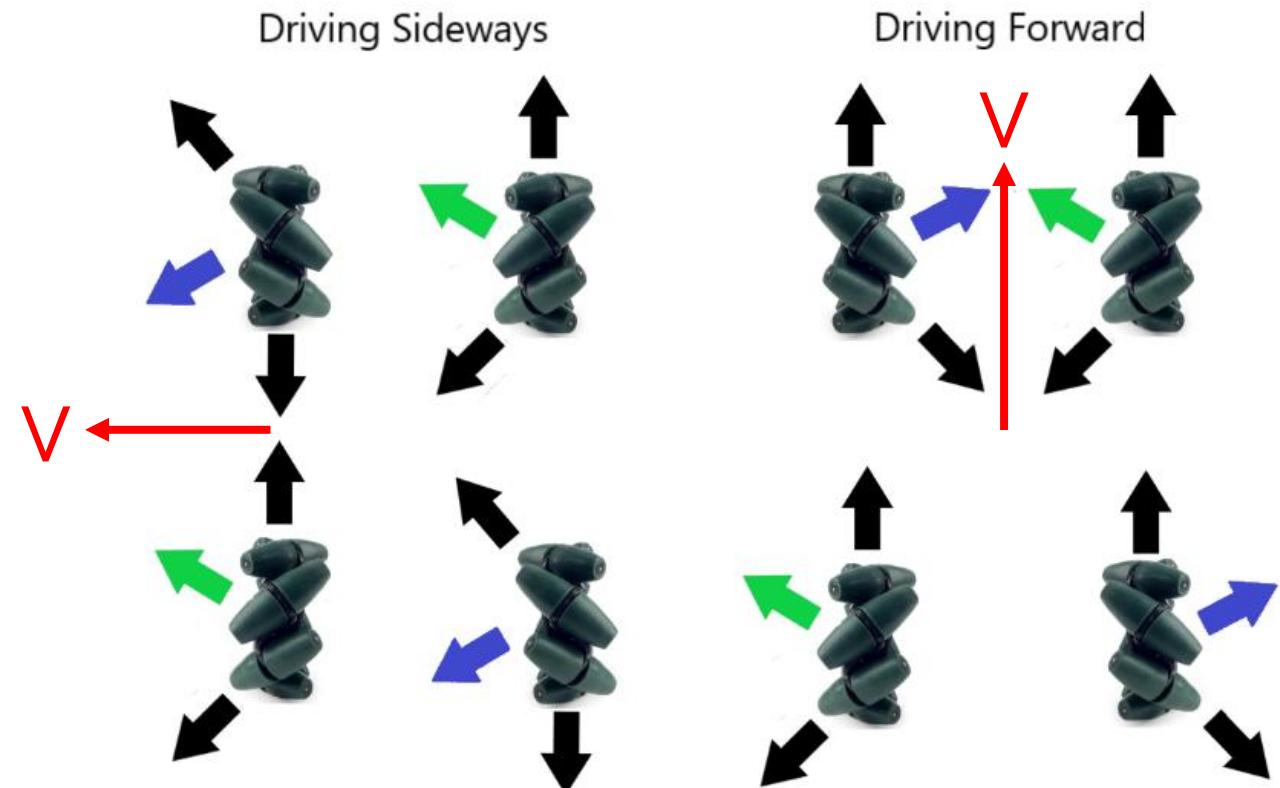
$$R = \frac{L(V_l + V_r)}{2(V_r - V_l)}$$

$$\omega = \frac{(V_r - V_l)}{L}$$

Wheel Locomotion (Mecanum Wheel Drive)



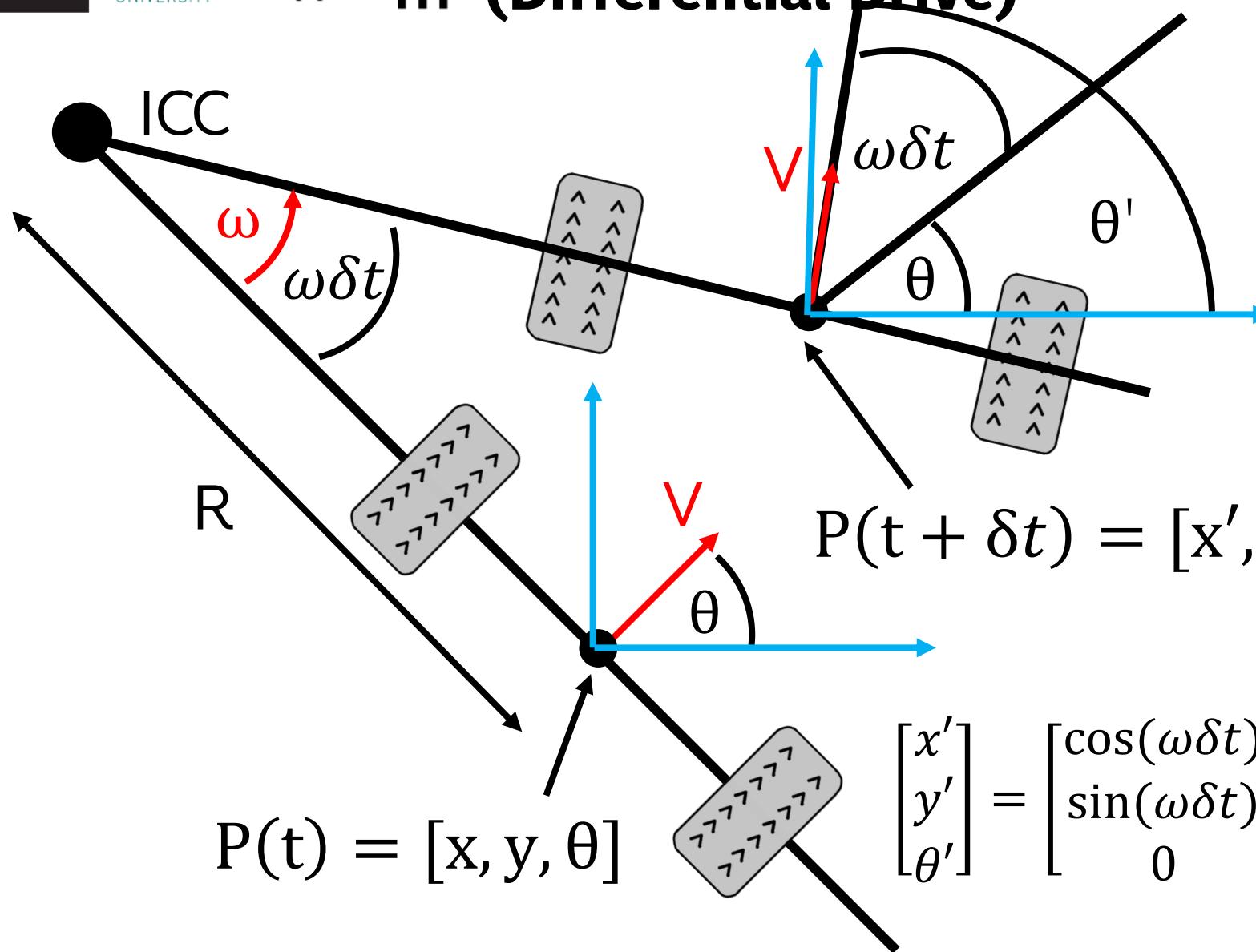
$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{R} \begin{bmatrix} 1 & -1 & -(a+b) \\ 1 & 1 & (a+b) \\ 1 & 1 & -(a+b) \\ 1 & -1 & (a+b) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_\theta \end{bmatrix}$$



Ref :

- [1] Maulana, E., Muslim, M.A., & Hendrayawan, V. (2015). Inverse kinematic implementation of four-wheels mecanum drive mobile robot using stepper motors.
- [2] <https://ftccats.github.io/software/ProgrammingMecanumWheels.html>

Wheel Locomotion (Differential Drive)



$$x(t) = \int_0^t v(t') \cos[\theta(t')] dt'$$

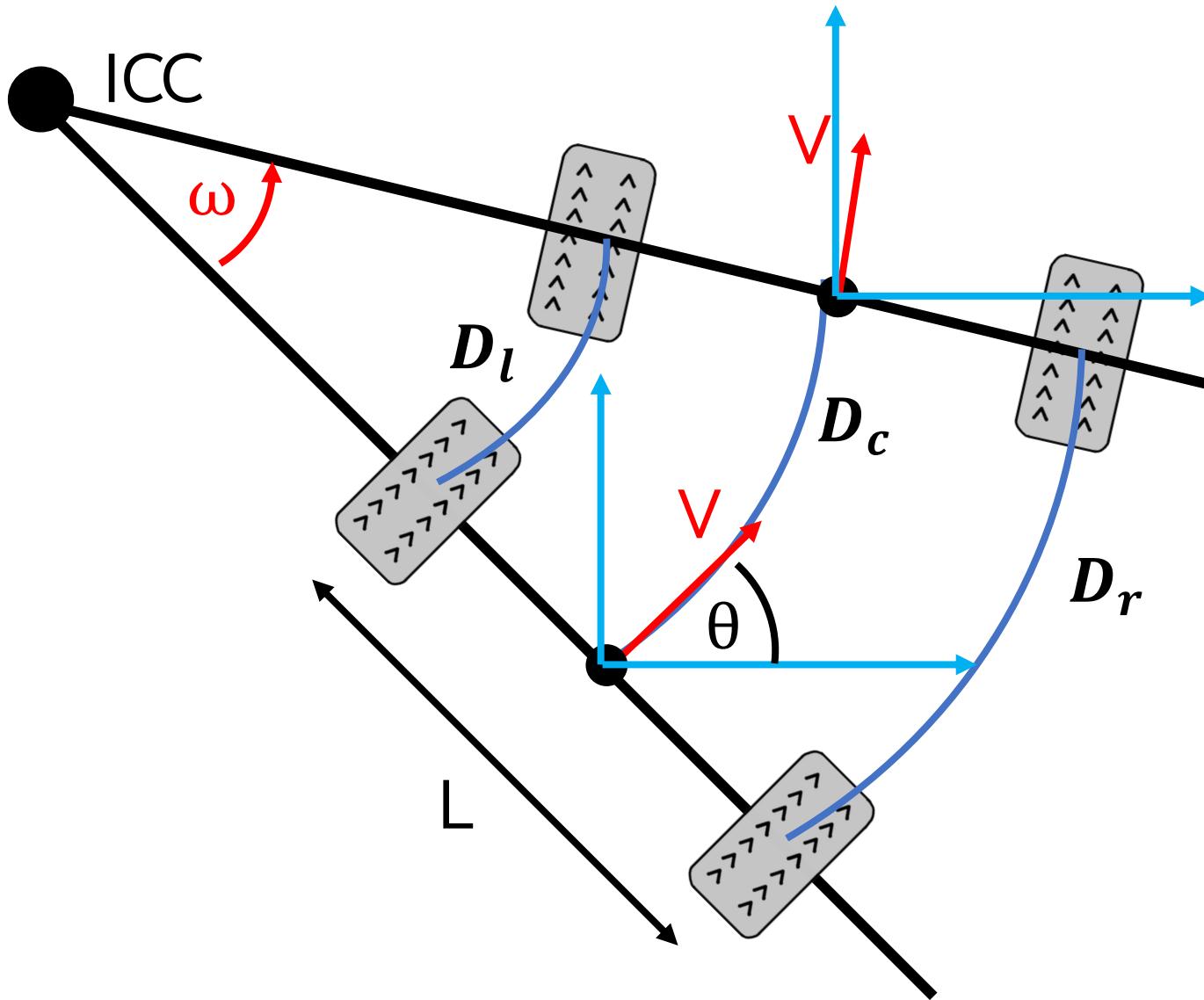
$$y(t) = \int_0^t v(t') \sin[\theta(t')] dt'$$

$$\theta(t) = \int_0^t \omega(t') dt'$$

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega\delta t) & -\sin(\omega\delta t) & 0 \\ \sin(\omega\delta t) & \cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix}$$



Wheel Locomotion (Differential Drive)

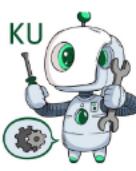


$$V\delta t = D_c$$
$$D_c = \frac{(D_l + D_r)}{2}$$

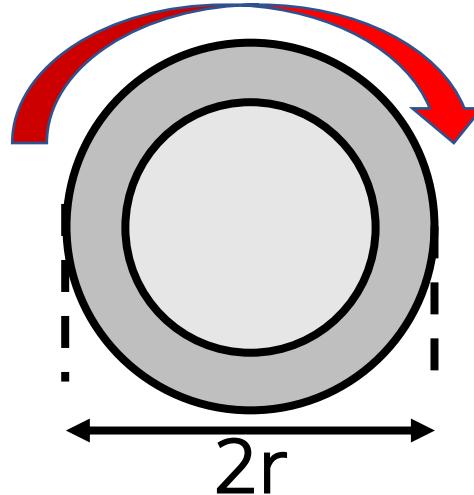
$$x' = x + D_c \cos(\theta)$$

$$y' = y + D_c \sin(\theta)$$

$$\theta' = \theta + \frac{(D_r - D_l)}{L}$$



Encoder

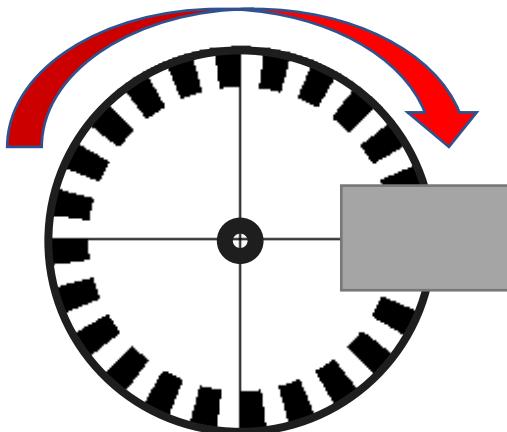


- Encoder has N “ticks” per revolution
- Most wheel Encoder give the **total tick** count since the beginning (Increment Encoder)

For Each Wheel

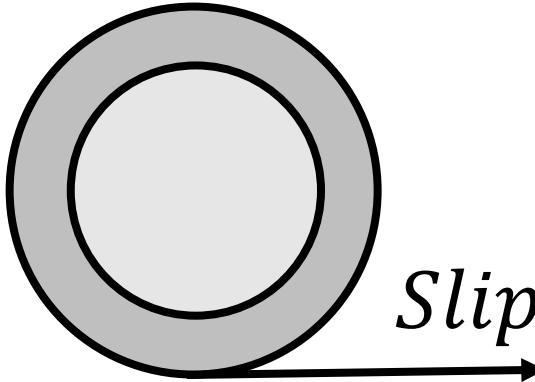
$$\Delta\text{tick} = \text{tick}' - \text{tick}$$

$$D = 2\pi r \frac{\Delta\text{tick}}{N}$$

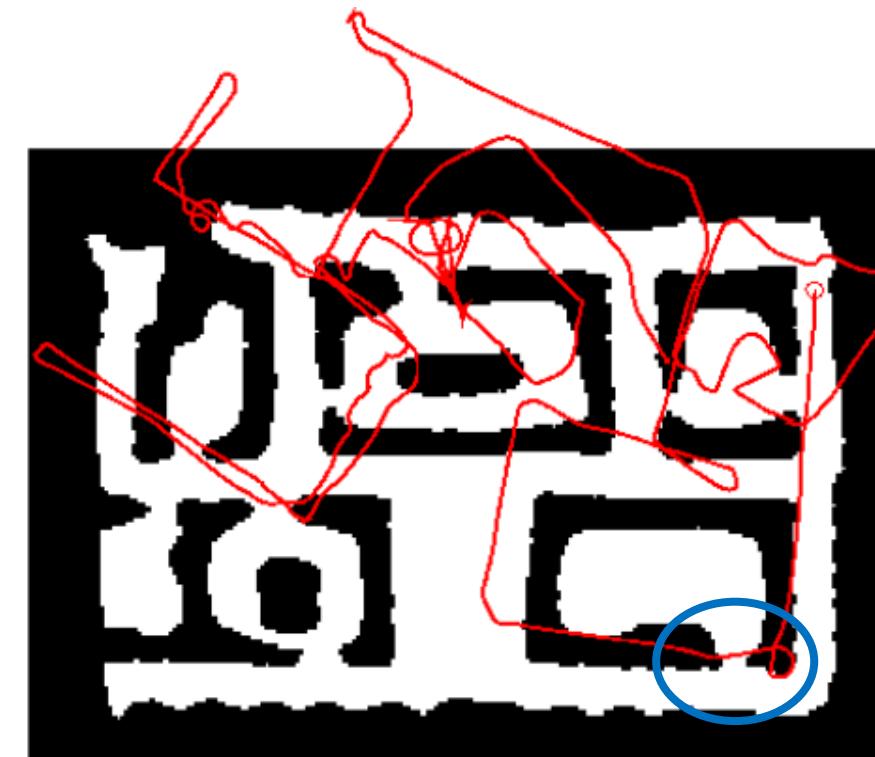




Dead Reckoning and Odometry



- Estimating the motion based on the issued controls/wheel **encoder readings**
- Integrated over time (**Cumulative Error**)
- **Drift**



|| Software Setup

Install all kobuki package

```
$ sudo apt-get install ros-kinetic-kobuki*
```

Install turtlebot package

```
$ sudo apt-get install ros-kinetic-turtlebot
$ sudo apt-get install ros-kinetic-turtlebot-*
```

Ref : <https://robots.ros.org/kobuki/>
<http://wiki.ros.org/turtlebot/Tutorials/indigo>

Software Setup

Environment Setup

Add following command in `~/.bashrc` file

```
export TURTLEBOT_BASE=kobuki
export TURTLEBOT_STACKS=hexagons
export TURTLEBOT_3D_SENSOR=kinect
export TURTLEBOT_SERIAL_PORT=/dev/kobuki
```

|| Software Setup

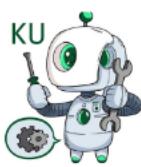
Verify software setup

- 1) Turn ON Kobuki
- 2) Connect computer to Kobuki via USB
- 3) Run following command:

```
$ roscore
```

- 4) Open new terminal

```
$ roslaunch turtlebot_bringup minimal.launch
```



Control The Robot

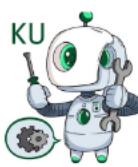
Control with the Controller

```
$ roslaunch turtlebot_teleop logitech.launch
```

LB : Hold to Enable



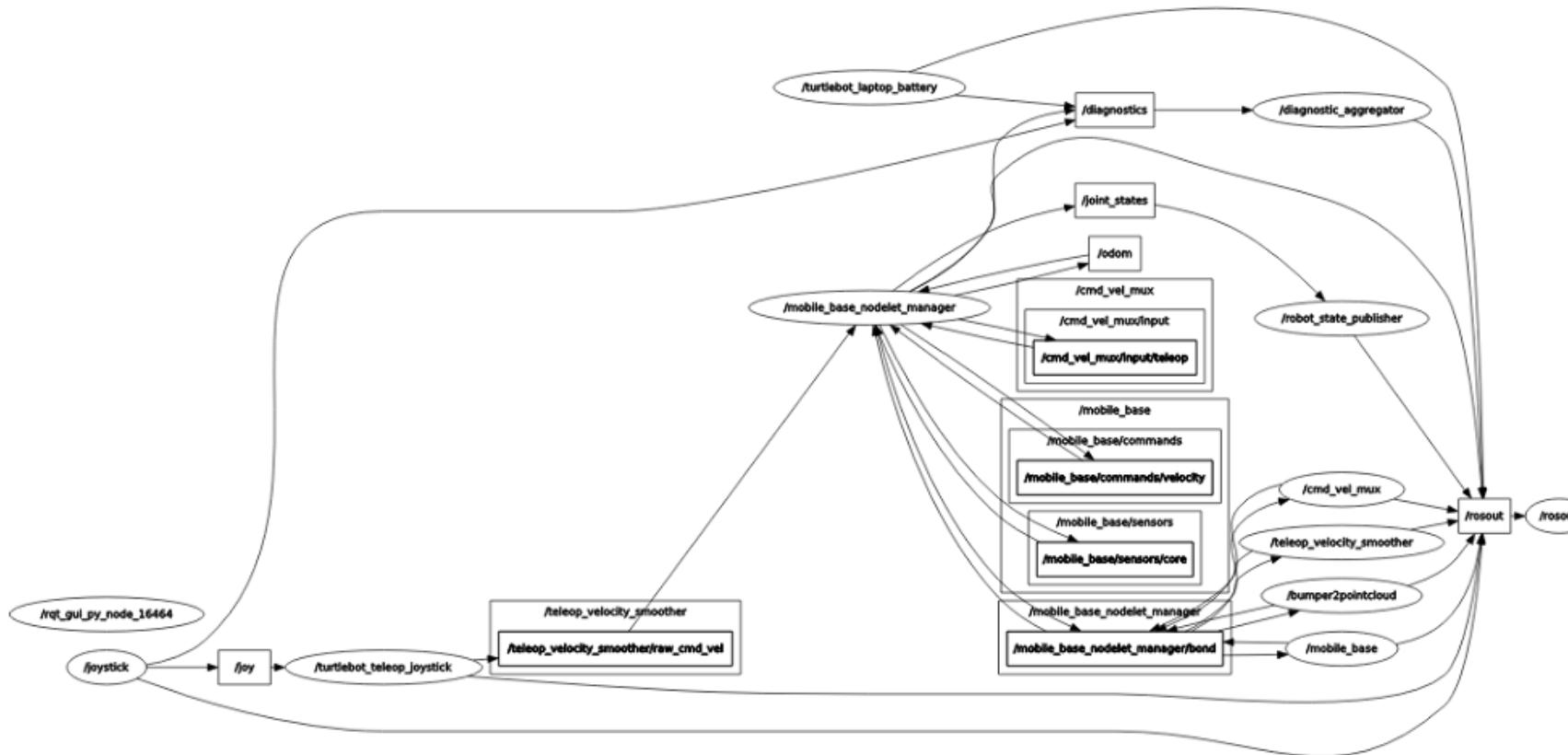
Using Left Analog to Control



Control The Robot

To create a dynamic graph of what's going on in the system run:

```
$ rosrun rqt_graph rqt_graph
```

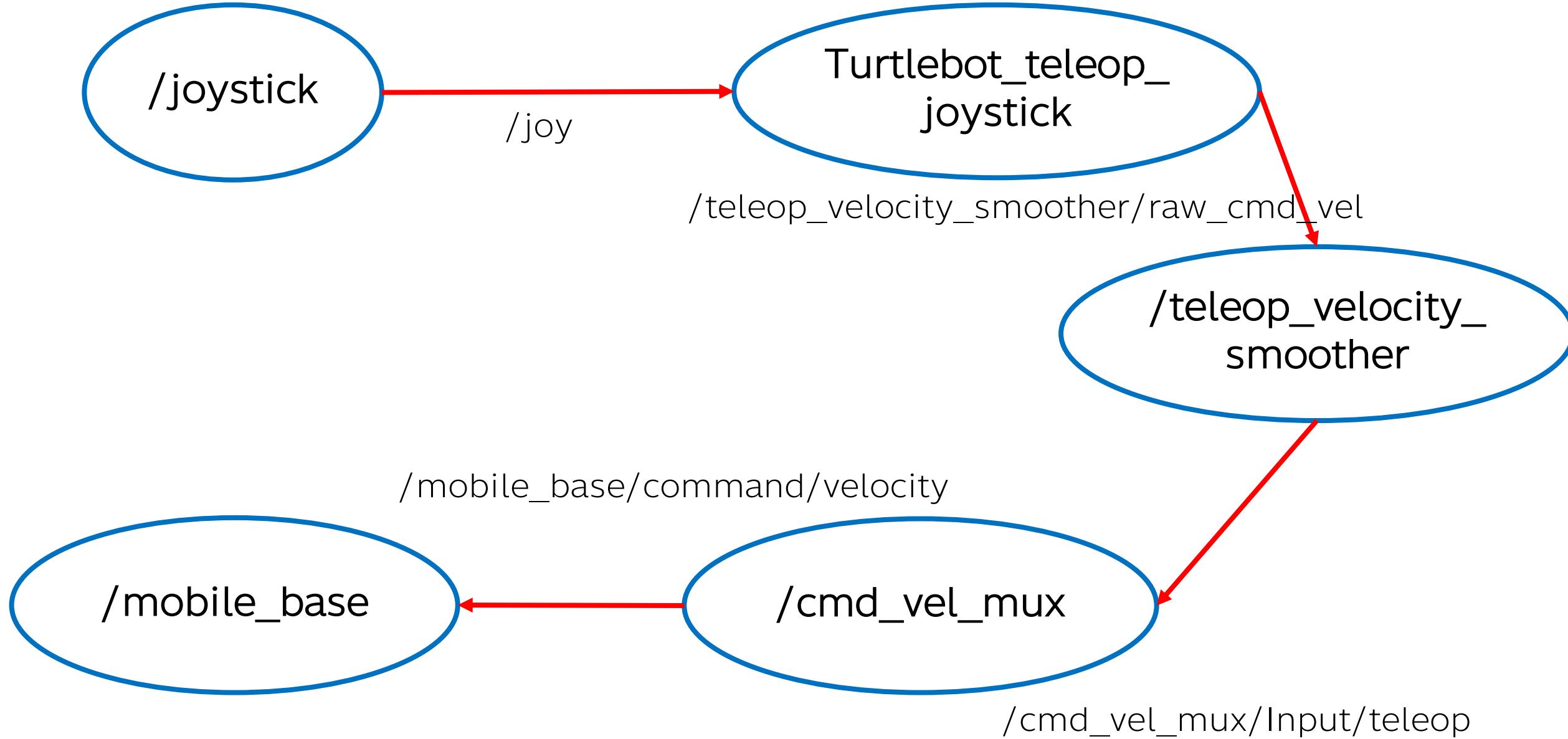


Display the results as:

Node/Topics (active)



Control The Robot



Control The Robot

```
user@ros-workshop:~$ rostopic type /joy
sensor_msgs/Joy
user@ros-workshop:~$ rostopic type /teleop_velocity_smoother/raw_cmd_vel
geometry_msgs/Twist
user@ros-workshop:~$ rostopic type /cmd_vel_mux/input/teleop
geometry_msgs/Twist
user@ros-workshop:~$ rostopic type /mobile_base/commands/velocity
geometry_msgs/Twist
```

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3 linear
Vector3 angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Control The Robot

File: **geometry_msgs/Vector3.msg**

Raw Message Definition

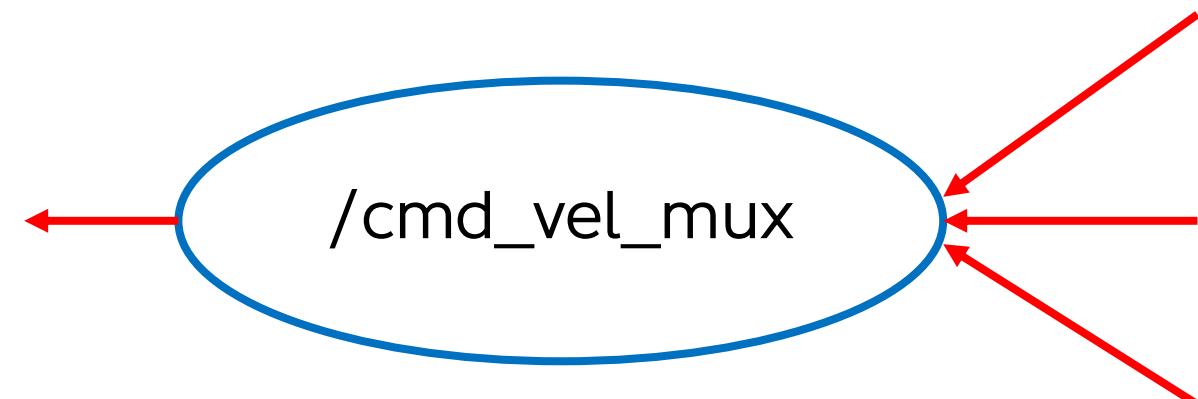
```
# This represents a vector in free space.  
# It is only meant to represent a direction. Therefore, it does not  
# make sense to apply a translation to it (e.g., when applying a  
# generic rigid transformation to a Vector3, tf2 will only apply the  
# rotation). If you want your data to be translatable too, use the  
# geometry_msgs/Point message instead.  
  
float64 x  
float64 y  
float64 z
```

Control The Robot

/opt/ros/kinetic/share/turtlebot_bringup/param/mux.yaml

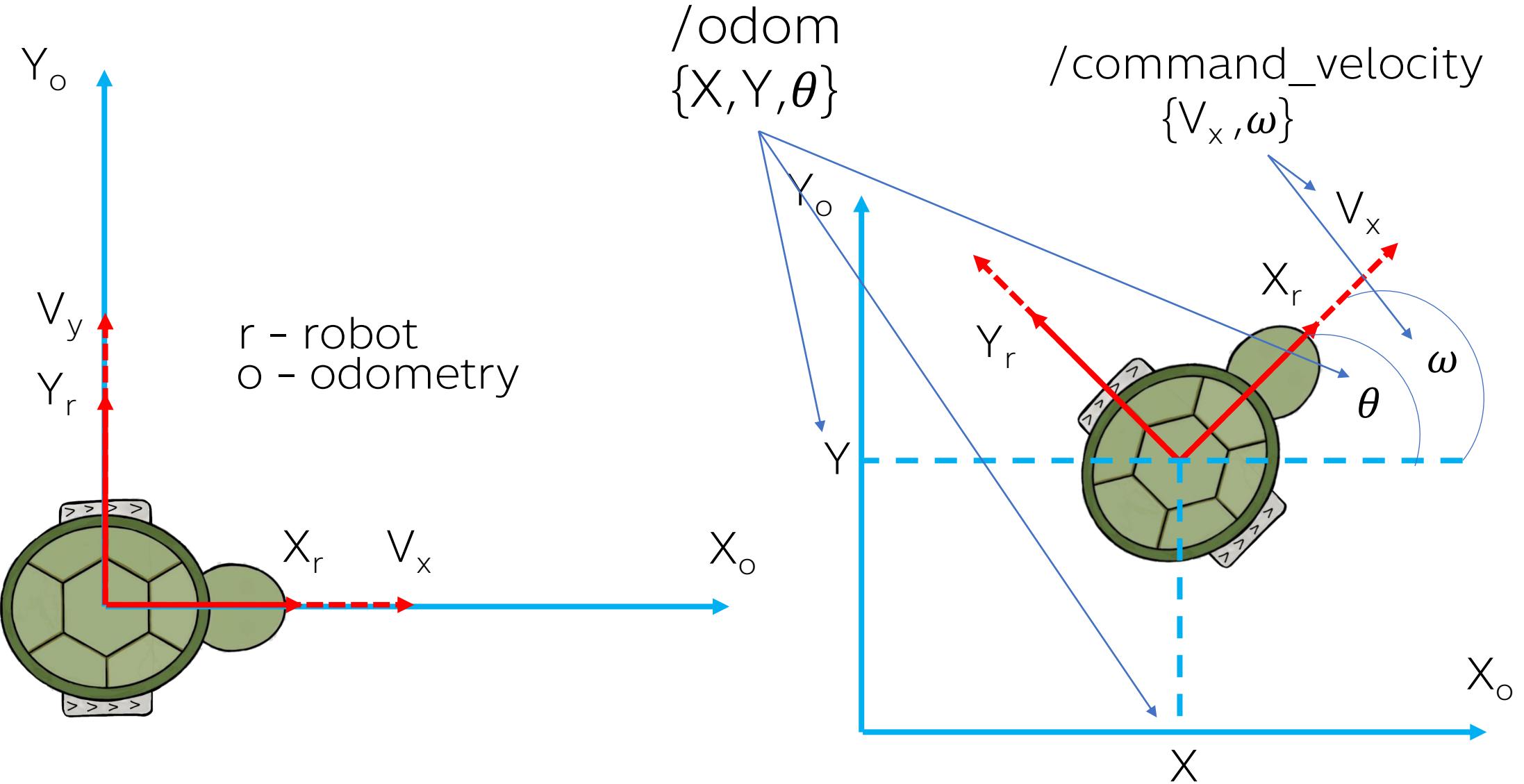
subscribers:

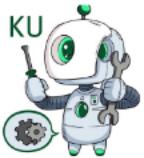
- **name:** "Safe reactive controller"
topic: "input/safety_controller"
timeout: 0.2
priority: 10
- **name:** "Teleoperation"
topic: "input/teleop"
timeout: 1.0
priority: 7
- **name:** "Switch"
topic: "input/switch"
timeout: 1.0
priority: 6
- **name:** "Navigation"
topic: "input/navi"
timeout: 1.0
priority: 5



[https://github.com/yujinrobot/yujin_ocs/tree/
devel/yocs_cmd_vel_mux](https://github.com/yujinrobot/yujin_ocs/tree/devel/yocs_cmd_vel_mux)

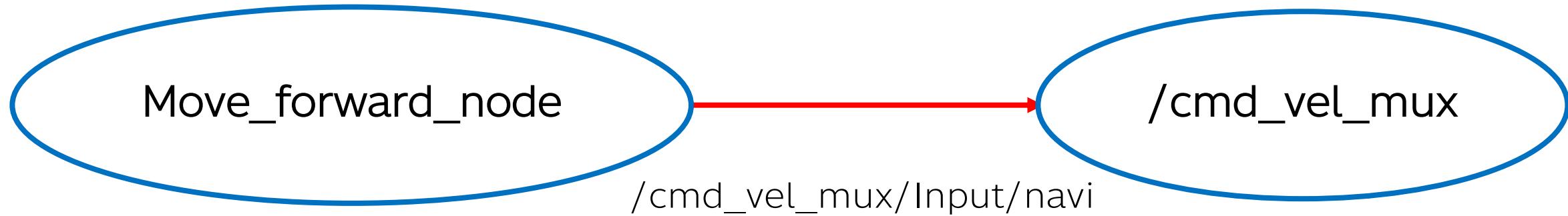
Control The Robot





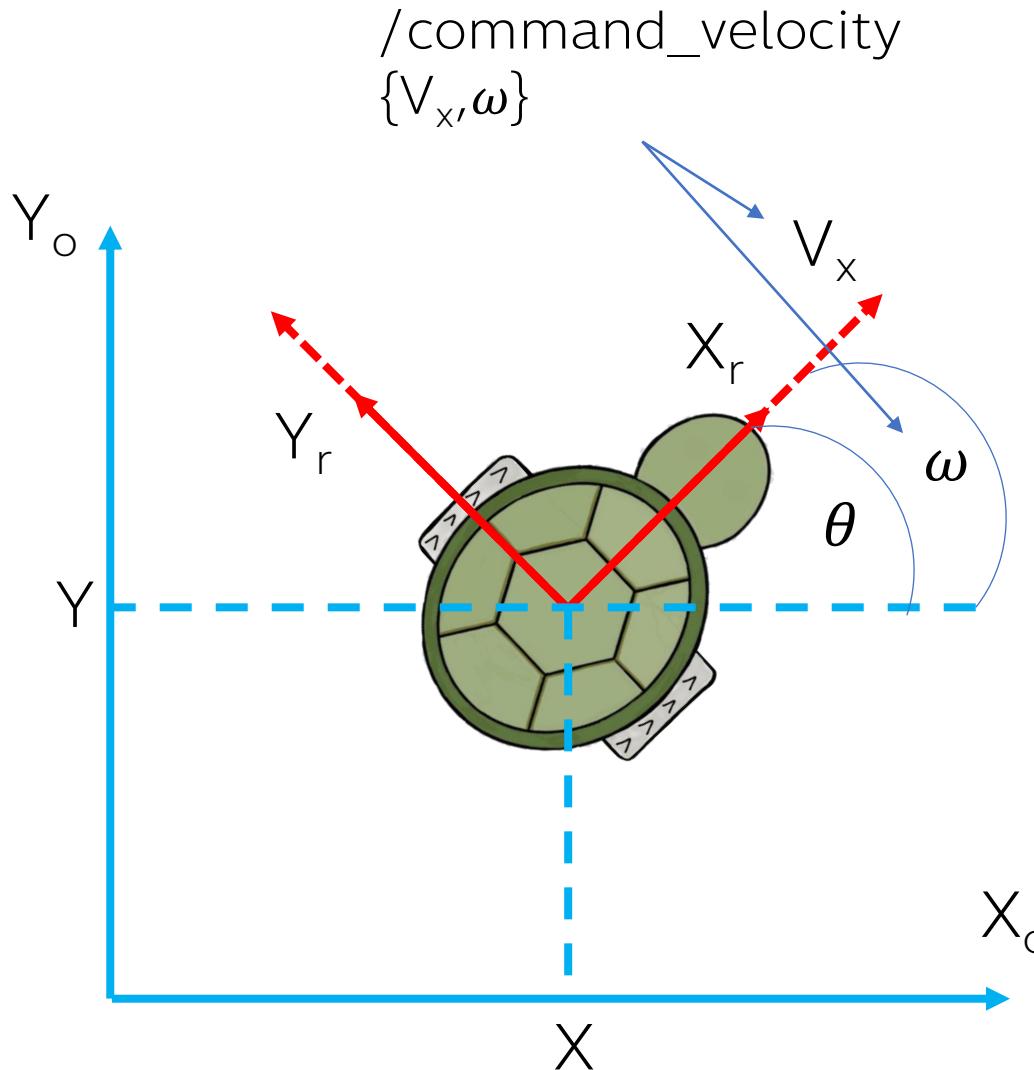
Control The Robot

1. Control robot velocity without feedback





Control The Robot



geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular

Twist

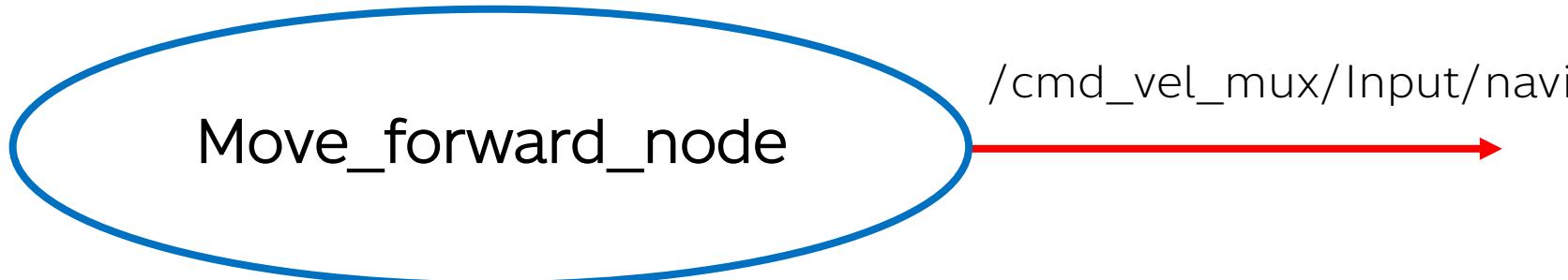
linear

$x \leftarrow V_x$
 y
 z

angular

x
 y
 $z \leftarrow \omega$

Control The Robot



```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

if __name__ == '__main__':
    try:
        MoveForward()
    except:
        rospy.loginfo(" MoveForward node terminated.")
```

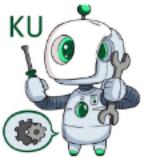
Control The Robot

```
class MoveForward():
    def __init__(self):
        rospy.init_node('move_forward_node', anonymous=False)
        rospy.loginfo("To stop Turtlebot press CTRL + C")
        rospy.on_shutdown(self.shutdown)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input/navi', Twist,
queue_size=10)
        self.move_forward(0.05, 0.0)
```

```
def move_forward(self, vel_x, vel_z):
    r = rospy.Rate(10)
    move_cmd = Twist()
    move_cmd.linear.x = vel_x
    move_cmd.angular.z = vel_z
    while not rospy.is_shutdown():
        self.cmd_vel_pub.publish(move_cmd)
        r.sleep()
```

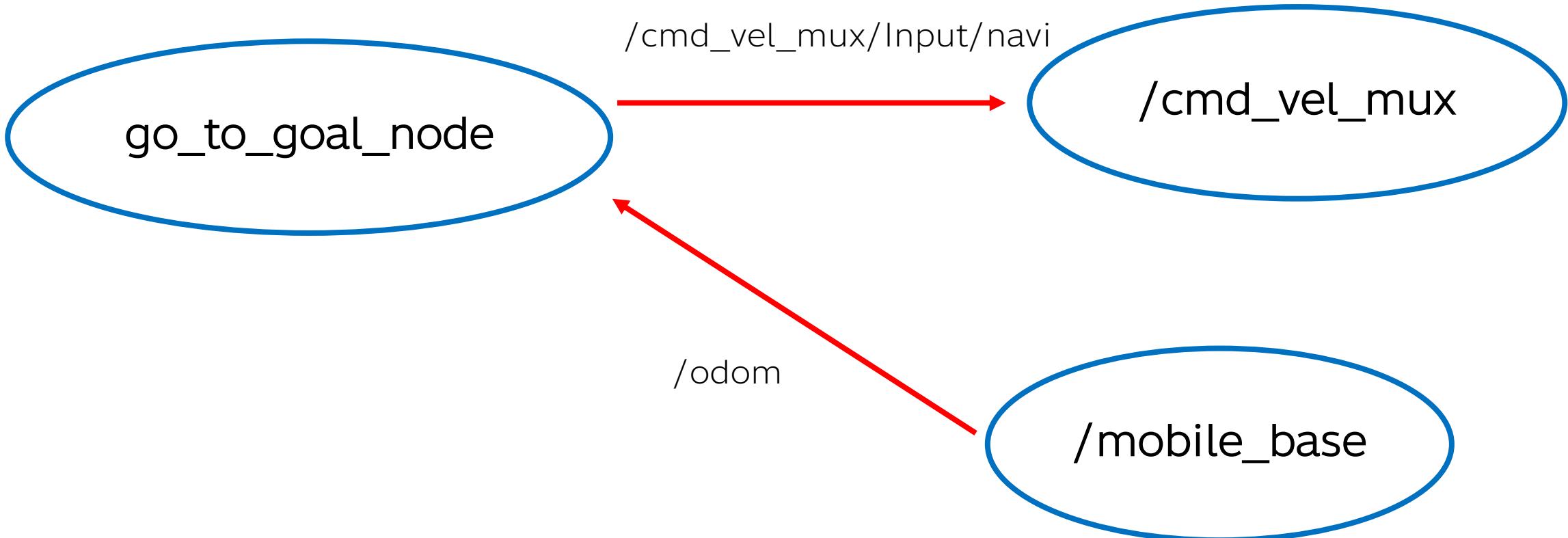
Control The Robot

```
def shutdown(self):
    # stop turtlebot
    rospy.loginfo("Stop TurtleBot")
    # a default Twist has linear.x of 0 and angular.z of 0. So it'll
stop TurtleBot
    self.cmd_vel_pub.publish(Twist())
    # sleep just makes sure TurtleBot receives the stop command prior
to shutting down the script
    rospy.sleep(1)
```

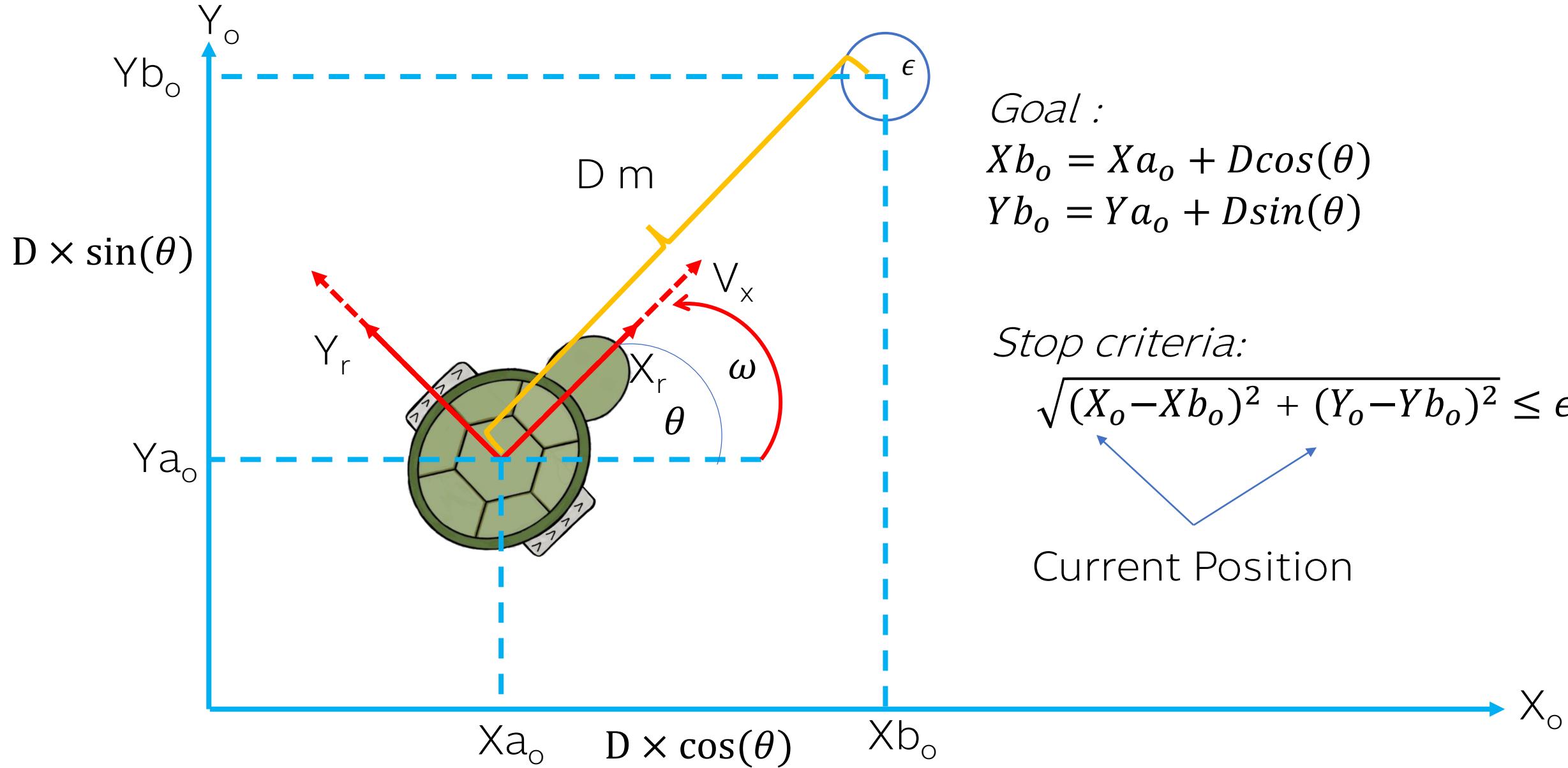


Control The Robot

2. Control robot velocity with feedback



Control The Robot



Control The Robot

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from math import cos, sin, sqrt
class GoToGoal():
    def __init__(self):
        rospy.init_node('go_to_goal_node', anonymous=False, log_level=rospy.DEBUG)
        rospy.loginfo("To stop Turtlebot press CTRL + C")
        rospy.on_shutdown(self.shutdown)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel_mux/input/navi' , Twist, queue_size=10)
        self.odom_sub = rospy.Subscriber('odom', Odometry, self.odom_callback)
        self.goal_pose_x = 0
        self.goal_pose_y = 0
        self.current_pose_x = 0
        self.current_pose_y = 0
        self.current_theta = 0
        self.vel_x = 0
        self.tol = 0.1
        self.move_cmd = Twist()
        rospy.wait_for_message("odom", Odometry) # wait for odometry data
```

Control The Robot

```
def odom_callback(self, odom_data):
    self.current_pose_x = odom_data.pose.pose.position.x
    self.current_pose_y = odom_data.pose.pose.position.y
    orientation_q = odom_data.pose.pose.orientation # orientation data
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]

    # change orientation data to radius
    (roll, pitch, yaw) = euler_from_quaternion(orientation_list)
    self.current_theta = yaw # currently radius of robot

def go_to_goal(self, vel_x):
    self.move_cmd.linear.x = vel_x # set velocity
    while not rospy.is_shutdown():
        if self.check_stop():
            self.shutdown()
            break
        else:
            self.cmd_vel_pub.publish(self.move_cmd)
```

Control The Robot

```

def set_goal(self, distance, tol):
    self.goal_pose_x = self.current_pose_x + (distance * cos(self.current_theta))
    self.goal_pose_y = self.current_pose_y + (distance * sin(self.current_theta))
    self.tol = tol
    rospy.logdebug(" Current Pose : " + str([self.current_pose_x, self.current_pose_y]))
    rospy.logdebug("Move Distance : " + str(distance) + " m with TOL : " + str(tol) + "m")
    rospy.logdebug("    Goal Pose : " + str([self.goal_pose_x, self.goal_pose_y]))
```



```

def check_stop(self):
    delta_x = self.goal_pose_x - self.current_pose_x
    delta_y = self.goal_pose_y - self.current_pose_y
    error = sqrt(delta_x ** 2 + delta_y ** 2)
    rospy.logdebug(error)
    if error <= self.tol:
        return True
    else:
        return False
```

Goal :

$$X_{b_0} = X_{a_0} + D\cos(\theta)$$

$$Y_{b_0} = Y_{a_0} + D\sin(\theta)$$

Stop criteria:

$$\sqrt{(X_o - X_{b_0})^2 + (Y_o - Y_{b_0})^2} \leq \epsilon$$

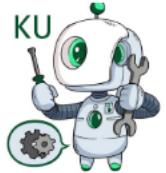
Control The Robot

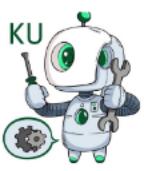
```
def shutdown(self):
    rospy.loginfo("Stop TurtleBot")
    self.cmd_vel_pub.publish(Twist())
    rospy.sleep(1)

if __name__ == "__main__":
    try:
        go_to_goal = GoToGoal()
        # Set go at 1m ahead with 0.05m tolerant
        go_to_goal.set_goal(1, 0.05)
        # Start moving forward at speed 0.1 m/s
        go_to_goal.go_to_goal(0.1)

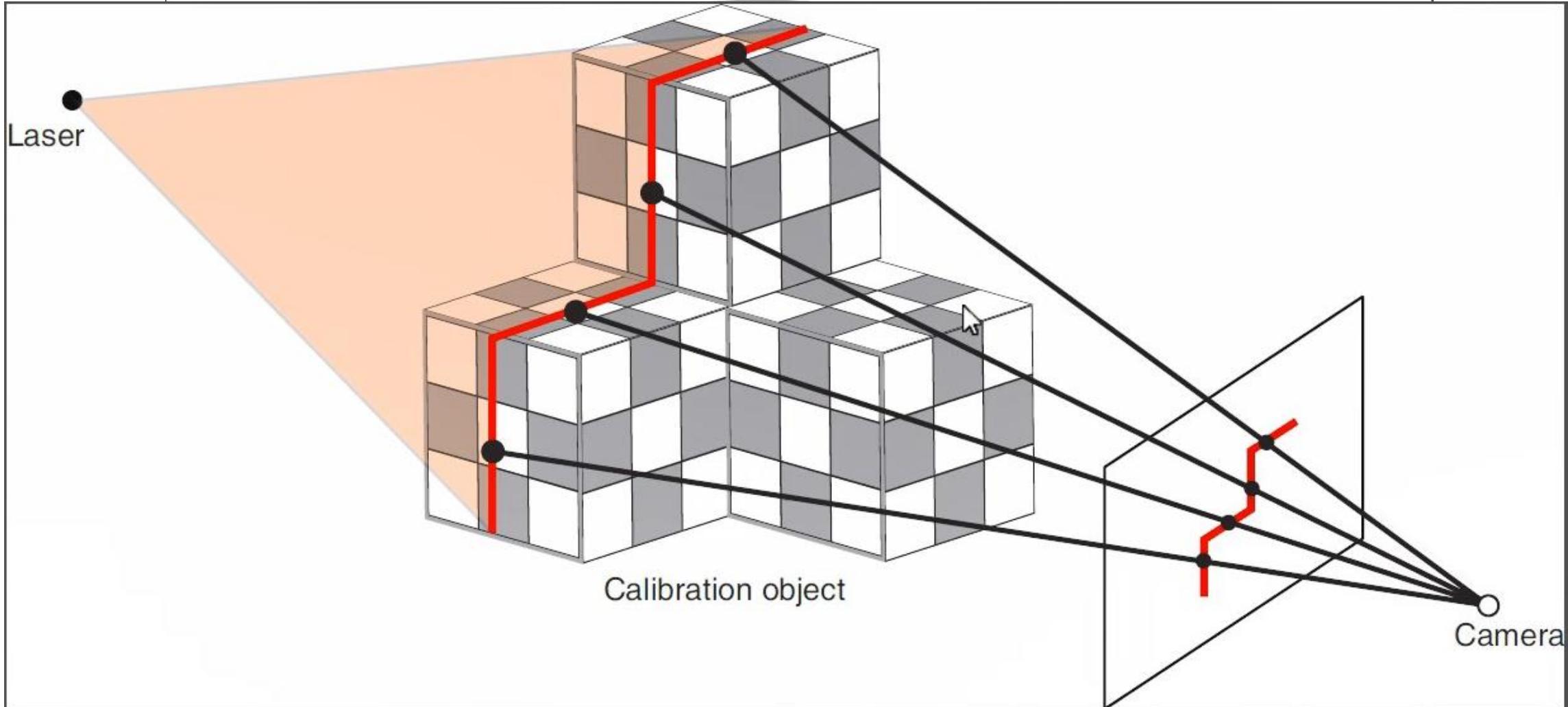
    except rospy.ROSInterruptException:
        rospy.loginfo("GoToGoal Forward node terminated")
```

3D sensor





Type of 3D sensor



Laser triangulation: one of the most common types of 3D sensing, laser triangulation works by projecting a laser onto a part or other surface. Depth is measured by the deformation of the laser, making this technology good for large depth of field applications

Type of 3D sensor

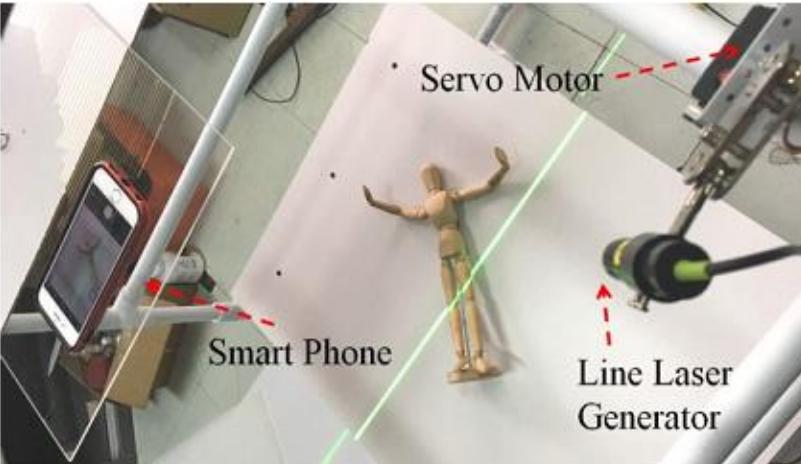


Fig. 1. Low Cost Smart Phone 3D Scanner.

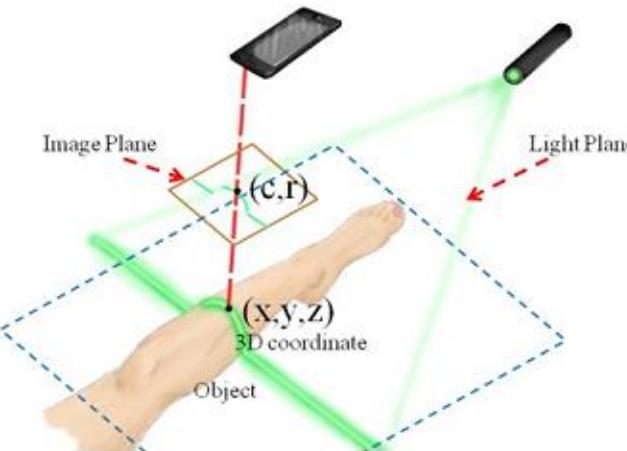


Fig. 2. Triangulation scanning method

3. Splint Reconstruction Method

Lanman [3] is not used because it is complex to a non-technical user. Therefore, the calibration is done simpler way by placing a smart phone camera to a scanning surface at fixed distance, then, they are perpendicular to each others. Each pixel point on the detected line is calculated for corresponding 3D point. This process is repeated for every frame in recorded video and the point cloud image is generated. The noise and unnecessary point cloud can be removed by using Statistical Outlier Removal or Mean distance to k-NN and Voxel grid uniform sampling respectively. Figure 3 shows the output point cloud image from this step.

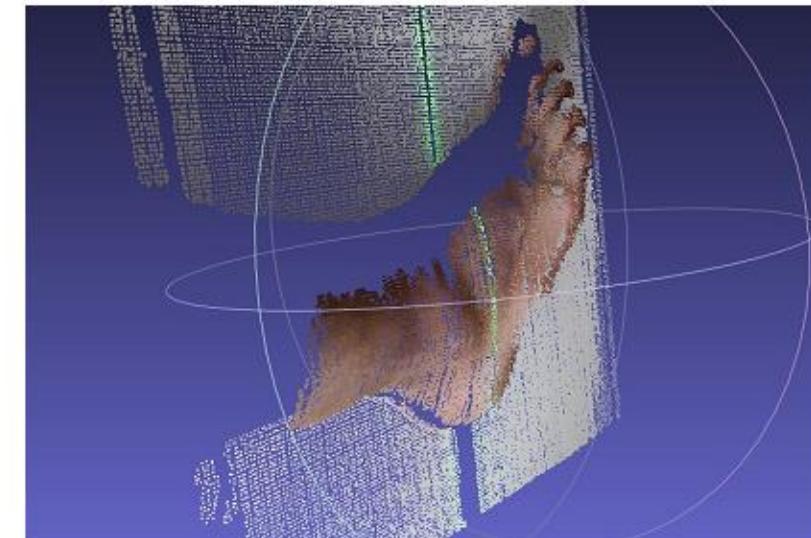
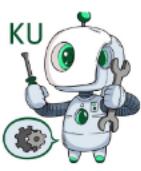
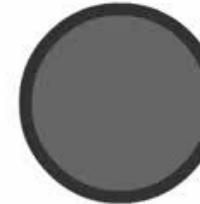


Fig. 3. 3D point cluod image from the scanner.



Type of 3D sensor

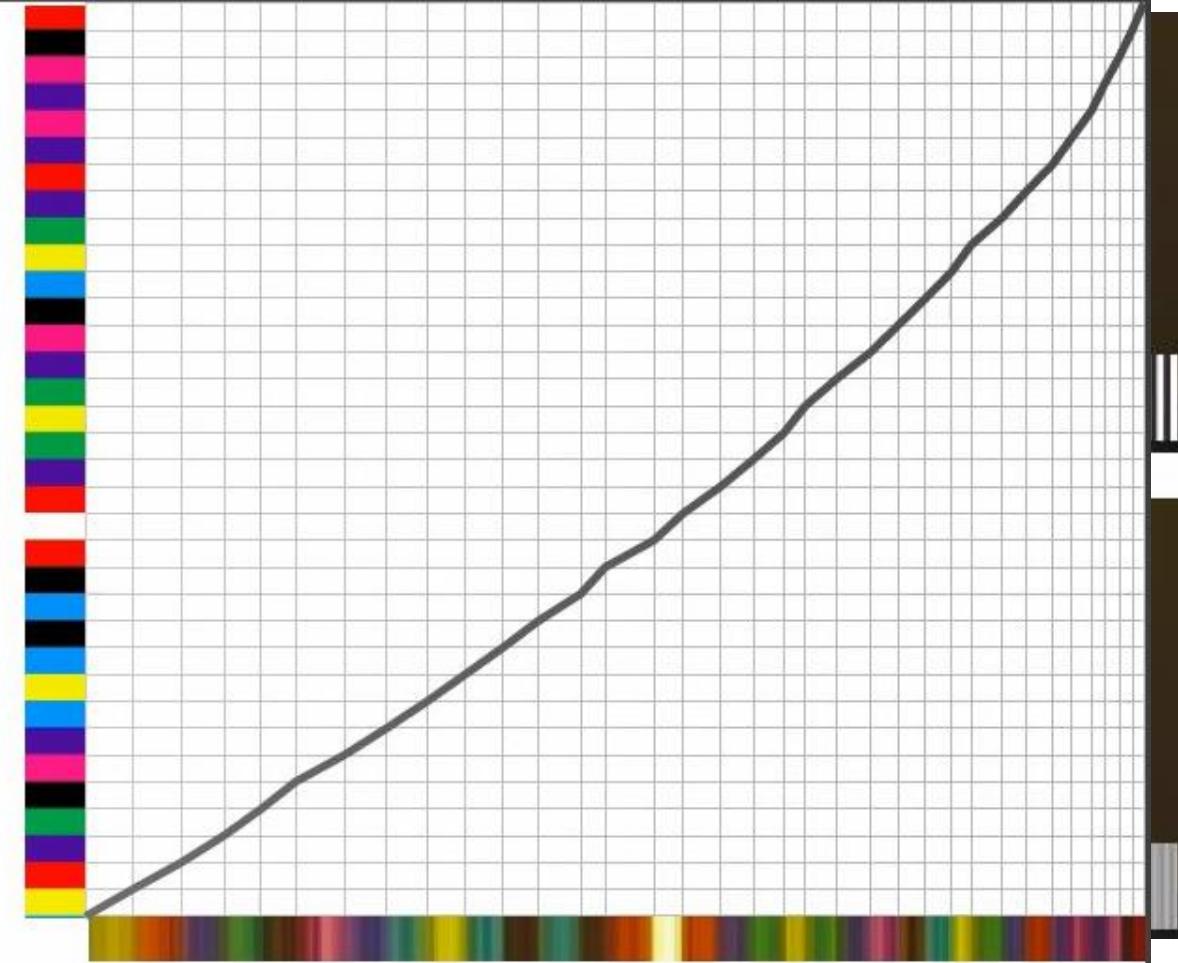
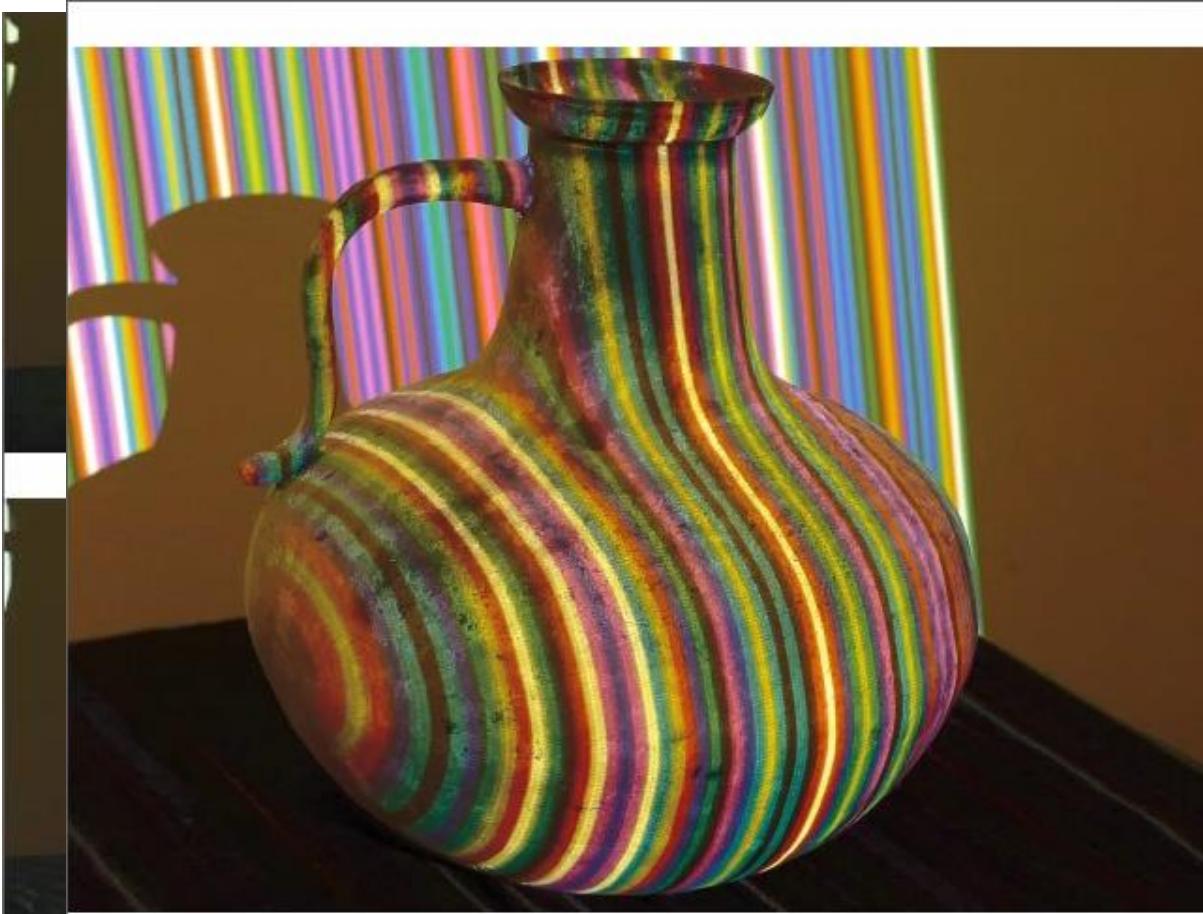
Top Down View



Round Object



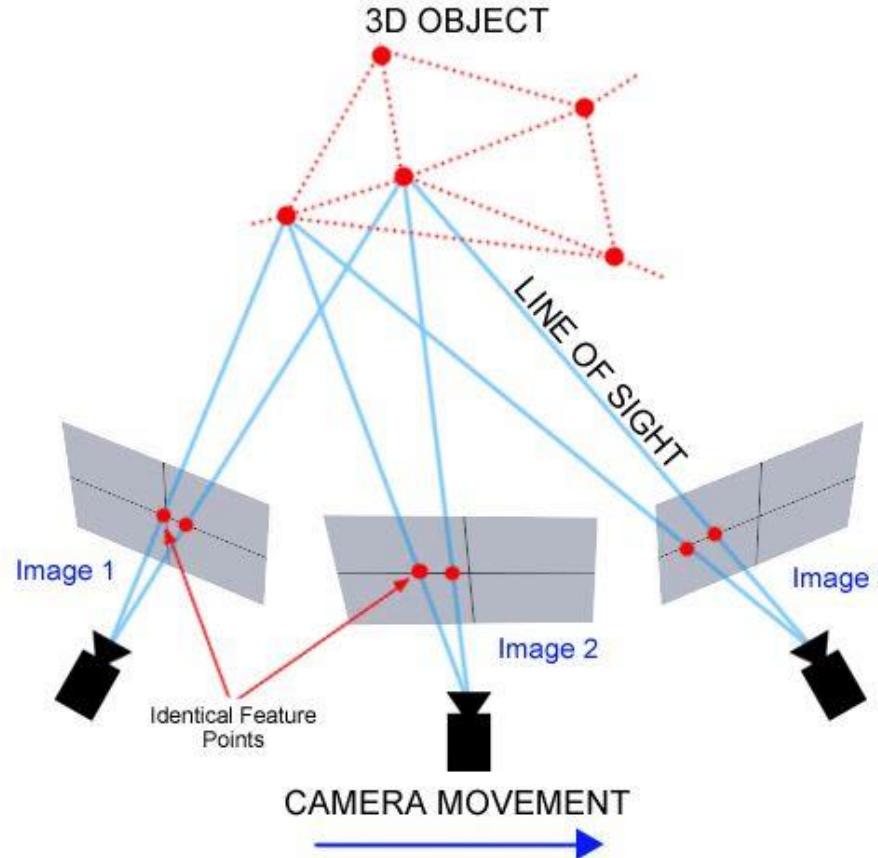
Type of 3D sensor



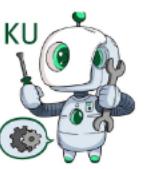
Structured light: with no need for moving parts, structured light 3D sensing systems are some of the most sophisticated solutions out there. They leverage an innovative projection technique to encode 3D information directly on the scene. (**Apple face id**, **Kinect v1**)



Type of 3D sensor

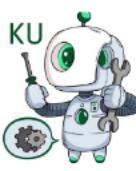


Photogrammetry: a method of computing a 3D reconstruction based on a large number of 2D images of an object. Photogrammetry is a common alternative to LIDAR systems.

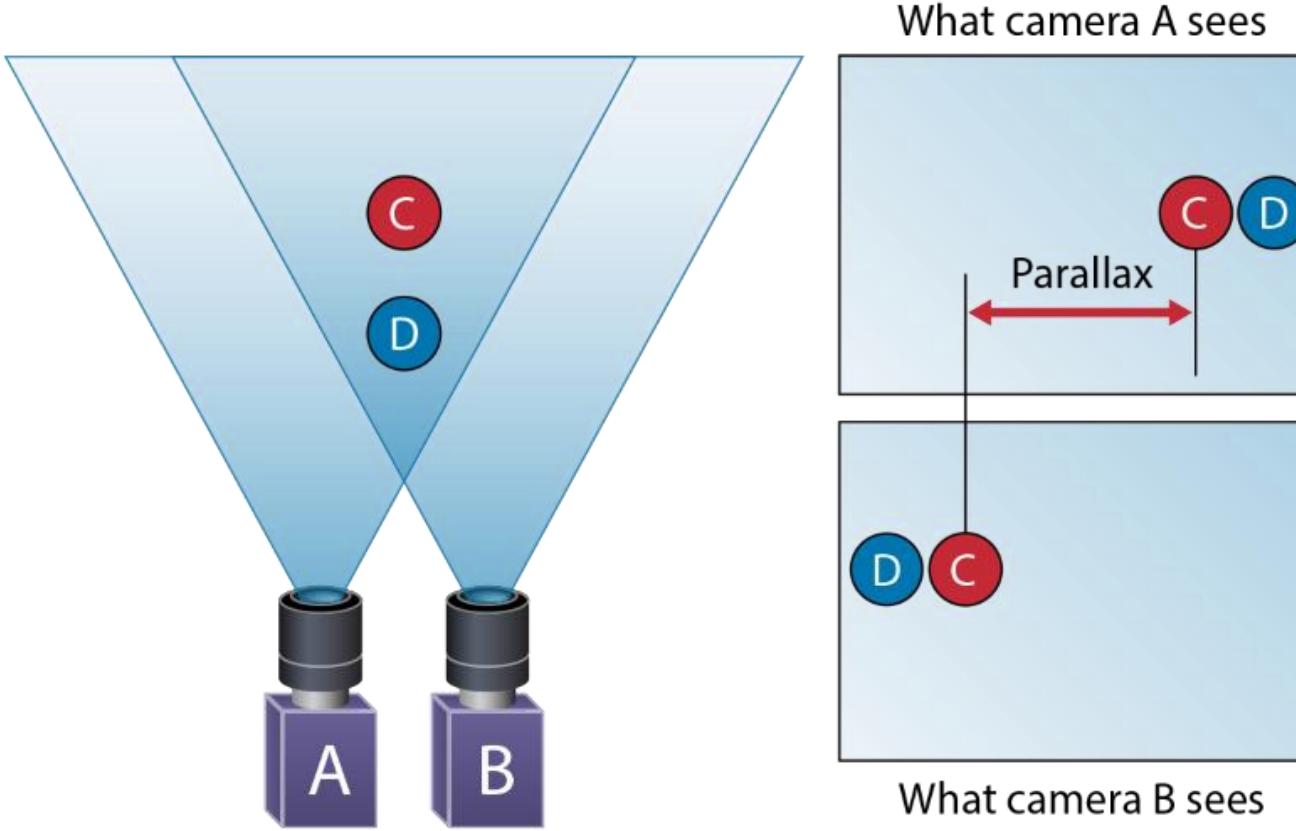


Type of 3D sensor

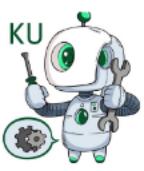




Type of 3D sensor

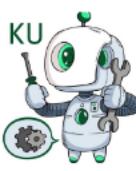


Stereo vision: this type of 3D imaging mimics the way humans perceive 3D images – two sensors are used at different angles to calculate and represent depth. Typically stereo vision is used for applications that don't require measurements, such as counting people.

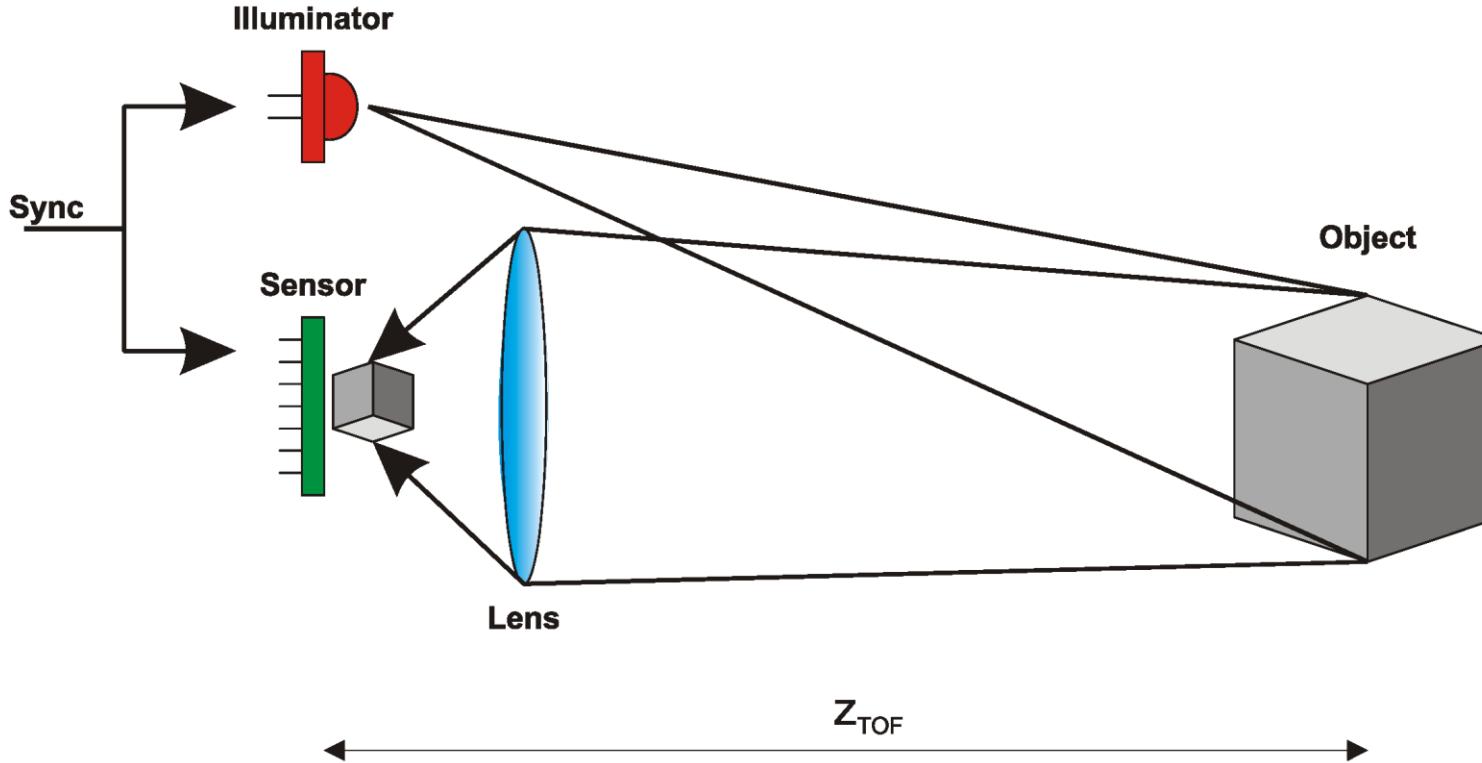


Type of 3D sensor





Type of 3D sensor

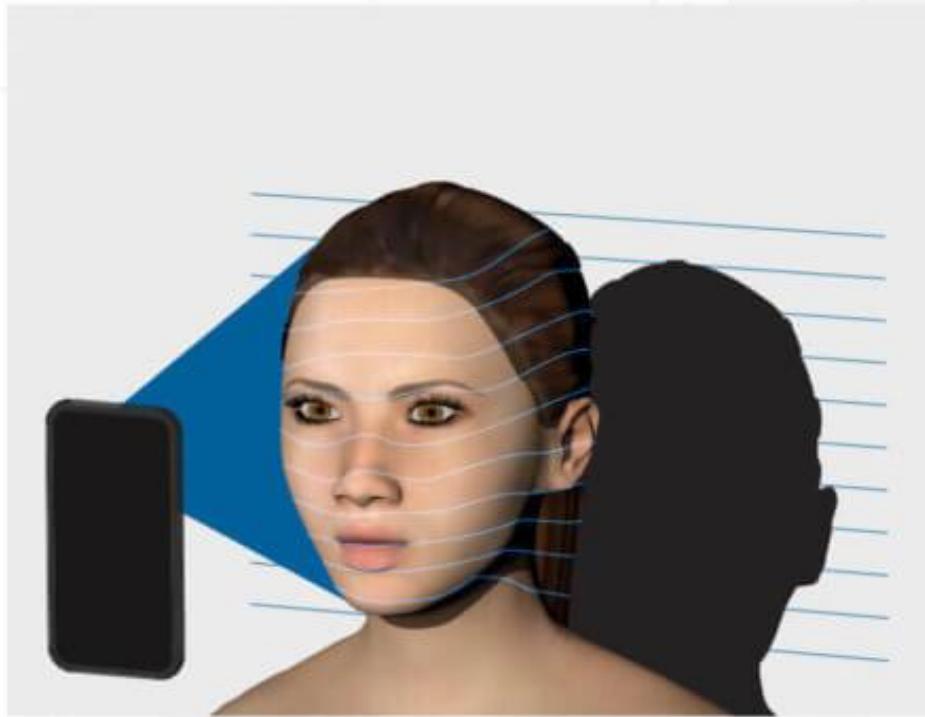


Time-of-flight: these systems detect the time of light travel between the light emitter, an inspected object, and back to the detector. Time-of-flight systems are either area sensing or LIDAR systems. (Apple is considering using ToF sensors for 2020's phones)



Type of 3D sensor

HOW STRUCTURED LIGHT SYSTEMS WORK

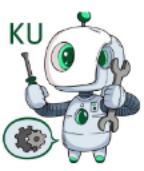


Structured light emitter projects patterns in infrared light.
The patterns are projected in pulses.
By understanding how the pattern distorts on each object, depth can be calculated.

HOW TIME OF FLIGHT SYSTEMS WORK



Time-of-flight emitter floods the scene with infrared light.
By measuring the time it takes for the light to return from each pixel, the depth map of the scene is computed.



Kinect

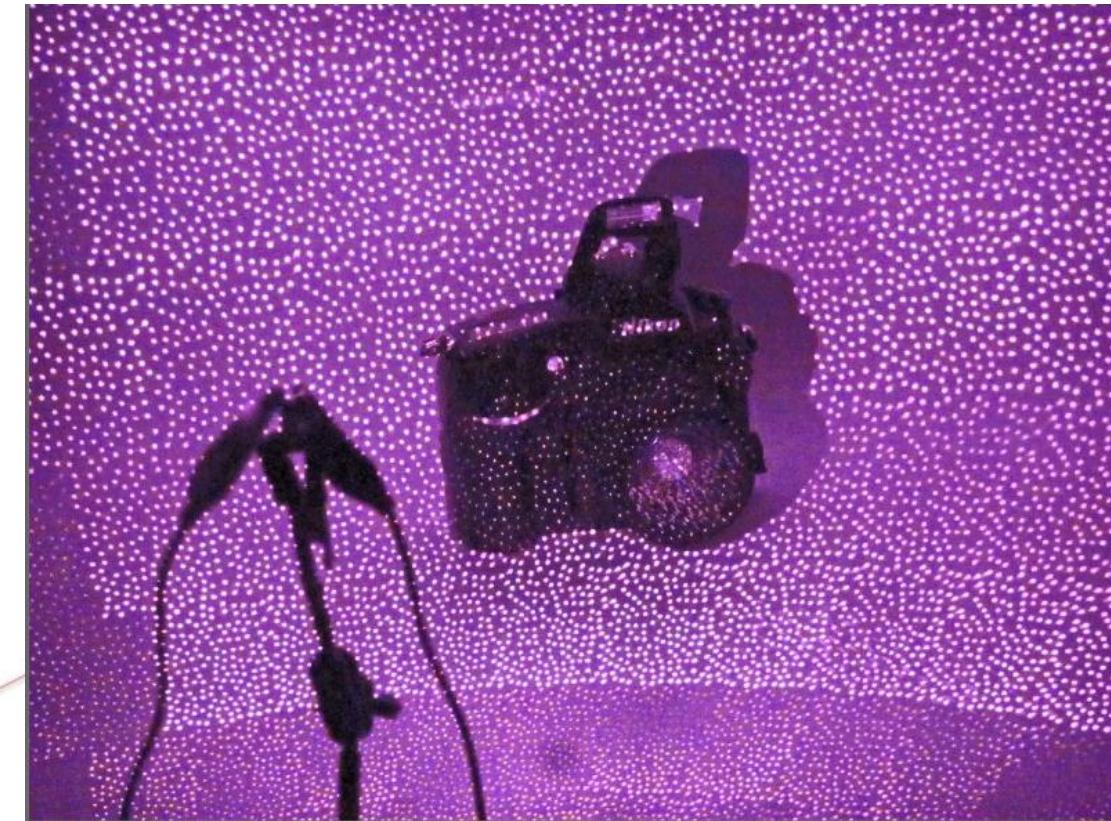
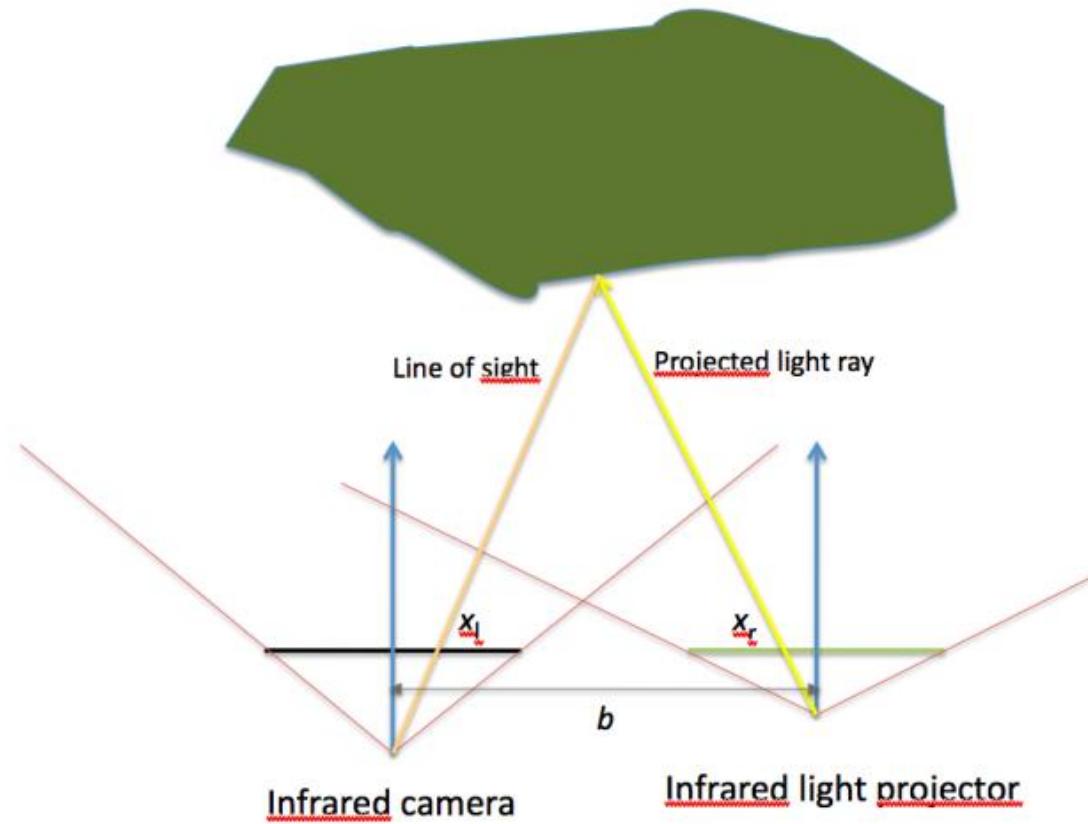
Inside Kinect

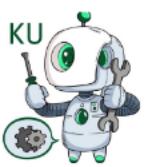




Kinect

Geometry of Kinect (Rectified)

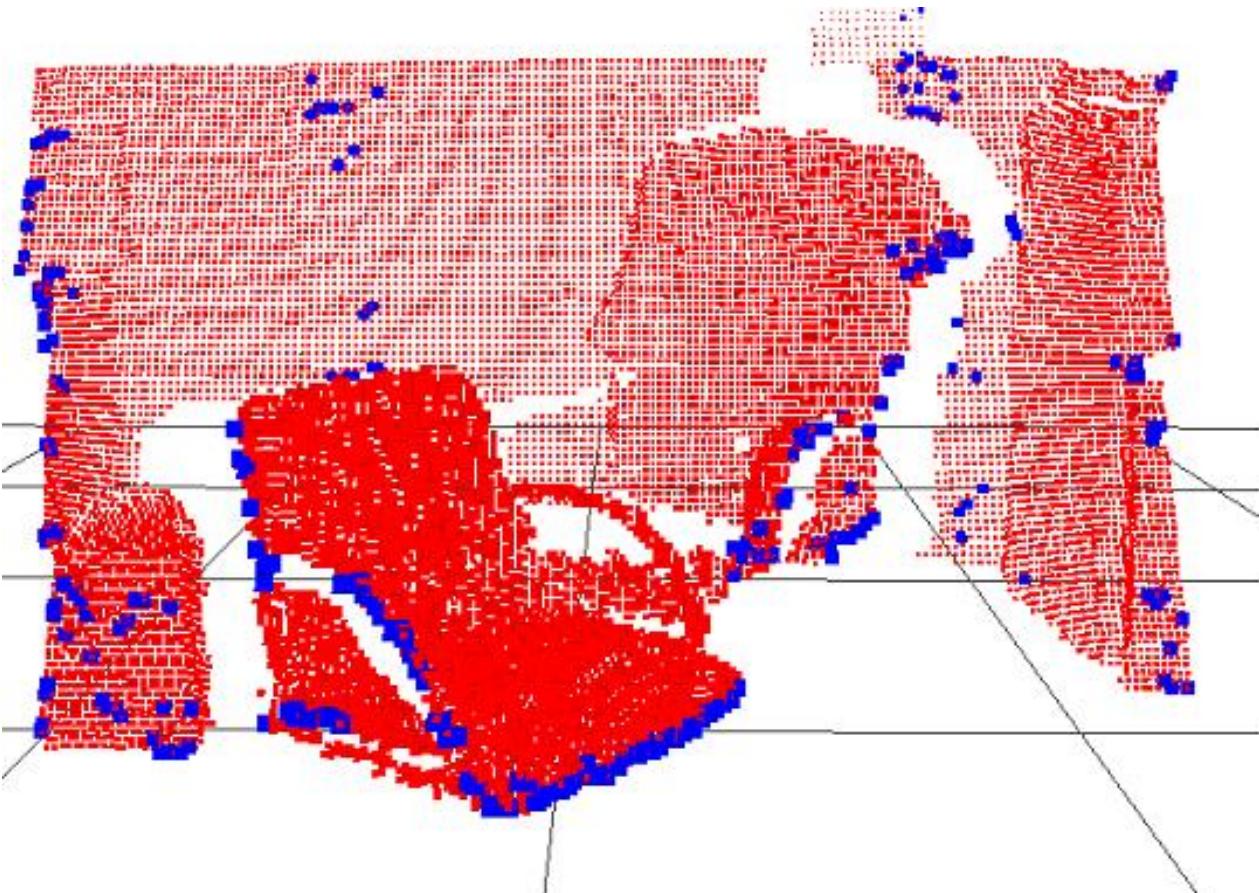




Kinect



|| Point Cloud = Output of the 3D sensor ||

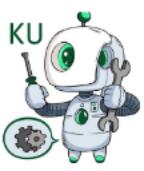


Install and testing Kinect (3D sensor)

- 1) Connect Kinect to Computer (USB)
- 2) Connect Kinect power wire to Kobuki base
- 3) Run:

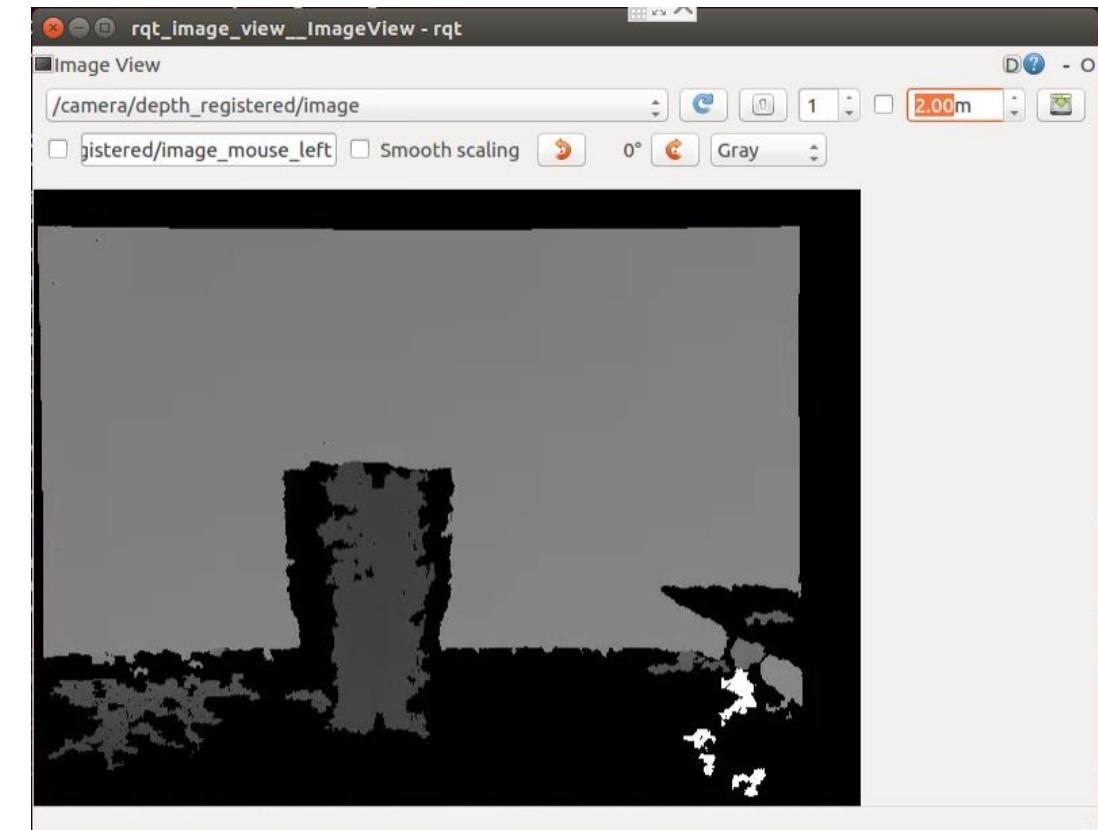
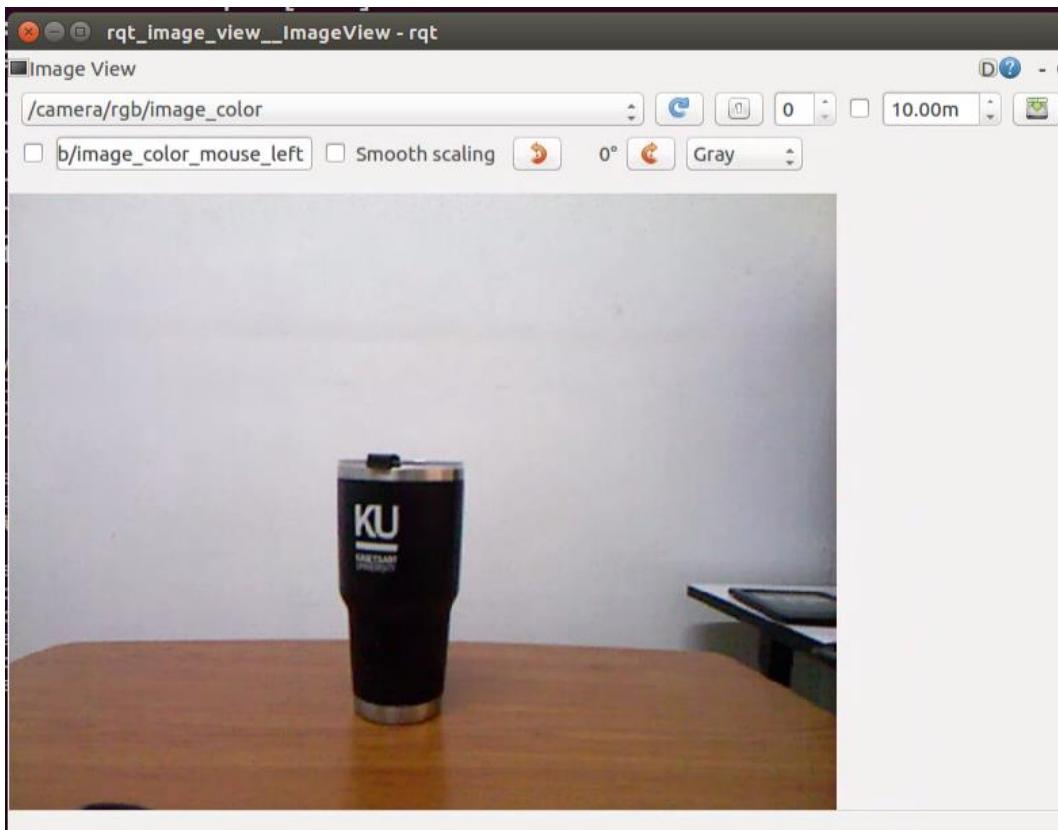
```
$ rosrun turtlebot_bringup 3dsensor.launch
```

```
[ INFO] [1588910795.024392227]: Searching for device with index = 1
[ INFO] [1588910795.194397100]: Starting a 3s RGB and Depth stream flush.
[ INFO] [1588910795.194562060]: Opened 'Xbox NUI Camera' on bus 0:0 with serial number 'B00362706888042B'
```



Install and testing Kinect (3D sensor)

```
$rosrun rqt_image_view rqt_image_view
```



Install and testing Kinect (3D sensor)

View Depth image in 3D with robot visualization (Rviz)

```
$ roslaunch turtlebot_bringup minimal.launch
```

```
$ roslaunch turtlebot_bringup 3dsensor.launch
```

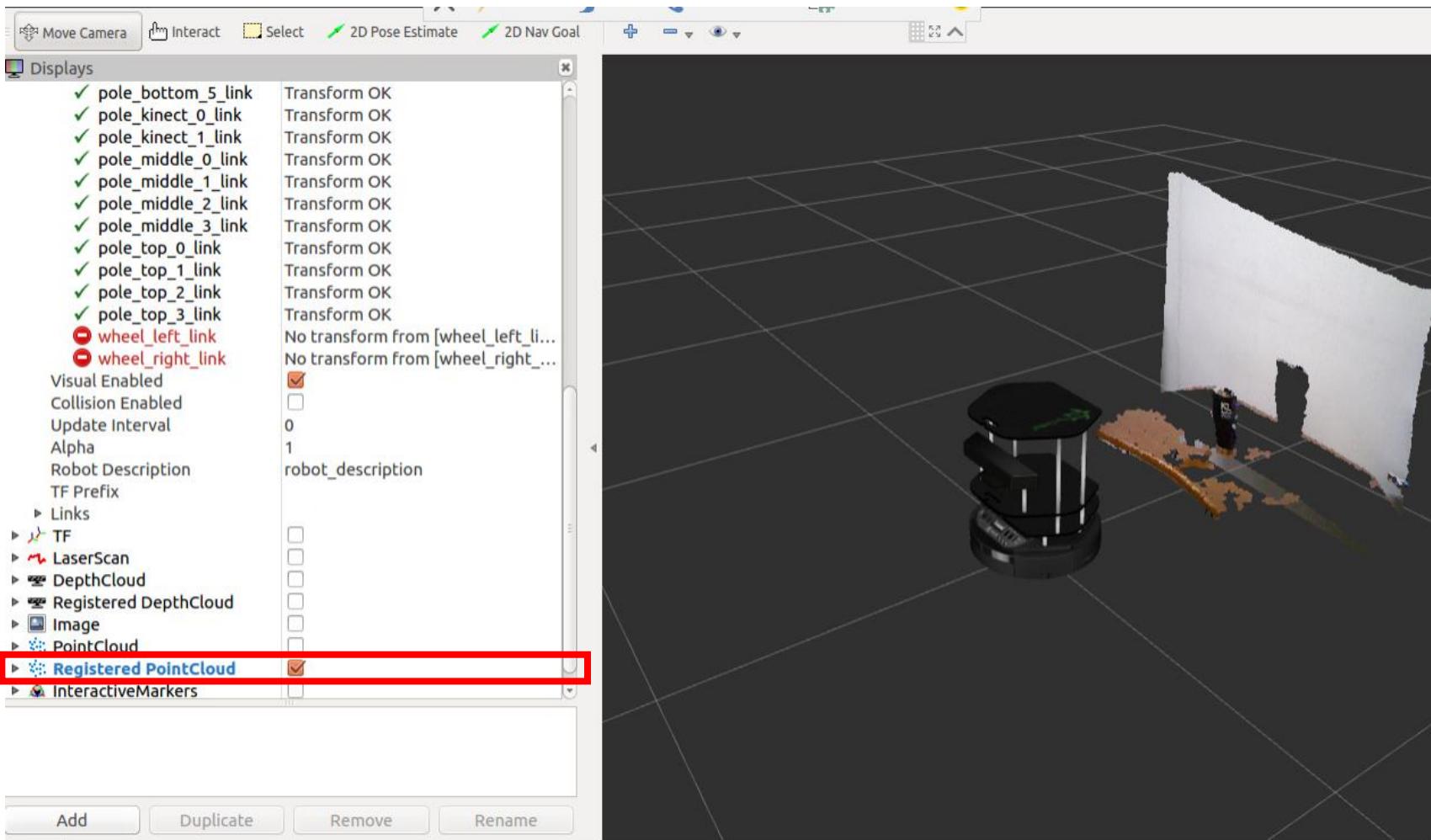
Run RVIZ

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```



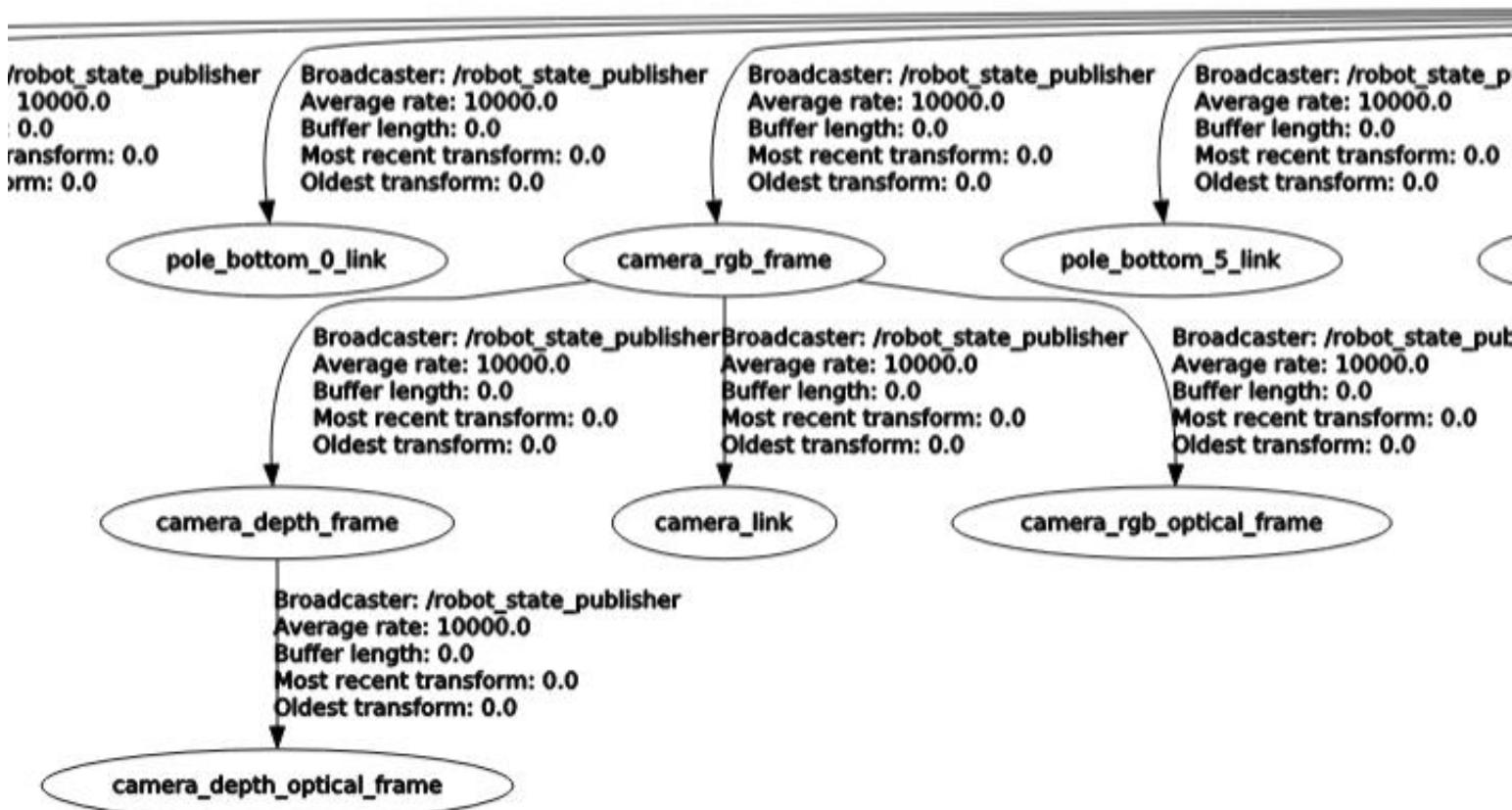
Install and testing Kinect (3D sensor)

Check the Registered PointCloud in menu



Install and testing Kinect (3D sensor)

```
$ rosrun rqt_tf_tree rqt_tf_tree
```



Human Follower Experiment

Start the follower demo

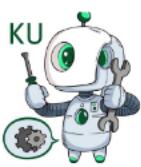
```
$ roslaunch turtlebot_bringup minimal.launch
```

```
$ roslaunch turtlebot_teleop logitech.launch
```

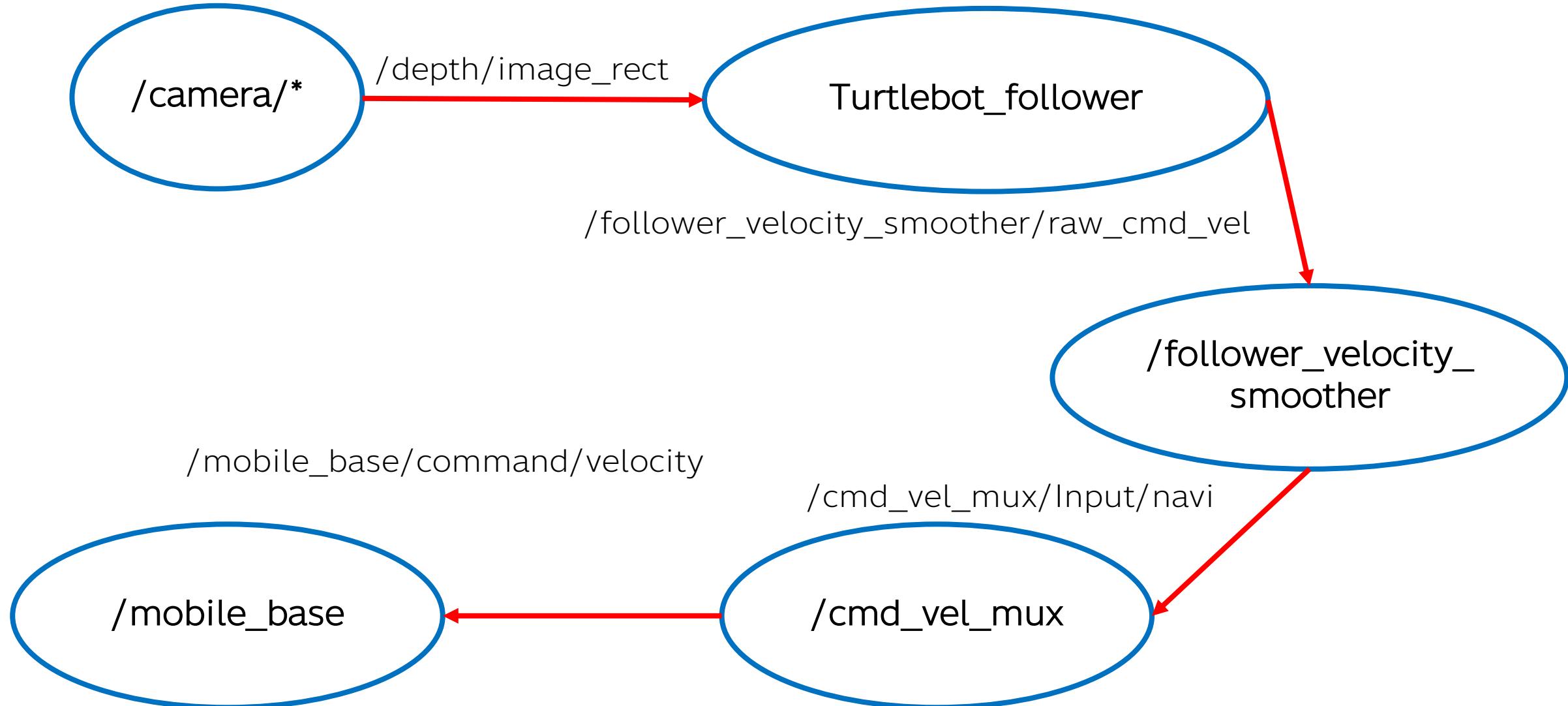
```
$ roslaunch turtlebot_follower follower.launch
```

Changing Follower Parameters

```
$ rosrun rqt_reconfigure rqt_reconfigure
```



Human Follower Experiment



Human Follower Experiment

How it works:

```

uint32_t image_width = depth_msg->width;
float x_radians_per_pixel = 60.0/57.0/image_width;
float sin_pixel_x[image_width];
for (int x = 0; x < image_width; ++x) {
    sin_pixel_x[x] = sin((x - image_width/ 2.0) * x_radians_per_pixel);
}
uint32_t image_height = depth_msg->height;
float y_radians_per_pixel = 45.0/57.0/image_width;
float sin_pixel_y[image_height];

for (int y = 0; y < image_height; ++y) { // Sign opposite x for y up values
    sin_pixel_y[y] = sin((image_height/ 2.0 - y) * y_radians_per_pixel);
}

```

Note:

$$\begin{aligned} \text{radians} &= \text{degrees} * \pi / 180^\circ \\ &= \text{degree} / 57.2957 \end{aligned}$$

Kinect

Viewing angle

Specifications

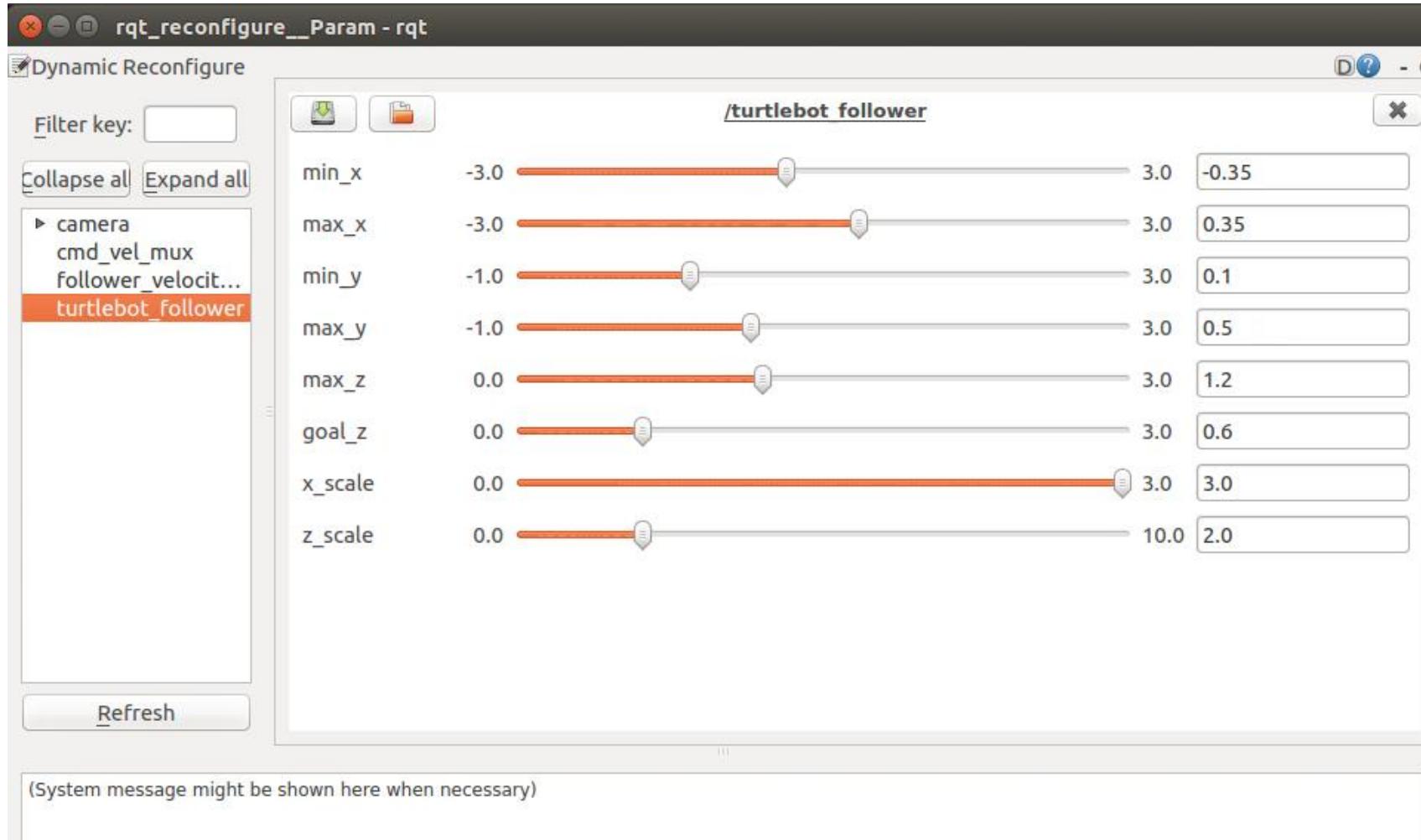
Field of View (FoV): 43° vertical x 57° horizontal

Human Follower Experiment

```
for (int v = 0; v < (int)depth_msg->height; ++v, depth_row += row_step)
{
    for (int u = 0; u < (int)depth_msg->width; ++u)
    {
        float depth = depth_image_proc::DepthTraits<float>::toMeters(depth_row[u]);
        if (!depth_image_proc::DepthTraits<float>::valid(depth) || depth > max_z_)
            continue;
        float y_val = sin_pixel_y[v] * depth;
        float x_val = sin_pixel_x[u] * depth;
        if (y_val > min_y_ && y_val < max_y_ &&
            x_val > min_x_ && x_val < max_x_)
        {
            x += x_val;
            y += y_val;
            z = std::min(z, depth); //approximate depth as forward.
            n++;
        }
    }
}
```

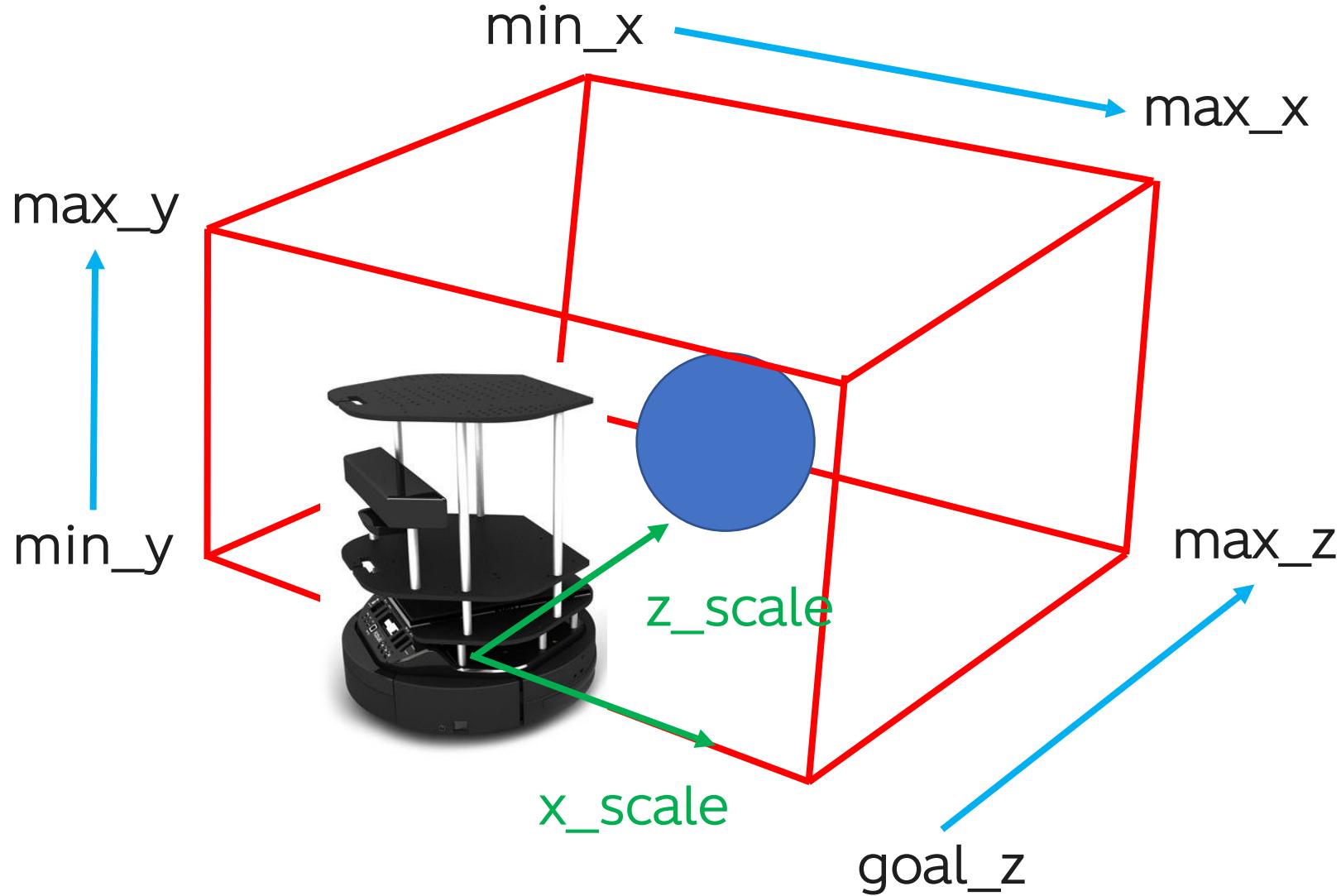
Human Follower Experiment

Changing Follower Parameters

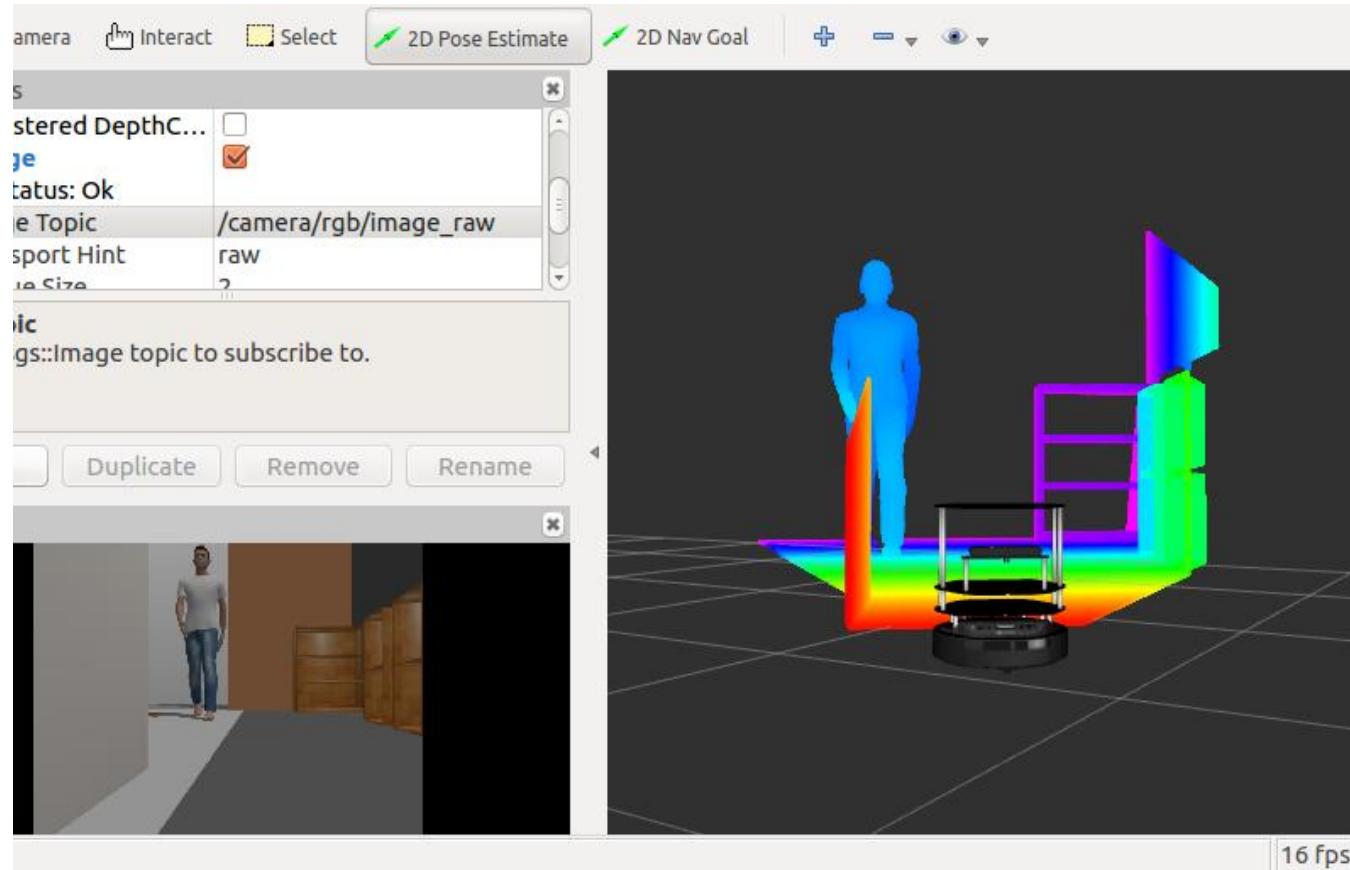




Human Follower Experiment



World Simulation



Setup a ROS catkin workspace

Create a ROS package for the robot

Create a ROS node to launch the robot with Rviz

Create a node to publish joint state instructions to the robot

Launch these ROS nodes with rosrun

World Simulation

Place gazebo_models.tar.gz to home directory

```
$ tar -zxf gazebo_models.tar.gz
$ cd gazebo_models
$ cp -fR * $HOME/.gazebo/models/
```

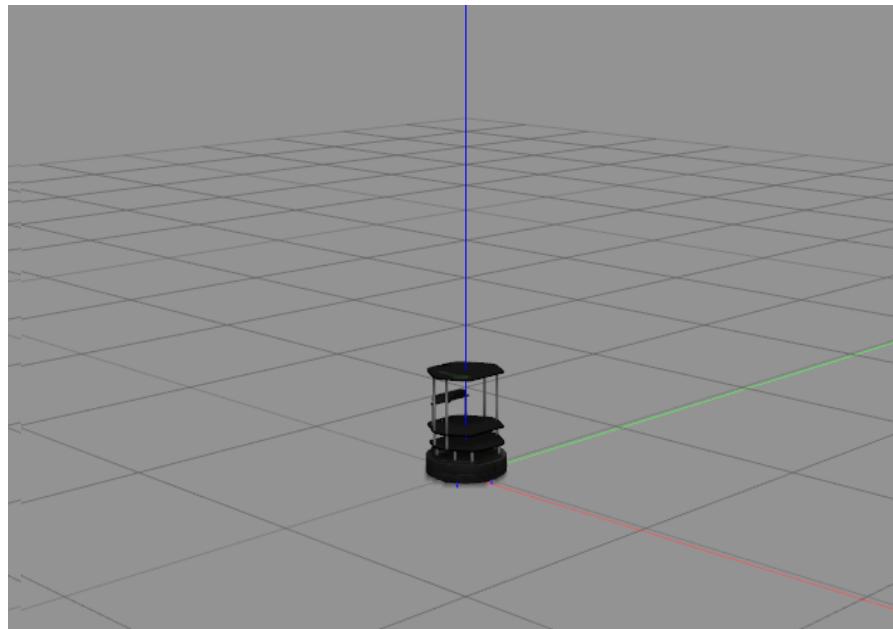
```
ku@ku:~/myROS$ cp -fR fitness $HOME/.gazebo/models
ku@ku:~/myROS$ ls $HOME/.gazebo/models
ambulance cube_20k fitness jersey_barrier living person_standing sun table
bookshelf dumpster ground_plane kitchen_dining oak_tree person_walking suv
ku@ku:~/myROS$
```

World Simulation

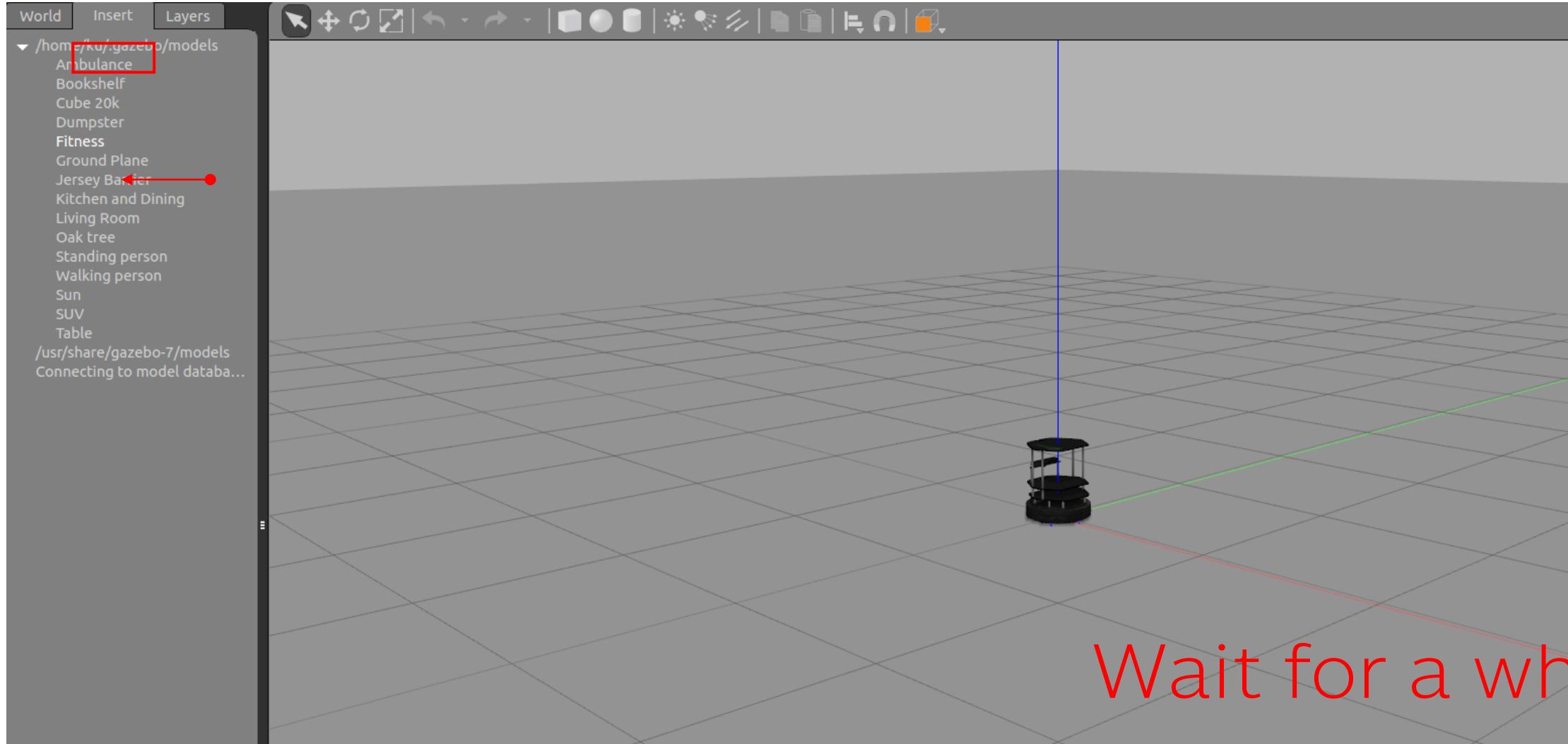
Launch gazebo turtlebot simulator with empty world

```
$ roscore
```

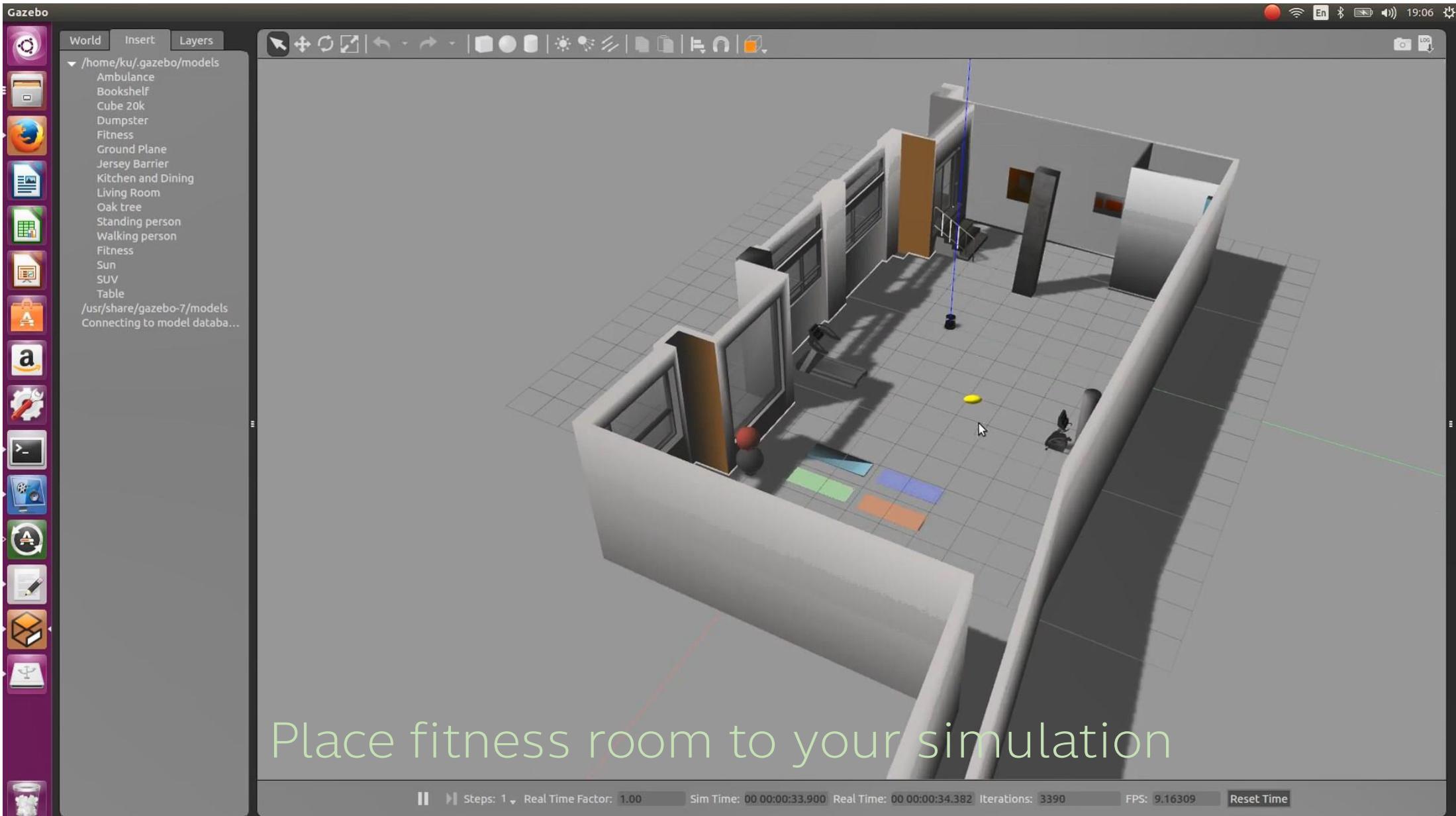
```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
world_file:=/opt/ros/kinetic/share/turtlebot_gazebo/worlds/empty.world
```



World Simulation



World Simulation



World Simulation

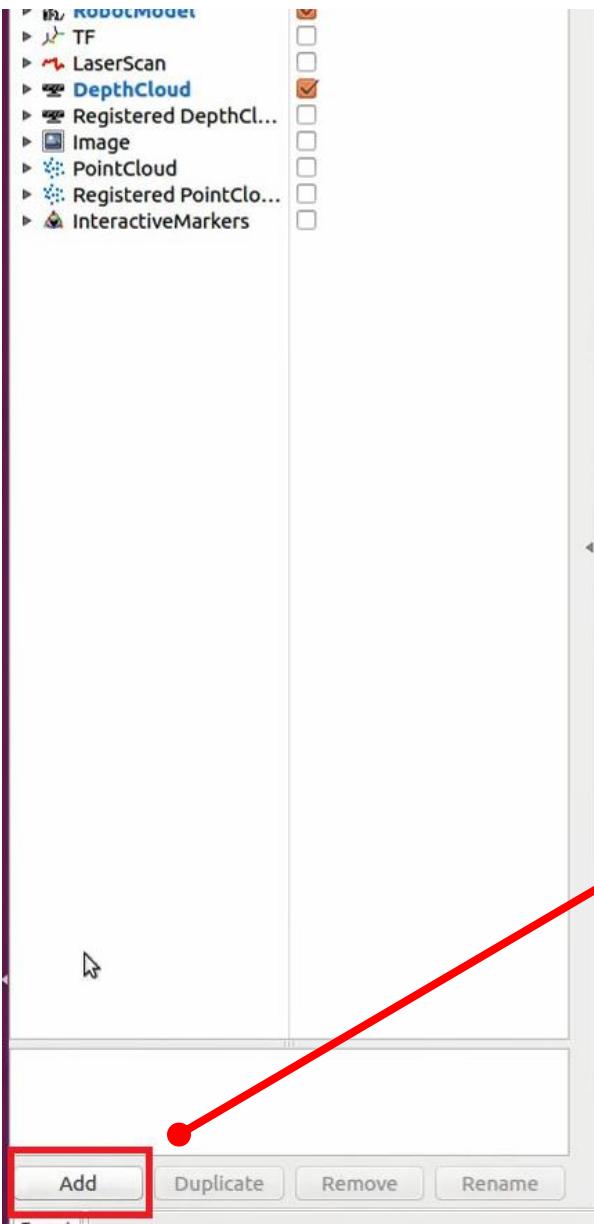
Launch turtlebot rviz

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

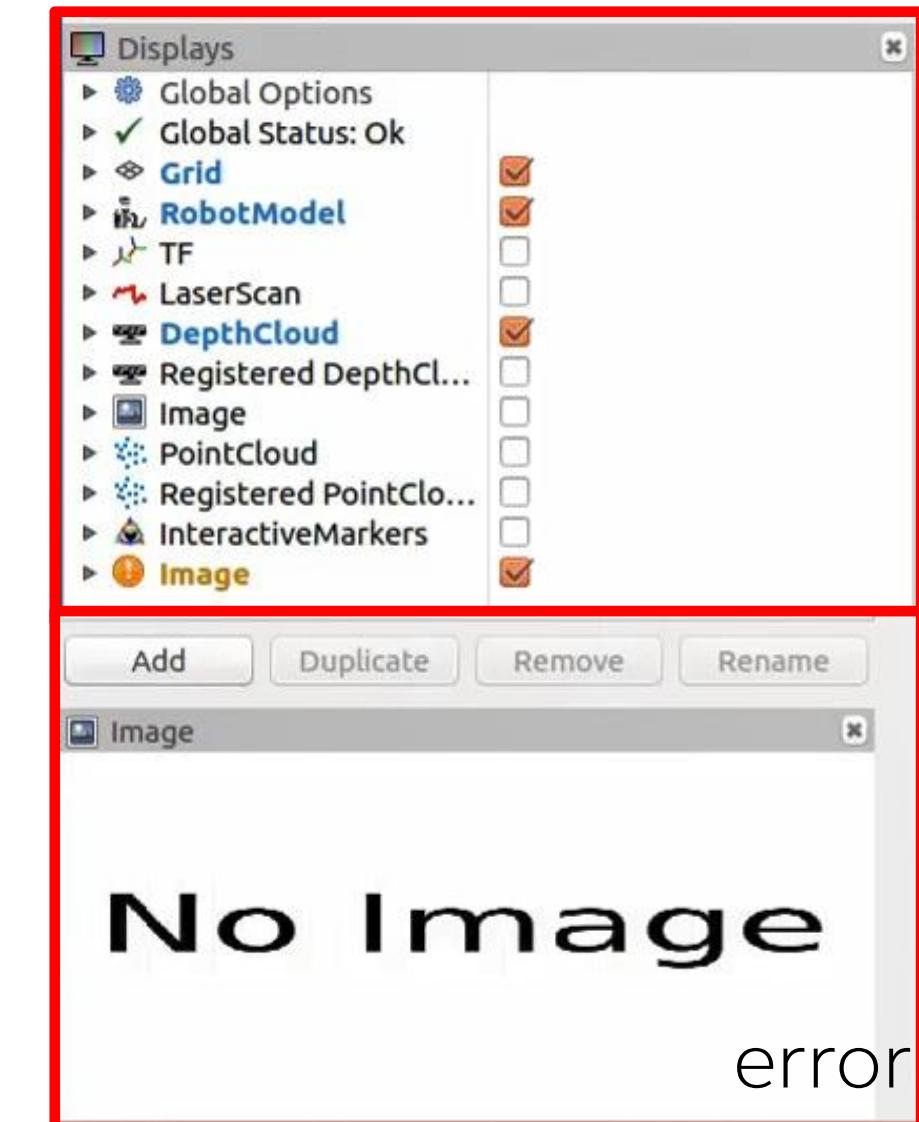
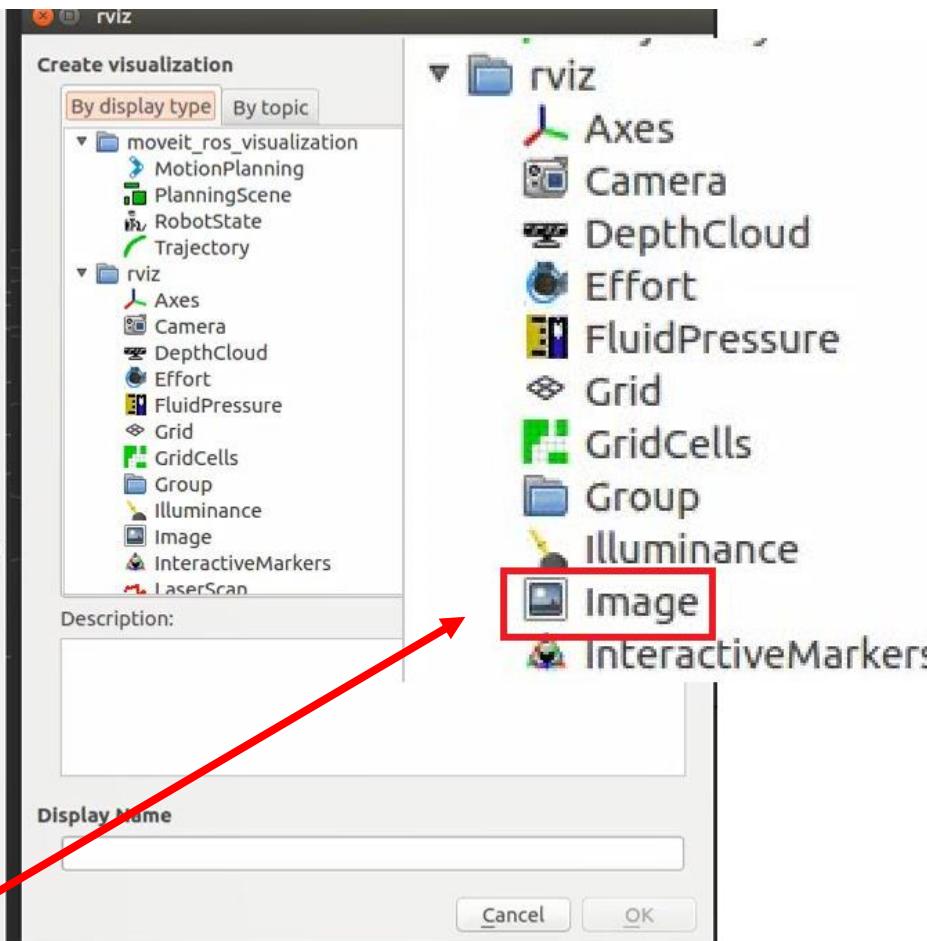
Enable DepthCloud



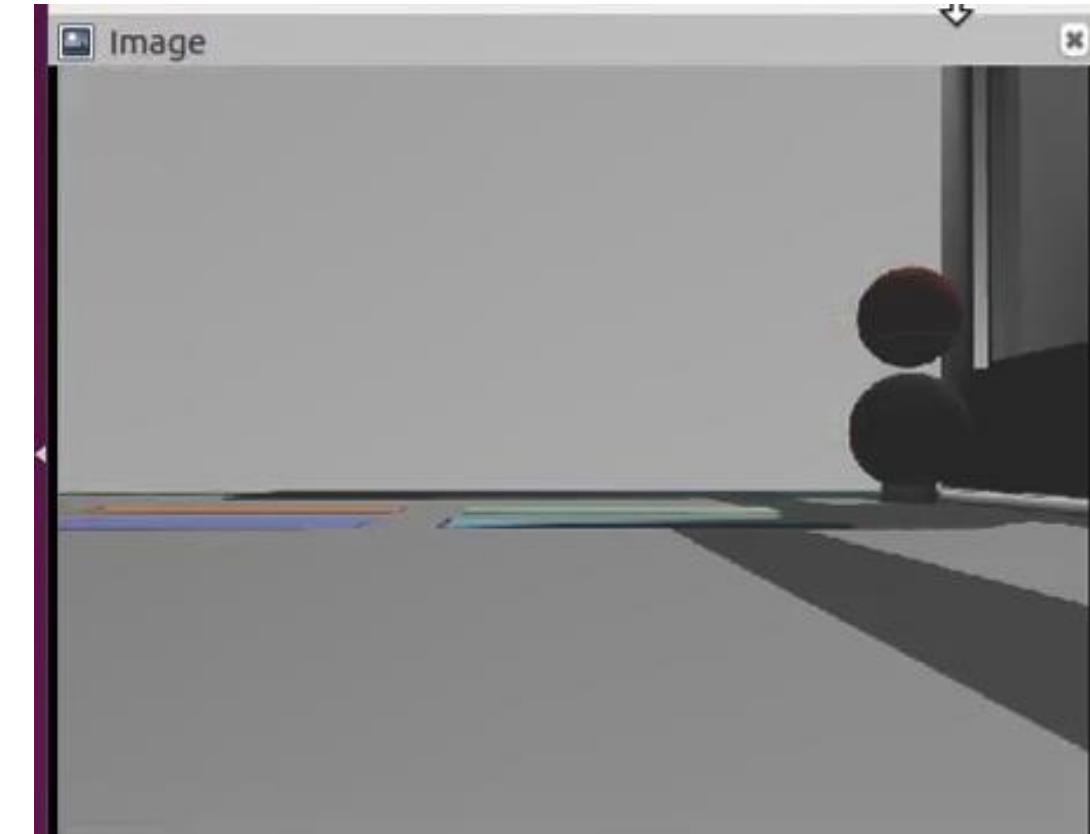
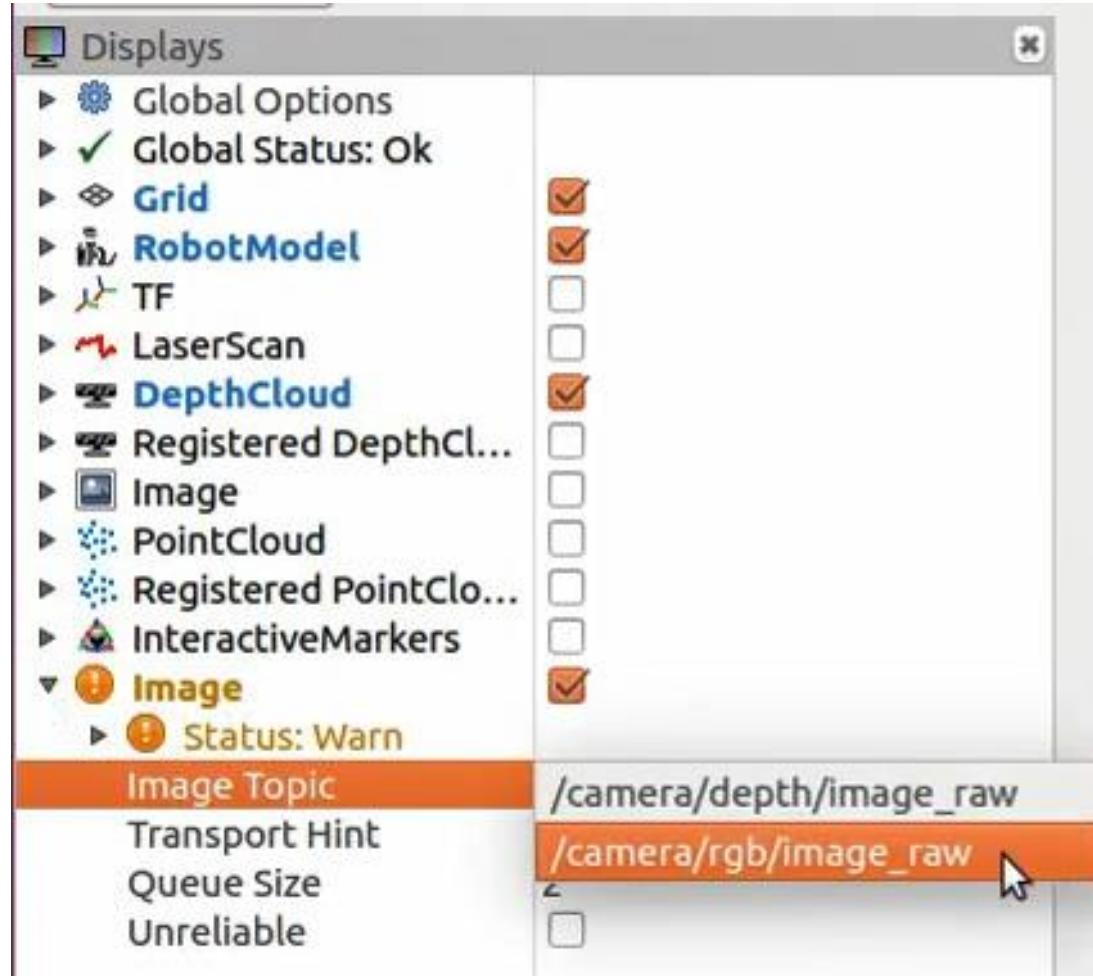
World Simulation



Add image



World Simulation



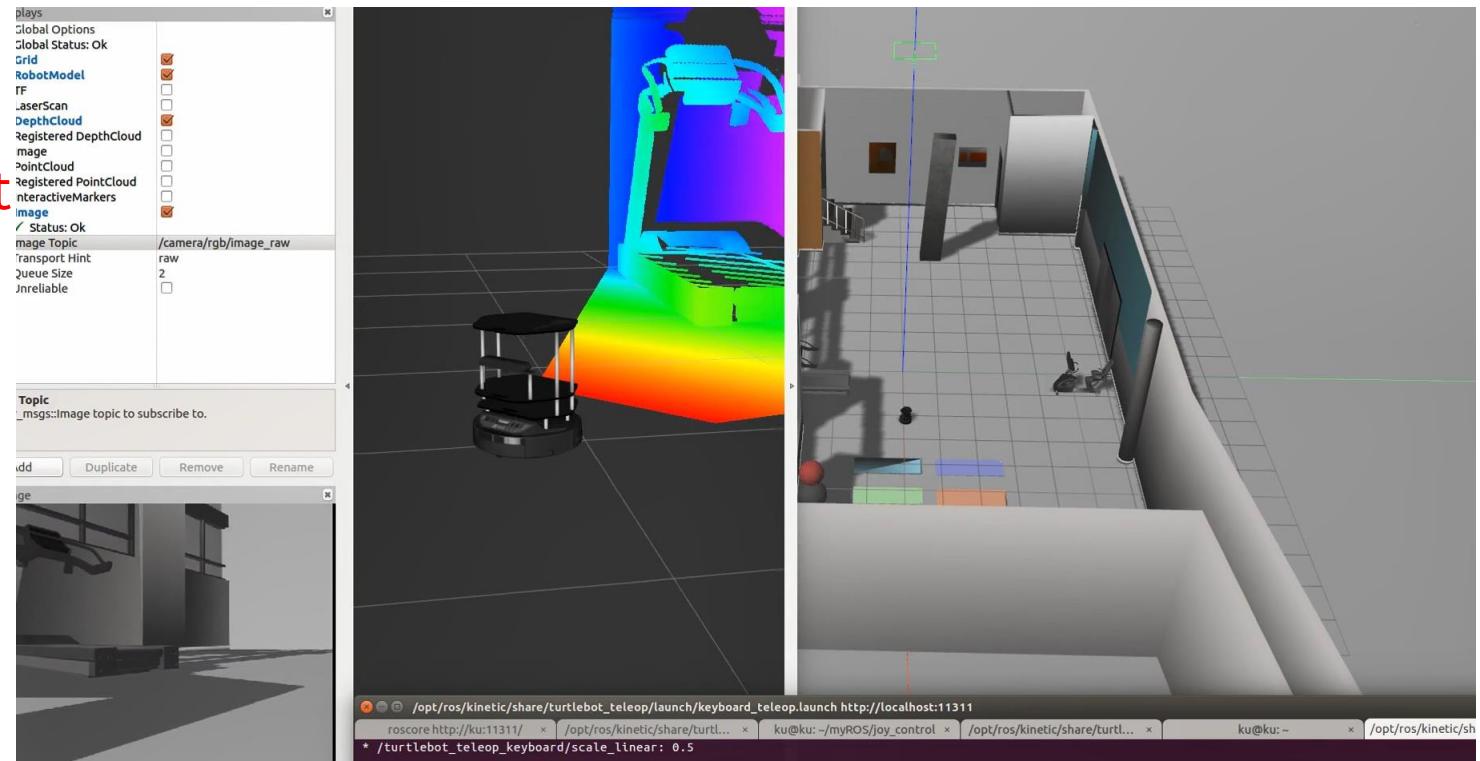
World Simulation

Test turtlebot in environment

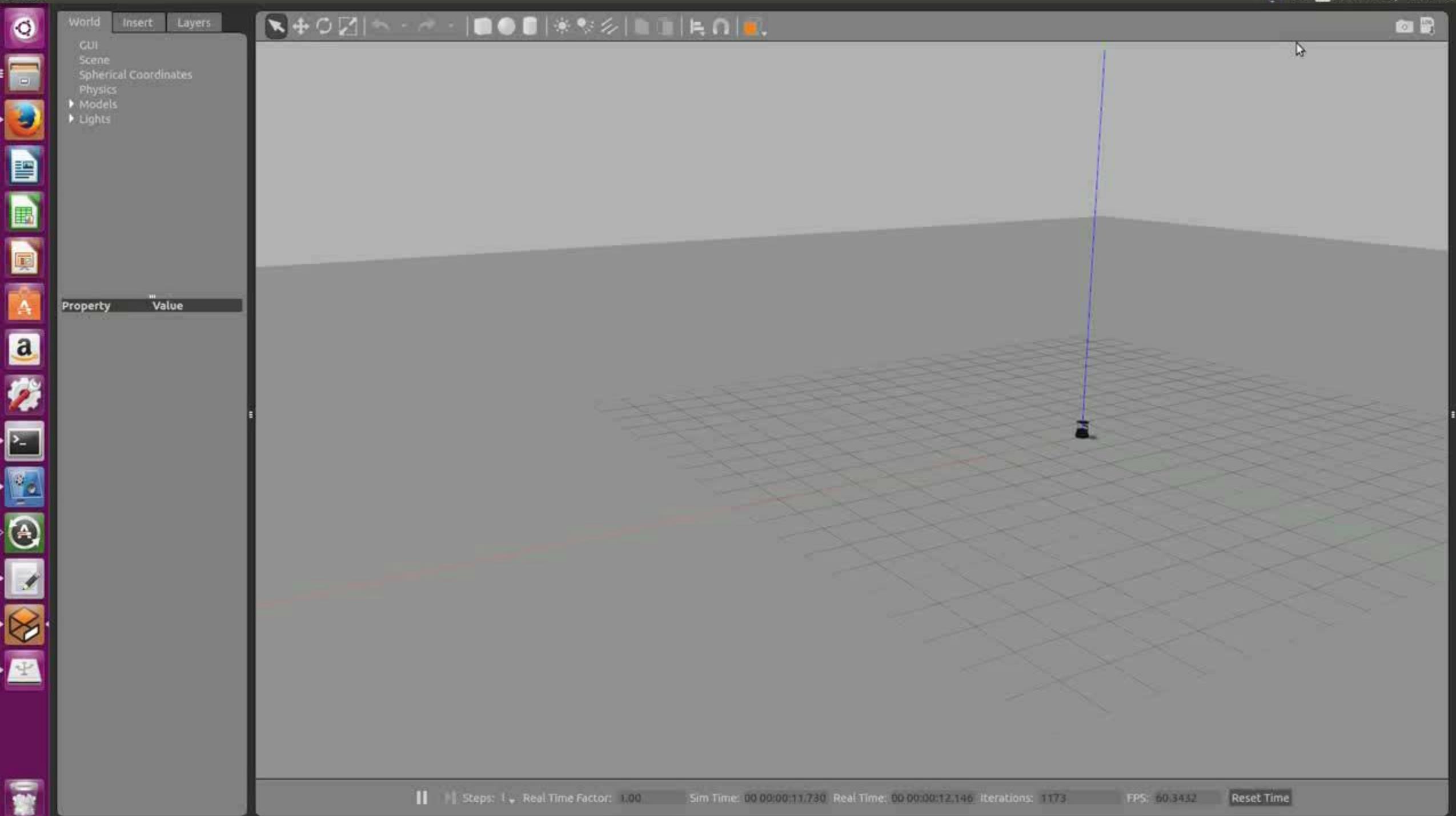
```
$ rosrun turtlebot_teleop keyboard_teleop.launch
```

```
$ rosrun turtlebot_teleop logitech.launch
```

You can get image directly to test
your image processing algorithm



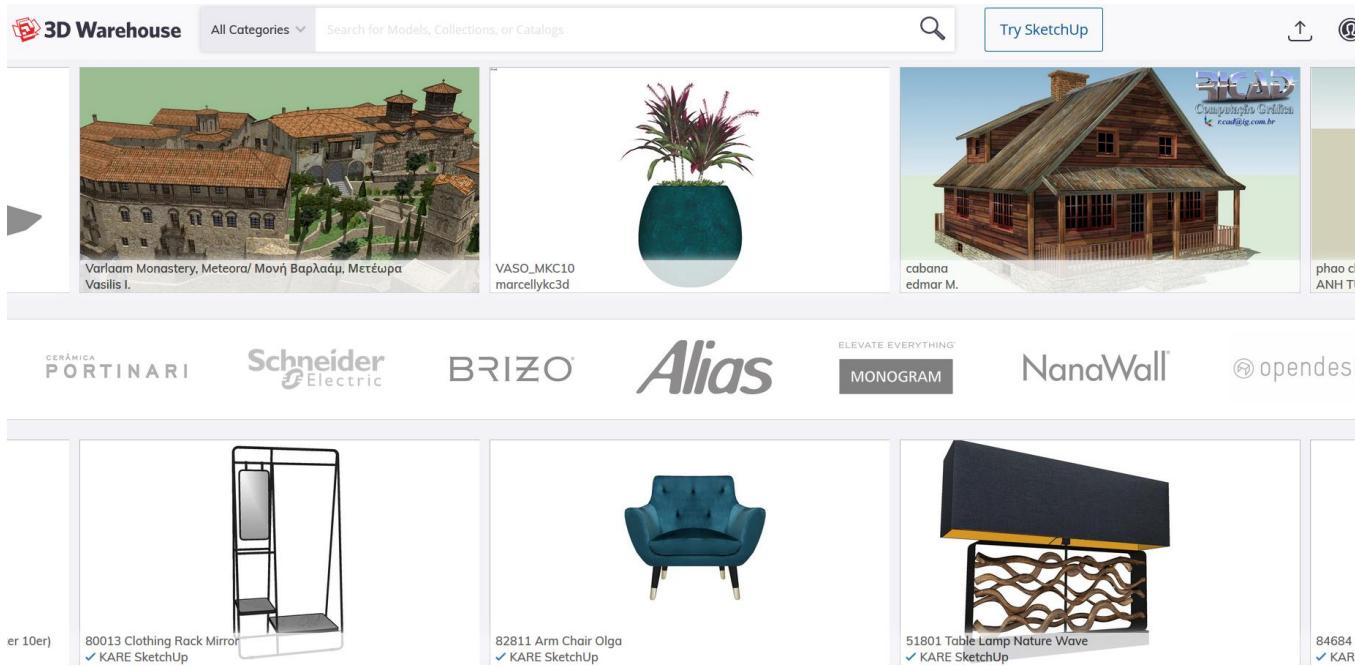
Save .world to use in the future





Where can we get environment ?

skp export to Collada file
convert solidwork to Collada file via MeshLab



or create by yourself

http://gazebosim.org/tutorials?tut=build_model

THANK YOU