Computer Vision Enabled Localization of Mobile Robot Networks

Katherine Liu
University of California, San Diego
Winter 2014

0 ABSTRACT

This paper summaries my individual efforts to provide a computer vision enabled localization module in support of the Robot Operating System (ROS) Robot Project at UC San Diego. An algorithm, relying on simple Hue Saturation Value (HSV) space mapping and blob detection was integrated into the existing ROS framework. Initial results indicate that with further filtering, this will be a viable, stable method of localization.

1 BIG PICTURE

This section will cover both the overall objectives of the project, as well the objectives of my own sub-project.

1.1 Objective and Overall Goals

Realization of control algorithms in the physical world necessitates a readily deployable system. The Cortes and Martinez labs at UCSD currently possess ten TurtleBot platforms to be integrated into a multi-agent robotic network. The open source ROS has been chosen as the software platform to enable the network structure, relying on a publisher-subscriber model to allow for inter-robot communication and coordination.

The objective of the ROS TurtleBot project is to provide a stable system to which distributed control algorithms can be readily deployed for testing and validation on hardware. There are a plethora of interesting algorithms that could potentially be deployed using the TurtleBot network, such as cooperative task completion. Of particular interest to the group is cooperative simultaneous localization and mapping (SLAM).

1.2 Localization of Mobile Robots

Allowing individual agents in a network of autonomous robots access to their location information is a valuable tool in deploying many control algorithms. Discrete sampling of agent location can be used to generate velocity estimates as well as heading information. In a GPS denied environment, computer vision algorithms serve as a means by which to extract location information from images collected by a camera. This information can be leveraged as a ground truth by which to benchmark other control algorithms, such as SLAM.

To this end, we desire a computer vision enabled localizing node capable of integration with the Linux based ROS that leverages the open source C++ library, OpenCV. The node should be capable of delivering both location information in the form of Cartesian coordinate and heading information describing the general direction of the robot. Furthermore, the localization scheme

should be scalable, so that as the number of agents in the network increases, the capability is still supported.

2 PERSONAL CONTRIBUTIONS

This quarter I focused on prototyping the localization subsystem described in 1.2. My contribution can therefore be described as three overarching tasks: (1) to design a localization system to work in the ROS environment, (2) research and implement each subsystem through modification of existing open source code, and (3) integrate all the separate components into a cohesive structure capable of delivering the necessary information. To efficiently manage the project, I divided the project into two distinct efforts: the computer vision (CV) algorithm development and the integration of a USB webcam with ROS. This allowed for in-tandem development.

I first researched computer vision algorithms, focusing on simple blob detection via HSV filters, which is a fundamental method in many segmentation algorithms. I utilized a simple piece of open source code that implements object tracking, and also allows for multiple object tracking. This code was when modified to filter not once, but twice, with two distinct sets of HSV values. Aside from changing various hardcoded values to better suit this application, I also implemented the localization and heading calculations. Next, I searched for an appropriate camera driver and example code to bridge between ROS sensor images and the Mat type images of OpenCV. Finally, after some fine-tuning of each individual subsystem, I integrated them into the localization node.

3 PRELIMINARIES

In this section we will briefly introduce the concept of HSV mapping.

3.1 HSV Mapping

HSV mapping is an alternative domain to characterize optical information. Unlike the Red Green Blue (RGB) space, which is traditionally and perhaps most commonly known, HSV is a cylindrical coordinate representation (as opposed to cubic). The HSV plane is defined by the “hue”, “saturation”, and “value”. Mapping from the RGB plane to the HSV plane is a common approach to blob detection (implemented in this project), as it allows for effects such as lighting to be taken into account.

4 METHODOLOGY

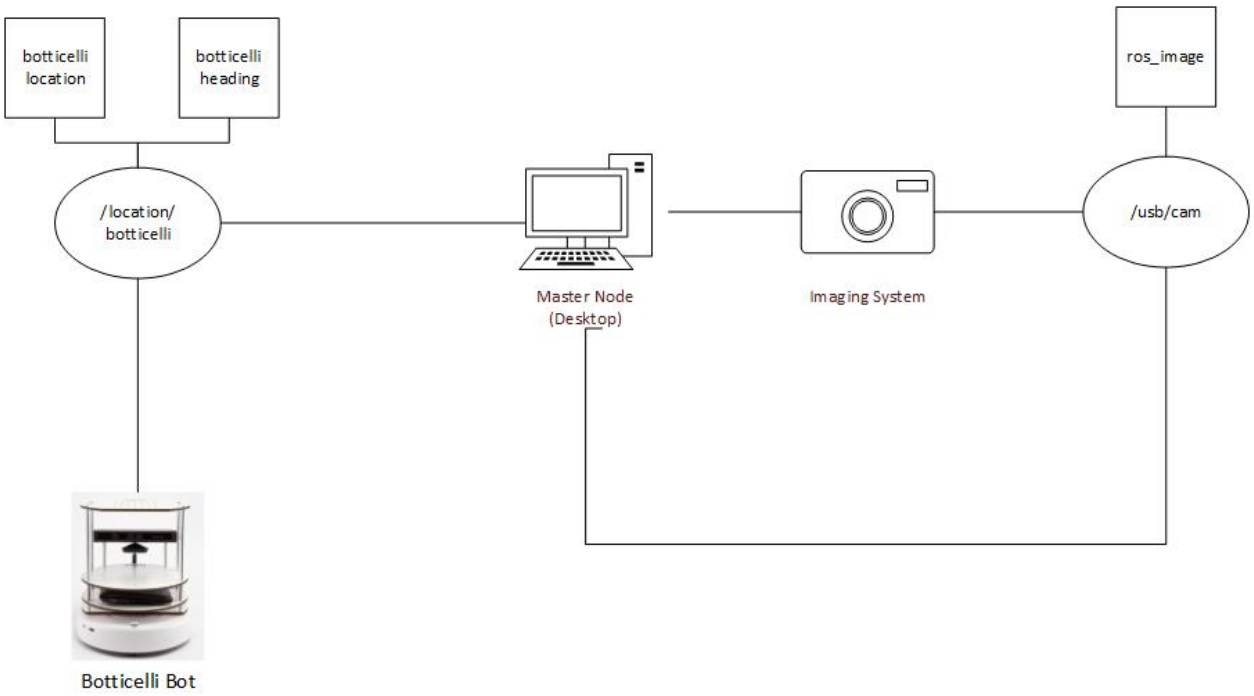
The following section describes the design of the localization module.

4.1 Proposed System (Scalable)

ROS supports the modularity required for a scalable localization package through its node and subscriber model. With this system, it is feasible to create a package that will locate a single TurtleBot. As the number of agents working increases, more nodes can be deployed to relay localization information. Different robots can be differentiated using different colored identification tags. A few examples can be seen in (4.2.4.1). Each robot would then have a code

base specifically tailored to the necessary HSV filter values required to isolate its individual identification tag, and each node will publish to a topic that is unique to each robot.

4.1.1 System Diagram



4.2 Code Overview

Covers the resources used in the code, where to find the code in its most current state, and high level descriptions of the algorithm.

4.2.1 Code Resources/Location

The computer vision algorithm is a slightly modified version of Kyle Hounslow's tutorial series on the topic. Launch files and OpenCV integration code modified from Siddhant Ahuja. Links to pertinent resources can be found in (6).

The code in its most current state can be found in the Appendix of this document, and will be uploaded to SVN.

4.2.2 High Level Description

At the highest level, the algorithm is contained within the localization node, which subscribes to information published by the camera node, and in turn publishes localization data and heading information (note that at this juncture, the code does not yet publish localization data, but this is soon to be implemented). The localization node then bridges to OpenCV, converting the ROS sensor image to a format that can be parsed using the build-in functions. The image is converted to the HSV plane (see 3.1), and then filtered according to user preset values into a binary matrix.

These filter values may need to be calibrated when the environment changes (for example, the lighting the room changes drastically, etc).

The function for finding contours in OpenCV is then applied to the binary image, and a matrix of objects is stored in vector form. Various filters then check to make sure there is not too much noise in the system, and that the object found falls within the appropriate and expected size range. After this is confirmed, the x and y moments of inertia of the object are calculated and used to find the centroid of the object. The Cartesian coordinates of the object are ultimately returned as the robot's location.

A second filter is also applied such that the second colored area of the identification pattern is mapped to its own binary matrix (utilizing its own HSV filter). After it has been identified through the contour and size filter method, and its centroid is calculated. A simple trigonometric operation is used to relate the main location and the pointer location into a heading for the robot.

These three data points – x location, y location, and heading – are published to the location topic, which each TurtleBot can subscribe to.

To run this module the following code should be executed:

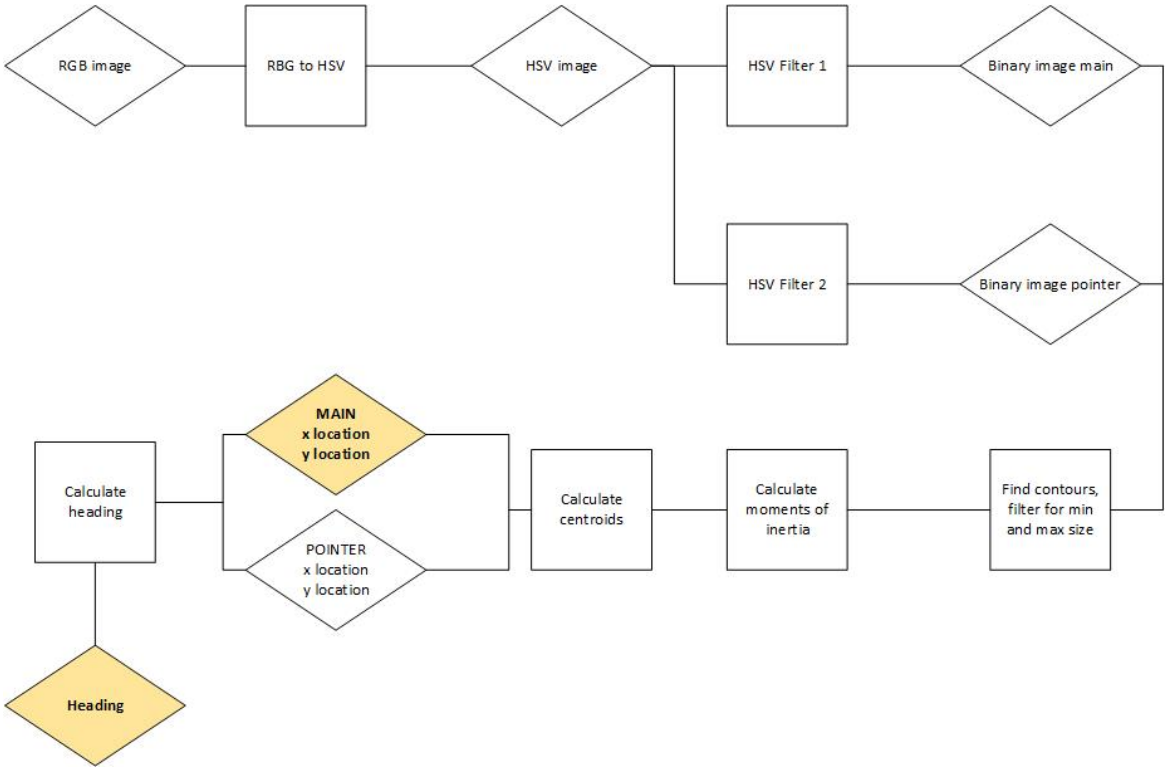
1. Running the camera node and viewing the image feed

```
In TERMINAL 1
roslaunch tutorialROSOpenCV uvcCameraLaunch.launch
In TERMINAL 2
rosrun image_view image_view image:=/camera/image_raw
```

2. Running the CV module

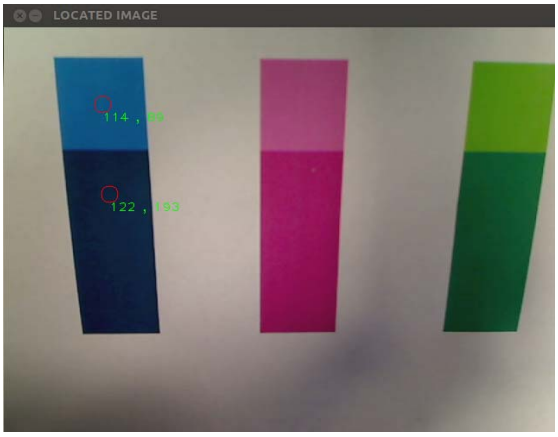
```
In TERMINAL 3
roscd tutorialROSOpenCV/src
rosrun tutorialROSOpenCV tutorialROSOpenCV
```

4.2.3 Software Diagram

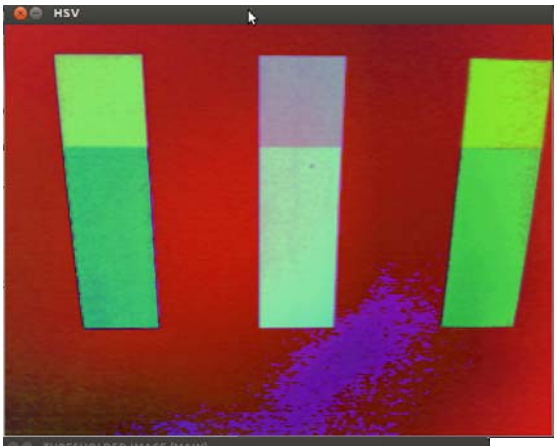


4.2.4 Visual Examples of Algorithm

This section includes various figures which serve to provide a visual representation of key algorithmic steps.

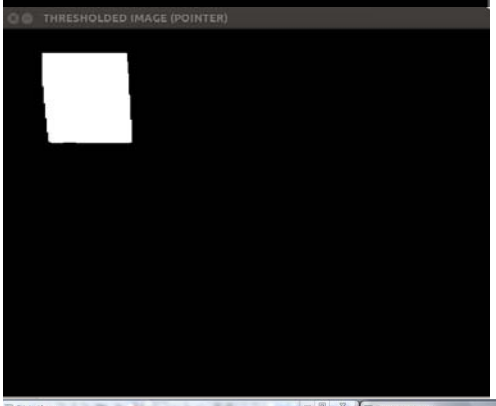


4.2.4.1
Example targets. The following steps will illustrate localization of the blue identifier on the far left. The same steps can of course be applied to the other targets with different filtering values.

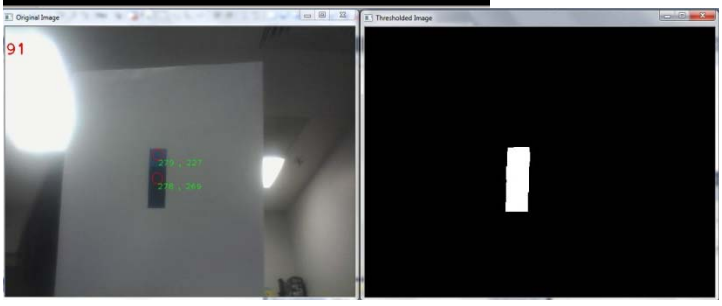


4.2.4.2
The resulting image after mapping to the HSV space.

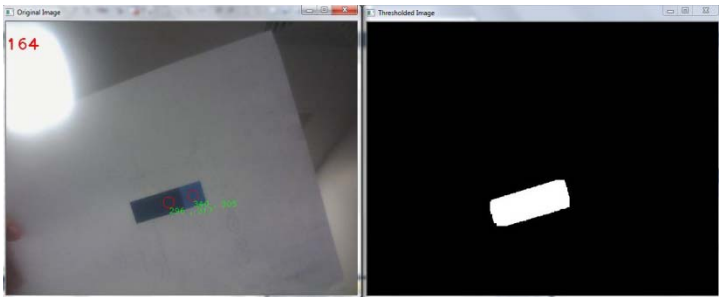
4.2.4.3
The thesholded, binary image. White indicates pixels that fall within the user defined HSV values. This is the thresholded image of the entire identification tag ("main"). The x and y locations of the centroid of this object is then saved as the x and y location of the target. We see this in the marker at 122,193 in (4.2.4.1)**.**



4.2.4.3
This is the thresholded image of the "pointer" component of the idenfication tag. The x and y locations of the centroid of this object is then saved as the x and y location of the pointer, and simple trignometry used to calculate the heading of the robot.



4.2.4.5
Example of heading calculation. These images are from the intial stages of protoyping (prior to integration of the CV algorithm into ROS). Headings are indicated in the upper left corner of the image.



5 CONCLUSIONS

This section discusses tips for others beginning to use ROS, as well as a discussion of the path forward for the project.

5.1 Tips/Pitfalls

The `uvc_camera` driver does not seem to allow for dynamic parameter settings. A work around to this issue is to use `guvcview` in the terminal. These settings appear to save.

Each terminal in ROS is its own “node”. Therefore, if the camera node is running in one terminal, another terminal tab *must* be opened to run other processes dependent on the camera node.

5.2 Path Forward

This quarter, I made significant progress on making a stand-alone module to provide localization and heading data to the ROS network. In terms of the bigger picture, we are on our way to having a ground truth to benchmark SLAM results against. The following sections address the path forward for this subproject, as well as the UC San Diego ROS Robot project.

5.2.1 Algorithm Robustness and Deployment

To account of the noise in the algorithm, a Kalman filter will be implemented next. This will be a node that subscribes to the location information published by the module discussed in this report, and republishes more accurate location information to a topic which can again be accessed by the robots.

Additionally, steps will be taken to make the CV algorithm more robust. This will require extensive testing to find the best filtering techniques. Calibration procedures may be instituted; more sophisticated shape detection algorithms may be incorporated to reinforce the results of blob detection.

Finally, the module should be deployed in the next quarter to provide localization data for multiple robots.

5.2.2 Coordinated Motion Deployment, Cooperative SLAM

As stated earlier in this report, of particular interest to the group is SLAM and coordinated motion deployment. After this localization module has been made more robust, it can be used to do some

preliminary experiments with multiple TurtleBots interacting by pulling their location information and autonomously deciding their behavior. It might also be very interesting to mount the webcam system on an aerial vehicle, such as a quadcopter.

Beyond this, the module should be used to verify information as the team moves forward with cooperative SLAM projects.

6 SOURCES

Launch files and ROS/OpenCV Bridge, Siddhant Ahuja

"Working with ROS and OpenCV"

http://siddhantahuja.wordpress.com/2011/07/20/working-with-ros-and-opencv-draft/

Object Tracking, Kyle Hounslow

"Tutorial: Real-Time Object Tracking Using OpenCV"

https://www.youtube.com/watch?v=bSeFrPrqZ2A

For a soft introduction to HSV: http://en.wikipedia.org/wiki/HSV

APPENDIX (CODE)

```
//Katherine Liu
//UCSD, Dynamics Systems and Controls
//main.cpp

//Modifed from objectTrackingTutorial.cpp written by  Kyle Hounslow 2013

//Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software")
//, to deal in the Software without restriction, including without
limitation the rights to use, copy, modify, merge, publish, distribute,
sublicense,
//and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

//The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

//THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
//FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
//LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
//IN THE SOFTWARE.

//Based in part off of Siddhant Ahuja's “Working with ROS and OpenCV”
tutorial, found here:http://siddhantahuja.wordpress.com/2011/07/20/working-
with-ros-and-opencv-draft/


//Includes all the headers necessary to use the most common public pieces of
the ROS system.
#include <ros/ros.h>
//Use image_transport for publishing and subscribing to images in ROS
#include <image_transport/image_transport.h>
//Use cv_bridge to convert between ROS and OpenCV Image formats
#include <cv_bridge/cv_bridge.h>
//Include some useful constants for image encoding. Refer to:
http://www.ros.org/doc/api/sensor_msgs/html/namespacesensor__msgs_1_1image__
encodings.html for more info.
#include <sensor_msgs/image_encodings.h>
//Include headers for OpenCV Image processing
#include <opencv2/imgproc/imgproc.hpp>
//Include headers for OpenCV GUI handling
#include <opencv2/highgui/highgui.hpp>
#include "std_msgs/Int32.h"
#include <sstream>
//Store all constants for image encodings in the enc namespace to be used
later.
namespace enc = sensor_msgs::image_encodings;
```

```
//Declare a string with the name of the window that we will create using OpenCV
where processed images will be displayed.
static const char WINDOW[] = "HSV";
static const char WINDOW2[] = "THRESHOLDED IMAGE (MAIN)";
static const char WINDOW3[] = "LOCATED IMAGE";
static const char WINDOW4[] = "THRESHOLDED IMAGE (POINTER)";

//for now, use global variables to return information - this should be changed,
but this allows for the robot to hold it's original position if no object is
found, rather than appearing to travel through time and space

//makes an myLocs array to save the "main" data to
int myLocs[3];
//make an myLocs2 array to save the "pointer" data to
int myLocs2[2];

//HSV Filter Values, set one (main target)
int H_MIN = 95;
int H_MAX = 135;
int S_MIN = 51;
int S_MAX = 256;
int V_MIN = 0;
int V_MAX = 137;

//HSV Filter Values, set two (pointer)
int H_MIN2 = 95;
int H_MAX2 = 135;
int S_MIN2 = 51;
int S_MAX2 = 256;
int V_MIN2 = 90;
int V_MAX2 = 150;

//size values
//minimum and maximum object area
const int MIN_OBJECT_AREA = 15*15;
//const int MAX_OBJECT_AREA = FRAME_HEIGHT*FRAME_WIDTH/1.5;
const int MAX_OBJECT_AREA = 50*50;
//maximum number of objects before declaring noise
const int MAX_NUM_OBJECTS = 100;

std::string intToString(int number){
    std::stringstream ss;
    ss << number;
    return ss.str();
}

void drawObject(int x,int y, cv::Mat &frame){
    cv::circle(frame,cv::Point(x,y),10,cv::Scalar(0,0,255));
    cv::putText(frame,intToString(x)+ " , " +
intToString(y),cv::Point(x,y+20),1,1, cv::Scalar(0,255,0));
}
//Use method of ImageTransport to create image publisher
image_transport::Publisher pub;
```

```
//This function dilates and errodes the image
void morphOps(cv::Mat &thresh){
    //create structuring element that will be used to "dilate" and "erode"
image.
    //the element chosen here is a 3px by 3px rectangle
    cv::Mat erodeElement = cv::getStructuringElement(cv::MORPH_RECT,
cv::Size(3,3));
    //dilate with larger element so make sure object is nicely visible
    cv::Mat dilateElement = cv::getStructuringElement(cv::MORPH_RECT,
cv::Size(8,8));
    cv::erode(thresh,thresh,erodeElement);
    cv::erode(thresh,thresh,erodeElement);
    cv::dilate(thresh,thresh,dilateElement);
    cv::dilate(thresh,thresh,dilateElement);
}
void trackFilteredObject(int myLocs[], cv::Mat threshold, cv::Mat HSV,
cv::Mat &cameraFeed, int mode)
{
    //Note that mode 1 indicates "main" position is desired; mode 2
indicates "pointer" position is desired
    cv::Mat temp;
    threshold.copyTo(temp);
    //these two vectors needed for output of findContours
    std::vector<std::vector<cv::Point> > contours;
    std::vector<cv::Vec4i> hierarchy;
    //find contours of filtered image using openCV findContours function
    findContours(temp,contours,hierarchy,CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPL
E);
    //use moments method to find our filtered object
    double refArea = 0;
    bool objectFound = false;
    if (hierarchy.size() > 0)
    {
       int numObjects = hierarchy.size();
       //if number of objects greater than MAX_NUM_OBJECTS we have a noisy
filter
       if(numObjects<MAX_NUM_OBJECTS)
           {
           //for (int index = 0; index >= 0; index = hierarchy[index][0]) {
               int index = 0;
               cv::Moments moment = moments((cv::Mat)contours[index]);
               double area = moment.m00;
               //if the area is less than 20 px by 20px then it is probably
just noise
               //if the area is the same as the 3/2 of the image size,
probably just a bad filter
               //we only want the object with the largest area so we safe a
reference area each
               //iteration and compare it to the area in the next iteration.
               if(area>MIN_OBJECT_AREA)
           {
                       if (mode==1)
           {
               myLocs[0] = (moment.m10/area);
```

```
                        myLocs[1] = (moment.m01/area);
                        objectFound = true;
            }
            else
            {
                myLocs2[0] = (moment.m10/area);
                        myLocs2[1] = (moment.m01/area);
                        objectFound = true;
            }
                }
                else objectFound = false;
             }
        if(objectFound ==true)
            {
                //draw object location on screen, if you're looking for the
main pointer
                    if (mode==1)
                    {
                            drawObject(myLocs[0],myLocs[1],cameraFeed);
                    }
            else
            {
            //calculate the heading
                myLocs[2] = atan(myLocs2[0]/myLocs2[1]);
                drawObject(myLocs2[0],myLocs2[1],cameraFeed);
            }
                  }
        } else putText(cameraFeed,"TOO MUCH NOISE! ADJUST
FILTER",cv::Point(0,50),1,2,cv::Scalar(0,0,255),2);
}


//This function is called everytime a new image is published
void imageCallback(const sensor_msgs::ImageConstPtr& original_image)
{
    //Convert from the ROS image message to a CvImage suitable for working
with OpenCV for processing
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
       //Always copy, returning a mutable CvImage
       //OpenCV expects color images to use BGR channel order.
       cv_ptr = cv_bridge::toCvCopy(original_image, enc::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
       //if there is an error during conversion, display it
       ROS_ERROR("tutorialROSOpenCV::main.cpp::cv_bridge exception: %s",
e.what());
       return;
    }
    //translate to HSV plane
    cv::Mat HSV;
    cv::Mat HSV2;
```

```
    cv::Mat threshold;
    cv::Mat threshold2;
    cvtColor(cv_ptr->image, HSV, cv::COLOR_BGR2HSV);
    cv::inRange(HSV, cv::Scalar(H_MIN, S_MIN, V_MIN), cv::Scalar(H_MAX,
S_MAX, V_MAX), threshold);
    cv::inRange(HSV, cv::Scalar(H_MIN2, S_MIN2, V_MIN2), cv::Scalar(H_MAX2,
S_MAX2, V_MAX2), threshold2);
    morphOps(threshold);
    morphOps(threshold2);

    //Display the image using OpenCV
    //cv::imshow(WINDOW, cv_ptr->image);

    //DISPLAY THE HSV IMAGE
    cv::imshow(WINDOW, HSV);
    //DISPLAY THE THRESHOLDED IMAGE (main)
    cv::imshow(WINDOW2, threshold);
    //DISPLAY THE THRESHOLDED IMAGE (pointer)
    cv::imshow(WINDOW4, threshold2);

    //TRACK THE FILTERED OBJECT
    trackFilteredObject(myLocs, threshold,HSV, cv_ptr->image, 1);
    trackFilteredObject(myLocs, threshold2, HSV2, cv_ptr->image, 2);

    //DISPLAY THE TRACKED OBJECT
    cv::imshow(WINDOW3, cv_ptr->image);

    //Add some delay in miliseconds. The function only works if there is at
least one HighGUI window created and the window is active. If there are
several HighGUI windows, any of them can be active.
    cv::waitKey(3);
    /**
    * The publish() function is how you send messages. The parameter
    * is the message object. The type of this object must agree with the
type
    * given as a template parameter to the advertise<>() call, as was done
    * in the constructor in main().
    */
    //Convert the CvImage to a ROS image message and publish it on the
"camera/image_processed" topic.
       pub.publish(cv_ptr->toImageMsg());
}
/**
* This tutorial demonstrates simple image conversion between ROS image
message and OpenCV formats and image processing
*/
int main(int argc, char **argv)
{
    /**
    * The ros::init() function needs to see argc and argv so that it can
perform
    * any ROS arguments and name remapping that were provided at the command
line. For programmatic
    * remappings you can use a different version of init() which takes
```

```
remappings
    * directly, but for most command-line programs, passing argc and argv is
the easiest
    * way to do it.  The third argument to init() is the name of the node.
Node names must be unique in a running system.
    * The name used here must be a base name, ie. it cannot have a / in it.
    * You must call one of the versions of ros::init() before using any
other
    * part of the ROS system.
    */
       ros::init(argc, argv, "image_processor");

    /**
    * NodeHandle is the main access point to communications with the ROS
system.
    * The first NodeHandle constructed will fully initialize this node, and
the last
    * NodeHandle destructed will close down the node.
    */
       ros::NodeHandle nh;

    //Create an ImageTransport instance, initializing it with our
NodeHandle.
       image_transport::ImageTransport it(nh);
    //OpenCV HighGUI call to create a display window on start-up.
       cv::namedWindow(WINDOW, CV_WINDOW_AUTOSIZE);
    /**
    * Subscribe to the "camera/image_raw" base topic. The actual ROS topic
subscribed to depends on which transport is used.
    * In the default case, "raw" transport, the topic is in fact
"camera/image_raw" with type sensor_msgs/Image. ROS will call
    * the "imageCallback" function whenever a new image myLocsives. The 2nd
argument is the queue size.
    * subscribe() returns an image_transport::Subscriber object, that you
must hold on to until you want to unsubscribe.
    * When the Subscriber object is destructed, it will automatically
unsubscribe from the "camera/image_raw" base topic.
    */
       image_transport::Subscriber sub = it.subscribe("camera/image_raw", 1,
imageCallback);
    //OpenCV HighGUI call to destroy a display window on shut-down.
       cv::destroyWindow(WINDOW);
    /**
    * The advertise() function is how you tell ROS that you want to
    * publish on a given topic name. This invokes a call to the ROS
    * master node, which keeps a registry of who is publishing and who
    * is subscribing. After this advertise() call is made, the master
    * node will notify anyone who is trying to subscribe to this topic name,
    * and they will in turn negotiate a peer-to-peer connection with this
    * node.  advertise() returns a Publisher object which allows you to
    * publish messages on that topic through a call to publish().  Once
    * all copies of the returned Publisher object are destroyed, the topic
    * will be automatically unadvertised.
    *
```

```
    * The second parameter to advertise() is the size of the message queue
    * used for publishing messages.  If messages are published more quickly
    * than we can send them, the number here specifies how many messages to
    * buffer up before throwing some away.
    */
       pub = it.advertise("camera/image_processed", 1);

    /**
    * In this application all user callbacks will be called from within the
ros::spin() call.
    * ros::spin() will not return until the node has been shutdown, either
through a call
    * to ros::shutdown() or a Ctrl-C.
    */
       ros::spin();
    //ROS_INFO is the replacement for printf/cout.
    ROS_INFO("tutorialROSOpenCV::main.cpp::No error.");
}
```