##### +

Matthew Parker - Harvard University - Spring 2022

# Assignment Details

Assignment 3

Use ROS Environment and Gazebo to simulate a RasPi Robot

[**Assignment Details**](#_qaghn8okgeki) **1**

[**Initial Package Build**](#_xwit1t6o6chl) **1**

[**Start Gazebo Simulator**](#_rk01rlthinp7) **3**

[**Configure Robot Files**](#_lot81zv7zpvo) **4**

[**Load Robot Into Simulator**](#_41wzgrzcleas) **6**

[**Prepare to Run**](#_pkrj0c4mtnzw) **7**

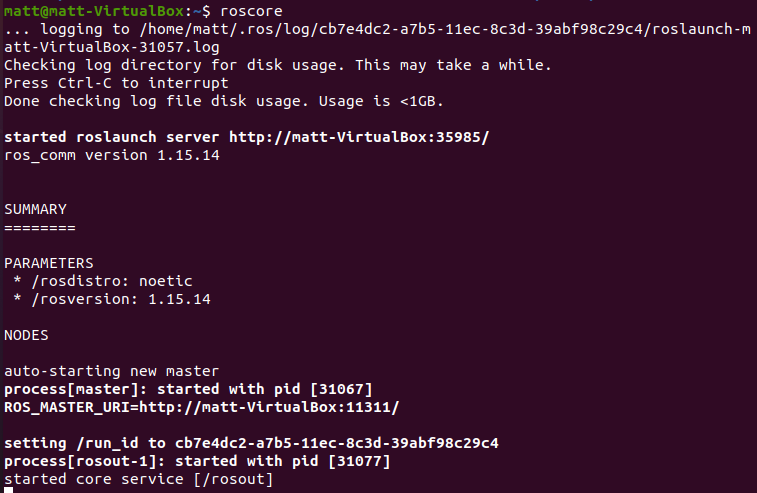
[**Appendix 1: Source Code**](#_er1wnwzapoa) **11**

# Initial Package Build

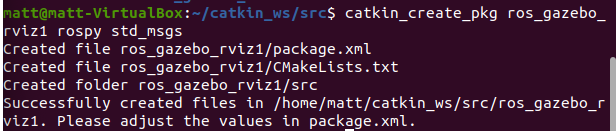
setup the ROS environment:



Start roscore

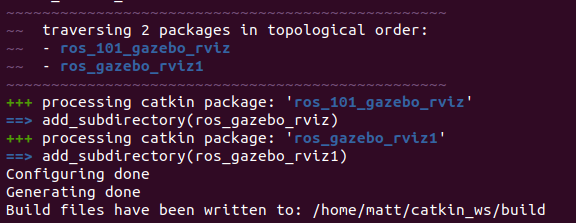


Then create new package.



Now build package



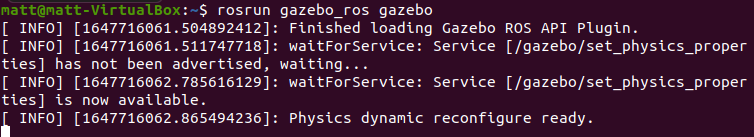


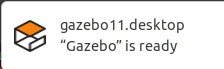
Don’t forget to add development setup to ROS path:



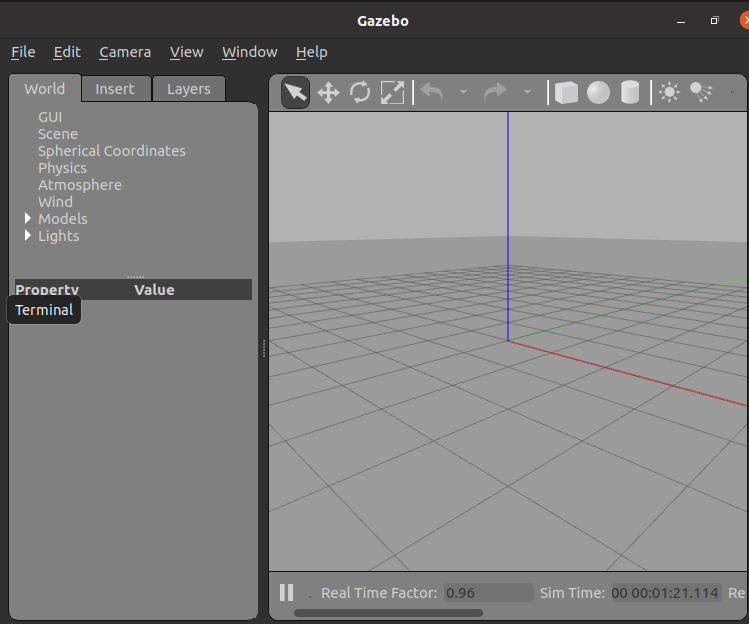
# Start Gazebo Simulator

Run command to confirm gazebo is installed and running correctly.

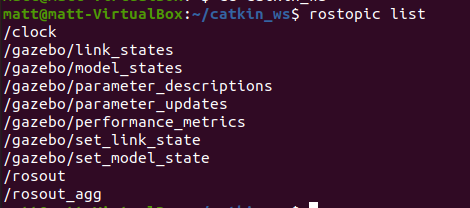




Gazebo GUI View

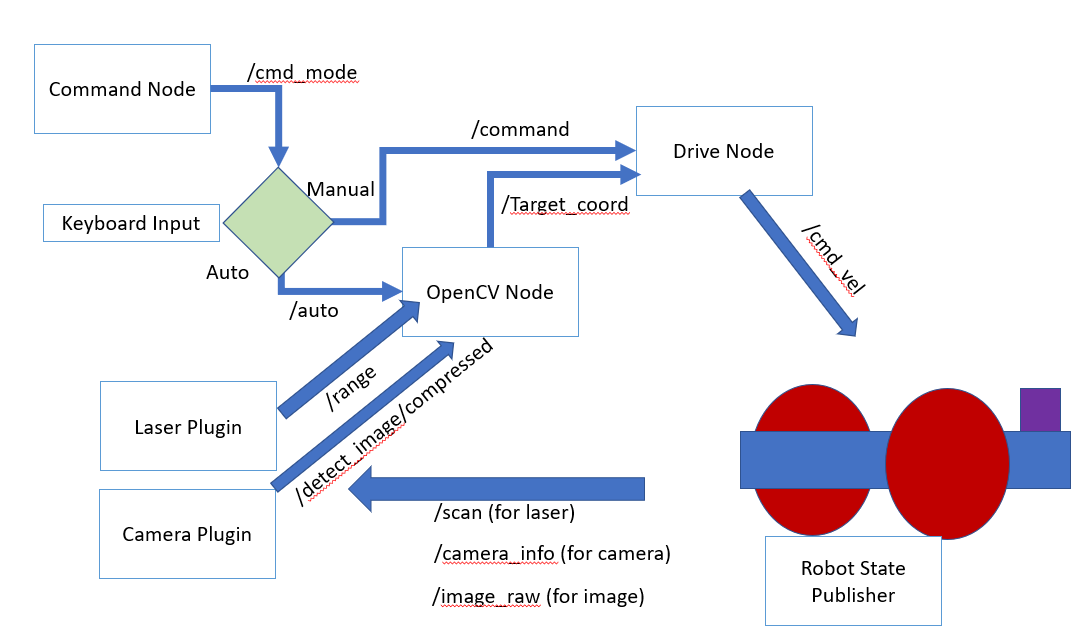


Verify all topics are publishing correctly



# Configure Robot Files

First, let’s look at the overall data flow diagram for this project:



A description of each file is provided below, with the full source code copied to the appendix:

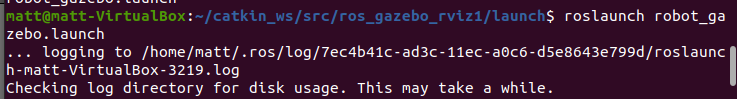
1. Command Node
   1. Command\_node.py file
   2. Similar to what was used in the first HW, but with the addition of the “a” key sending an “auto” command to enable autonomous operations.
2. OpenCV Node
   1. Image\_processor.py file
   2. This is the main brains behind the driving. The file takes in an image from the Gazebo Camera Plugin instead of the webcam from the RPi example. From here it performs the same target detection and identification of target coordinates. This node then publishes the coordinates to the drive node for turning decisions.
3. Drive Node
   1. Drive\_node.py file
   2. Again, this is similar to the provided code, but we had to modify the inputs and outputs to publish to ROS topics instead of publishing to RPi GPIO pins directly since there is no hardware in this simulation. Additionally, we had to add direct linear and angular velocity commands for the /cmd\_vel topic to send the correctly formatted commands to the Gazebo differential drive plugin module.
4. Robot State Publisher
   1. Node is started when the xacro\_robot\_gazebo.launch file is executed. This handles the incoming velocity commands and then publishes the robot state, including information from sensors.
5. Camera Plugin
   1. This node is also started when the xacro\_robot\_gazebo.launch file is executed. This node is taking in raw camera images and metadata from camera\_info to provide a pre-processed compressed image to be sent to the computer vision module.
   2. Note: the Laser Plugin was shown because we did optionally add a laser tracker to this module. It’s not directly required for the current form of autonomous driving, but it will be useful when the robot needs to perform more complex movements like collision avoidance.

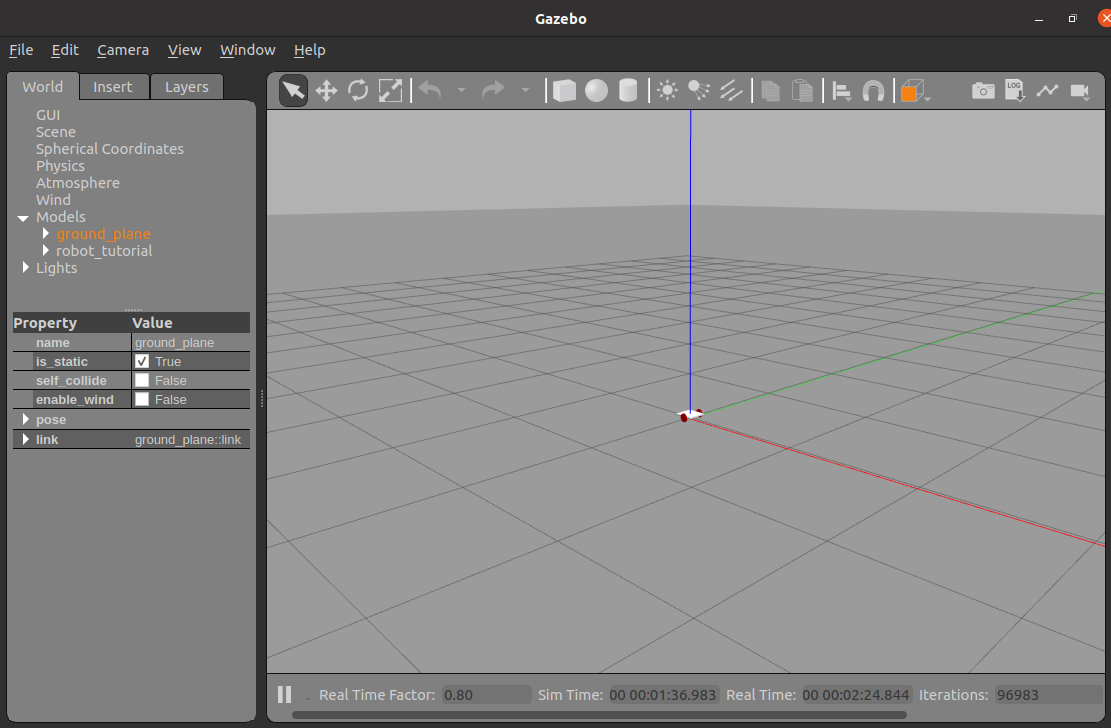
The package file structure looks like this:

* Ros\_gazebo\_rviz1
  + CMakeLists.txt
  + Package.xml
  + Launch
    - Robot\_gazebo.launch
  + Src
    - Sonar.py (not required, used for laser scanner)
  + Urdf
    - Robot\_tutorial.xacro
    - Robot\_tutorial\_gazebo.xacro
    - Camera\_gazebo.xacro
    - laser\_gazebo.xacro
  + Scripts
    - Command\_node.py
    - Drive\_node.py
    - Image\_processor.py
  + Rviz
    - main.rviz

# Load Robot Into Simulator

Let’s try the original robot, verify we can load into Gazebo:

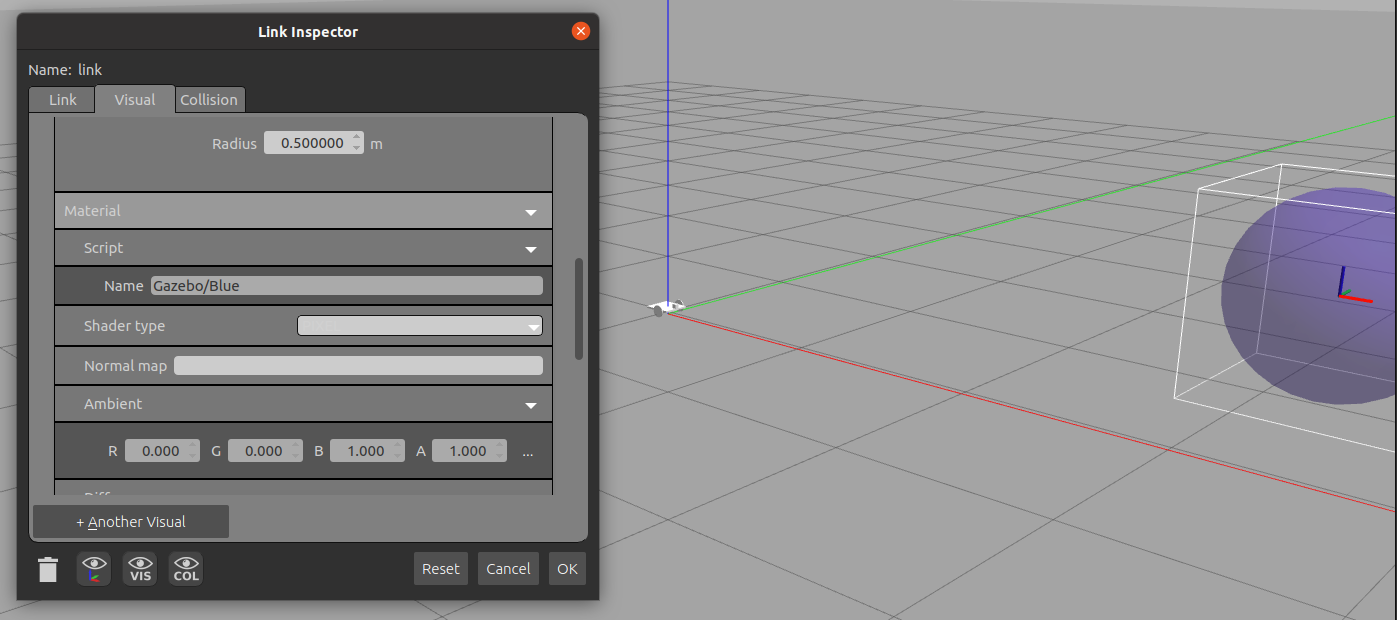




Now after adding the camera to the front of the robot:

| Rviz | Gazebo |
| --- | --- |
|  |  |

Add a blue sphere to the world:



# Prepare to Run

For this task we will utilize the command\_node and drive\_node files used for the original robot car controller.

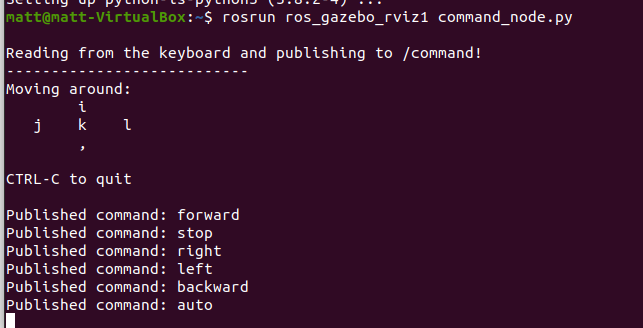
These two files were added to the ros\_gazebo\_rviz1 package in the scripts folder. Now make them executable:



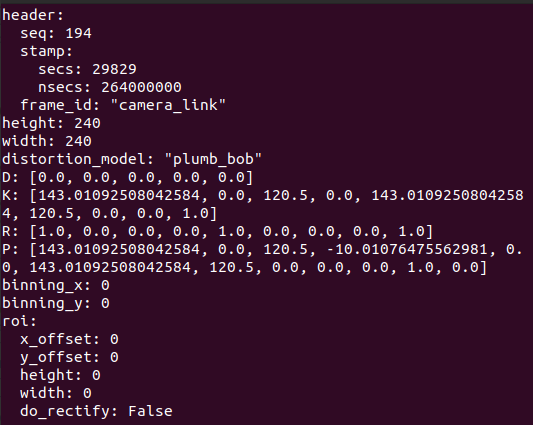
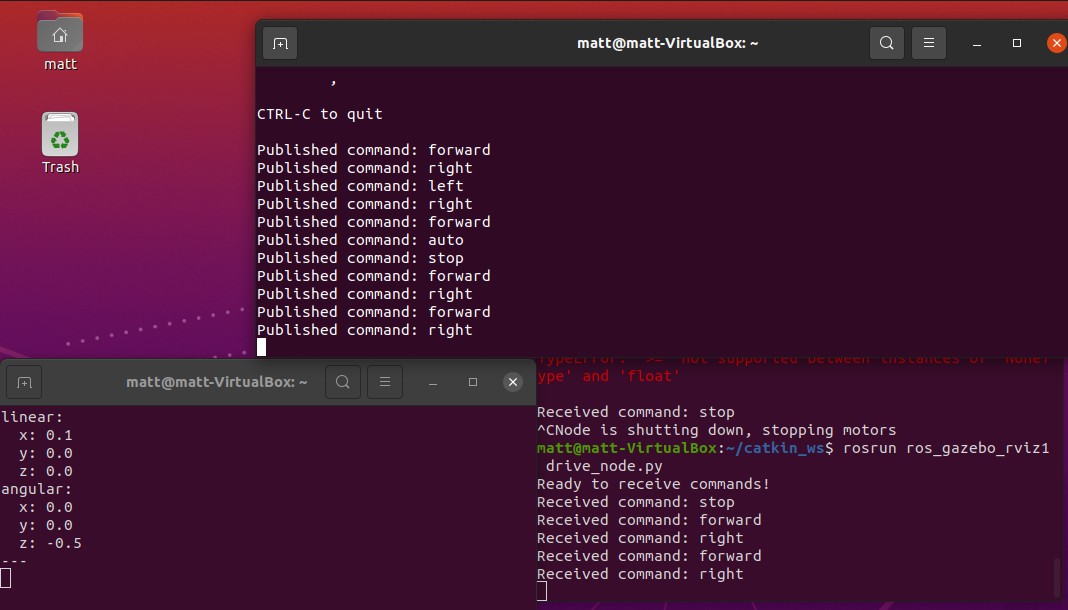
Add new “image\_processor.py” script, based on the OpenCV code but expanded to take in images from Gazebo instead of a webcam. Also make executable.



Verify command\_node is working:



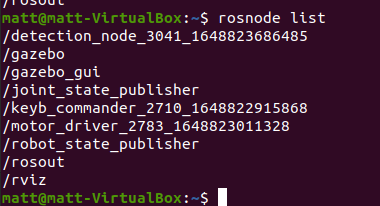
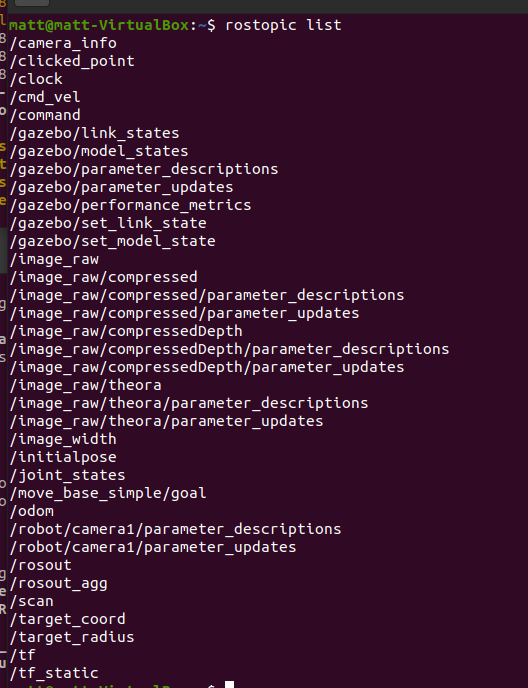
Check to see everything is publishing correctly:

* Camera info topic
  + 
  + 
* Do some manual robot motions to confirm the /cmd\_vel
  + Great, can see the linear and angular velocities being sent correctly to the robot
  + 

Now start the image processor node:

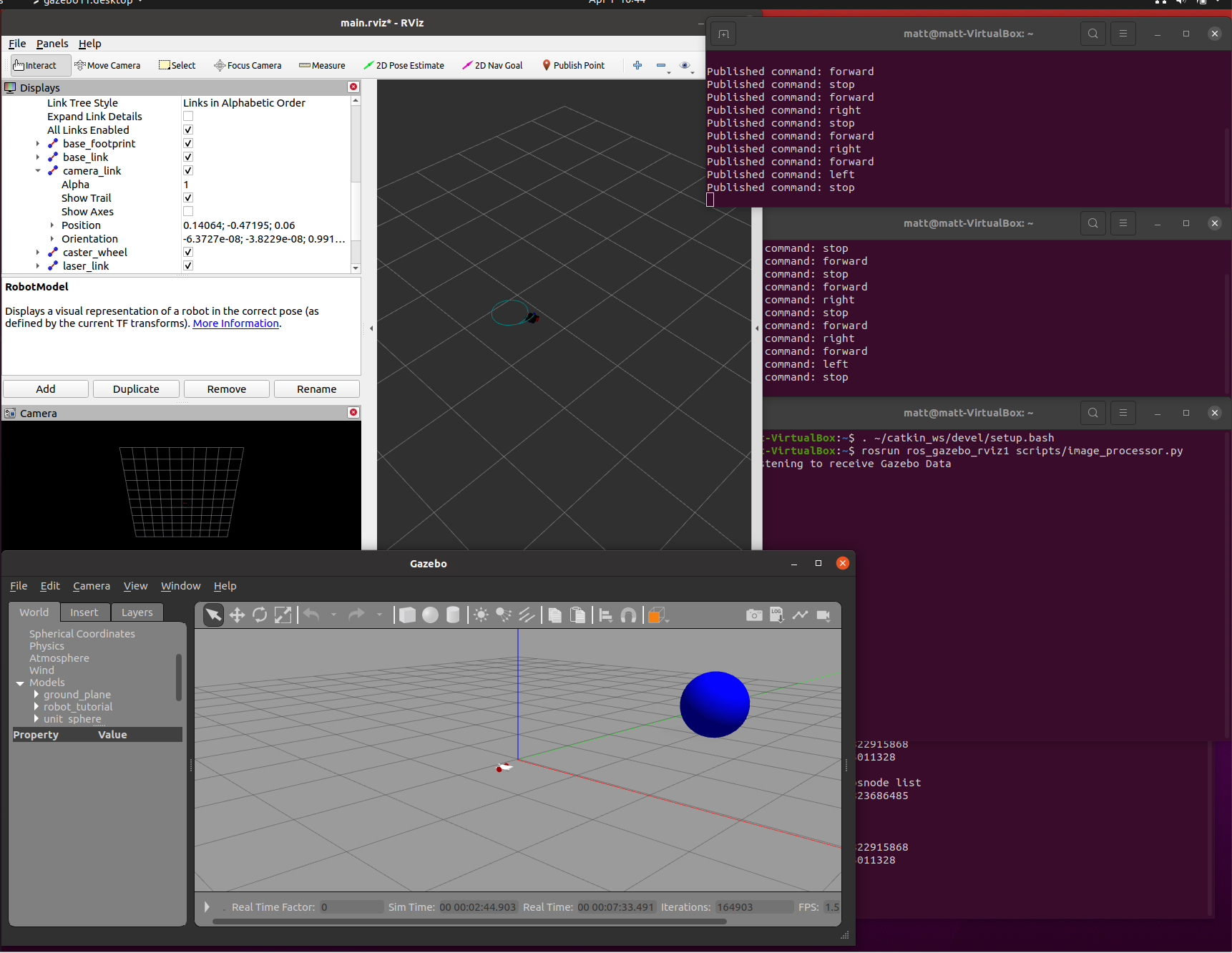


Let’s see that everything is available:

1. Check nodes:
   1. 
   2. Good we have everything expected
      1. Detection node is for image\_processor
      2. Gazebo and Rviz are both running
      3. Keyb\_commander and motor\_driver are running for command\_node and drive\_nodes respectively
      4. And the robot\_state\_publisher is sending back live robot feedback from Gazebo.
2. And check topics:
   1. 
   2. Excellent, we have all the image data, robot pose, image coordinates, gazebo information, and robot commands. It’s all now communicating.

Next will do a driving demo to ensure everything is talking correctly:

Good, the robot is accepting driving and turning commands as expected



# Appendix 1: Source Code

1. **Gazebo Launch File**

<launch>

<!--Robot Description from URDF-->

<param name="robot\_description" command="$(find xacro)/xacro --inorder $(find ros\_gazebo\_rviz1)/urdf/robot\_tutorial.xacro"/>

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="robot\_state\_publisher"/>

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher"/>

<!--RViz-->

<node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find ros\_gazebo\_rviz1)/rviz/main.rviz" />

<!--Gazebo empty world launch file-->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="debug" value="false" />

<arg name="gui" value="true" />

<arg name="paused" value="false"/>

<arg name="use\_sim\_time" value="false"/>

<arg name="headless" value="false"/>

<arg name="verbose" value="true"/>

</include>

<!--Gazebo Simulator-->

<node name="spawn\_model" pkg="gazebo\_ros" type="spawn\_model" args="-urdf -param robot\_description -model robot\_tutorial" output="screen"/>

</launch>

1. **Drive Node**

#!/usr/bin/env python

import rospy # Python library for ROS

from std\_msgs.msg import String, UInt16 # String and Unsigned integer message types

from geometry\_msgs.msg import Point, Twist # Point (x, y, z) message type

#import RPi.GPIO as GPIO # Raspberry i GPIO library

# Set the GPIO mode

#GPIO.setmode(GPIO.BCM)

#GPIO.setwarnings(False) # Disable GPIO warnings

# Set variables for the GPIO motor driver pins

motor\_left\_fw\_pin = 10

motor\_left\_bw\_pin = 9

motor\_right\_fw\_pin = 8

motor\_right\_bw\_pin = 7

# PWM signal frequency in Hz

pwm\_freq = 2000 # Use between 2000 - 20000

# PWM % duty cycle (change these to the values that work best for you)

fw\_bw\_duty\_cycle = 60

turn\_duty\_cycle = 40

# Set the GPIO Pin mode to output

#GPIO.setup(motor\_left\_fw\_pin, GPIO.OUT)

#GPIO.setup(motor\_left\_bw\_pin, GPIO.OUT)

#GPIO.setup(motor\_right\_fw\_pin, GPIO.OUT)

#GPIO.setup(motor\_right\_bw\_pin, GPIO.OUT)

# Create PWM objects to handle GPIO pins with 'pwm\_freq' frequency

#motor\_left\_fw = GPIO.PWM(motor\_left\_fw\_pin, pwm\_freq)

#motor\_left\_bw = GPIO.PWM(motor\_left\_bw\_pin, pwm\_freq)

#motor\_right\_fw = GPIO.PWM(motor\_right\_fw\_pin, pwm\_freq)

#motor\_right\_bw = GPIO.PWM(motor\_right\_bw\_pin, pwm\_freq)

# Start PWM with a duty cycle of 0 by default

#motor\_left\_fw.start(0)

#motor\_left\_bw.start(0)

#motor\_right\_fw.start(0)

#motor\_right\_bw.start(0)

# Global variables for storing received ROS messages

received\_command = ''

last\_received\_command = ''

received\_coord = Point(0, 0, 0)

target\_radius = None

MIN\_TGT\_RADIUS\_PERCENT = 0.05

image\_width = 0

CENTER\_WIDTH\_PERCENT = 0.30

# Publish the cmd\_vel values

motor\_command = rospy.Publisher('/cmd\_vel', Twist, queue\_size=10)

def listener():

# Initialize this node with a the name 'motor\_driver'

rospy.init\_node('motor\_driver', anonymous=True)

# Subscribe to the '/command' topic

rospy.Subscriber('/command', String, commandCallback)

# Subscribe to the '/target\_coord' topic

rospy.Subscriber('/target\_coord', Point, targetCoordCallback)

# Subscribe to the '/target\_radius' topic

rospy.Subscriber('/target\_radius', UInt16, targetRadiusCallback)

# Subscribe to the '/image\_width' topic

rospy.Subscriber('/image\_width', UInt16, imageWidthCallback)

# Put this node in an inifinite loop to execute when new messages arrive

rospy.spin()

# '/command' topic message handler

def commandCallback(message):

global received\_command

global last\_received\_command

received\_command = message.data

if received\_command == 'forward':

forward()

elif received\_command == 'backward':

backward()

elif received\_command == 'left':

left()

elif received\_command == 'right':

right()

elif received\_command == 'stop':

stopMotors()

elif received\_command == 'auto':

autonomous()

else:

print('Unknown command!')

if received\_command != last\_received\_command:

print('Received command: ' + received\_command)

last\_received\_command = received\_command

# Follow the target in autonomous mode:

def autonomous():

global image\_width

global target\_radius

global MIN\_TGT\_RADIUS\_PERCENT

if target\_radius >= image\_width\*MIN\_TGT\_RADIUS\_PERCENT and target\_radius <= image\_width/3:

if abs(received\_coord.x) <= image\_width\*CENTER\_WIDTH\_PERCENT:

forward()

elif received\_coord.x > 0:

right()

elif received\_coord.x < 0:

left()

else:

stopMotors()

print('stopMotors')

# '/image\_width' topic message handler

def imageWidthCallback(message):

global image\_width

image\_width = message.data

# '/target\_radius' topic message handler

def targetRadiusCallback(message):

global target\_radius

target\_radius = message.data

# '/target\_coord' topic message handler

def targetCoordCallback(message):

# global received\_coord

received\_coord.x = message.x

received\_coord.y = message.y

# print("received\_coord = ", received\_coord.x, received\_coord.y)

# Turn both motors forwards

def forward():

motor\_left\_fw = fw\_bw\_duty\_cycle

motor\_left\_bw = 0

motor\_right\_fw = fw\_bw\_duty\_cycle

motor\_right\_bw = 0

#configure cmd\_vel output

cmd\_vel = Twist()

cmd\_vel.linear.x = 0.1

cmd\_vel.angular.z = 0.0

motor\_command.publish(cmd\_vel)

# Turn both motors backwards

def backward():

motor\_left\_fw = 0

motor\_left\_bw = fw\_bw\_duty\_cycle

motor\_right\_fw = 0

motor\_right\_bw = fw\_bw\_duty\_cycle

#configure cmd\_vel output

cmd\_vel = Twist()

cmd\_vel.linear.x = -0.1

cmd\_vel.angular.z = 0.0

motor\_command.publish(cmd\_vel)

# Turn left motor backward, right motor forward

def left():

motor\_left\_fw = 0

motor\_left\_bw = turn\_duty\_cycle

motor\_right\_fw = turn\_duty\_cycle

motor\_right\_bw = 0

#configure cmd\_vel output

cmd\_vel = Twist()

cmd\_vel.linear.x = 0.1

cmd\_vel.angular.z = 0.5

motor\_command.publish(cmd\_vel)

# Turn right motor backward, left motor forward

def right():

motor\_left\_fw = turn\_duty\_cycle

motor\_left\_bw = 0

motor\_right\_fw = 0

motor\_right\_bw = turn\_duty\_cycle

#configure cmd\_vel output

cmd\_vel = Twist()

cmd\_vel.linear.x = 0.1

cmd\_vel.angular.z = -0.5

motor\_command.publish(cmd\_vel)

# Turn all motors off

def stopMotors():

motor\_left\_fw = 0

motor\_left\_bw = 0

motor\_right\_fw = 0

motor\_right\_bw = 0

#configure cmd\_vel output

cmd\_vel = Twist()

cmd\_vel.linear.x = 0.0

cmd\_vel.angular.z = 0.0

motor\_command.publish(cmd\_vel)

if \_\_name\_\_ == '\_\_main\_\_':

print('Ready to receive commands!')

listener()

print('Node is shutting down, stopping motors')

stopMotors()

1. **Command Node**

#!/usr/bin/env python

# Parts of this code are based on the 'teleop\_twist\_keyboard' node

import rospy

from std\_msgs.msg import String

import sys, select, termios, tty

command = 'stop'

last\_command = 'stop'

msg = """

Reading from the keyboard and publishing to /command!

---------------------------

Moving around:

i

j k l

,

CTRL-C to quit

"""

def talker():

global command

global last\_command

pub = rospy.Publisher('/command', String, queue\_size=10)

rospy.init\_node('keyb\_commander', anonymous=True)

rate = rospy.Rate(10) # 10hz

print(msg)

# rospy.rosinfo(msg)

while not rospy.is\_shutdown():

key\_timeout = 0.6

k = getKey(key\_timeout)

if k == "i":

command = 'forward'

elif k == ",":

command = 'backward'

elif k == "j":

command = 'left'

elif k == "l":

command = 'right'

elif k == "k":

command = 'stop'

elif k == "a":

command = 'auto'

elif k == '\x03': # To detect CTRL-C

break

if command != last\_command:

print("Published command: " + command)

last\_command = command

pub.publish(command)

rate.sleep()

def getKey(key\_timeout):

tty.setraw(sys.stdin.fileno())

rlist, \_, \_ = select.select([sys.stdin], [], [], key\_timeout)

if rlist:

key = sys.stdin.read(1)

else:

key = ''

termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)

return key

if \_\_name\_\_ == '\_\_main\_\_':

settings = termios.tcgetattr(sys.stdin)

try:

talker()

except rospy.ROSInterruptException:

pass

1. **Image\_Processor.py**

#!/usr/bin/env python

# Import libraries

import rospy # Python library for ROS

from sensor\_msgs.msg import CompressedImage, CameraInfo # CompressedImage message type

from geometry\_msgs.msg import Point # Point (x, y, z) message type

from std\_msgs.msg import UInt16, String # Unsigned integer message type

import cv2 # OpenCV library

from cv\_bridge import CvBridge # Converts between OpenCV and ROS images

import time

from PIL import Image

import numpy as np

# Global constants and variables

NUM\_FILT\_POINTS = 5 # Number of filtering points for the Moving Average Filter

DESIRED\_IMAGE\_HEIGHT = 240 # A smaller image makes the detection less CPU intensive

# A dictionary of two empty buffers (arrays) for the Moving Average Filter

filt\_buffer = {'width':[], 'height':[]}

# A dictionary of general parameters derived from the camera image size,

# which will be populated later with the 'get\_image\_params' function

params = {'image\_height':None, 'image\_width': None,'resized\_height':None,'resized\_width': None,

'x\_ax\_pos':None, 'y\_ax\_pos':None, 'scaling\_factor':None}

def listener():

# Set the node's name

rospy.init\_node('detection\_node', anonymous=True)

# Subscribe to the '/command' topic

rospy.Subscriber('/image\_raw', UInt16, RawImageCallback, queue\_size=10)

# Subscribe to the '/camera\_info' topic

rospy.Subscriber('/camera\_info', CameraInfo, CameraInfoCallback, queue\_size=10)

# Put this node in an inifinite loop to execute when new messages arrive

#rospy.spin()

# '/image\_raw' topic message handler

def callbackImage(message):

global image\_raw

image\_raw = message.data

print(image\_raw)

# '/image\_raw' topic message handler

def callbackImageComp(message):

global image\_comp

global np\_img

#image\_comp = message.data

#image\_comp = cv2.imread(img)

#print(image\_comp)

np\_arr = np.fromstring(message.data, np.uint8)

#print(np\_arr)

np\_img = np\_arr

image\_comp = cv2.imdecode(np\_arr, cv2.IMREAD\_COLOR)

image\_comp = np\_arr

#print(image\_comp)

# '/camera\_info' topic message handler

def CameraInfoCallback(message):

global camera\_info

# all the camera data (header, height, width, distortion\_model, D, K, R, P, binning\_x, binning\_y, roi)

camera\_info = message

# pull useful bits

image\_height = camera\_info.height

image\_width = camera\_info.width

def detect\_target():

""" Main entry function for this node. """

global image\_raw

global image\_comp

global np\_img

# Publishes the video frames from the detection process

# detect\_image\_pub = rospy.Publisher('detect\_image', Image, queue\_size=10)

detect\_image\_pub = rospy.Publisher('detect\_image/compressed', CompressedImage, queue\_size=10)

# Publishes the (x, y, 0) coordinates for the detected target

target\_coord\_pub = rospy.Publisher('/target\_coord', Point, queue\_size=10)

#print(target\_coord\_pub)

# Publishes the detected target's computed enclosing radius

target\_radius\_pub = rospy.Publisher('/target\_radius', UInt16, queue\_size=10)

#print(target\_radius\_pub)

# Publishes the image frame width after scaling used in the detection process

image\_width\_pub = rospy.Publisher('/image\_width', UInt16, queue\_size=10)

# Set the node's name

rospy.init\_node('detection\_node', anonymous=True)

# The node will run 30 times per second

rate = rospy.Rate(0.2) # 30 Hz

# Subscribe to the '/command' topic

rospy.Subscriber('/image\_raw', UInt16, callbackImage)

image\_comp = rospy.Subscriber('/image\_raw/compressed', CompressedImage, callbackImageComp)

#print(image\_comp.data)

# Subscribe to the '/camera\_info' topic

rospy.Subscriber('camera\_info', CameraInfo, CameraInfoCallback, queue\_size=10)

# Create a VideoCapture object

#vid\_cam = cv2.VideoCapture(0) # '0''is de index for the default webcam

# Check if the camera opened correctly

#if vid\_cam.isOpened() is False:

# print('[ERROR] Couldnt open the camera.')

# return

#print('-- Camera opened successfully')

# define image

#image = image\_raw

# Compute general parameters

#get\_image\_params(vid\_cam)

#print("-- Original image width, height: ", {params['image\_width']}, {params['image\_height']})

# To convert between OpenCV and ROS images

bridge = CvBridge()

# While ROS is still running.

while not rospy.is\_shutdown():

start\_time = time.time()

# Set the node's name

rospy.init\_node('detection\_node', anonymous=True)

# Subscribe to the '/command' topic

rospy.Subscriber('/image\_raw/compressed', CompressedImage, callbackImageComp)

pix = np.array(image\_comp)

print(pix)

#img = cv2.resize(image\_comp, (240,240))

img = image\_comp

print("img: ", img)

# Compute general parameters

get\_image\_params(img)

# Get the target coordinates

tgt\_cam\_coord, frame, contour, radius = get\_target\_coordinates(img)

# If a target was found, filter their coordinates

if tgt\_cam\_coord['width'] is not None and tgt\_cam\_coord['height'] is not None:

# Apply Moving Average filter to target camera coordinates

tgt\_filt\_cam\_coord = moving\_average\_filter(tgt\_cam\_coord)

# No target was found, set target camera coordinates to the Cartesian origin,

# so the drone doesn't move

else:

# The Cartesian origin is where the x and y Cartesian axes are located

# in the image, in pixel units

tgt\_cam\_coord = {'width':params['y\_ax\_pos'], 'height':params['x\_ax\_pos']} # Needed just for drawing objects

tgt\_filt\_cam\_coord = {'width':params['y\_ax\_pos'], 'height':params['x\_ax\_pos']}

# Convert from camera coordinates to Cartesian coordinates (in pixel units)

tgt\_cart\_coord = {'x':(tgt\_filt\_cam\_coord['width'] - params['y\_ax\_pos']),

'y':(params['x\_ax\_pos'] - tgt\_filt\_cam\_coord['height'])}

# Draw objects over the detection image frame just for visualization

frame = draw\_objects(tgt\_cam\_coord, tgt\_filt\_cam\_coord, frame, contour)

# Publish the detection image after convertin from OpenCV to ROS

# detect\_image\_pub.publish(bridge.cv2\_to\_imgmsg(frame))

detect\_image\_pub.publish(bridge.cv2\_to\_compressed\_imgmsg(frame))

# Publish the detected target's coordinates (x, y, 0)

tgt\_coord\_msg = Point(tgt\_cart\_coord['x'], tgt\_cart\_coord['y'], 0)

target\_coord\_pub.publish(tgt\_coord\_msg)

print(tgt\_coord\_msg)

# Publish de detected target's enclosing radius

target\_radius\_pub.publish(radius)

print(radius)

# Publish the image frame's resized width

image\_width\_pub.publish(params['resized\_width'])

# Show the detection image frame on screen

# Optionally you can comment this line when running this node remotely through SSH:

cv2.imshow("Detect and Track", frame)

delta\_time = end\_time = time.time()

detection\_time = round(end\_time-start\_time, 3)

print("Detection time: " + str(detection\_time ))

# Catch aborting key from computer keyboard

key = cv2.waitKey(1) & 0xFF

# If the 'q' key is pressed, break the 'while' infinite loop

if key == ord("q"):

break

# Sleep just enough to maintain the desired rate

rate.sleep()

def get\_image\_params(image):

""" Computes useful general parameters derived from the camera image size."""

# Grab a frame and get its size

#is\_grabbed, frame = vid\_cam.read()

frame=image

try:

params['image\_height'], params['image\_width'], \_ = image.shape

except:

params['image\_height'] = 240

params['image\_width'] = 240

# Compute the scaling factor to scale the image to a desired size

if params['image\_height'] != DESIRED\_IMAGE\_HEIGHT:

# Rounded scaling factor. Convert 'DESIRED\_IMAGE\_HEIGHT' to float or the division will throw zero

params['scaling\_factor'] = round((float(DESIRED\_IMAGE\_HEIGHT) / params['image\_height']), 3)

else:

params['scaling\_factor'] = 1

print("params['scaling\_factor']: ", params['scaling\_factor'])

print("params['scaling\_factor']: ", DESIRED\_IMAGE\_HEIGHT / params['image\_height'])

# Compute resized width and height and resize the image

params['resized\_width'] = int(params['image\_width'] \* params['scaling\_factor'])

params['resized\_height'] = int(params['image\_height'] \* params['scaling\_factor'])

dimension = (params['resized\_width'], params['resized\_height'])

# dimension = (int(params['resized\_width']), int(params['resized\_height']))

#frame = cv2.resize(frame, dimension, interpolation = cv2.INTER\_AREA)

# Compute the position for the X and Y Cartesian coordinates in camera pixel units

params['x\_ax\_pos'] = int(params['resized\_height']/2 - 1)

params['y\_ax\_pos'] = int(params['resized\_width']/2 - 1)

return

def get\_target\_coordinates(image):

""" Detects a target by using color range segmentation and returns its 'camera pixel' coordinates."""

# Use the 'threshold\_inRange.py' script included with the code to get

# your own bounds with any color

# To detect a blue target:

HSV\_LOWER\_BOUND = (107, 119, 41)

HSV\_UPPER\_BOUND = (124, 255, 255)

# Grab a frame in BGR (Blue, Green, Red) space color

#is\_grabbed, frame = vid\_cam.read()

frame = image

# Resize the image frame for the detection process, if needed

if params['scaling\_factor'] != 1:

dimension = (params['resized\_width'], params['resized\_height'])

frame = cv2.resize(frame, dimension, interpolation = cv2.INTER\_AREA)

# Blur the image to remove high frequency content

blurred = cv2.GaussianBlur(frame, (11, 11), 0)

# Change color space from BGR to HSV

hsv = cv2.cvtColor(blurred, cv2.COLOR\_BGR2HSV)

# Histogram equalisation to minimize the effect of variable lighting

# hsv[:, :, 0] = cv2.equalizeHist(hsv[:, :, 0]) # on the H-channel

# hsv[:, :, 1] = cv2.equalizeHist(hsv[:, :, 1]) # on the S-channel

# hsv[:, :, 2] = cv2.equalizeHist(hsv[:, :, 2]) # on the V-channel

# Get a mask with all the pixels inside our defined color boundaries

mask = cv2.inRange(hsv, HSV\_LOWER\_BOUND, HSV\_UPPER\_BOUND)

# Erode and dilate to remove small blobs

mask = cv2.erode(mask, None, iterations=2)

mask = cv2.dilate(mask, None, iterations=2)

# Find all contours in the masked image

\_, contours, \_ = cv2.findContours(mask,

cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

# Centroid coordinates to be returned:

cX = None

cY = None

# To save the larges contour, presumably the detected object

largest\_contour = None

tgt\_radius = 0

# Check if at least one contour was found

if len(contours) > 0:

# Get the largest contour of all posibly detected

largest\_contour = max(contours, key=cv2.contourArea)

# Compute the radius of an enclosing circle aorund the largest contour

(x,y), tgt\_radius = cv2.minEnclosingCircle(largest\_contour)

center = (int(x),int(y))

tgt\_radius = int(tgt\_radius)

# cv2.circle(frame, center, tgt\_radius , (3, 186, 252), 3)

# Compute contour raw moments

M = cv2.moments(largest\_contour)

# Get the contour's centroid

cX = int(M["m10"] / M["m00"])

cY = int(M["m01"] / M["m00"])

# Return centroid coordinates (camera pixel units), the analized frame and the largest contour

return {'width':cX, 'height':cY}, frame, largest\_contour, tgt\_radius

def moving\_average\_filter(coord):

""" Applies Low-Pass Moving Average Filter to a pair of coordinates."""

# Append new coordinates to filter buffers

filt\_buffer['width'].append(coord['width'])

filt\_buffer['height'].append(coord['height'])

# If the filters were full already with a number of NUM\_FILT\_POINTS values,

# discard the oldest value (FIFO buffer)

if len(filt\_buffer['width']) > NUM\_FILT\_POINTS:

filt\_buffer['width'] = filt\_buffer['width'][1:]

filt\_buffer['height'] = filt\_buffer['height'][1:]

# Compute filtered camera coordinates

N = len(filt\_buffer['width']) # Get the number of values in buffers (will be < NUM\_FILT\_POINTS at the start)

# Sum all values for each coordinate

w\_sum = sum( filt\_buffer['width'] )

h\_sum = sum( filt\_buffer['height'] )

# Compute the average

w\_filt = int(round(w\_sum / N))

h\_filt = int(round(h\_sum / N))

# Return filtered coordinates as a dictionary

return {'width':w\_filt, 'height':h\_filt}

def draw\_objects(cam\_coord, filt\_cam\_coord, frame, contour):

""" Draws visualization objects from the detection process.

Position coordinates of every object are always in 'camera pixel' units"""

# Draw the Cartesian axes

cv2.line(frame, (0, params['x\_ax\_pos']), (params['resized\_width'], params['x\_ax\_pos']), (0, 128, 255), 1)

cv2.line(frame, (params['y\_ax\_pos'], 0), (params['y\_ax\_pos'], params['resized\_height']), (0, 128, 255), 1)

cv2.circle(frame, (params['y\_ax\_pos'], params['x\_ax\_pos']), 1, (255, 255, 255), -1)

# Draw the detected object's contour, if any

if contour is not None:

cv2.drawContours(frame, [contour], -1, (0, 255, 0), 2)

# Compute Cartesian coordinates of unfiltered detected object's centroid

x\_cart\_coord = cam\_coord['width'] - params['y\_ax\_pos']

y\_cart\_coord = params['x\_ax\_pos'] - cam\_coord['height']

# Compute Cartesian coordinates of filtered detected object's centroid

x\_filt\_cart\_coord = filt\_cam\_coord['width'] - params['y\_ax\_pos']

y\_filt\_cart\_coord = params['x\_ax\_pos'] - filt\_cam\_coord['height']

# Draw unfiltered centroid as a red dot, including coordinate values

cv2.circle(frame, (cam\_coord['width'], cam\_coord['height']), 5, (0, 0, 255), -1)

cv2.putText(frame, str(x\_cart\_coord) + ", " + str(y\_cart\_coord),

(cam\_coord['width'] + 25, cam\_coord['height'] - 25), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 255), 1)

# Draw filtered centroid as a yellow dot, including coordinate values

cv2.circle(frame, (filt\_cam\_coord['width'], filt\_cam\_coord['height']), 5, (3, 186, 252), -1)

cv2.putText(frame, str(x\_filt\_cart\_coord) + ", " + str(y\_filt\_cart\_coord),

(filt\_cam\_coord['width'] + 25, filt\_cam\_coord['height'] - 25), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (3, 186, 252), 1)

return frame # Return the image frame with all drawn objects

if \_\_name\_\_ == '\_\_main\_\_':

# try:

print("Start Listening to receive Gazebo Data")

#listener()

print("Prepare for Autonomous Target Detection")

detect\_target()

#except rospy.ROSInterruptException:

# pass

1. **Robot\_tutorial.xacro**

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot\_tutorial">

<xacro:property name="base\_width" value="0.16"/>

<xacro:property name="base\_len" value="0.16"/>

<xacro:property name="wheel\_radius" value="0.035"/>

<xacro:property name="base\_wheel\_gap" value="0.007"/>

<xacro:property name="wheel\_separation" value="0.15"/>

<xacro:property name="wheel\_joint\_offset" value="0.02"/>

<xacro:property name="caster\_wheel\_radius" value="${wheel\_radius/2}"/>

<xacro:property name="caster\_wheel\_mass" value="0.001"/>

<xacro:property name="caster\_wheel\_joint\_offset" value="-0.052"/>

<!--Color Properties-->

<material name="blue">

<color rgba="0 0 0.8 1"/>

</material>

<material name="black">

<color rgba="0 0 0 1"/>

</material>

<material name="white">

<color rgba="1 1 1 1"/>

</material>

<material name="red">

<color rgba="0.8 0.0 0.0 1.0"/>

</material>

<!--Interial macros-->

<xacro:macro name="cylinder\_inertia" params="m r h">

<inertial>

<mass value="${m}"/>

<inertia ixx="${m\*(3\*r\*r+h\*h)/12}" ixy = "0" ixz = "0" iyy="${m\*(3\*r\*r+h\*h)/12}" iyz = "0" izz="${m\*r\*r/2}"/>

</inertial>

</xacro:macro>

<xacro:macro name="box\_inertia" params="m w h d">

<inertial>

<mass value="${m}"/>

<inertia ixx="${m / 12.0 \* (d\*d + h\*h)}" ixy="0.0" ixz="0.0" iyy="${m / 12.0 \* (w\*w + h\*h)}" iyz="0.0" izz="${m / 12.0 \* (w\*w + d\*d)}"/>

</inertial>

</xacro:macro>

<xacro:macro name="sphere\_inertia" params="m r">

<inertial>

<mass value="${m}"/>

<inertia ixx="${2.0\*m\*(r\*r)/5.0}" ixy="0.0" ixz="0.0" iyy="${2.0\*m\*(r\*r)/5.0}" iyz="0.0" izz="${2.0\*m\*(r\*r)/5.0}"/>

</inertial>

</xacro:macro>

<!--Base Footprint-->

<link name="base\_footprint">

<xacro:box\_inertia m="10" w="0.001" h="0.001" d="0.001"/>

<visual>

<origin xyz="0 0 0" rpy="0 0 0" />

<geometry>

<box size="0.001 0.001 0.001" />

</geometry>

</visual>

</link>

<!--Base link-->

<link name="base\_link">

<xacro:box\_inertia m="10" w="${base\_len}" h="${base\_width}" d="0.01"/>

<visual>

<geometry>

<box size="${base\_len} ${base\_width} 0.01"/>

</geometry>

<material name="black"/>

</visual>

<collision>

<geometry>

<box size="${base\_len} ${base\_width} 0.01"/>

</geometry>

</collision>

</link>

<!--base\_link to base\_footprint Joint-->

<joint name="base\_link\_joint" type="fixed">

<origin xyz="0 0 ${wheel\_radius + 0.005}" rpy="0 0 0" />

<parent link="base\_footprint"/>

<child link="base\_link" />

</joint>

<!--Wheel link & joint macro-->

<xacro:macro name="wheel" params="prefix reflect">

<link name="${prefix}\_wheel">

<visual>

<origin xyz="0 0 0" rpy="${pi/2} 0 0"/>

<geometry>

<cylinder radius="${wheel\_radius}" length="0.005"/>

</geometry>

<material name="red"/>

</visual>

<collision>

<origin xyz="0 0 0" rpy="${pi/2} 0 0"/>

<geometry>

<cylinder radius="${wheel\_radius}" length="0.005"/>

</geometry>

</collision>

<xacro:cylinder\_inertia m="10" r="${wheel\_radius}" h="0.005"/>

</link>

<joint name="${prefix}\_wheel\_joint" type="continuous">

<axis xyz="0 1 0" rpy="0 0 0" />

<parent link="base\_link"/>

<child link="${prefix}\_wheel"/>

<origin xyz="${wheel\_joint\_offset} ${((base\_width/2)+base\_wheel\_gap)\*reflect} -0.005" rpy="0 0 0"/>

</joint>

</xacro:macro>

<!--Create Left & Right Wheel links/joints-->

<xacro:wheel prefix="left" reflect="1"/>

<xacro:wheel prefix="right" reflect="-1"/>

<!--Caster Wheel Link-->

<link name="caster\_wheel">

<visual>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<sphere radius="${caster\_wheel\_radius}"/>

</geometry>

<material name="blue"/>

</visual>

<collision>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<sphere radius="${caster\_wheel\_radius}"/>

</geometry>

</collision>

<xacro:sphere\_inertia m="10" r="${caster\_wheel\_radius}"/>

</link>

<!--Caster Wheel Joint-->

<joint name="caster\_wheel\_joint" type="continuous">

<axis xyz="0 1 0" rpy="0 0 0" />

<parent link="base\_link"/>

<child link="caster\_wheel"/>

<origin xyz="${caster\_wheel\_joint\_offset} 0 -${caster\_wheel\_radius+0.005}" rpy="0 0 0"/>

</joint>

<!-- Laser Link-->

<link name="laser\_link">

<visual>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<box size="0.025 0.025 0.025" />

</geometry>

<material name="blue"/>

</visual>

<collision>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<box size="0.025 0.025 0.025" />

</geometry>

</collision>

<xacro:box\_inertia m="1" w="0.1" h="0.1" d="0.1" />

</link>

<!--Laser Joint-->

<joint name="laser\_joint" type="fixed">

<axis xyz="0 1 0" />

<origin xyz="0.075 0 0.02" rpy="0 0 0" />

<parent link="base\_link"/>

<child link="laser\_link"/>

</joint>

<!-- Camera Link-->

<link name="camera\_link">

<visual>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<box size="0.025 0.025 0.025" />

</geometry>

<material name="red"/>

</visual>

<collision>

<origin xyz="0 0 0" rpy="0 0 0"/>

<geometry>

<box size="0.025 0.025 0.025" />

</geometry>

</collision>

<xacro:box\_inertia m="1" w="0.1" h="0.1" d="0.1" />

</link>

<!--Camera Joint-->

<joint name="camera\_joint" type="fixed">

<axis xyz="0 1 0" />

<origin xyz="0.075 0 0.02" rpy="0 0 0" />

<parent link="base\_link"/>

<child link="camera\_link"/>

</joint>

<xacro:include filename="$(find ros\_gazebo\_rviz1)/urdf/robot\_tutorial\_gazebo.xacro"/>

</robot>

1. **Robot\_tutorial\_gazebo.xacro**

<?xml version="1.0"?>

<robot>

<gazebo>

<plugin name="differential\_drive\_controller" filename="libgazebo\_ros\_diff\_drive.so">

<alwaysOn>false</alwaysOn>

<legacyMode>false</legacyMode>

<updateRate>20</updateRate>

<leftJoint>left\_wheel\_joint</leftJoint>

<rightJoint>right\_wheel\_joint</rightJoint>

<wheelSeparation>${wheel\_separation}</wheelSeparation>

<wheelDiameter>${wheel\_radius \* 2}</wheelDiameter>

<torque>20</torque>

<commandTopic>/cmd\_vel</commandTopic>

<odometryTopic>/odom</odometryTopic>

<odometryFrame>odom</odometryFrame>

<robotBaseFrame>base\_footprint</robotBaseFrame>

</plugin>

</gazebo>

<gazebo reference="base\_link">

<material>Gazebo/White</material>

</gazebo>

<gazebo reference="left\_wheel">

<material>Gazebo/Red</material>

</gazebo>

<gazebo reference="right\_wheel">

<material>Gazebo/Red</material>

</gazebo>

<gazebo reference="laser\_link">

<sensor type="ray" name="laser\_sensor">

<pose>0 0 0 0 0 0 </pose>

<visualize>false</visualize>

<update\_rate>40</update\_rate>

<ray>

<scan>

<horizontal>

<samples>5</samples>

<min\_angle>-0.0349066</min\_angle>

<max\_angle>0.0349066</max\_angle>

</horizontal>

</scan>

<range>

<min>0.10</min>

<max>30.0</max>

<resolution>0.01</resolution>

</range>

<noise>

<type>gaussian</type>

<mean>0.0</mean>

<stddev>0.01</stddev>

</noise>

</ray>

<plugin name="gazebo\_ros\_head\_hokuyo\_controller" filename="libgazebo\_ros\_laser.so">

<topicName>/scan</topicName>

<frameName>laser\_link</frameName>

</plugin>

</sensor>

</gazebo>

<gazebo reference="camera\_link">

<sensor type="camera" name="camera1">

<update\_rate>1.0</update\_rate>

<camera name="head">

<horizontal\_fov>1.3962634</horizontal\_fov>

<image>

<width>240</width>

<height>240</height>

<format>R8G8B8</format>

</image>

<clip>

<near>0.02</near>

<far>300</far>

</clip>

<noise>

<type>gaussian</type>

<!-- Noise is sampled independently per pixel on each frame.

That pixel's noise value is added to each of its color

channels, which at that point lie in the range [0,1]. -->

<mean>0.0</mean>

<stddev>0.007</stddev>

</noise>

</camera>

<plugin name="camera\_controller" filename="libgazebo\_ros\_camera.so">

<alwaysOn>true</alwaysOn>

<updateRate>0.0</updateRate>

<cameraName>robot/camera1</cameraName>

<imageTopicName>/image\_raw</imageTopicName>

<cameraInfoTopicName>/camera\_info</cameraInfoTopicName>

<frameName>camera\_link</frameName>

<hackBaseline>0.07</hackBaseline>

<distortionK1>0.0</distortionK1>

<distortionK2>0.0</distortionK2>

<distortionK3>0.0</distortionK3>

<distortionT1>0.0</distortionT1>

<distortionT2>0.0</distortionT2>

</plugin>

</sensor>

</gazebo>

</robot>