

# Computer Vision Enabled Localization and Deployment of Mobile Robot Networks

Katherine Liu  
University of California, San Diego  
Spring 2014

## 0 ABSTRACT

This paper summarizes my continued individual efforts to provide a computer vision enabled localization module in support of the Robot Operating System (ROS) Robot Project at UC San Diego. The tracking algorithm generated in Winter 2014 was tested, validated, and further integrated into ROS. Finally, the node was successfully scaled to three robots.

## 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 to access 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 integrating

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**

My contributions this quarter are continuations of my work of the previous quarter. For details regarding prior work, please refer to my Winter 2014 report.

The major contributions I have developed this quarter are primarily aimed at increasing robustness, flexibility, and integration into other functions of ROS. For example, I improved the tracking algorithm by providing a calibration function, which allows for dynamic readjustments. This is important for scalability (users can manually set tracking for new agents) as well as repeatability (users can adjust for environmental changes in lighting, etc). To facilitate possible integration into RVIZ and Gazebo, the tracking algorithm now mimics the ROS-included AMCL (a Monte-Carlo particle filter) node so that it the localization node publishes on all the topics expected from traditional ACML localization. This includes the integration of the robot's transform into the global transform tree, so that RVIZ can be used to visualize robot locations. The code was then adjusted to allow for several launch files to publish with different topic headings, verifying the scalability of the algorithm. Finally, I wrote and tested a function christened "centerTurtlebot", which simply uses a proportional controller to move the turtlebot to the center of the camera frame. This exercise was an important step forward towards both integrating the overhead camera localization and autonomous path planning, as it is the first successful instance of the turtlebot receiving actionable intelligence from the overhead camera, making some decision based on that information, and deciding and executing an action.

It is also worth noting the preliminary explorations into Android-leveraged ROS that were undertaken in the last few weeks of the quarter. As will be discussed later, Android smartphones could greatly extend the capabilities of the lab for potentially marginal start-up costs.

## **3 PRELIMINARIES**

In this section, several basic topics crucial for multi-agent deployment are discussed.

### **3.1 Transforms**

Transforms, supported under the tf package in ROS, are incorporated in a system which allows for the origin of several different bodies to be related to each other [1]. Figure 1

shows the different *reference frames* of each respective robot or carot related by *transforms*. Transforms exist between any two points in the graph which are connected by a tf definition. For example, while this is a not a theoretically “complete” graph, a transform path exists from every reference frame to another, and therefore the relationship between any two frames can be computed. The world or map frame is generally considered to be stationary. ROS handles transforms with transform “listeners” and “broadcasters” with tf “child” and “parent” frames defined in sending messages. It is important to note tf messages are *not* handled through traditional ROS topics.

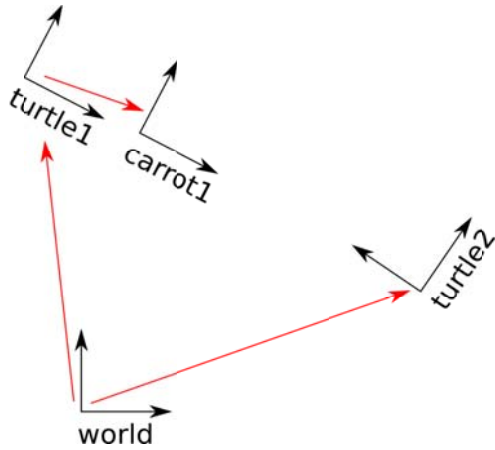


Figure 1: Example representation of transforms in ROS [2]

### 3.2 Namespaces

Namespaces in ROS are important when attempting to deploy several robots at once. Namespaces effectively allow for different robots to listen and act upon their own information. For example, good distributed structure allows for robot\_1 to listen to the robot\_1/odom topic for its odometry information, and robot\_2 to listen to the robot\_2/odom topic for its different, presumably unique odometry information. This functionality is generally implemented through launch files. See [3] for further documentation.

## 4 METHODOLOGY

In this section, each development in regards to robot tracking and deployment are explored in greater depth. For methodology and more detail on the actual tracking algorithm, refer to my Winter 2014 report. Code can be found in the SVN repository under lab/in-progress/localization-mounted-camera. Localization code is under the sub-folder /blueBot



Figure 2: Three turtlebots with different identifiers

## 4.1 Tracking Calibration

To enable on-the-fly, real-time calibration of HSV filter values, tracking slider functionality has been added to this iteration of code. This allows for immediate adjustment, especially as the environment can change subtly depending on the time of day.

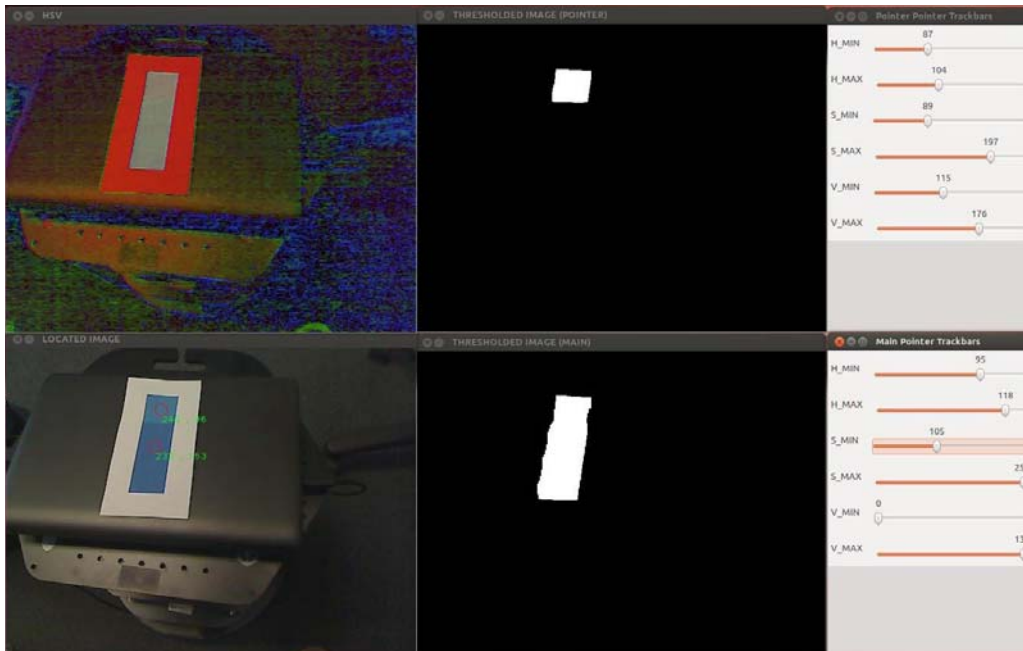


Figure 3: Thresholding of camera feed based on sliders

However, this type of calibration can be unwieldy and very undesirable when rapid debugging is desired. Therefore, an effort has been made to modularize this code, and

allow the user to specify when the calibration capability is desired. The following is an example of a dedicated launch file which will run the node in calibration mode so that filter values can be obtained. The user now has the option to specify whether the HSV representation and or thresholding and sliders are desired. These values then must be re-input into the code, as they are predefined values. The rationale for this is that filter values should not change drastically; however if it is determined that more flexibility is required, it is quite reasonable to allow the user to define filter values in the parameters of the launch file as well.

#### 4.1.1 Launch file (calibration)

localization\_calibration.launch

```
<launch>
  <!-- Camera -->
  <include file="$(find blueBot)/launch/uvvCameraLaunch.launch"/>

  <node pkg="blueBot" type="blueBot"
    args="/donatellobot" name="localization_green" output="screen" />
  <!--Set calibration settings-->
  <param name="calibration_mode" value="1" type="int"/>
  <param name="show_hsv" value="1" type="int"/>
</launch>
```

#### 4.1.2 Sample calibration values

Calibration values for the pink, green, and orange tracking identifiers are provided below for lab reference. They should serve well for general reference points to assist in fine-tuning.

Identifier	Main Filter Values	Pointer Filter Values
<b>Pink</b>	H 136-256 S 88-256 V 167-256	H 116-256 S 129-256 V 116-256
<b>Green</b>	H 45-103 S 86-256 V 88-216	H 44-96 S 106-256 V 72-256
<b>Orange</b>	H 5-39 S 89-175 V 203-245	H 5-39 S 146-205 V 210-242

## 4.2 Multi-agent Tracking

In the previous paper, claims were made to the scalability of the system. This quarter that functionality has been verified. To achieve this, a special argument was defined at the launch of a single node that defines the namespace of the robot (this was tested with the Botticelli, Donatello, and Raphael robots). Multi-agent tracking is crucial to achieving the goals of the Distributed Robotics group.

Allowing the localization for each robot to inhabit a specific namespace creates a structure such that each individual robot can access their locations and headings to make decisions based on this information. Furthermore, it also enables robots to subscribe to the localization data of other agents in their environments, which is necessary for path planning. Because tracking one robot requires one node, this algorithm is easily scalable, as the launch file simply needs to be updated to accommodate any number of robots, so long as filter values for the robot have been defined.

Figure 4 is a brief systems diagram of the transform tree, where ovals represent transform frames and arrows show broadcasted relationships. Each instance of the localization node provides a transform from it's calculated center of mass to the origin of the map. Figure 5 shows nodal relationship where ovals are nodes and arrows represent topics by which information is communicated. Figures 6-7 provide screenshots of the algorithm running for three robots. Each robot is also tagged with their name for user friendliness.

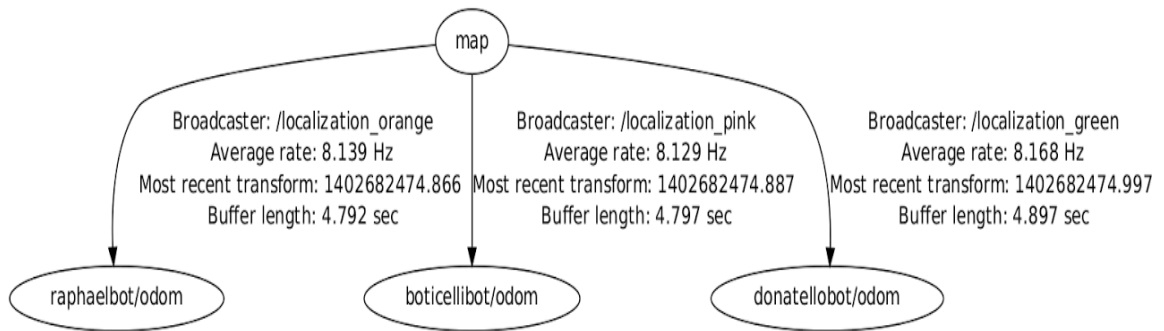


Figure 4: Multi-agent tracking transform graph



Figure 5: Node graph with localization node running

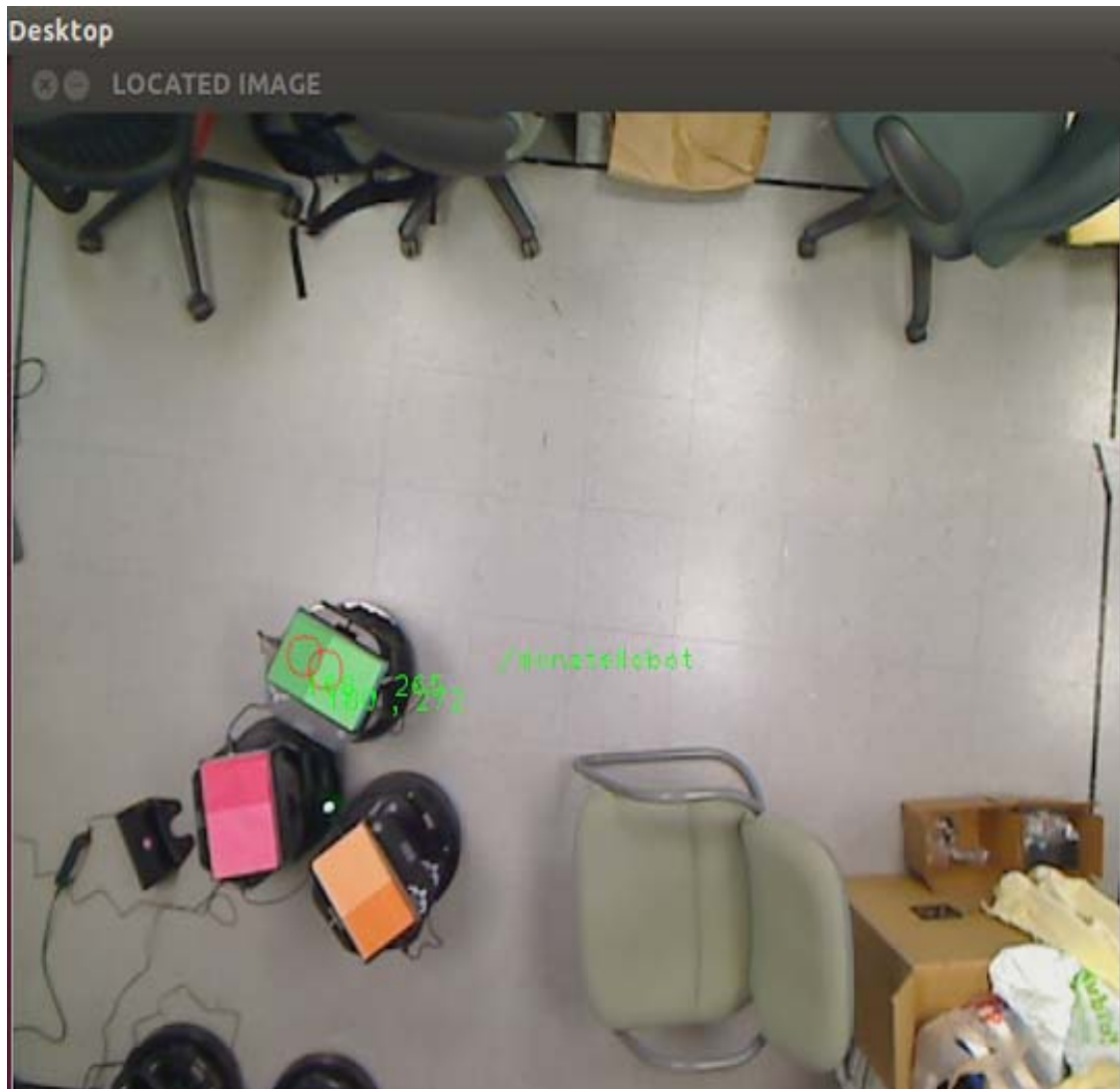


Figure 6: Tracking visualization of DonatelloBot, as identified using the green marker

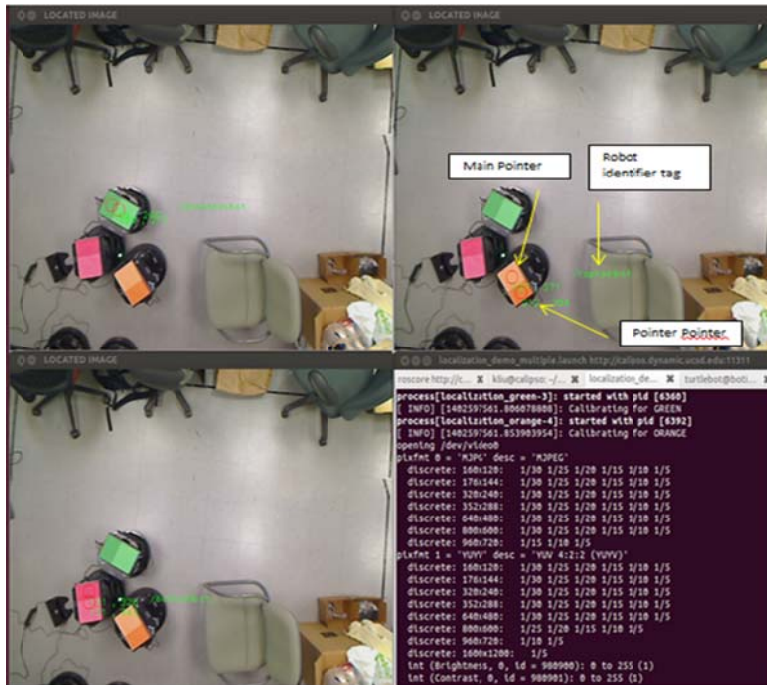


Figure 7: Tracking visualization of three robots

#### 4.2.1 Pseudo-code

```

Initialize node handle
Check launch file parameter for turtlebot name
Set filter values according to turtlebot name

if (calibration_mode==1)
    create trackbars

while (images are published from camera)
    if (calibration_mode==1)
        publish thresholded images
    if (show_hsv==1)
        publish HSV transformation
    //normal object tracking algorithm

```

#### 4.2.2 Launch file

localization\_demo\_multiple.launch

```

<launch>
  <!-- Camera -->
  <include file="$(find blueBot)/launch/uvvCameraLaunch.launch"/>

  <!-- Localization -->
  <node pkg="blueBot" type="blueBot"

```



```

args="/boticellibot" name="localization_pink" output="screen" />

<node pkg="blueBot" type="blueBot"
args="/donatellobot" name="localization_green" output="screen" />

<node pkg="blueBot" type="blueBot"
args="/raphaelbot" name="localization_orange" output="screen" />

</launch>

```

### 4.3 AMCL Substitution and Transform Definitions

To facilitate the integration of this computer vision enabled localization, the AMCL node [4] output topics were mimicked. This allows for the localization node to replace the AMCL node that is traditionally used in applications, usually by a simple replacement in launch file specifications. The localization node has been modified to publish transforms using the transform broadcaster, and also on the following topics (under the namespace of each robot that is tracked):

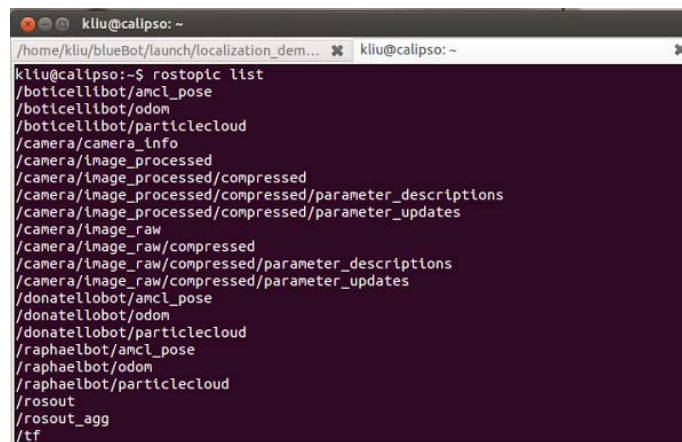
```

/odom

/amcl_pose

/particlecloud

```



```

kliu@calipso: ~
/home/kliu/blueBot/launch/localization_dem... x kliu@calipso: ~
kliu@calipso:~$ rostopic list
/boticellibot/amcl_pose
/boticellibot/odom
/boticellibot/particlecloud
/camera/camera_info
/camera/image_processed
/camera/image_processed/compressed
/camera/image_processed/compressed/parameter_descriptions
/camera/image_processed/compressed/parameter_updates
/camera/image_raw
/camera/image_raw/compressed
/camera/image_raw/compressed/parameter_descriptions
/camera/image_raw/compressed/parameter_updates
/donatellobot/amcl_pose
/donatellobot/odom
/donatellobot/particlecloud
/raphaelbot/amcl_pose
/raphaelbot/odom
/raphaelbot/particlecloud
/rosout
/rosout_agg
/tf

```

Figure 8: Result of running `rostopic list` after localization node has been started for three robots

Note that the `/particlecloud` topic has only one entry in its array of possible locations, rather than a multitude of pose estimates; this is because the camera data is taken to be ground truth, and therefore the "true" pose of the robot. Figures X-Y are examples of the topics shown in Figure 8 being utilized to visualize robot locations within the ROS software RVIZ. Note that the localization node is run for three robots; the robot not in the

frame is appropriately located outside of the map.



Figure 9: Camera feed

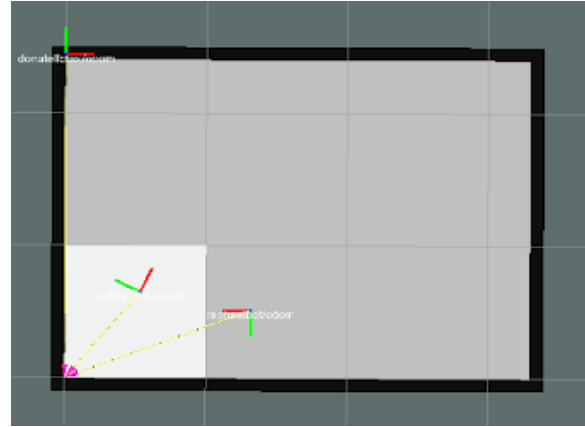


Figure 10: Robot transform representations in RVIZ

## 4.4 Preliminary Deployment

This quarter, significant steps were taken to achieve deployment. First, a simple proportional controller was written to move the turtlebot to the center of the camera feed.

### 4.4.1 Pseudocode (centerTurtle)

```
define distance_tolerance
define angle_gain
define distance_gain

create a transform listener
get my_xLocation, my_yLocation, my_heading

while (the transform exists)
    distance_delta = sqrt((my_xLocation-x_goal)^2+(my_yLocation-y_goal)^2)
    heading_delta = atan((my_yLocation-y_goal) , (my_xLocation-y_goal))
    create a command message = cmd

    if (abs(distance_delta) > distance_tolerance)
        set cmd linear velocity = distance_delta*distance_gain
        set cmd angular velocity =heading_delta*angle_gain
    else
        set cmd linear velocity and angular velocity = 0
    send the cmd
```

### 4.4.2 First tests

Figures 11-14 show the path and tracking of the robot at different time steps. Note that  $t_4$  is also the steady-state response of the system, as the robot has come within its threshold distance of .35 meters of its goal. Figures 15-16 show the difference in the HSV

interpretation of the environment with the blinds opened and closed.



Figure 11: centerTurtle, t\_1

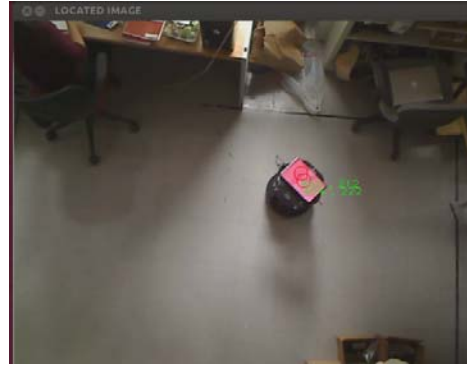


Figure 12: centerTurtle, t\_2

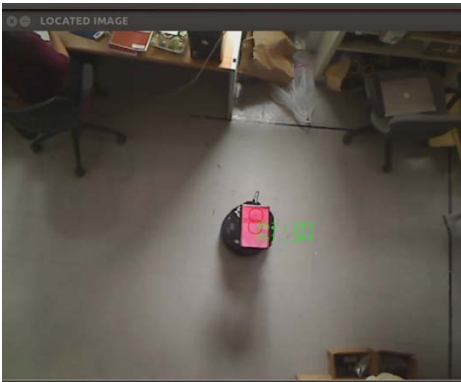


Figure 13: centerTurtle, t\_3

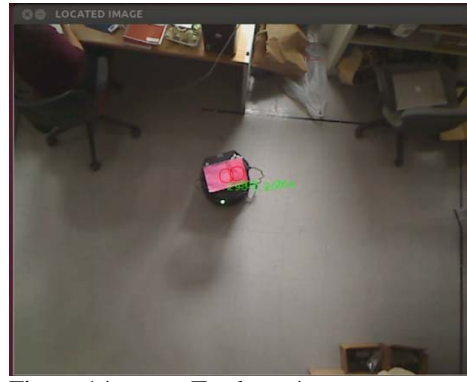


Figure 14: centerTurtle, t\_4

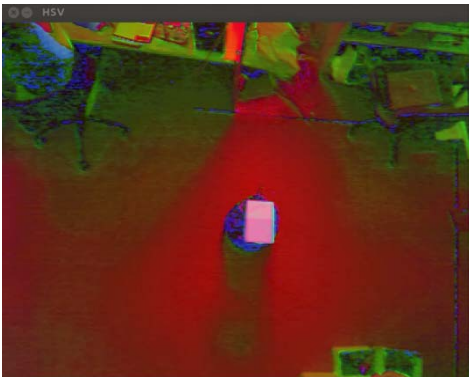


Figure 15: HSV space with blinds open

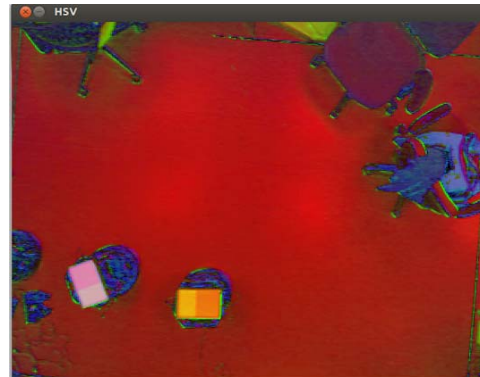


Figure 16: HSV space with blinds closed

#### 4.4.3 Problems and countermeasures

This experiment was rife with problems, which was expected due to the primitive nature of the control algorithm, but it did successfully validate sensor node integration. Here we briefly enumerate various issues and solutions as encountered in during the debugging of code. Note that this section is included here, rather than in the conclusion, for the sake of continuity.

<i>Problem</i>	<i>Countermeasure</i>	<i>Status</i>
<u>Filtering</u> HSV environment is non-uniform, leading to unsatisfactory filtering results	(1) Close blinds; regulate environment	Implemented with moderate success. Figures 15-16 indicate greater uniformity in HSV environment is achieved. Recommend further environmental control.
	(2) Use Extended <u>Kalman</u> Filter (EKF)	Implementation begun. Source [5] must be altered to account for individual transforms.
	(3) Incorporate adaptive filtering by extrapolating what filter values should shift to at different areas of the environment, based on HSV space ranges.	Investigation stages. See source [6].
<u>Transform syncing</u> Lack of transform with exact time stamp causes the algorithm to attempt to see into the "future"	(1) Code in a catch-and-wait control structure; the <u>turtlebot</u> will wait until the transform it desires is available, and then act.	Completed, problem completely fixed
<u>Controller</u> Controller is extremely unsophisticated, and lacks situational awareness (will exit camera frame)	(1) Implement two-tiered control loop structure; if the robot is too close to the camera frame edges, stop linear motion and adjusted heading to appropriate direction.	Coded; requires testing.
	(2) Add integral control	Not implemented.

## 4.5 Android-Leveraged ROS

In this section, the potential of integration of Android devices is explored.

### 4.5.1 Preliminaries

Several Android applications have already been created and are readily available on the Android Marketplace [7,8]. These apps allow devices to connect wirelessly to the ROS system running on their network, and publish sensor information- such as camera feed, IMU, etc. Using devices like this is particularly interesting because of their innate accessibility and ubiquity. Leveraging these resources, both a Nvidia Tegra Note tablet and Moto G smartphone were connected to ROS on Mencia.



Figure 17: Nvidia Tegra Note 7.0 acting as camera node on Menica

Both these low end devices have 5MP sensors, as compared to the 2MP webcam currently being used in the lab. Essentially, the conclusion is that devices such as smartphones are inherently more versatile than single-purpose hardware, especially when one considers the additional potential sensing and computing capabilities they have.

However, due to the inherent nature of software interface, image quality will generally be limited to 1080p. This is because we expect to rely on the live image feed, which due to hardware bandwidth limitations, is generally capped at 1080p. Various Android cameras have sensors capable of 10+ MP pictures, but for single-image capture only. Therefore, to truly leverage such sensors, a new sensor driver would have to be written that enabled triggered image capturing. Therefore, while the camera sensor specs are included in the following suggestions, I recommend purchase based on computing power, rather than camera capabilities.

#### 4.5.2 Purchase Recommendations

Three phones are compared in the table below: Google Nexus 5, Samsung Galaxy S5 Active, and HTC One (M8) [9]. Based on the specifications below, where the cells in bolded green text indicated features that are highly desirable, I suggest purchasing the Nexus 5. Criteria include weight, camera, and processing power. Furthermore, as the Nexus line is Google's development line, it is most likely to get quick system updates and patches.

SPECS	Google Nexus 5	Samsung Galaxy S5 Active	HTC One (M8)
OS	Android (4.4.2, 4.4)	Android (4.4.2)	Android (4.4) HTC Sense 6 UI
Dimensions	5.43 x 2.72 x 0.34 inches (137.84 x 69.17 x 8.59 mm)	5.72 x 2.89 x 0.35 inches (145 x 73 x 9 mm)	5.76 x 2.78 x 0.37 inches (146.36 x 70.6 x 9.35)
Weight	4.59 oz (130 g)	6 oz (170 g)	5.64 oz (160 g)
Camera	8 megapixels	16 megapixels	4 megapixels, Duo camera
Camera sensor size	1/3.2"		1/3"
Features	Back-illuminated sensor (BSI), Autofocus, Touch to focus, Optical image stabilization, Face detection, ISO control, white balance presets, Burst mode, Digital zoom, Geo tagging, High Dynamic Range mode (HDR), Panorama, Scenes, Self-timer	Autofocus, Touch to focus, Face detection, Smile detection, Burst mode, Digital zoom, Geo tagging, High Dynamic Range mode (HDR), Panorama	Back-illuminated sensor (BSI), Autofocus, Manual focus, Digital image stabilization, Face detection, Smile detection, Exposure compensation, ISO control, White balance presets, Burst mode, Geo tagging, High Dynamic Range mode (HDR), Panorama, Macro mode, Effects
Camcorder	1920x1080 (1080p HD) (30 fps)	3840x2160 (4K) (30 fps), 1920x1080 (1080p HD) (30 fps)	1920x1080 (1080p HD) (60 fps), 1280x720 (720p HD)
Features	Optical image stabilization, Video light	Picture-taking during video recording, Video sharing	High Dynamic Range mode (HDR), Video light, Video sharing
System chip	Qualcomm Snapdragon 800 MSM8974	Qualcomm Snapdragon 801	Qualcomm Snapdragon 801
Processor	Quad core, 2260 MHz, Krait 400	Quad core, 2500 MHz, Krait 400	Quad core, 2300 MHz, Krait 400
Graphics processor	Adreno 330	Adreno 330	Adreno 330



<b>System Memory</b>	2048 MB RAM	2048 MB RAM	2048 MB RAM
<b>Built-in storage</b>	32 GB	16 GB	32 GB
<b>Maximum User Storage</b>		11 GB	24 GB
<b>Storage expansion</b>		microSD, microSDHC, microSDXC up to 128 GB	microSD, microSDHC, microSDXC up to 128 GB
<b>Talk time</b>	17.00 hours	29.00 hours	
<b>Stand-by time</b>	12.5 days (300 hours)	20.0 days (480 hours)	
<b>Talk time (3G)</b>			20.00 hours
<b>Stand-by time (3G)</b>			20.7 days (496 hours)
<b>Capacity</b>	2300 mAh	2800 mAh	2600 mAh
<b>Not user replaceable</b>	Yes		Yes
<b>Wireless charging</b>	Built-in		

## 5 CONCLUSIONS

This section includes general troubleshooting tips for working with ROS, and a suggested path forward.

### 5.1 Troubleshooting/Tips

If camera exposure leads to a dark image, run `gview` and set exposure to about 81.

When using OpenCV, clear all new `cv::Mat` objects, otherwise image noise will remain.

Use the calibration function to find values for filter; hardcode new values into `main.cpp`.

The turtlebot expects commands not on `/cmd_vel`, as many sources suggest, but on

```
/mobile_base/commands/velocity.
```

## 5.2 Path forward

There are several interesting ways that Android devices could be integrated into some efforts the lab is already pursuing. For example, my localization software can easily be expanded to work using a smartphone. Furthermore, due to the portability and wireless nature of these devices, one could be strapped to the bottom of a quadcopter, and used to provide localization to Turtlebots in more interesting environments. I believe that this functionality could be immediately implemented with manual flight. They could also perhaps be used in computer vision enabled cooperative SLAM, using a sensor fusion of IMU and camera data to creating local maps that they then send back to the master computer for map-merging.

To continue increasing the robustness of the localization node, an EKF should be implemented. As mentioned briefly in a prior section, ROS-supported functionality exists, however clever transform manipulations may be required to leverage the resource.



## 6 SOURCES

- [1] ROS: tf Information. <<http://wiki.ros.org/tf>>
- [2] ROS: tf Tutorial.  
<<http://wiki.ros.org/tf/Tutorials/Adding%20a%20frame%20%28C%2B%2B%29>>
- [3] ROS: Namespaces. <<http://wiki.ros.org/Names>>
- [4] ROS: amcl Node Information. <<http://wiki.ros.org/amcl>>
- [5] ROS: robot\_post\_ekf Node Information. <[http://wiki.ros.org/robot\\_pose\\_ekf](http://wiki.ros.org/robot_pose_ekf)>
- [6] GPU Conference: Adaptive Filtering. <<http://on-demand.gputechconf.com/gtc/2014/presentations/S4328-object-tracking-nonuniform-illumination-conditions.pdf>> \_
- [7] ROS Android Sensors Driver.  
<[https://play.google.com/store/apps/details?id=org.ros.android.sensors\\_driver](https://play.google.com/store/apps/details?id=org.ros.android.sensors_driver)>
- [8] ROS Camera Viewer.  
<[https://play.google.com/store/apps/details?id=org.ros.android.android\\_camera\\_viewer](https://play.google.com/store/apps/details?id=org.ros.android.android_camera_viewer)>
- [9] Android Camera Spec Review.  
<<http://www.phonearena.com/phones/compare/Google-Nexus-5,Samsung-Galaxy-S5-Active,HTC-One-M8/phones/8148,8665,8242>>