Rodney - A long time coming autonomous robot (Part 2)

Second part in a series on a ROS (Robot Operating System) House Bot.

## **Introduction**

The Rodney Robot project is a hobbyist robotic project to design and build an autonomous house-bot. This article is the second in the series describing the project.

## **Background**

In [part one](https://www.codeproject.com/Articles/1249436/Rodney-A-long-time-coming-autonomous-robot-Part-1) to help define the requirements for our robot we selected our first mission and split it down into a number of Design Goals to make it more manageable.

The mission was taken from the article [Let's build a robot!](https://www.codeproject.com/Articles/1115414/Lets-build-a-robot) and was: Take a message to... - Since the robot will [have] the ability to recognize family members, how about the ability to make it the 'message taker and reminder'. I could say 'Robot, remind (PersonName) to pick me up from the station at 6pm'. Then, even if that household member had their phone turned on silent, or were listening to loud music or (insert reason to NOT pick me up at the station), the robot could wander through the house, find the person, and give them the message.

The design goals for this mission were:

* Design Goal 1: To be able to look around using the camera, search for faces, attempt to identify any people seen and display a message for any identified
* Design Goal 2: Facial expressions and speech synthesis. Rodney will need to be able to deliver the message
* Design Goal 3: Locomotion controlled by a remote keyboard and/or joystick
* Design Goal 4: Addition of a laser ranger finder or similar ranging sensor used to aid navigation
* Design Goal 5: Autonomous locomotion
* Design Goal 6: Task assignment and completion notification

In the first part we used ROS (Robot Operating System) to add pan/tilt functionality to move the head and camera. In this part I'll add the face recognition and control node to complete Design Goal 1.

## **Mission 1, Design Goal 1 continued**

### **Access images from the Raspberry Pi Camera**

As stated in part 1 we can make use of work already carried out by the ROS community to make our life easier. The Raspberry Pi Ubuntu image I have installed includes a ROS package [raspicam\_node](https://github.com/ubiquityRobotics/raspicam_node), we will make use of this package to access the camera. If you are using a different OS image on your Raspberry Pi you can still install the node from the GitHub site.

To add the node to our system we just need to include one of the supplied ROS launch files in our launch file. I'm going to use an image resolution of 1280 x 960 pixels, so adding the following to our launch file will start the raspicam\_node in the required resolution.

Hide   Copy Code

<include file="$(find raspicam\_node)/launch/camerav2\_1280x960.launch" />

ROS uses its own image format to pass images between nodes. In the case of raspicam\_node this image is in a compressed format. Now when we come to use the image to detect and recognise a face we will be using OpenCV to process the image. We therefore need a method to convert ROS images to OpenCV images and back again. Not a problem, those nice people in the ROS community have produced a package called [cv\_bridge](http://wiki.ros.org/cv_bridge) which will do the work for us.

I will write the face recognition node in Python since 1) the pan\_tilt node was written in C++ and writing the next node in Python will give us examples in both languages and 2) I already have a Python face recognition library I would like to make use of.

Now I'm using the Raspberry Pi camera but you can use a different camera. You may find a ROS node is already available for your camera or you may need to wrap the camera library in a ROS node of your own. Although the topic name our face recognition node subscribes to is called /raspicam/image/compressed, you can always remap your topic name to match this name in the launch file if you don't want to change the code.

Hide   Copy Code

<remap from="my\_camera\_node/image/compressed" to="raspicam\_node/image compressed" />

### **Detect and recognise faces**

Before the system can attempt to recognise a face we need to train it with the subjects we wish to recognise. We will write two non ROS Python scripts to train our system. The first, data\_set\_generator.py, will use the camera to capture facial images of each of our subjects. The second, training.py, uses the images collected by the first script to do the actual training. The output of this second script is a yaml file containing the training data, the ROS node will load this training data during its initialisation. If you wish to add new people to your list of subjects you will need to rerun each script.

Our ROS package for the node is called face\_recognition and is available in the face\_recognition folder. The sub folder scripts contains our two training scripts.

Each of the scripts makes use of face detection and face recognition built in to OpenCV. If you wish to fully understand how this works then may I suggest you read some of the many articles available on the internet. One of these can be found [here](https://github.com/informramiz/opencv-face-recognition-python/blob/master/README.md). Not one article I came across had the code for exactly what I wanted so I have borrowed code from a number of articles. Here I'll give a high level description of each of the scripts starting with data\_set\_generator.py.

After the required imports we load the classifier using the OpenCV library, declare a helper function which we use to ensure that folders we require exist. The folder dataset will hold all the captured images and trainer will hold both the yaml file with the training data and a file containing names and ids of our subjects.

Hide   Copy Code

import cv2

import os

import io

import numpy

import yaml

import picamera

*# Detect object in video stream using Haarcascade Frontal Face*

face\_detector = cv2.CascadeClassifier('../classifiers/haarcascade\_frontalface\_default.xml')

def assure\_path\_exists(path):

dir = os.path.dirname(path)

if not os.path.exists(dir):

os.makedirs(dir)

assure\_path\_exists("dataset/")

assure\_path\_exists("../trainer/")

Next we set the camera resolution, set up some variables including the file name which holds our list of subjects and open the file.

We then create a window to display the image read from the camera. This will enable the subject to position themselves within the camera field of view.

The script will then prompt the user for the subjects unique ID, the subjects name and whether it is low light conditions or not. The unique IDs should start at 1 and be incremented by 1 each time you add a new subject and the name is the name that you wish the robot to use for this subject. You should ideally run this script twice per subject, once in bright light and a second time in low light conditions. This will improve the chances of the recognition algorithm having success. Each run of the script will take 100 images of the subject, the file name of each image is constructed from the subject id and an image number. Image numbers are numbered 0 to 99 for those taken in bright light and 100 to 199 for those taken in low light.

The next step is to add the subject to the names file if they don't already exist.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

with picamera.PiCamera() as camera:

camera.resolution = (1280, 960)

looping = True

count = 0

end = 99

names\_dict = {}

name\_file = '../trainer/names.yml'

*# Open the file of IDs and names to append the new one to*

if os.path.exists(name\_file):

with open(name\_file, 'r') as stream:

names\_dict = yaml.load(stream)

cv2.namedWindow('frame', cv2.WINDOW\_NORMAL)

face\_id = raw\_input("What is this persons ID number? ")

name = raw\_input("What is this persons name? ")

low\_light = raw\_input("Low light Y/N?" )

if low\_light == 'Y' or low\_light == 'y':

count = 100

end = 199

*# If not already in the dictionary add details*

if not face\_id in names\_dict:

names\_dict[int(face\_id)]=name

with open(name\_file, 'w') as outfile:

yaml.dump(names\_dict, outfile, default\_flow\_style=False)

A loop is then entered to capture the images. Each pass of the loop captures an image from the camera and converts it to a numpy array for the OpenCV calls.

Using OpenCV we then convert the image to a grey scale image and attempt to detect a face in the image. If a face is detected the image is cropped around the face, the number of image samples is incremented and the cropped grey scale image is stored in the dataset folder. The original image from the camera along with a superimposed frame around the face is displayed to the user.

A check is then made to see if the user has pressed the 'q' key on the keyboard which is used to exit the program early. Otherwise a check is made to see if we have captured the one hundred images of the face, if so this main loop is exited.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

while(looping):

*# Create a memory stream so image doesn't need to be saved to a file*

stream = io.BytesIO()

camera.capture(stream, format='jpeg')

*#Convert picture to numpy array*

buff = numpy.fromstring(stream.getvalue(), dtype=numpy.uint8)

*# Now create an OpenCV image*

image\_frame = cv2.imdecode(buff, 1)

*# Convert frame to grayscale*

gray = cv2.cvtColor(image\_frame, cv2.COLOR\_BGR2GRAY)

*# Detect frames of different sizes, list of faces rectangles*

faces = face\_detector.detectMultiScale(gray, 1.3, 5)

*# Although faces could contain more than one face we only expect one*

*# person to be in the data set image otherwise it would confuse*

*# the whole thing*

if (len(faces) != 0):

*# Expecting one face only on the data set image*

(x, y, w, h) = faces[0]

*# Crop the image frame into rectangle*

cv2.rectangle(image\_frame, (x,y), (x+w,y+h), (255,0,0), 4)

*# Increment sample face image*

count += 1

*# Save the captured image into the datasets folder*

cv2.imwrite("dataset/User." + str(face\_id) + '.' + str(count) + ".jpg", gray[y:y+h,x:x+w])

*# Display the video frame, with bounded rectangle on the person's face*

cv2.imshow('frame', image\_frame)

*# To stop taking video, press 'q' for at least 100ms*

if cv2.waitKey(100) & 0xFF == ord('q'):

looping = False

*# If image taken reach 100, stop taking video*

elif count>end:

looping = False

The last two lines close the window displaying the image and prints a message indicating the process is complete.

Hide   Copy Code

*# Close all started windows*

cv2.destroyAllWindows()

print("Data prepared")

Once you have run the script for each subject, or if you have rerun the script for a new subject, you then run the training.py script.

The training.py script starts with the imports and the assure\_path\_exists function definition, it then creates an instance of the OpenCV classes LBPHFaceRecognizer\_create and CascadeClassifier using the same classifier file.

Hide   Copy Code

import cv2

import os

import numpy as np

def assure\_path\_exists(path):

dir = os.path.dirname(path)

if not os.path.exists(dir):

os.makedirs(dir)

*# Create Local Binary Patterns Histograms for face recognization*

recognizer = cv2.face.LBPHFaceRecognizer\_create()

*# Using prebuilt frontal face training model, for face detection*

detector = cv2.CascadeClassifier("../classifiers/haarcascade\_frontalface\_default.xml");

The get\_images\_and\_labels function reads in each stored image, detects the face and obtains the id from the file name.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# Create method to get the images and label data*

def get\_images\_and\_labels(path):

*# Get all file path*

image\_paths = [os.path.join(path,f) for f in os.listdir(path)]

*# Initialize empty face sample*

face\_samples=[]

*# Initialize empty id*

ids = []

*# Loop all the file path*

for image\_path in image\_paths:

*# The stored image is grayscale so read in in gray scale*

gray = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

*# Get the image id*

id = int(os.path.split(image\_path)[-1].split(".")[1])

*# Get the face from the training images*

*# Don't need any scaling as these images already full face*

faces = detector.detectMultiScale(gray);

*# During testing not always detected face on image, which*

*# is odd as it should be just an image that was saved*

if (len(faces) == 0):

print "No face on " + image\_path

else:

*# We know each image is only of one face*

(x, y, w, h) = faces[0]

*# Add the image to face samples*

face\_samples.append(gray[y:y+h,x:x+w])

*# Add the ID to IDs*

ids.append(id)

*# Pass the face array and IDs array*

return face\_samples,ids

Once all the faces and ids are obtained they are passed to the OpenCV face recognizer and the data from the recognizer is saved to disk. The face recognition library that will be used by our node will later load this data to train the recognizer.

Hide   Copy Code

*# Get the faces and IDs*

faces,ids = get\_images\_and\_labels('dataset')

*# Train the model using the faces and IDs*

recognizer.train(faces, np.array(ids))

*# Save the model into trainer.yml*

assure\_path\_exists('../trainer/')

recognizer.save('../trainer/trainer.yml')

print("Done")

The code for the ROS node itself is in the sub folder src in the file face\_recognition\_node.py. The code makes use of a library file, face\_recognition\_lib.py, which contains the class FaceRecognition. This file is in the sub folder src/face\_recognition\_lib.

Before describing the code for the node I'll discus the FaceRecognition class. After the required imports and the declaration of the class it defines a number of functions.

The class constructor creates the OpenCV face recognizer and then reads the training file created by the training script. It then opens the file containing the list of names and the ids, and creates the classifier. It finally stores a confidence value passed to it. This value will be used to determine if the suggested ID for the face is accepted.

Hide   Copy Code

def \_\_init\_\_(self, path, confidence):

*# Create Local Binary Patterns Histograms for face recognization*

self.\_\_face\_recognizer = cv2.face.LBPHFaceRecognizer\_create()

*# Load the trained mode*

self.\_\_face\_recognizer.read(path + '/trainer/trainer.yml')

*# Load the names file*

with open(path + '/trainer/names.yml', 'r') as stream:

self.\_\_names\_dict = yaml.load(stream)

*# Detect object in image using Haarcascade Frontal Face*

self.\_\_face\_detector = cv2.CascadeClassifier(path + '/classifiers/haarcascade\_frontalface\_default.xml')

*# Confidence level, the confidence of the system in recognising a face must be greater than*

*# this level to be accepted by the system as a recognised face.*

self.\_\_confidence\_level = confidence

Two functions are declared which will be used to modify the captured image if a face is detected. The first will draw a rectangle on the image and the second will draw the supplied text on the image.

Hide   Copy Code

*# Function to draw rectangle on image according to given (x, y) coordinates*

*# and the given width and height*

def draw\_rectangle(self, img, rect, bgr):

(x, y, w, h) = rect

cv2.rectangle(img, (x, y), (x+w, y+h), bgr, 4)

*# Function to draw text on give image starting at the passed (x, y) coordinates.*

def draw\_text(self, img, text, x, y, bgr):

cv2.putText(img, text, (x, y), cv2.FONT\_HERSHEY\_PLAIN, 3.0, bgr, 4)

The next function detect\_faces, is used to detect any faces in the supplied image. It converts the image to grey scale so that OpenCV can be used to detect any faces. If faces are detected then the data for each face and a location of the face in the image is returned. Note that this part of the code is written to allow more than one face to be seen in the image.

Hide   Copy Code

*# This function detects any faces using OpenCV from the supplied image*

def detect\_faces(self, img):

face\_data = []

*#convert the test image to gray image as opencv face detector expects gray images*

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

*#let's detect multiscale (some images may be closer to camera than others) images*

*#result is a list of faces*

faces\_detected = self.\_\_face\_detector.detectMultiScale(gray, 1.3, 5);

*#if no faces are detected then return None*

if (len(faces\_detected) == 0):

return None, None

*#return only the face part of the image*

for face in faces\_detected:

(x, y, w, h) = face

face\_data.append(gray[y:y+w, x:x+h])

*# faces\_detected is a list of rectangles where faces have been detected.*

*# face\_data is a list of the data for the faces detected*

return face\_data, faces\_detected

The final function in the class, scan\_for\_faces, is the one which will be called by the node to do the face detecting and recognition on the supplied image. This function calls the detect\_faces function and if any faces are detected it loops through each face calling the OpenCV face predictor which returns the id of the recognised face and a value indicating the confidence in the prediction. This value is converted to a percentage of confidence. From the supplied id a name is obtained. A rectangle is drawn around the face along with the name of the subject and the confidence level. If the confidence level of the prediction exceeds the stored threshold value the box and text drawn on the image will be in the colour green, otherwise in the colour red. Also if the confidence threshold is exceeded a dictionary entry of the id and name will be entered. Once all the faces detected have been analysed this dictionary is returned to the caller.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# This class function will be called from outside to scan the supplied img.*

*# First it attempts to detect faces in the image and then if any are found*

*# it attempts for recognise them against know subjects. It will adjust the*

*# supplied image.*

def scan\_for\_faces(self, img):

*# First do the face detection, returned faces is a list of the faces detected*

*# and rects is a list of rectangles of where the faces are in the image*

faces, rects = self.detect\_faces(img)

*# Create a dictionary of IDs and Names of those detected in the image*

detected\_dict = {}

*# If we detected faces then process each one*

if(faces != None):

for index in range(len(faces)):

*# Predict the image using our face recognizer*

label, confidence = self.\_\_face\_recognizer.predict(faces[index])

our\_confidence = round(100 - confidence, 2)

*# Get name of respective label returned by face recognizer*

name\_text = self.\_\_names\_dict[label]

name\_text\_confidence = name\_text + " {0:.2f}%".format(our\_confidence)

if(our\_confidence > self.\_\_confidence\_level):

colour = (0, 255, 0)

else:

colour = (0, 0, 255)

*#draw a rectangle around face(s) detected*

self.draw\_rectangle(img, rects[index], colour)

*#draw name of predicted person(s) and the confidence value*

self.draw\_text(img, name\_text\_confidence, rects[index,0], rects[index,1]-5, colour)

if(our\_confidence > self.\_\_confidence\_level):

*# Add details to the dictionary to be returned*

detected\_dict[label]=name\_text

return detected\_dict

Next I'll describe the ROS node itself.

The main routine initialises ROS for the node and creates an instance of FaceRecognitionNode and then calls spin to process the topics subscribed to and published.

Hide   Copy Code

def main(args):

rospy.init\_node('face\_recognition\_node', anonymous=False)

frn = FaceRecognitionNode()

rospy.loginfo("Face recognition node started")

try:

rospy.spin()

except KeyboardInterrupt:

print("Shutting down")

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

main(sys.argv)

The rest of the file contains the FaceRecognitionNode class.

The constructor for FaceRecognitionNode creates an instance of CVBridge, as previously stated this will be used to convert the ROS image to an OpenCV image. It then registers that it will publish the topic face\_recognition\_node/image/compressed which will be used to send the image with a box drawn around any faces recognised with the subject name and confidence level also printed on the image. This topic is used for testing the node.

The node subscribes to the topic raspicam\_node/image/compressed which will contain the latest image from the camera, the function callback is registered to be called when a new image is received.

The confidence threshold is read from the parameter server and is set to the default value of 20% if a value is not held in the parameter server.

The constructor then creates an instance of the FaceRecognition class described above.

Finally we create and start an action server. An action is used when a node is required to perform a non-blocking task which may take some time. A request is sent to perform the task and a reply is received once the task is complete. Feedback during the task can also be received. The task request can also be cancelled. More information on ROS actions can be [found here](http://wiki.ros.org/actionlib). The file containing our user defined action message is described later in this section.

Hide   Copy Code

class FaceRecognitionNode:

def \_\_init\_\_(self):

self.\_\_bridge = CvBridge()

self.\_\_image\_pub = rospy.Publisher('face\_recognition\_node/image/compressed', CompressedImage, queue\_size=1)

self.\_\_image\_sub = rospy.Subscriber('raspicam\_node/image/compressed', CompressedImage, self.callback)

confidence\_level = rospy.get\_param('/face\_rec\_python/confidence\_level', 20)

rospy.loginfo("FaceRecognitionNode: Confidence level %s", str(confidence\_level))

*# Create the face\_recognition\_lib class instance*

self.\_\_frc = face\_recognition\_lib.FaceRecognition(roslib.packages.get\_pkg\_dir('face\_recognition', required=True), confidence\_level)

*# Create the Action server*

self.\_\_as = actionlib.SimpleActionServer('face\_recognition', scan\_for\_facesAction, self.do\_action, False)

self.\_\_as.start()

The function do\_action is called by the action server when a request to conduct the action is received.

This function converts the last received image from a ROS image to an OpenCV image. A call to the scan\_for\_faces is made to check the image for known faces. The adjusted image, which may now contain superimposed boxes and names is converted back to a ROS image and published on the face\_recognition\_node/image/compressed topic.

The names and ids returned from the call to scan\_for\_faces is then used to create the result of the action. If no faces were recognised the data in the result will be empty.

Hide   Copy Code

def do\_action(self, goal):

*# Scan the current image for faces recognised*

*# The image may show more than one face.*

*# The returned dictionary will contain the unique IDs*

*# and Names of any subjects recognised.*

*# If no detection/recognition dictionary will be empty*

image = self.\_\_bridge.compressed\_imgmsg\_to\_cv2(self.\_\_current\_image)

*# In the next call image will be altered if faces are recognised*

detected\_dict = self.\_\_frc.scan\_for\_faces(image)

try:

self.\_\_image\_pub.publish(self.\_\_bridge.cv2\_to\_compressed\_imgmsg(image))

except CvBridgeError as e:

print(e)

*# Now post a message with the list of IDs and names*

ids = []

names = []

for k, v in detected\_dict.items():

ids.append(k)

names.append(v)

*# Set result for the action*

result = scan\_for\_facesResult()

result.ids\_detected = ids

result.names\_detected = names

self.\_\_as.set\_succeeded(result)

The function callback is called each time a message is received on the raspicam/image/compressed topic.

This function simply stores the current image should it be required for facial recognition.

Hide   Copy Code

*# Callback for new image received*

def callback(self, data):

*# Each time we receive an image we store it ready in case then asked to scan it*

self.\_\_current\_image = data

The node package also contains a config.yaml file which can be used to set the confidence level without having to recompile the code. The package also contains a launch file, test.launch, which can be used to test the node. As well as launching this node, it will launch the camera node.

### **Face recognition action**

As stated above the face\_recognition package uses a user defined action message to kick off the operation and to return the result of looking for known faces. The package face\_recognition\_msgs contains our first ROS action scan\_for\_faces.action.

The files for the package are available in the face\_recognition\_msgs folder and the file for the action is stored in the action sub folder.

An action specification contains a goal, result and feedback section. It looks similar to a message definition file except each of these parts is divided by the three dashes (---).

Hide   Copy Code

# This action scans the next camera image for faces that are recognised

# There are no parameters to start the action

---

# Results of the scan

uint16[] ids\_detected

string[] names\_detected

---

# No parameters for feedback

Above the first three dashes is the goal. In our case we don't have any parameters for the goal, just the receipt of the goal will start our action.

The result parameters are below the first three dashes and in our case contains an array of ids and an array of names for any recognised faces.

Below the second three dashes is the feedback parameters. In our case we will not return any feedback.

### **Controlling the head**

We now have a node to carry out the facial recognition operation on an image from the camera and from part 1 of this article we have the pan/tilt functionality to move the servos connected to the head. We will now add a node which accepts a position the head should be in, but moves the head to that target in steps so as not to rock the robot when the head moves from one extreme to the other. The node will also accept not only absolute position to move to, but a relative distance to move from the current position.

Our ROS package for the node is called head\_control and is available in the head\_control folder. The package contains all the usual ROS files and folders.

The action folder contains the point\_head.action file. This action will pass a goal containing the position to move to and feedback of where the head is on its way to the target position.

Hide   Copy Code

# This action moves the head to a given position

# The goal definition is the position to move the head to if absolute is true,

# otherwise the pan/tilt values are how much relative to move the head.

# Values are radians.

bool absolute

float64 pan

float64 tilt

---

# There are no parameters for the result

---

# Feedback gives the current position

float64 current\_pan

float64 current\_tilt

The goal contains a pan and tilt value in radians. It also contains a boolean flag absolute. If the flag is true then the pan and tilt values are the absolute target position. If the flag is false then the values represent how much the head should be moved relative to the current position.

The feedback values represent the current position of the head on its way to the target position.

The config folder contains a config.yaml file which can be used to override some of the default configuration values. You can configure:

* The default position of the head
* The maximum value the pan and tilt device should move per request. This is to stop the servo moving a requested angle in one jump and causing the head to shudder

Hide   Copy Code

head:

position:

pan: 0.0

tilt: 0.0

max\_step:

pan: 0.0872665

tilt: 0.0872665

The include/head\_control and src folders contain the C++ code for the package. For this package we have one C++ class, HeadControlNode and a main routine contained within the head\_control\_node.cpp file.

The main routine informs ROS of our node, creates an instance of the class for the node and passes it the node handle and node name. For the first time in the project we are not going to hand total control of the node to ROS. In all our other nodes so far we have handed off control to ROS with a call to ros::spin in C++ or rospy.spin in our Python code. These calls hand control to ROS and our code only gets to run when ROS calls one of our callback functions when a message is received. In this node I want to retain control as we want to move the servos in small incremental steps to a target position. If we allow the servos to move a large angle in one go the head comes to a shuddering stop when the target position is reached. The way that we retain control is to call ros::spinOnce. In this function ROS will publish any topics we have requested and process any incoming topics by calling our callbacks but will then return control to main.

Before we enter the while loop we create an instance of ros::Rate and pass it the timing we would like to maintain, in our case I'm setting the rate to 10Hz. When inside the loop we call r.sleep, this Rate instance will attempt to keep the loop at 10Hz by accounting for the time used to complete the work in the loop.

Our loop will continue while the call to ros::ok returns true, it will return false when the node has finished shutting down e.g. when you press Ctrl-c on the keyboard.

In the loop we will call moveServo which is described later in the article.

Hide   Copy Code

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

{

ros::init(argc, argv, "head\_control\_node");

ros::NodeHandle n;

std::string node\_name = ros::this\_node::getName();

HeadControlNode head\_control(n, node\_name);

ROS\_INFO("%s started", node\_name.c\_str());

*// We need control of the node so that we can step the servos to the target*

*// position in small steps to stop the head shuddering if it was to move in one big step*

ros::Rate r(10); *// 10Hz*

while(ros::ok())

{

*// See if the servos need moving*

head\_control.moveServo();

ros::spinOnce();

r.sleep();

}

return 0;

}

The rest of the file contains the HeadControlNode class.

The constructor for HeadControlNode registers our callback function, pointHeadCallback,with the action server. This callback will be called when the action server receives the action goal, this in effect kicks off the action.

We then start the server running with the as\_.start call.

The constructor then advertises that it will publish the topic pan\_tilt\_node/joints which will be used to pass the required pan/tilt position to the pan/tilt node.

We then set some configuration defaults and read any overrides from the parameter server should they be available.

Next we set names of our joints in the joint state message as these will not change.

Finally we publish a message to move the head to a known starting point. We can't move to this position in small steps as we don't know our starting position after power up.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// Constructor*

HeadControlNode::HeadControlNode(ros::NodeHandle n, std::string name) : as\_(n, name, false)

{

nh\_ = n;

as\_.registerGoalCallback(boost::bind(&HeadControlNode::pointHeadCallback, this));

as\_.start();

*// Topic to move head*

move\_head\_pub\_ = nh\_.advertise<sensor\_msgs::JointState>("pan\_tilt\_node/joints", 10, true);

*// Obtain any configuration values from the parameter server. If they don't exist use the defaults*

*// Joint names*

nh\_.param<std::string>("/servo/index0/pan/joint\_name", pan\_joint\_name\_, "reserved\_pan0");

nh\_.param<std::string>("/servo/index0/tilt/joint\_name", tilt\_joint\_name\_, "reserved\_tilt0");

*// Maximum angle we can move in one go*

nh\_.param("/head/max\_step/pan", pan\_step\_, 0.174533);

nh\_.param("/head/max\_step/tilt", tilt\_step\_, 0.174533);

double pan; *// Pan default position to return to*

double tilt; *// Tilt default position to return to*

nh\_.param("/head/position/pan", pan, 0.0);

nh\_.param("/head/position/tilt", tilt, 0.0);

default\_position\_.pan = pan;

default\_position\_.tilt = tilt;

*// Set up the the message we will publish*

msg\_.name.push\_back(pan\_joint\_name\_);

msg\_.name.push\_back(tilt\_joint\_name\_);

msg\_.position.push\_back(0.0);

msg\_.position.push\_back(0.0);

*// We will often return to this position when a task is completed*

current\_pan\_tilt\_ = default\_position\_;

*// We don't know where the servo starts from so just jump to the required position*

*// Publish a start position to get the head in a known position.*

publishJointState(current\_pan\_tilt\_);

move\_head\_ = false;

movement\_complete\_ = false;

target\_pan\_tilt\_ = current\_pan\_tilt\_;

}

I'll now briefly describe the functions that make up the class.

The pointHeadCallback is called by ROS when the action server receives a goal message. The goal data is either the absolute or relative target position depending on the state of the absolute flag.

The function calls the action server to inform it that the goal has been accepted, it stores the new target position and sets the flag move\_head to true indicating that the head is required to be moved.

Hide   Copy Code

*// This callback is for the point head action*

void HeadControlNode::pointHeadCallback()

{

head\_control::point\_headGoal::ConstPtr goal;

goal = as\_.acceptNewGoal();

*// Set the target position to the request position*

if (goal->absolute == true)

{

target\_pan\_tilt\_.pan = goal->pan;

target\_pan\_tilt\_.tilt = goal->tilt;

}

else

{

target\_pan\_tilt\_.pan += goal->pan;

target\_pan\_tilt\_.tilt += goal->tilt;

}

*// Indicate that the servos should be moved*

move\_head\_ = true;

movement\_complete\_ = false;

}

The function, moveServo, is the function called by main in our loop. It checks to see if a request to move the head was made and if so enters an 'if', 'else if', 'else' construct.

The 'if' part of this construct checks to see if the action has been pre-empted. If so accepts the pre-emption and tidies up.

The 'else if' part checks to see if the head movement is complete, if so a counter is decremented. This counter is used to include time for the head to stop moving and blurring any camera images after the servos reach the target position. When the counter reaches zero the fact that the action is complete is reported to the action server. There are no parameters in the result.

The 'else' part is responsible for calculating the next step movement of the servos towards the target position, it also publishes the joint state message with the next required servo position using the helper function publishJointState and reports the feedback to the action server. This feedback will contain the position that the servos were just asked to move to.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// Function to move the servos if required by a step amount. This is to stop the head shuddering if the servo*

*// is moved to the target position in one movement.*

void HeadControlNode::moveServo()

{

if(move\_head\_ == true)

{

if(as\_.isPreemptRequested() || !ros::ok())

{

as\_.setPreempted();

movement\_complete\_ = false;

move\_head\_ = false;

}

else if(movement\_complete\_ == true)

{

*// We have reached the target but give time to settle*

loop\_count\_down\_--;

if(loop\_count\_down\_ <= 0)

{

movement\_complete\_ = false;

move\_head\_ = false;

head\_control::point\_headResult result;

as\_.setSucceeded(result);

}

}

else

{

if((target\_pan\_tilt\_.pan == current\_pan\_tilt\_.pan) && (target\_pan\_tilt\_.tilt == current\_pan\_tilt\_.tilt))

{

*// Last time around we must have requested the final move*

movement\_complete\_ = true;

loop\_count\_down\_ = 8;

}

else

{

*// Still moving, calculate pan movement*

if(std::abs(target\_pan\_tilt\_.pan - current\_pan\_tilt\_.pan) > pan\_step\_)

{

*// Distance to target to great to move in one go*

if(target\_pan\_tilt\_.pan > current\_pan\_tilt\_.pan)

{

*// Add the step to current*

current\_pan\_tilt\_.pan += pan\_step\_;

}

else

{

*// Subtract step from current*

current\_pan\_tilt\_.pan -= pan\_step\_;

}

}

else

{

*// Can move to the target position in one go (or pan is in fact already there but tilt is not)*

current\_pan\_tilt\_.pan = target\_pan\_tilt\_.pan;

}

*// Calculate tilt movement*

if(std::abs(target\_pan\_tilt\_.tilt - current\_pan\_tilt\_.tilt) > tilt\_step\_)

{

*// Distance to target to great to move in one go*

if(target\_pan\_tilt\_.tilt > current\_pan\_tilt\_.tilt)

{

*// Add the step to current*

current\_pan\_tilt\_.tilt += tilt\_step\_;

}

else

{

*// Subtract step from current*

current\_pan\_tilt\_.tilt -= tilt\_step\_;

}

}

else

{

*// Can move to the target position in one go (or tilt is in fact already there but pan is not)*

current\_pan\_tilt\_.tilt = target\_pan\_tilt\_.tilt;

}

*// Publish the movement*

publishJointState(current\_pan\_tilt\_);

*// Publish feedback*

head\_control::point\_headFeedback feedback;

feedback.current\_pan = current\_pan\_tilt\_.pan;

feedback.current\_tilt = current\_pan\_tilt\_.tilt;

as\_.publishFeedback(feedback);

}

}

}

}

The final function in the class is the helper function publishJointState. This function updates the position values in the joint state message and then publishes the message.

Hide   Copy Code

*// This function creates and publishes a joint state message*

void HeadControlNode::publishJointState(struct position pan\_tilt)

{

msg\_.position[0] = pan\_tilt.pan;

msg\_.position[1] = pan\_tilt.tilt;

msg\_.header.stamp = ros::Time::now();

move\_head\_pub\_.publish(msg\_);

}

The package also includes a launch file in the launch folder. This file, test.launch, will launch all the nodes developed to move the head.

Hide   Copy Code

<?xml version="1.0" ?>

<launch>

<rosparam command="load" file="$(find pan\_tilt)/config/config.yaml" />

<rosparam command="load" file="$(find head\_control)/config/config.yaml" />

<node pkg="pan\_tilt" type="pan\_tilt\_node" name="pan\_tilt\_node" output="screen" />

<node pkg="rosserial\_python" type="serial\_node.py" name="serial\_node" output="screen" args="/dev/ttyUSB0"/>

<node pkg="head\_control" type="head\_control\_node" name="head\_control\_node" output="screen"/>

</launch>

### **Action client**

Having included action servers in both our nodes you would expect to have action clients to connect with them. This is definitely one method of communicating with the server. In a later part of the article I'm going to introduce a ROS package which will allow us to create state machines and sub-state machines to control our robot missions. Using this package it is possible to assign an individual state to be the action client and all the communication is done behind the scenes for us.

In order to test the system we have developed so far and to show how to write an action client, we will write two test nodes here which will include an action client in each one.

Our first node is a very simple node written in Python and will be used to test the face recognition node. The ROS package for this node is called rodney\_recognition\_test and is available in the rodney\_recognition\_test folder. The package contains all the usual ROS files and folders.

All the code is contained in the rodney\_recognition\_test\_node.py file in the src folder.

The code initialises our node and creates an action client. Note that the name passed to the SimpleActionClient, in our case 'face\_recognition' must match the name given to the action server.

We then call wait\_for\_server, the code will wait here until it is able to make contact with the server. We then create a goal, which in this case contains no data, and send the goal to the server.

In our simple example we then wait until the result is returned and the node finishes by printing the id and names of any faces recognised and returned in the result.

Hide   Copy Code

*#!/usr/bin/env python*

import rospy

import actionlib

from face\_recognition\_msgs.msg import scan\_for\_facesAction, scan\_for\_facesGoal, scan\_for\_facesResult

rospy.init\_node('face\_recognition\_client')

client = actionlib.SimpleActionClient('face\_recognition', scan\_for\_facesAction)

client.wait\_for\_server()

goal = scan\_for\_facesGoal()

client.send\_goal(goal)

client.wait\_for\_result()

result = client.get\_result()

print(result.ids\_detected)

print(result.names\_detected)

In our next package to test the head\_control node we will write a slightly more complicated node this time written in C++.

Our ROS package is called rodney\_head\_test and is available in the rodney\_head\_test folder. The package contains all the usual ROS files and folders.

The include/rodney\_head\_test and src folders contain the C++ code for the package. For this package we have one C++ class, RodneyHeadTestNode and a main routine contained within the rodney\_head\_test\_node.cpp file.

The main routine informs ROS of our node, creates an instance of the class for the node and passes it the node handle, logs that the node has started and hands control to ROS with the call to ros::spin.

Hide   Copy Code

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

{

ros::init(argc, argv, "rodney\_head\_test");

ros::NodeHandle n;

RodneyHeadTestNode rodney\_head\_test\_node(n);

std::string node\_name = ros::this\_node::getName();

ROS\_INFO("%s started", node\_name.c\_str());

ros::spin();

return 0;

}

The constructor creates an instance of our action client, ac\_, and passes it the name of the action server which in our case is head\_control\_node. This must match the name we gave to our action server when we created it in the HeadControlNode constructor.

We then read the config parameters to limit the movement of the servos.

We are going to use a keyboard node which is available from <https://github.com/lrse/ros-keyboard>, to interact with the system. In the constructor we subscribe to the topic keyboard/keydown and call the function keyboardCallBack when a message is received on that topic.

The call ac\_.waitForServer will wait in the constructor until our action server is running.

Hide   Copy Code

*// Constructor*

RodneyHeadTestNode::RodneyHeadTestNode(ros::NodeHandle n) : ac\_("head\_control\_node", true)

{

nh\_ = n;

*// Subscribe to receive keyboard input*

key\_sub\_ = nh\_.subscribe("keyboard/keydown", 100, &RodneyHeadTestNode::keyboardCallBack, this);

nh\_.param("/servo/index0/pan/max", max\_pan\_radians\_, M\_PI/2.0);

nh\_.param("/servo/index0/pan/min", min\_pan\_radians\_, -(M\_PI/2.0));

nh\_.param("/servo/index0/tilt/max", max\_tilt\_radians\_, M\_PI/2.0);

nh\_.param("/servo/index0/tilt/min", min\_tilt\_radians\_, -(M\_PI/2.0));

ROS\_INFO("RodneyHeadTestNode: Waiting for action server to start");

*// wait for the action server to start*

ac\_.waitForServer(); *//will wait for infinite time*

moving\_ = false;

ROS\_INFO("RodneyHeadTestNode: Action server started");

}

The function keyboardCallBack checks the received message and runs a test dependent on the key pressed.

It creates an instance of our action goal, sets the goal parameters and passes it to the action server with a call to ac\_.sendGoal. With the call we pass three callback functions, 1) doneCB which is called when the action is completed 2) activeCB which is called when the action goes active and 3) feedbackCB which is called when the feedback on the progress of the action is received.

The action can be pre-empted, so if the 'c' key is pressed and moving the head is in progress we will cancel the action with a call to ac\_.cancelGoal.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

void RodneyHeadTestNode::keyboardCallBack(const keyboard::Key::ConstPtr& msg)

{

head\_control::point\_headGoal goal;

*// Check for key 1 with no modifiers apart from num lock is allowed*

if((msg->code == keyboard::Key::KEY\_1) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 1, Test 1 move to max pan and tilt*

goal.absolute = true;

goal.pan = max\_pan\_radians\_;

goal.tilt = max\_tilt\_radians\_;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_2) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 2, test 2 move to min pan and tilt*

goal.absolute = true;

goal.pan = min\_pan\_radians\_;

goal.tilt = min\_tilt\_radians\_;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_3) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 3, test 3 move to pan 0, tilt 0*

goal.absolute = true;

goal.pan = 0.0;

goal.tilt = 0.0;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_4) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 4, test 4 move to pan 0, tilt -45 degress*

goal.absolute = true;

goal.pan = 0.0;

goal.tilt = -0.785398;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_5) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 5, test 5 move tilt up by 10 degrees*

goal.absolute = false;

goal.pan = 0;

goal.tilt = -0.174533;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_6) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 6, test 6 move pan by 20 anti-clockwise*

goal.absolute = false;

goal.pan = 0.349066;

goal.tilt = 0;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

if((msg->code == keyboard::Key::KEY\_7) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 7, test 7 move pan by 20 clockwise and tilt by 10 down*

goal.absolute = false;

goal.pan = -0.349066;

goal.tilt = 0.174533;

*// Need boost::bind to pass in the 'this' pointer*

ac\_.sendGoal(goal,

boost::bind(&RodneyHeadTestNode::doneCB, this, \_1, \_2),

boost::bind(&RodneyHeadTestNode::activeCB, this),

boost::bind(&RodneyHeadTestNode::feedbackCB, this, \_1));

}

else if((msg->code == keyboard::Key::KEY\_c) && ((msg->modifiers & ~RodneyHeadTestNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// Key 'c' or 'C', cancel action*

if(moving\_ == true)

{

ac\_.cancelGoal();

}

}

else

{

;

}

}

The callback function activeCB is called when the action goes active, here we log the fact and set a member variable indicating that movement is taking place.

Hide   Copy Code

*// Called once when the goal becomes active*

void RodneyHeadTestNode::activeCB()

{

ROS\_INFO("RodneyHeadTestNode: Goal just went active");

moving\_ = true;

}

The callback function feedbackCB is called when feedback on the progress of the action is received. If you remember our feedback includes the current position of the servos on their way to the target position.

Hide   Copy Code

*// Called every time feedback is received for the goal*

void RodneyHeadTestNode::feedbackCB(const head\_control::point\_headFeedbackConstPtr& feedback)

{

ROS\_INFO("Feedback pan=%f, tilt=%f", feedback->current\_pan, feedback->current\_tilt);

}

The callback function donCB is called when the action is completed. In this case the result data is empty.

Hide   Copy Code

*// Called once when the goal completes*

void RodneyHeadTestNode::doneCB(const actionlib::SimpleClientGoalState& state,

const head\_control::point\_headResultConstPtr& result)

{

ROS\_INFO("RodneyHeadTestNode: Finished in state [%s]", state.toString().c\_str());

moving\_ = false;

}

## **Using the code**

In this article we will test the two nodes individually. In part four we will put the two together so that the robot can scan a room within its head movement range looking for faces it recognises.

As previously when testing the code, I'm going to run the system code on the Raspberry Pi and the test code on a separate Linux workstation. The Raspberry Pi will also be connected to the Arduino nano which in turn is connected to the servos and running the sketch from part one of the article.

Note that to distinguish between the Pi and the workstation in the instructions below, the code is in a folder (workspace) called "rodney\_ws" on the Pi and "test\_ws" on the workstation.

### **Building the ROS packages on the Pi**

If not already done create a catkin workspace on the Raspberry Pi and initialise it with the following commands:

Hide   Copy Code

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, pan\_tilt and servo\_msgs into the ~/rodney\_ws/src folder and then build the code. As a little tip I don't copy the code into the src folder but create a symbolic link in the src folder to the code location. That way I can have a number of workspaces using the same code files.

Hide   Copy Code

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

### **Building the ROS test packages on the workstation**

You can build and run the test packages on the Raspberry Pi but I'm going to use a Linux workstation which is on the same network as the Pi.

Create a workspace with the following commands:

Hide   Copy Code

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, pan\_tilt, servo\_msgs, *rodney\_recognition\_test, rodney\_head\_test* and *ros-keyboard*  (from <https://github.com/lrse/ros-keyboard>) into the ~/test\_ws/src folder and then build the code with the following commands:

Hide   Copy Code

$ cd ~/test\_ws/

$ catkin\_make

Check that the build completes without any errors.

### **Tip**

When running ROS code and tools on a workstation and the Raspberry Pi, there can be a lot of repeat typing of commands at a number of terminals. In the next section I have included the full commands to type but here are a few tips that can save you all that typing.

On the Raspberry Pi to save typing "source devel/setup.bash" I have added it to the .bashrc file for the Raspberry Pi.

Hide   Copy Code

$ cd ~/

$ nano .bashrc

Then add "source /home/ubuntu/rodney\_ws/devel/setup.bash" to the end of the file, save and exit.

When running test code and tools on the workstation it also needs to know where the ROS master is so I have added the following to the .bashrc file for the workstation.

Hide   Copy Code

alias rodney='source ~/test\_ws/devel/setup.bash; \

export ROS\_MASTER\_URI=http://ubiquityrobot:11311'

Then by just typing "rodney" at a terminal the two commands are run and a lot of typing is saved.

### **Running the code**

First we will test the face recognition node. Use the launch file to start the nodes with the following commands on the Raspberry Pi:

Hide   Copy Code

$ cd ~/rodney\_ws/

$ source devel/setup.bash

$ roslaunch face\_recognition test.launch

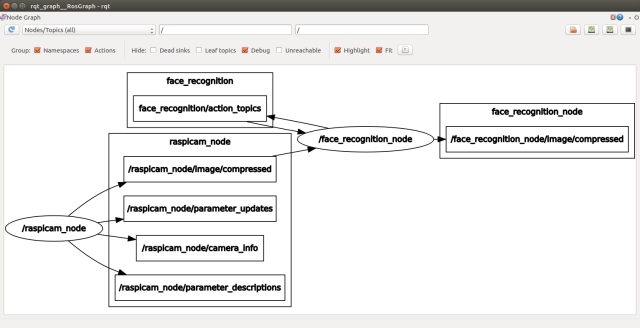
With the nodes running on the Raspberry Pi I'm going to use a Linux workstation on the same network to run some test. Note: as we will use our user defined topics the code also needs to be built on this workstation. You can if you wish run the tests on the same Raspberry Pi running the system nodes.

At the workstation run the following to check that the nodes are running and connected to the correct topics. You can see the name of master in the output from running roslaunch. As I'm using the Ubiquity ROS Ubuntu image and have not changed the name my master is ubiquityrobot.

Hide   Copy Code

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rqt\_graph



If any topics have been misspelt in one part of the code then it will be obvious from the graph as the nodes will not be joined by the topics.

In another terminal enter the following in order to be able to view the images.

Hide   Copy Code

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rqt\_image\_view

In the Image View GUI you can select the topic */raspicam/image/compressed* to view the current camera image. For the test I'm going to select the topic */face\_recognition\_node/image/compressed*, the image will currently be blank but when we request a face recognition operation we will be able to view the result.

In a new terminal run the test node to conduct the face recognition process on an image from the camera.

Hide   Copy Code

$ cd ~/test\_ws

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ source devel/setup.bash

$ rosrun rodney\_recognition\_test rodney\_recognition\_test\_node.py

You can rerun the process by just entering the last line again in the same terminal. The result of each run will be shown in the rqt\_image\_view window and output in the terminal.

When I ran the test without anyone in view of the camera, the image viewer displayed an image of the room and the terminal reported empty results with:

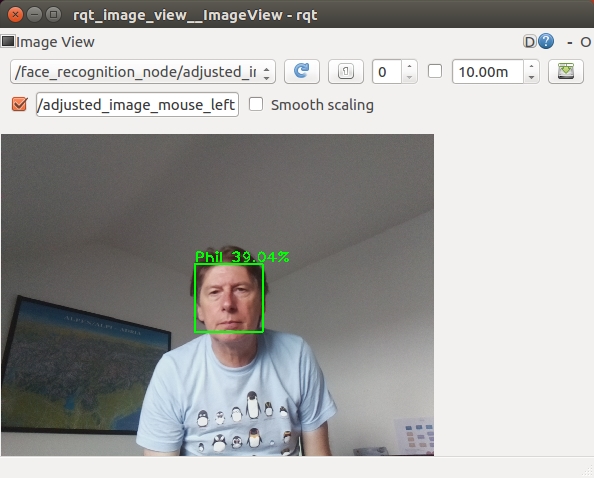
Hide   Copy Code

()

[]

When run with myself in view of the camera the terminal and the image viewer displayed the following:

Hide   Copy Code



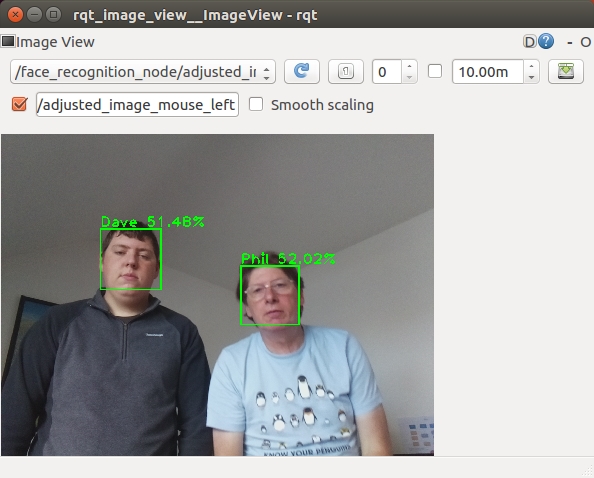
When testing with two people in the image, it's trained for both these subjects, I got the following results.

Hide   Copy Code

(1, 2,)

[Phil, Dave]

---



You can close down each terminal on the workstation and the Pi by entering Ctrl-C in the terminal.

Next we will test the head control node. With the Arduino connected to a USB port of the Pi use the launch file to start the nodes with the following commands.

Hide   Copy Code

$ cd ~/rodney\_ws/

$ source devel/setup.bash

$ roslaunch head\_control test.launch

When the code starts the head will move to the default position.

Next I'm going to use rqt\_graph and our test code to test the system. On the workstation run the following commands to start the keyboard node:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosrun keyboard keyboard

On the workstation in a second terminal run the following commands to start our test node:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosrun rodney\_head\_test rodney\_head\_test\_node

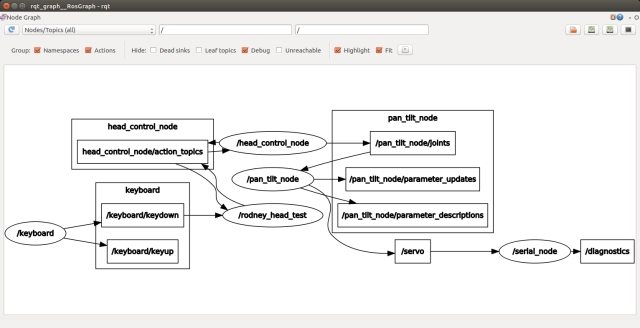
In a third terminal run the following commands to start rqt\_graph:

Hide   Copy Code

$ cd ~/test\_ws

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rqt\_graph



From the graph you should see the nodes under test and the test code running. You should also see the nodes linked by the topics. Any broken links is an indication of misspelt topics in the code.

The workstation should also be running a small window whose title is "ROS keyboard input". Make sure this window has the focus and then press a key for the following tests. During a scan head movement you can press the 'c' key to cancel the action.

* Key '1' - The head will move to the maximum pan and tilt position (left and down)
* Key '2' - The head will move to the minimum pan and tilt position (right and up)
* Key '3' - The head will move back to zero pan and tilt position
* Key '4' - The head will tilt to up to the 45 degrees position
* Key '5' - The head will move up from the current position by 5 degrees
* Key '6' - The head will move anti-clockwise (left) from the current position by 20 degrees
* Key '7' - The head will move clockwise (right) from the current position by 20 degrees and down by 10 degrees

## **Points of Interest**

In this part of the article we have added the face recognition and control of the head to the code from part 1 to complete our Design Goal 1.

In the next article I'll give Rodney facial expressions and speech to complete Design Goal 2.

A short video showing Rodney running a test of the combined Design Goal 1 and 2.