**Visual object recognition**

You can run this tutorial on:

* [ROSbot 2.0](https://store.husarion.com/products/rosbot)
* [ROSbot 2.0 PRO](https://store.husarion.com/collections/dev-kits/products/rosbot-pro)
* [ROSbot 2.0 simulation model (Gazebo)](https://github.com/husarion/rosbot_description)

**Introduction**

Objects can be recognized by a robot with use of a vision system. It is based on image characteristics like points, lines, edges colours and their relative positions.

Processing of object recognition consists of two steps. First is teaching and should be executed before main robot operation. During this step object is presented to the vision system, image and extracted set of features are saved as a pattern. Many objects can be presented to the system.

Second step is actual recognition which is executed constantly during robot operation. Every frame of camera is processed, image features are extracted and compared to data set in the memory. If enough features matches the pattern, then the object is recognized.

In this tutorial we will use find\_object\_2d node from find\_object\_2d package for both teaching and recognition.

As an image source we will use nodes from astra.launch as in tutorial 1.

**Teaching objects**

Anything could be an object to recognize, but remember, that the more edges and contrast colours it has, the easier it will be recognized. A piece of paper with something drawn on it would be enough for this tutorial.

First you should run find\_object\_2d and astra.launch. Node find\_object\_2d by default subscribe to image topic, you should remap it to topic /camera/rgb/image\_raw.

You can use below launch file:

<**launch**>

<**arg** name="use\_rosbot" default="false"/>

<**arg** name="use\_gazebo" default="true"/>

<**arg** name="teach" default="true"/>

<**arg** name="recognize" default="false"/>

<**arg** if="$(arg teach)" name="chosen\_world" value="rosbot\_world\_teaching"/>

<**arg** if="$(arg recognize)" name="chosen\_world" value="rosbot\_world\_recognition"/>

<**include** if="$(arg use\_rosbot)" file="$(find astra\_launch)/launch/astra.launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_gazebo)/launch/$(arg chosen\_world).launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_gazebo)/launch/rosbot.launch"/>

<**node** name="teleop\_twist\_keyboard" pkg="teleop\_twist\_keyboard" type="teleop\_twist\_keyboard.py" output="screen"/>

<**node** pkg="find\_object\_2d" type="find\_object\_2d" name="find\_object\_2d">

<**remap** from="image" to="/camera/rgb/image\_raw"/>

<**param** name="gui" value="$(arg teach)"/>

<**param** if="$(arg recognize)" name="objects\_path" value="$(find tutorial\_pkg)/image\_rec/"/>

</**node**>

</**launch**>

**Copy**

To start teaching objects on ROSbot:

roslaunch tutorial\_pkg tutorial\_4.launch use\_rosbot:=true use\_gazebo:=false teach:=true recognize:=false

**Copy**

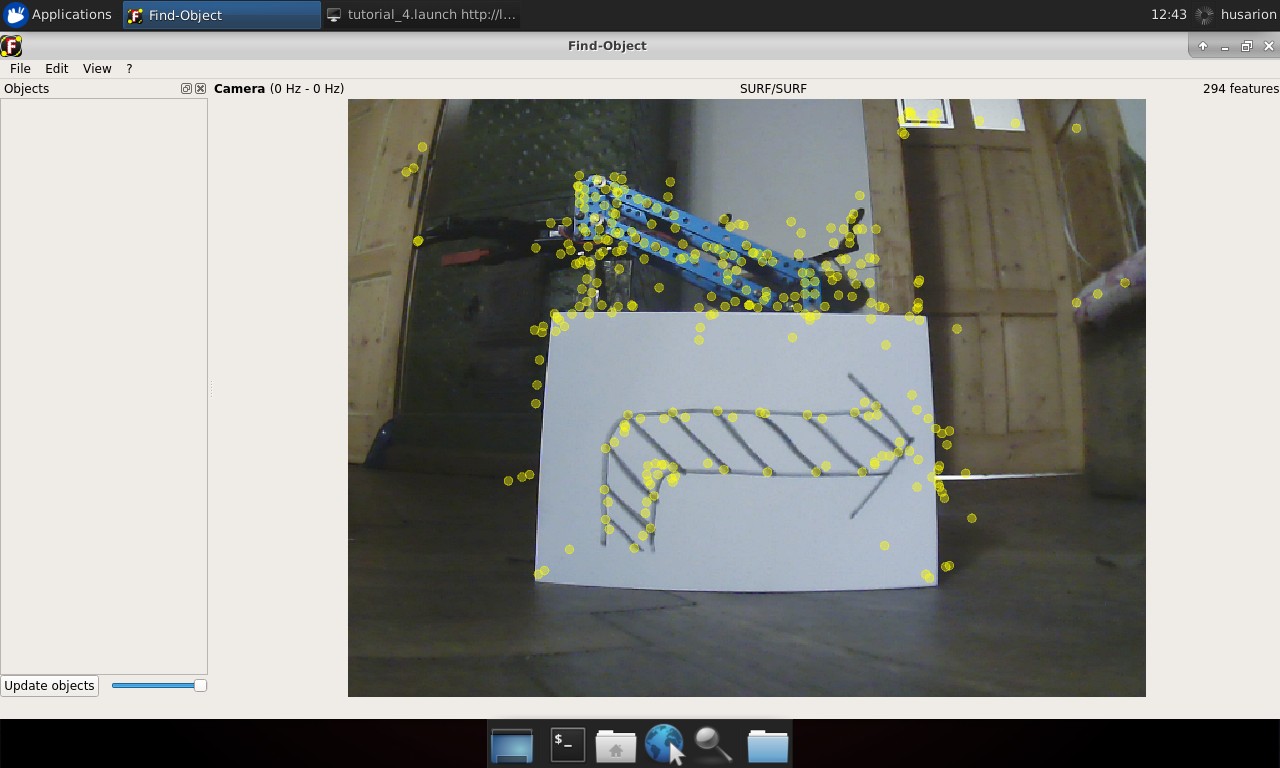
To start teaching objects in Gazebo:

roslaunch tutorial\_pkg tutorial\_4.launch use\_rosbot:=false use\_gazebo:=true teach:=true recognize:=false

**Copy**

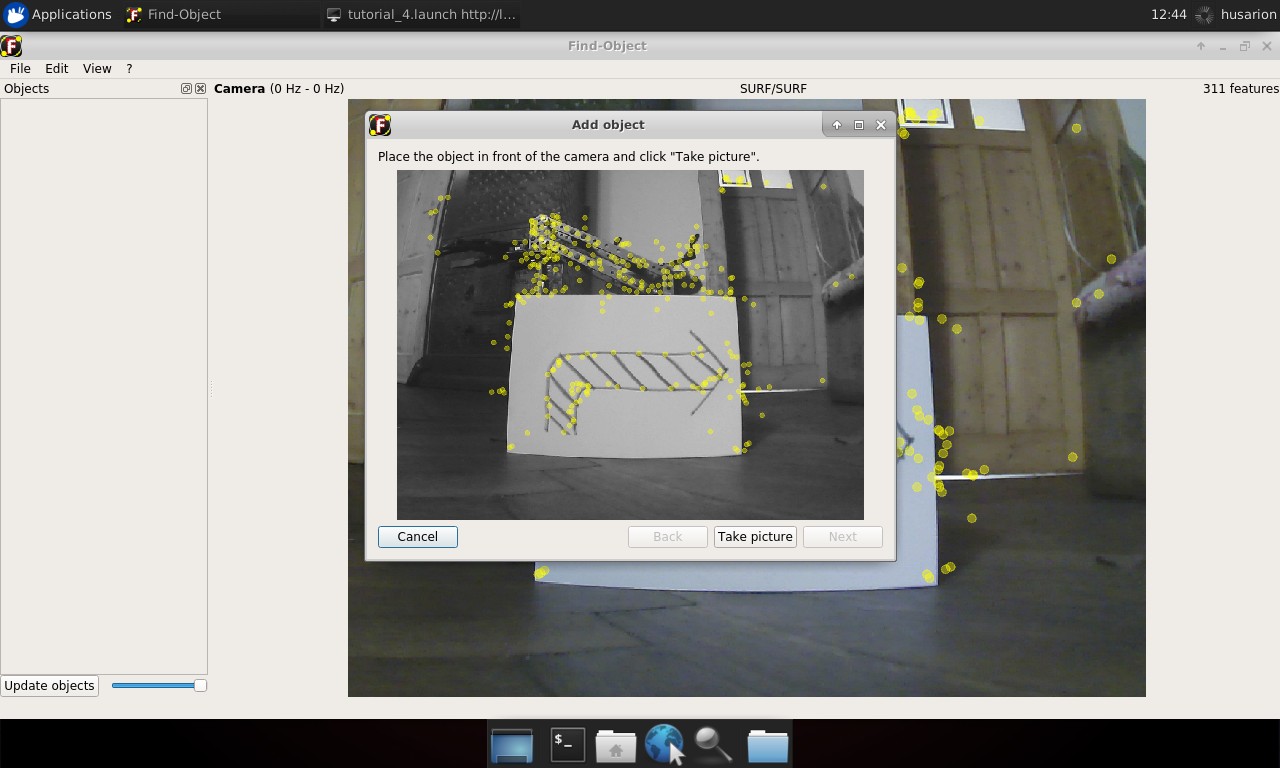
To place objects in front of camera using Gazebo, you can use buttons **Translation mode** and **Rotation mode**in top left corner and drag objects to desired position. Another option is to drive ROSbot to look at the selected object.

After launching the find\_object\_2d node with properly adjusted image topic, new window should appear:

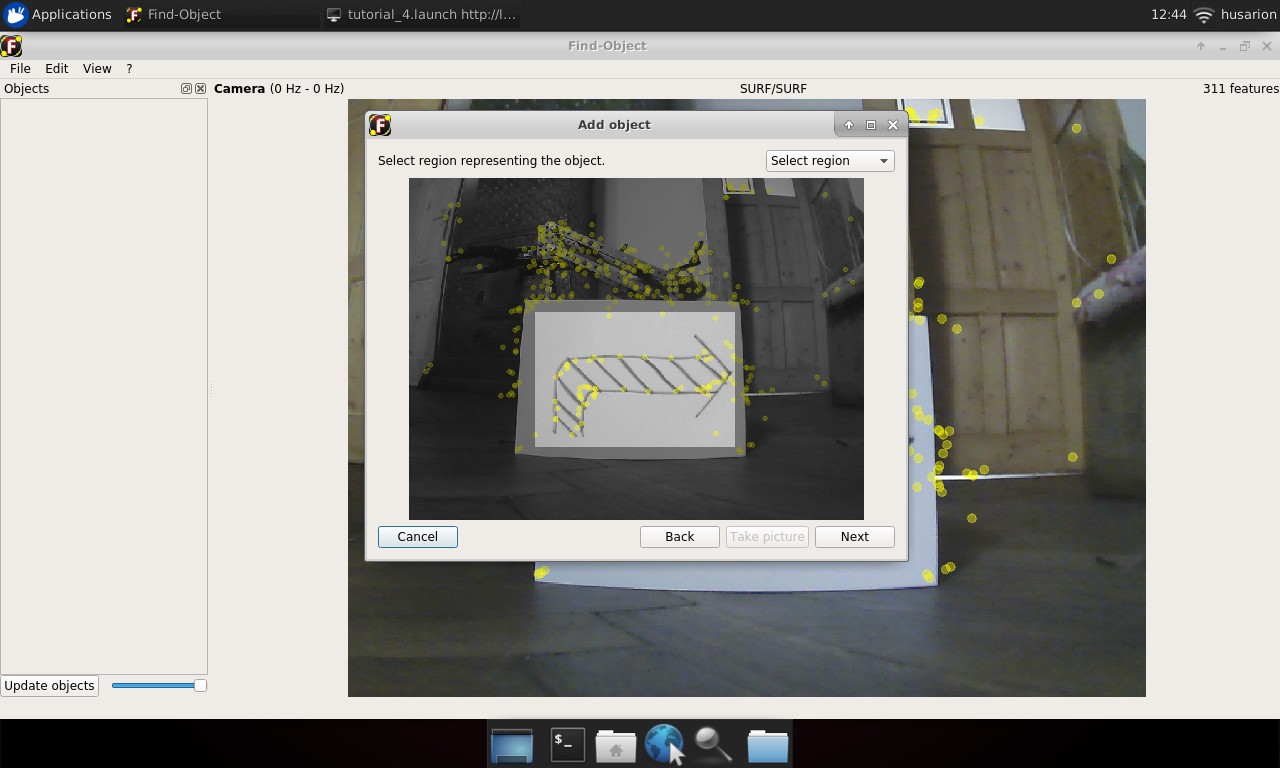


On the left side of the window there are thumbnails of saved images (should be empty at first run). Application main window contains camera view. Yellow circles on the image are marking extracted image features.

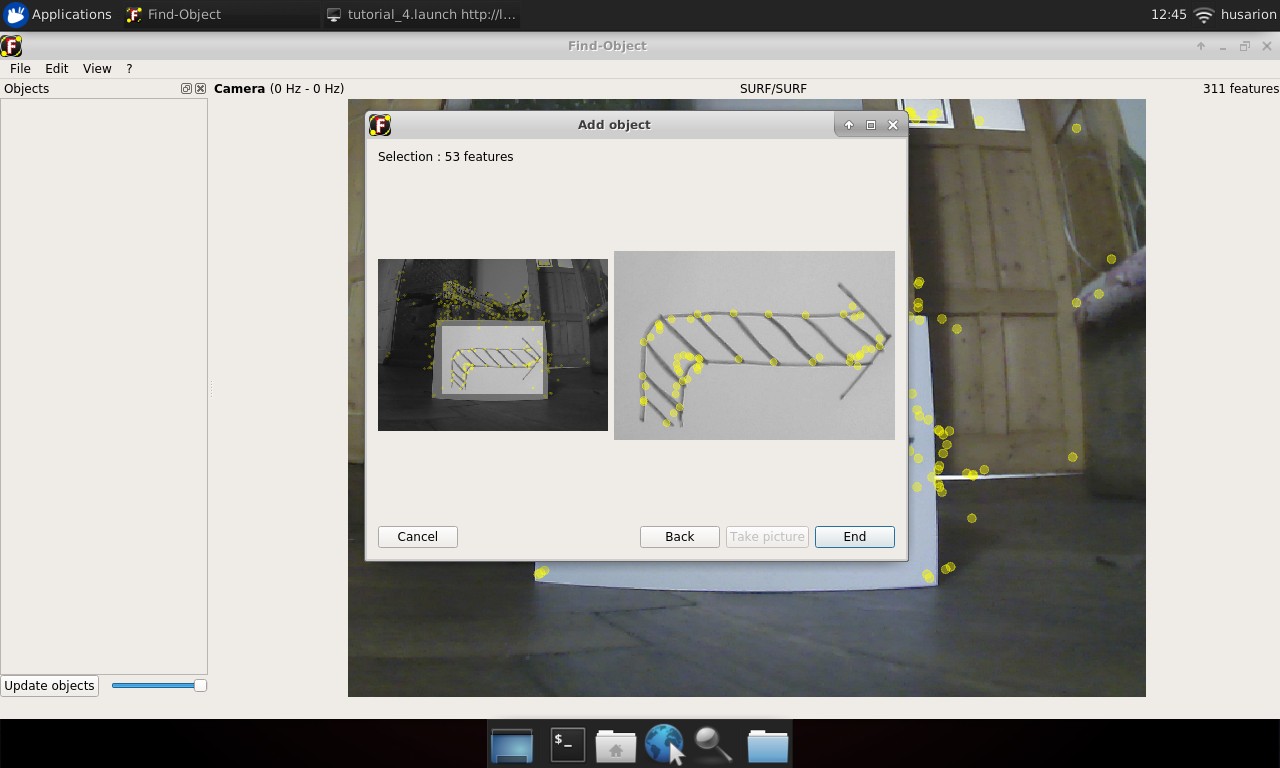
To begin teaching process choose from the main toolbar **Edit** → **Add object from scene....** New window will appear:



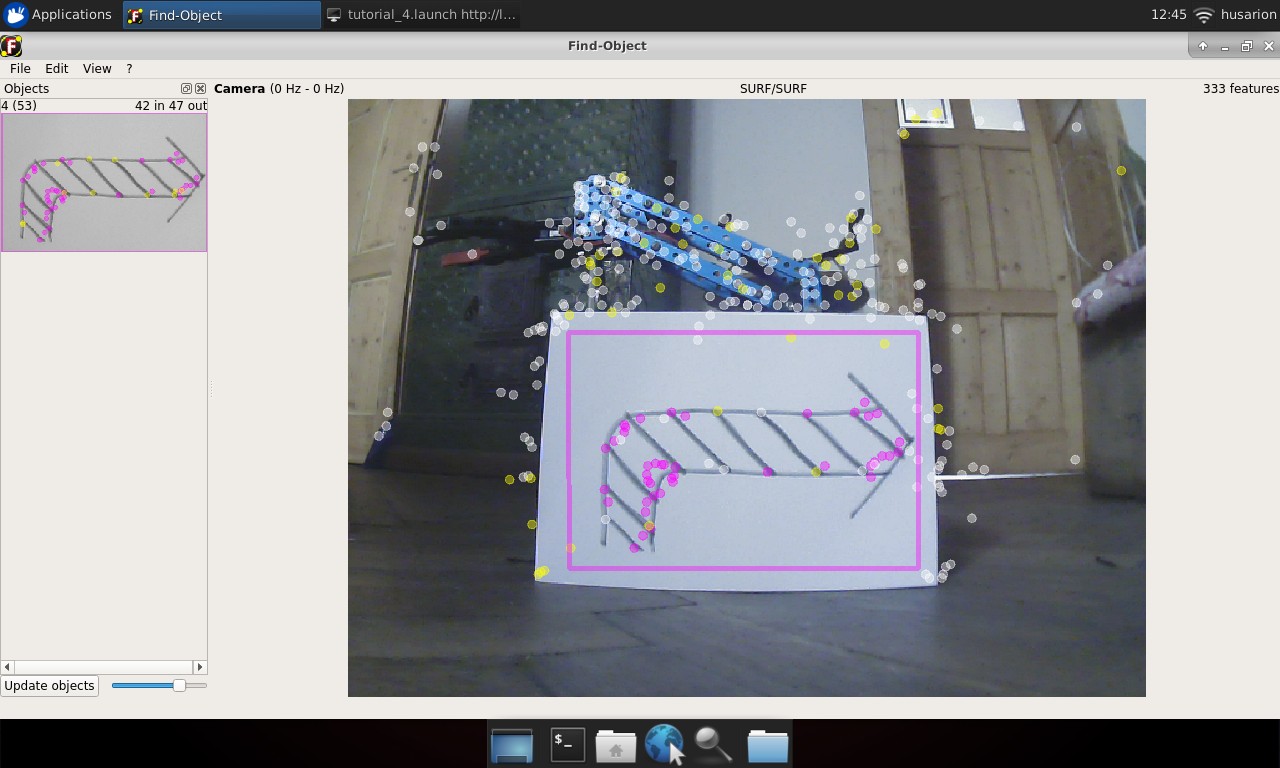
Now move the object and camera in order to cover as many features of the object as possible. While doing that try not to catch object surroundings. When it’s done, click **Take picture**.



In next view click **Select region** and select only that part of taken picture, that covers desired object and click **Next**.

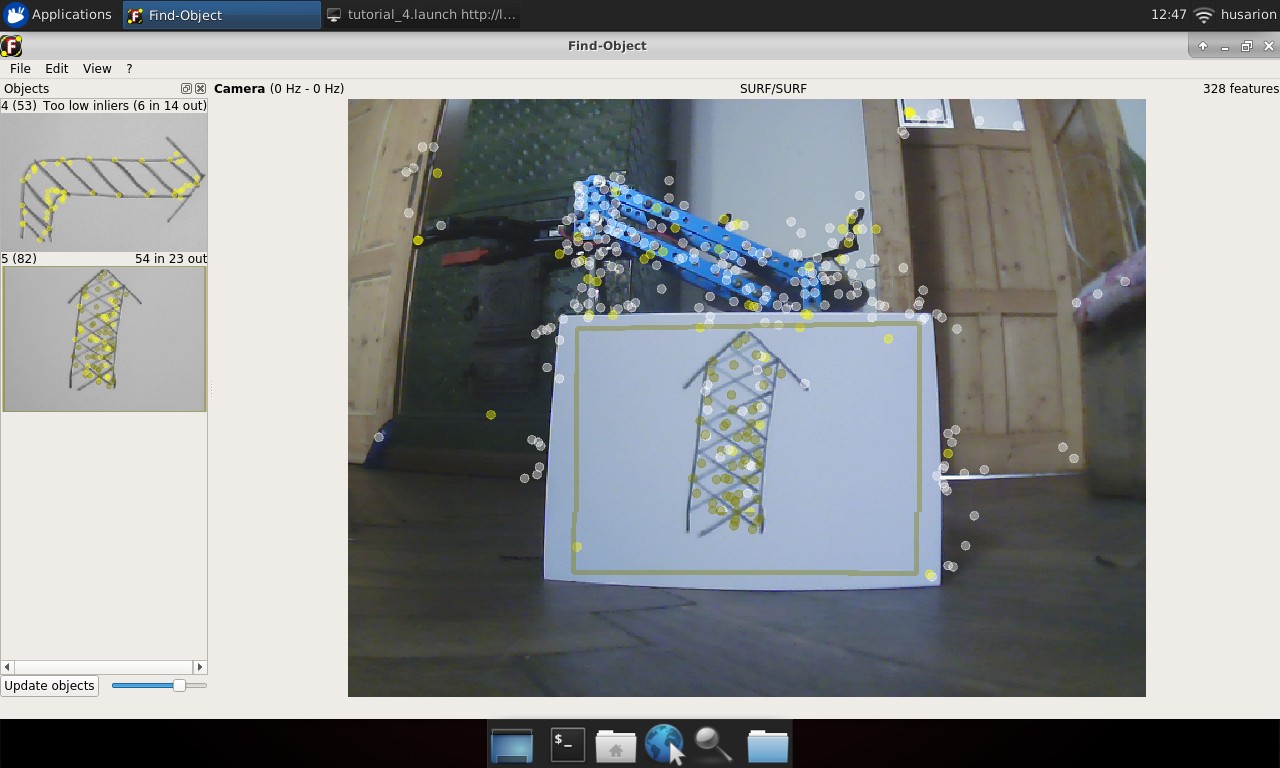


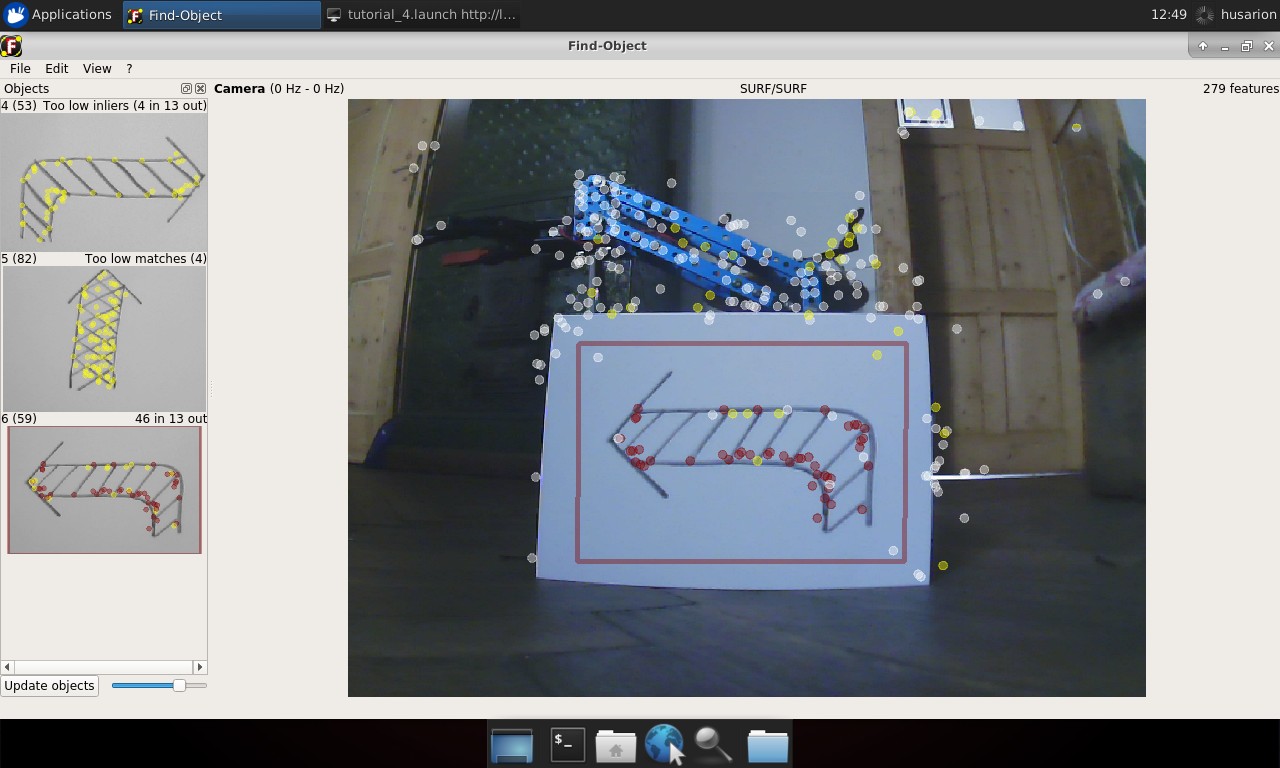
You will get confirmation of features extracted from selected image area. If presented image is in accordance with what you selected, click **End**



You should see new thumbnail in the left panel. Notice the number outside of parentheses on the left of the image, this is the object ID.

Now you can add some more objects to be recognized. Remember their IDs, you will need them later:





When you have enough objects in the database choose from the main toolbar **File** → **Save objects...** and choose a folder to store recognized objects. Close the window and stop all running nodes.

**Recognizing objects**

Objects will be recognized by the same node which was used for teaching but it works in slightly different configuration. We will set two new parameters for the node. For parameter gui we will set value false, this will run node without window for learning objects as we no longer need it. Another parameter will be objects\_path, this should be a path to a folder that you have just chosen to store recognized objects.

You can use the same launch file as for teaching, but with different parameter values.

On ROSbot:

roslaunch tutorial\_pkg tutorial\_4.launch use\_rosbot:=true use\_gazebo:=false teach:=false recognize:=true

**Copy**

In Gazebo:

roslaunch tutorial\_pkg tutorial\_4.launch use\_rosbot:=false use\_gazebo:=true teach:=false recognize:=true

**Copy**

Node is publishing to /objects topic with message type std\_msgs/Float32MultiArray. Data in this message is covering: object ID, size of recognized object and its orientation. The most interesting for us is object ID, this is first element of array.

Whenever object is recognized, it will be published in that topic. Place different objects in front of camera and observe as their data is published in the topic.

To watch messages published in the topic, you can use rostopic tool:

rostopic echo /objects

**Copy**

**Making decision with recognized object**

To perform a robot action based on recognized object, we will make a new node. It’s task will be to subscribe /objects topic and publish message to /cmd\_vel with speed and direction depending on the object.

Create a new file, name it action\_controller.cpp and place it in src folder under tutorial\_pkg. Then open it in text editor and paste below code:

#**include** <ros/ros.h>

#**include** <std\_msgs/Float32MultiArray.h>

#**include** <geometry\_msgs/Twist.h>

#**define** SMILE 4

#**define** ARROW\_LEFT 3

#**define** ARROW\_UP 5

#**define** ARROW\_DOWN 6

**int** id = 0;

ros::Publisher action\_pub;

geometry\_msgs::Twist set\_vel;

**void** **objectCallback**(**const** std\_msgs::Float32MultiArrayPtr &object)

{

**if** (object->data.size() > 0)

{

id = object->data[0];

**switch** (id)

{

**case** ARROW\_LEFT:

set\_vel.linear.x = 0;

set\_vel.angular.z = 1;

**break**;

**case** ARROW\_UP:

set\_vel.linear.x = 1;

set\_vel.angular.z = 0;

**break**;

**case** ARROW\_DOWN:

set\_vel.linear.x = -1;

set\_vel.angular.z = 0;

**break**;

**default**: // other object

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

}

action\_pub.publish(set\_vel);

}

**else**

{

// No object detected

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

action\_pub.publish(set\_vel);

}

}

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

{

ros::init(argc, argv, "action\_controller");

ros::NodeHandle **n**("~");

ros::Rate **loop\_rate**(50);

ros::Subscriber sub = n.subscribe("/objects", 1, objectCallback);

action\_pub = n.advertise<geometry\_msgs::Twist>("/cmd\_vel", 1);

set\_vel.linear.x = 0;

set\_vel.linear.y = 0;

set\_vel.linear.z = 0;

set\_vel.angular.x = 0;

set\_vel.angular.y = 0;

set\_vel.angular.z = 0;

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

Below is an explanation of the code line by line.

Including required headers:

#**include** <ros/ros.h>

#**include** <std\_msgs/Float32MultiArray.h>

#**include** <geometry\_msgs/Twist.h>

**Copy**

Defining constants for recognized objects, adjusting values to IDs of objects recognized by your system:

#**define** SMILE 8

#**define** ARROW\_LEFT 9

#**define** ARROW\_UP 10

#**define** ARROW\_DOWN 11

**Copy**

Integer for storing object identifier:

**int** id = 0;

**Copy**

Publisher for velocity commands:

ros::Publisher action\_pub;

**Copy**

Velocity command message:

geometry\_msgs::Twist set\_vel;

**Copy**

Callback function for handling incoming messages with recognized objects data:

**void** **objectCallback**(**const** std\_msgs::Float32MultiArrayPtr &object) {

**Copy**

Checking if size of data field is non zero, if it is, then object is recognized. When data field size is zero, then no object was recognized.

**if** (object->data.size() > 0) {

**Copy**

Reading id of recognized object:

id = object->data[0];

**Copy**

Depending on recognized object, setting appropriate speed values:

**switch** (id)

{

**case** ARROW\_LEFT:

set\_vel.linear.x = 0;

set\_vel.angular.z = 1;

**break**;

**case** ARROW\_UP:

set\_vel.linear.x = 1;

set\_vel.angular.z = 0;

**break**;

**case** ARROW\_DOWN:

set\_vel.linear.x = -1;

set\_vel.angular.z = 0;

**break**;

**default**: // other object

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

}

**Copy**

Publishing velocity command message:

action\_pub.publish(set\_vel);

**Copy**

Stopping all motors when no object was detected:

**else**

{

// No object detected

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

action\_pub.publish(set\_vel);

}

**Copy**

Main function, node initialization and setting main loop interval:

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

{

ros::init(argc, argv, "action\_controller");

ros::NodeHandle **n**("~");

ros::Rate **loop\_rate**(50);

**Copy**

Subscribing to /objects topic:

ros::Subscriber sub = n.subscribe("/objects", 1, objectCallback);

**Copy**

Preparing publisher for velocity commands:

action\_pub = n.advertise<geometry\_msgs::Twist>("/cmd\_vel", 1);

**Copy**

Setting zeros for initial speed values:

set\_vel.linear.x = 0;

set\_vel.linear.y = 0;

set\_vel.linear.z = 0;

set\_vel.angular.x = 0;

set\_vel.angular.y = 0;

set\_vel.angular.z = 0;

**Copy**

Main loop, waiting for trigger messages:

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

**Copy**

Last thing to do is editting the CMakeLists.txt file. Find line:

add\_executable(tutorial\_pkg\_node src/tutorial\_pkg\_node.cpp)

**Copy**

and add below code after it:

add\_executable(action\_controller\_node src/action\_controller.cpp)

**Copy**

Find also:

**target\_link\_libraries**(tutorial\_pkg\_node

${catkin\_LIBRARIES}

)

**Copy**

and add below code after it:

**target\_link\_libraries**(action\_controller\_node

${catkin\_LIBRARIES}

)

**Copy**

Now you can build your node and test it.

**Task 1** Run your node along with find\_object\_2d and astra.launch or **Gazebo** simulator. Then use rosnode, rostopic and rqt\_graph tools to examine the system. Place different objects in front of your robot one by one. Observe how it drives and turns depending on differnt objects.

You can use below launch file:

<**launch**>

<**arg** name="use\_rosbot" default="true"/>

<**arg** name="use\_gazebo" default="false"/>

<**arg** name="teach" default="false"/>

<**arg** name="recognize" default="true"/>

<**arg** if="$(arg teach)" name="chosen\_world" value="rosbot\_world\_teaching"/>

<**arg** if="$(arg recognize)" name="chosen\_world" value="rosbot\_world\_recognition"/>

<**include** if="$(arg use\_rosbot)" file="$(find astra\_launch)/launch/astra.launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_gazebo)/launch/$(arg chosen\_world).launch"/>

<**include** if="$(arg use\_gazebo)" file="$(find rosbot\_gazebo)/launch/rosbot.launch"/>

<**node** name="teleop\_twist\_keyboard" pkg="teleop\_twist\_keyboard" type="teleop\_twist\_keyboard.py" output="screen"/>

<**node** pkg="find\_object\_2d" type="find\_object\_2d" name="find\_object\_2d">

<**remap** from="image" to="/camera/rgb/image\_raw"/>

<**param** name="gui" value="$(arg teach)"/>

<**param** if="$(arg recognize)" name="objects\_path" value="$(find tutorial\_pkg)/image\_rec/"/>

</**node**>

<**node** pkg="tutorial\_pkg" type="action\_controller\_node" name="action\_controller" output="screen"/>

</**launch**>

**Copy**

**Getting position of recognized object**

Besides the ID of recognized object, find\_object\_2d node is also publishing a homography matrix of recognized object. In computer vision homography is used to define position of object relative to the camera. We will use it to obtain horizontal position of the object. Homography matrix is published in the same topic as the ID, but in the next cells of array, they are formatted as

[objectId1, objectWidth, objectHeight, h11, h12, h13, h21, h22, h23, h31, h32, h33, object2].

We will modify node to rotate robot to the direction of recognized object.

Open action\_controller.cpp file in text editor.

Begin with including of required header file:

#**include** <opencv2/opencv.hpp>

**Copy**

Variable for storing camera centre- this should be half of your camera horizontal resolution:

**int** camera\_center = 320; // left 0, right 640

**Copy**

Variables for defining rotation speed:

**float** max\_ang\_vel = 0.6;

**float** min\_ang\_vel = 0.4;

**float** ang\_vel = 0;

**Copy**

Variable for object width and height:

**float** objectWidth = object->data[1];

**float** objectHeight = object->data[2];

**Copy**

Variable for storing calculated object centre:

**float** x\_pos;

**Copy**

Variable defining how much rotation speed should increase with every pixel of object displacement:

**float** speed\_coefficient = (**float**) camera\_center / max\_ang\_vel /4;

**Copy**

Object for homography matrix:

cv::Mat **cvHomography**(3, 3, CV\_32F);

**Copy**

Vectors for storing input and output planes:

std::vector<cv::Point2f> inPts, outPts;

**Copy**

Adding new case in switch statement:

**case** SMILE:

**Copy**

Extracting coefficients homography matrix:

cvHomography.at<**float**>(0, 0) = object->data[3];

cvHomography.at<**float**>(1, 0) = object->data[4];

cvHomography.at<**float**>(2, 0) = object->data[5];

cvHomography.at<**float**>(0, 1) = object->data[6];

cvHomography.at<**float**>(1, 1) = object->data[7];

cvHomography.at<**float**>(2, 1) = object->data[8];

cvHomography.at<**float**>(0, 2) = object->data[9];

cvHomography.at<**float**>(1, 2) = object->data[10];

cvHomography.at<**float**>(2, 2) = object->data[11];

**Copy**

Defining corners of input plane:

inPts.push\_back(cv::Point2f(0, 0));

inPts.push\_back(cv::Point2f(objectWidth, 0));

inPts.push\_back(cv::Point2f(0, objectHeight));

inPts.push\_back(cv::Point2f(objectWidth, objectHeight));

**Copy**

Calculating perspective transformation:

cv::perspectiveTransform(inPts, outPts, cvHomography);

**Copy**

Calculating centre of object from its corners:

x\_pos = (**int**) (outPts.at(0).x + outPts.at(1).x + outPts.at(2).x + outPts.at(3).x) / 4;

**Copy**

Calculating angular speed value proportional to position of object and putting it into velocity message:

ang\_vel = -(x\_pos - camera\_center) / speed\_coefficient;

**if** (ang\_vel >= -(min\_ang\_vel / 2) && ang\_vel <= (min\_ang\_vel / 2))

{

set\_vel.angular.z = 0;

}

**else** **if** (ang\_vel >= max\_ang\_vel)

{

set\_vel.angular.z = max\_ang\_vel;

}

**else** **if** (ang\_vel <= -max\_ang\_vel)

{

set\_vel.angular.z = -max\_ang\_vel;

}

**else**

{

set\_vel.angular.z = ang\_vel;

}

**Copy**

Your final file should look like this:

#**include** <ros/ros.h>

#**include** <std\_msgs/Float32MultiArray.h>

#**include** <geometry\_msgs/Twist.h>

#**include** <std\_msgs/String.h>

#**include** <opencv2/opencv.hpp>

#**define** SMILE 4

#**define** ARROW\_LEFT 3

#**define** ARROW\_UP 5

#**define** ARROW\_DOWN 6

**int** id = 0;

ros::Publisher action\_pub;

geometry\_msgs::Twist set\_vel;

**int** camera\_center = 320; // left 0, right 640

**float** max\_ang\_vel = 0.6;

**float** min\_ang\_vel = 0.4;

**float** ang\_vel = 0;

**void** **objectCallback**(**const** std\_msgs::Float32MultiArrayPtr &object)

{

**if** (object->data.size() > 0)

{

id = object->data[0];

**float** objectWidth = object->data[1];

**float** objectHeight = object->data[2];

**float** x\_pos;

**float** speed\_coefficient = (**float**)camera\_center / max\_ang\_vel / 4;

// Find corners OpenCV

cv::Mat **cvHomography**(3, 3, CV\_32F);

std::vector<cv::Point2f> inPts, outPts;

**switch** (id)

{

**case** ARROW\_LEFT:

set\_vel.linear.x = 0;

set\_vel.angular.z = 1.0;

**break**;

**case** ARROW\_UP:

set\_vel.linear.x = 1;

set\_vel.angular.z = 0;

**break**;

**case** ARROW\_DOWN:

set\_vel.linear.x = -1;

set\_vel.angular.z = 0;

**break**;

**case** SMILE:

cvHomography.at<**float**>(0, 0) = object->data[3];

cvHomography.at<**float**>(1, 0) = object->data[4];

cvHomography.at<**float**>(2, 0) = object->data[5];

cvHomography.at<**float**>(0, 1) = object->data[6];

cvHomography.at<**float**>(1, 1) = object->data[7];

cvHomography.at<**float**>(2, 1) = object->data[8];

cvHomography.at<**float**>(0, 2) = object->data[9];

cvHomography.at<**float**>(1, 2) = object->data[10];

cvHomography.at<**float**>(2, 2) = object->data[11];

inPts.push\_back(cv::Point2f(0, 0));

inPts.push\_back(cv::Point2f(objectWidth, 0));

inPts.push\_back(cv::Point2f(0, objectHeight));

inPts.push\_back(cv::Point2f(objectWidth, objectHeight));

cv::perspectiveTransform(inPts, outPts, cvHomography);

x\_pos = (**int**)(outPts.at(0).x + outPts.at(1).x + outPts.at(2).x +

outPts.at(3).x) /

4;

ang\_vel = -(x\_pos - camera\_center) / speed\_coefficient;

**if** (ang\_vel >= -(min\_ang\_vel / 2) && ang\_vel <= (min\_ang\_vel / 2))

{

set\_vel.angular.z = 0;

}

**else** **if** (ang\_vel >= max\_ang\_vel)

{

set\_vel.angular.z = max\_ang\_vel;

}

**else** **if** (ang\_vel <= -max\_ang\_vel)

{

set\_vel.angular.z = -max\_ang\_vel;

}

**else**

{

set\_vel.angular.z = ang\_vel;

}

**break**;

**default**: // other object

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

}

action\_pub.publish(set\_vel);

}

**else**

{

// No object detected

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

action\_pub.publish(set\_vel);

}

}

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

{

std\_msgs::String s;

std::string str;

str.clear();

str.append("");

std::to\_string(3);

s.data = str;

ros::init(argc, argv, "action\_controller");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/objects", 1, objectCallback);

ros::Rate **loop\_rate**(50);

action\_pub = n.advertise<geometry\_msgs::Twist>("/cmd\_vel", 1);

set\_vel.linear.x = 0;

set\_vel.linear.y = 0;

set\_vel.linear.z = 0;

set\_vel.angular.x = 0;

set\_vel.angular.y = 0;

set\_vel.angular.z = 0;

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

Last thing to do is to edit the CMakeLists.txt file. Find line:

find\_package(catkin REQUIRED COMPONENTS roscpp )

**Copy**

and add below code after it:

**find\_package**( OpenCV REQUIRED )

**Copy**

Find also:

**include\_directories**(

${catkin\_INCLUDE\_DIRS}

)

**Copy**

and change it to:

**include\_directories**(

${catkin\_INCLUDE\_DIRS}

${OpenCV\_INCLUDE\_DIRS}

)

**Copy**

Find:

**target\_link\_libraries**(action\_controller\_node

${catkin\_LIBRARIES}

)

**Copy**

and change it to:

**target\_link\_libraries**(action\_controller\_node

${catkin\_LIBRARIES}

${OpenCV\_LIBRARIES}

)

**Copy**

Now you can build your node and test it.

**Task 2** Run your node along with find\_object\_2d and astra.launch or **Gazebo** simulator. Place the object with ID bonded to new case in switch statement in front of your robot. Observe how it turns towards the object.

**Following the object**

In this section you will modify your robot to turn and also drive towards the object while keeping distance to it. For keeping the distance we will use one of two different types of sensors.

Log in to Husarion Cloud and open project that you created in previous manual. You will need to edit it a little.

Include required header files:

**void** **initDistanceSensorsPublisher**()

{

range\_fl.header.frame\_id = "range\_fl";

range\_fr.header.frame\_id = "range\_fr";

range\_rl.header.frame\_id = "range\_rl";

range\_rr.header.frame\_id = "range\_rr";

**switch** (sensor\_type)

{

**case** SENSOR\_LASER:

range\_fl.field\_of\_view = 0.26;

range\_fl.min\_range = 0.03;

range\_fl.max\_range = 0.90;

range\_fr.field\_of\_view = 0.26;

range\_fr.min\_range = 0.03;

range\_fr.max\_range = 0.90;

range\_rl.field\_of\_view = 0.26;

range\_rl.min\_range = 0.03;

range\_rl.max\_range = 0.90;

range\_rr.field\_of\_view = 0.26;

range\_rr.min\_range = 0.03;

range\_rr.max\_range = 0.90;

**break**;

**case** SENSOR\_INFRARED:

range\_fl.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_fl.field\_of\_view = 0.26;

range\_fl.min\_range = 0.05;

range\_fl.max\_range = 0.299;

range\_fr.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_fr.field\_of\_view = 0.26;

range\_fr.min\_range = 0.05;

range\_fr.max\_range = 0.299;

range\_rl.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_rl.field\_of\_view = 0.26;

range\_rl.min\_range = 0.05;

range\_rl.max\_range = 0.299;

range\_rr.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_rr.field\_of\_view = 0.26;

range\_rr.min\_range = 0.05;

range\_rr.max\_range = 0.299;

**break**;

**case** NO\_DISTANCE\_SENSOR:

// Do your own implementation

**break**;

}

**if** (sensor\_type != SensorType::NO\_DISTANCE\_SENSOR)

{

range\_pub\_fl = **new** ros::Publisher("/range/fl", &range\_fl);

range\_pub\_fr = **new** ros::Publisher("/range/fr", &range\_fr);

range\_pub\_rl = **new** ros::Publisher("/range/rl", &range\_rl);

range\_pub\_rr = **new** ros::Publisher("/range/rr", &range\_rr);

nh.advertise(\*range\_pub\_fl);

nh.advertise(\*range\_pub\_fr);

nh.advertise(\*range\_pub\_rl);

nh.advertise(\*range\_pub\_rr);

}

}

**Copy**

Assign values to variables, put into messages and publish them:

**if** (sensor\_type != SensorType::NO\_DISTANCE\_SENSOR)

{

// get ranges from distance sensors

ranges = rosbot.getRanges(sensor\_type);

range\_fl.range = ranges[0];

range\_fr.range = ranges[1];

range\_rl.range = ranges[2];

range\_rr.range = ranges[3];

Serial.printf("Ranges %f %f %f %f\n", ranges[0], ranges[1], ranges[2], ranges[3]);

// publish ranges

range\_pub\_fl->publish(&range\_fl);

range\_pub\_fr->publish(&range\_fr);

range\_pub\_rl->publish(&range\_rl);

range\_pub\_rr->publish(&range\_rr);

}

**Copy**

Whole file should look like this:

#**include** "hFramework.h"

#**include** "hCloudClient.h"

#**include** "ros.h"

#**include** "geometry\_msgs/Twist.h"

#**include** "geometry\_msgs/PoseStamped.h"

#**include** "sensor\_msgs/BatteryState.h"

#**include** "std\_msgs/Bool.h"

#**include** "sensor\_msgs/Range.h"

#**include** "tf/tf.h"

#**include** "ROSbot.h"

**using** **namespace** hFramework;

// Uncomment one of these lines, accordingly to range sensor type of your ROSbot

// If you have version with infrared sensor:

// static const SensorType sensor\_type = SENSOR\_INFRARED;

// If you have version with laser sensor:

**static** **const** SensorType sensor\_type = SENSOR\_LASER;

// If you want to use your own sensor:

// static const SensorType sensor\_type = NO\_DISTANCE\_SENSOR;

// Uncomment one of these lines, accordingly to IMU sensor type of your device

// If you have version with MPU9250:

**static** **const** ImuType imu\_type = MPU9250;

// If you want to use your own sensor:

// static const ImuType imu\_type = NO\_IMU;

ros::NodeHandle nh;

sensor\_msgs::BatteryState battery;

ros::Publisher \*battery\_pub;

geometry\_msgs::PoseStamped pose;

ros::Publisher \*pose\_pub;

sensor\_msgs::Range range\_fl;

sensor\_msgs::Range range\_fr;

sensor\_msgs::Range range\_rl;

sensor\_msgs::Range range\_rr;

ros::Publisher \*range\_pub\_fl;

ros::Publisher \*range\_pub\_fr;

ros::Publisher \*range\_pub\_rl;

ros::Publisher \*range\_pub\_rr;

std::vector<**float**> rosbot\_pose;

std::vector<**float**> rpy;

std::vector<**float**> ranges;

**int** publish\_counter = 0;

**void** **twistCallback**(**const** geometry\_msgs::Twist &twist)

{

rosbot.setSpeed(twist.linear.x, twist.angular.z);

}

**void** **initCmdVelSubscriber**()

{

ros::Subscriber<geometry\_msgs::Twist> \*cmd\_sub = **new** ros::Subscriber<geometry\_msgs::Twist>("/cmd\_vel", &twistCallback);

nh.subscribe(\*cmd\_sub);

}

**void** **resetCallback**(**const** std\_msgs::Bool &msg)

{

**if** (msg.data == true)

{

rosbot.reset\_odometry();

}

}

**void** **initResetOdomSubscriber**()

{

ros::Subscriber<std\_msgs::Bool> \*odom\_reset\_sub = **new** ros::Subscriber<std\_msgs::Bool>("/reset\_odom", &resetCallback);

nh.subscribe(\*odom\_reset\_sub);

}

**void** **initDistanceSensorsPublisher**()

{

range\_fl.header.frame\_id = "range\_fl";

range\_fr.header.frame\_id = "range\_fr";

range\_rl.header.frame\_id = "range\_rl";

range\_rr.header.frame\_id = "range\_rr";

**switch** (sensor\_type)

{

**case** SENSOR\_LASER:

range\_fl.field\_of\_view = 0.26;

range\_fl.min\_range = 0.03;

range\_fl.max\_range = 0.90;

range\_fr.field\_of\_view = 0.26;

range\_fr.min\_range = 0.03;

range\_fr.max\_range = 0.90;

range\_rl.field\_of\_view = 0.26;

range\_rl.min\_range = 0.03;

range\_rl.max\_range = 0.90;

range\_rr.field\_of\_view = 0.26;

range\_rr.min\_range = 0.03;

range\_rr.max\_range = 0.90;

**break**;

**case** SENSOR\_INFRARED:

range\_fl.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_fl.field\_of\_view = 0.26;

range\_fl.min\_range = 0.05;

range\_fl.max\_range = 0.299;

range\_fr.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_fr.field\_of\_view = 0.26;

range\_fr.min\_range = 0.05;

range\_fr.max\_range = 0.299;

range\_rl.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_rl.field\_of\_view = 0.26;

range\_rl.min\_range = 0.05;

range\_rl.max\_range = 0.299;

range\_rr.radiation\_type = sensor\_msgs::Range::INFRARED;

range\_rr.field\_of\_view = 0.26;

range\_rr.min\_range = 0.05;

range\_rr.max\_range = 0.299;

**break**;

**case** NO\_DISTANCE\_SENSOR:

// Do your own implementation

**break**;

}

**if** (sensor\_type != SensorType::NO\_DISTANCE\_SENSOR)

{

range\_pub\_fl = **new** ros::Publisher("/range/fl", &range\_fl);

range\_pub\_fr = **new** ros::Publisher("/range/fr", &range\_fr);

range\_pub\_rl = **new** ros::Publisher("/range/rl", &range\_rl);

range\_pub\_rr = **new** ros::Publisher("/range/rr", &range\_rr);

nh.advertise(\*range\_pub\_fl);

nh.advertise(\*range\_pub\_fr);

nh.advertise(\*range\_pub\_rl);

nh.advertise(\*range\_pub\_rr);

}

}

**void** **initBatteryPublisher**()

{

battery\_pub = **new** ros::Publisher("/battery", &battery);

nh.advertise(\*battery\_pub);

}

**void** **initPosePublisher**()

{

pose.header.frame\_id = "base\_link";

pose.pose.orientation = tf::createQuaternionFromYaw(0);

pose\_pub = **new** ros::Publisher("/pose", &pose);

nh.advertise(\*pose\_pub);

}

**void** **hMain**()

{

Serial.printf("init ROSbot\n");

rosbot.initROSbot(sensor\_type);

Serial.printf("init with dvice\n");

platform.begin(&RPi);

nh.getHardware()->initWithDevice(&platform.LocalSerial);

nh.initNode();

initBatteryPublisher();

initPosePublisher();

initDistanceSensorsPublisher();

initCmdVelSubscriber();

initResetOdomSubscriber();

**while** (true)

{

nh.spinOnce();

publish\_counter++;

**if** (publish\_counter > 10)

{

// get ROSbot pose

rosbot\_pose = rosbot.getPose();

pose.pose.position.x = rosbot\_pose[0];

pose.pose.position.y = rosbot\_pose[1];

pose.pose.orientation = tf::createQuaternionFromYaw(rosbot\_pose[2]);

// publish pose

pose\_pub->publish(&pose);

**if** (sensor\_type != SensorType::NO\_DISTANCE\_SENSOR)

{

// get ranges from distance sensors

ranges = rosbot.getRanges(sensor\_type);

range\_fl.range = ranges[0];

range\_fr.range = ranges[1];

range\_rl.range = ranges[2];

range\_rr.range = ranges[3];

Serial.printf("Ranges %f %f %f %f\n", ranges[0], ranges[1], ranges[2], ranges[3]);

// publish ranges

range\_pub\_fl->publish(&range\_fl);

range\_pub\_fr->publish(&range\_fr);

range\_pub\_rl->publish(&range\_rl);

range\_pub\_rr->publish(&range\_rr);

}

// get battery voltage

battery.voltage = rosbot.getBatteryLevel();

// publish battery voltage

battery\_pub->publish(&battery);

publish\_counter = 0;

}

sys.delay(10);

}

}

**Copy**

Build the project and upload it to your device. Then open action\_controller.cpp file in text editor.

Begin with including required header file:

#**include** <sensor\_msgs/Range.h>

**Copy**

Add variables for measured object distance, average distance and desired distance to obstacle:

**float** distFL = 0;

**float** distFR = 0;

**float** average\_dist = 0;

**float** desired\_dist = 0.2;

**Copy**

Callback functions for incoming sensor messages, their task is only to put values into appropriate variables:

**void** **distFL\_callback**(**const** sensor\_msgs::Range &range)

{

distFL = range.range;

}

**void** **distFR\_callback**(**const** sensor\_msgs::Range &range)

{

distFR = range.range;

}

**Copy**

Then, in switch statement, calculate average distance and set velocity proportional to it only if both sensors found an obstacle, else set zero value for linear velocity:

**if** (distFL > 0 && distFR > 0)

{

average\_dist = (distFL + distFR) / 2;

set\_vel.linear.x = (average\_dist - desired\_dist) / 4;

}

**else**

{

set\_vel.linear.x = 0;

}

**Copy**

In main function, subscribe to sensor topics:

ros::Subscriber distFL\_sub = n.subscribe("/range/fl", 1, distFL\_callback);

ros::Subscriber distFR\_sub = n.subscribe("/range/fr", 1, distFR\_callback);

**Copy**

Final file should look like this:

#**include** <ros/ros.h>

#**include** <std\_msgs/Float32MultiArray.h>

#**include** <std\_msgs/Int32MultiArray.h>

#**include** <geometry\_msgs/Twist.h>

#**include** <std\_msgs/String.h>

#**include** <opencv2/opencv.hpp>

#**include** <sensor\_msgs/Range.h>

#**define** SMILE 4

#**define** ARROW\_LEFT 3

#**define** ARROW\_UP 5

#**define** ARROW\_DOWN 6

**int** id = 0;

ros::Publisher action\_pub;

geometry\_msgs::Twist set\_vel;

**int** camera\_center = 320; // left 0, right 640

**float** max\_ang\_vel = 0.6;

**float** min\_ang\_vel = 0.4;

**float** ang\_vel = 0;

**float** distFL = 0;

**float** distFR = 0;

**float** average\_dist = 0;

**float** desired\_dist = 0.2;

**void** **distFL\_callback**(**const** sensor\_msgs::Range &range)

{

distFL = range.range;

}

**void** **distFR\_callback**(**const** sensor\_msgs::Range &range)

{

distFR = range.range;

}

**void** **objectCallback**(**const** std\_msgs::Float32MultiArrayPtr &object)

{

**if** (object->data.size() > 0)

{

id = object->data[0];

**float** objectWidth = object->data[1];

**float** objectHeight = object->data[2];

**float** x\_pos;

**float** speed\_coefficient = (**float**)camera\_center / max\_ang\_vel;

// Find corners OpenCV

cv::Mat **cvHomography**(3, 3, CV\_32F);

std::vector<cv::Point2f> inPts, outPts;

**switch** (id)

{

**case** ARROW\_LEFT:

set\_vel.linear.x = 0;

set\_vel.angular.z = 1;

**break**;

**case** ARROW\_UP:

set\_vel.linear.x = 1;

set\_vel.angular.z = 0;

**break**;

**case** ARROW\_DOWN:

set\_vel.linear.x = -1;

set\_vel.angular.z = 0;

**break**;

**case** SMILE:

cvHomography.at<**float**>(0, 0) = object->data[3];

cvHomography.at<**float**>(1, 0) = object->data[4];

cvHomography.at<**float**>(2, 0) = object->data[5];

cvHomography.at<**float**>(0, 1) = object->data[6];

cvHomography.at<**float**>(1, 1) = object->data[7];

cvHomography.at<**float**>(2, 1) = object->data[8];

cvHomography.at<**float**>(0, 2) = object->data[9];

cvHomography.at<**float**>(1, 2) = object->data[10];

cvHomography.at<**float**>(2, 2) = object->data[11];

inPts.push\_back(cv::Point2f(0, 0));

inPts.push\_back(cv::Point2f(objectWidth, 0));

inPts.push\_back(cv::Point2f(0, objectHeight));

inPts.push\_back(cv::Point2f(objectWidth, objectHeight));

cv::perspectiveTransform(inPts, outPts, cvHomography);

x\_pos = (**int**)(outPts.at(0).x + outPts.at(1).x + outPts.at(2).x +

outPts.at(3).x) /

4;

ang\_vel = -(x\_pos - camera\_center) / speed\_coefficient;

**if** (ang\_vel >= -(min\_ang\_vel / 2) && ang\_vel <= (min\_ang\_vel / 2))

{

set\_vel.angular.z = 0;

**if** (distFL > 0 && distFR > 0)

{

average\_dist = (distFL + distFR) / 2;

set\_vel.linear.x = (average\_dist - desired\_dist) / 4;

}

**else**

{

set\_vel.linear.x = 0;

}

}

**else** **if** (ang\_vel >= max\_ang\_vel)

{

set\_vel.angular.z = max\_ang\_vel;

}

**else** **if** (ang\_vel <= -max\_ang\_vel)

{

set\_vel.angular.z = -max\_ang\_vel;

}

**else**

{

set\_vel.angular.z = ang\_vel;

}

**break**;

**default**: // other object

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

}

action\_pub.publish(set\_vel);

}

**else**

{

// No object detected

set\_vel.linear.x = 0;

set\_vel.angular.z = 0;

action\_pub.publish(set\_vel);

}

}

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

{

std\_msgs::String s;

std::string str;

str.clear();

str.append("");

std::to\_string(3);

s.data = str;

ros::init(argc, argv, "action\_controller");

ros::NodeHandle **n**("~");

ros::Subscriber sub = n.subscribe("/objects", 1, objectCallback);

ros::Subscriber distL\_sub = n.subscribe("/range/fl", 1, distFL\_callback);

ros::Subscriber distR\_sub = n.subscribe("/range/fr", 1, distFR\_callback);

ros::Rate **loop\_rate**(50);

action\_pub = n.advertise<geometry\_msgs::Twist>("/cmd\_vel", 1);

set\_vel.linear.x = 0;

set\_vel.linear.y = 0;

set\_vel.linear.z = 0;

set\_vel.angular.x = 0;

set\_vel.angular.y = 0;

set\_vel.angular.z = 0;

**while** (ros::ok())

{

ros::spinOnce();

loop\_rate.sleep();

}

}

**Copy**

Now you can build your node and test it.

**Task 3** Run your node along with find\_object\_2d and astra.launch or **Gazebo** simulator. Place the same object as in Task 2 in front of your robot. Observe how it turns and drives towards the object.

**Summary**

After completing this tutorial you should be able to configure your CORE2 device with vision system to recognize objects. You should also be able to determine the position of recognized object relative to camera and create a node that perform specific action related to recognized objects. You should also know how to handle proximity sensors with sensor\_msgs/Range message type.

*by Łukasz Mitka, Husarion*

*Do you need any support with completing this tutorial or have any difficulties with software or hardware? Feel free to describe your thoughts on our community forum:*[*https://community.husarion.com/*](https://community.husarion.com/)*or to contact with our support:*[*support@husarion.com*](mailto:support@husarion.com)

[← 3. SIMPLE KINEMATICS FOR MOBILE ROBOT](https://husarion.com/tutorials/ros-tutorials/3-simple-kinematics-for-mobile-robot)[5. RUNNING ROS ON MULTIPLE MACHINES →](https://husarion.com/tutorials/ros-tutorials/5-running-ros-on-multiple-machines)