Massachusetts Institute of Technology

Robotics: Science and Systems I (6.141/16.405)

Lab 4: The Robot Operating System (ROS) and Visual Servoing

Distributed: Monday 2/25/2013, 3pm

Checkpoints: Wednesday 2/27/2013 and Monday 3/4/2013 in Lab

Wiki Materials Due, and Briefings: Wednesday 3/6/2013, 3pm

Objectives

In lecture, you have heard about sensing generally, and about camera sensors specifically. You were also introduced to the Robot Operating System (ROS) for planning and robot control. ROS allows you to abstract away most of the implementation-dependent details inherent in realizing a robot system that incorporates sensing, planning, navigation and control, freeing you to focus on higher-level issues when crafting algorithms that reason about your robot's observations of its environment and internal state.

Your objectives in this lab are to:

- Become familiar with the structure and use of ROS;
- Implement on-line digital image acquisition, along with an initial set of low-level image processing operators including blob detection and blob size and centroid estimation;
- Use the features extracted by your operators to aid in deciding how to control the robot, in this case to perform "visual servoing," by following a colored ball;
- Gain experience assessing your system's operating assumptions and robustness. You will write up the lab along with a discussion of your design assumptions and the conditions under which your implementation will likely fail.

Materials

For this lab you should have:

- One LogiTech QuickCam (spec. sheet on the RSS site), ball-joint mount and mount plate
- Three 1/4-20 screws and two sliders

Time Accounting and Self-Assessment:

Make a dated entry called "Start of Visual Servo Lab" on your Wiki's Self-Assessment page. Before doing any of the lab parts below, assign a number to describe your proficiency **as of the start of the lab**: 1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert

- **Java**: How proficient are you at programming in Java?
- **ROS**: How proficient are you at using ROS?
- **Image Features and Visual Servoing**: How proficient are you at extracting low-level features from digital images, and using these features for visual servoing?
- **Assumptions and Failure Modes**: How proficient are you at characterizing the assumptions inherent in a system that you have designed, and clearly stating the likely failure modes of the system?

1 The Robot Operating System (ROS)

ROS is an open-source, meta-operating system for your robot. ROS provides an inter-process communication (IPC) system that allows for data exchange among several processes running on one or more physical computers. We will use ROS to connect the different software modules we develop.

In addition to IPC, ROS provides hardware abstraction, implementation of commonly-used functionality, and tools for obtaining, building, writing, debugging, and running code across multiple computers. The ROS community has developed an extensive library of perception, planning and control algorithms.

At runtime, ROS processes form a peer-to-peer network (graph) of processes that communicate through the ROS infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

1.1 ROS Filesystem Level

Here are some ROS resources that you encounter on disk:

- Packages: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
- Manifests: Manifests (manifest.xml) provide metadata about a package, including its license information and dependencies, as well as language-specific information such as compiler flags.
- Stacks: Stacks are collections of packages that provide aggregate functionality, such as a "navigation stack." Stacks are also how ROS software is released and have associated version numbers
- Message (msg) Types: Message descriptions, stored in `my_package/msg/MyMessageType.msg`, define the data structures for ROS messages.
- Service (srv) types: Service descriptions, stored in `my_package/srv/MyServiceType.srv`, define the request and response data structures for services in ROS.

1.2 Peer-to-Peer Network of ROS Nodes (Processes)

- Nodes: A node is a software process that performs some computation. A robot control system will usually include many interconnected nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on.
- Master: The ROS Master provides name registration and lookup to the rest of the peer-to-peer network. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- Parameter Server: The Parameter Server allows data to be stored by key in a central location. It's part of the Master, and both are started by roscore.
- Message: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- Topics: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the communication channel for particular messages. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think

of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are sending (or trying to receive) the right type.

- Services: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.
- Bags: Bags are a format for saving ROS message data (e.g. sensor data). You can "replay" a bag file to recreate the sequence of messages that were collected during a live run of your robot. This is very useful for testing and debugging.

1.3 More on ROS Messages

We **strongly recommend** that you review the tutorials below for creating a custom ROS message, publishing it to a topic, and subscribing to that topic (no need to type URLs; links are in the PDF):
`http://www.ros.org/wiki/ROS/Tutorials` and
`http://mirror.umd.edu/roswiki/attachments/Documentation/ROScheatsheet.pdf`

1.4 Useful ROS Commands

- `rostopic list` Execute this command to show all available publications.
- `rostopic echo [topic]` Execute this command to print out a particular topic. For example, `rostopic echo /rss/Encoder` would print the current encoder readings.
- `roswtf` Performs a series of basic checks on your system's configuration.
- `rxgraph` Displays a graph showing publications and subscriptions.
- `rosnode list -a` Display a list of nodes and the machine they are running on.
- `rosnode kill [nodename]` Terminate a node. This does not always work, so you may need to restart `roscore` as well.

2 Useful Linux Commands

2.1 What processes are running?

```
ps -ef | grep -e regular-expression
```

The `ps` command lists all running processes. Its output is being sent as the input to `grep` which reports lines in its input contain `query-string`.

2.2 How do I kill a process?

```
sudo killall -9 process-name
```

This command will force-kill the running process. Use this only if the process is unresponsive. Process name can be replaced by process ID number (PID), which can be found via `ps` above (second column).

2.3 How do I inspect the END of a (growing) log file?

```
tail logfilename
```

Use this to look at the last few lines of a process's logfile. Use `tail -f` to "follow" the final lines of a file as it grows.

2.4 How do I learn more about specific linux commands?

```
man command
```

This command will bring up the command's manual page. It will list the options and expected arguments. Alternatively, many commands will give you this information if you run

```
<command> --help
```

2.5 How do I find the IP address of a machine?

Type `ifconfig` to show information about your machine's network configuration. The output will look something like the following. Notice that there are three network adapters: `eth0` is typically the wired Ethernet port, `lo` is the loopback adapter, and `wlan0` is typically the wireless adapter. The `inet addr` field shows the IPv4 Ethernet address.

```
rss_staff@RSSnetbook6:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr c8:0a:a9:93:0f:96
          inet addr:192.168.237.80  Bcast:192.168.237.255  Mask:255.255.255.0
          inet6 addr: fe80::ca0a:a9ff:fe93:f96/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:21074 errors:0 dropped:0 overruns:0 frame:0
          TX packets:201 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:22196056 (22.1 MB)  TX bytes:22339 (22.3 KB)
          Interrupt:16

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:170 errors:0 dropped:0 overruns:0 frame:0
          TX packets:170 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:13072 (13.0 KB)  TX bytes:13072 (13.0 KB)

wlan0     Link encap:Ethernet  HWaddr 20:7c:8f:07:f9:3f
          inet addr:192.168.0.50  Bcast:192.168.0.127  Mask:255.255.255.128
          inet6 addr: fe80::227c:8fff:fe07:f93f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:163 errors:0 dropped:0 overruns:0 frame:0
          TX packets:111 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:30284 (30.2 KB)  TX bytes:20106 (20.1 KB)
```

3 Mount Laptop and LogiTech QuickCam on the Robot

1. If you haven't already, mount your group's laptop on the back of your robot with Velcro, as shown on the exemplar. Connect your laptop to the μOrcBoard using the provided cable.
2. We have also provided the μOrcTest module (`uorc_test`) which you may find useful for low-level debugging. Run μOrcTest to check that your μOrcBoard hardware and peripherals (motors, encoders, etc.) are configured as expected by ROS. For safety, place your robot on blocks before execution.
3. Use the provided aluminum mounting plate to attach the camera to your robot.

4 Obtain and Build the New Source

ROS and rosjava have already been installed on the netbook.

1. On the netbook, one person should copy the VisualServo Lab java files to your working copy of your group repository in the usual manner:

```
cd ~/RSS-I-pub
svn up
cd ~/RSS-I-group/
svn export ~/RSS-I-pub/labs/uorc_listener/
svn export ~/RSS-I-pub/labs/uorc_publisher/
svn export ~/RSS-I-pub/labs/odometry/
svn export ~/RSS-I-pub/labs/lab4/
svn export ~/RSS-I-pub/labs/rss_msgs/
svn export ~/RSS-I-pub/labs/orc_utils/
svn export ~/RSS-I-pub/labs/roscv/
svn add *
svn commit -m "added new source for VisualServo lab"
```

2. Open `~/.bashrc` in your favorite editor, and scroll to the bottom of the file to see what these variables are set to. Make sure that the following lines/environment variables exist.

```
source /opt/ros/electric/setup.bash
export ROS_ROOT=/opt/ros/electric/ros
export PATH=$ROS_ROOT/bin:$PATH
export ROS_PACKAGE_PATH=~/RSS-I-group/:~/ros:/opt/ros/electric/stacks:$ROS_PACKAGE_PATH
export CLASSPATH=~/RSS-I-pub/scripts/orc.jar:$CLASSPATH
export PATH=~/RSS-I-pub/scripts:$PATH
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
alias rosjavarun='rosrun rosjava bootstrap run.py'
export ROS_HOSTNAME=netbook
export ROS_MASTER_URI=http://netbook:11311
```

3. Verify that you can compile the lab4 source by running:

```
cd ~/RSS-I-group
rosmake uorc_listener
rosmake uorc_publisher
rosmake odometry
rosmake lab4
rosmake gscam
```

Note: This may take a while. The next three steps (in part 5) are do not depend this step.

5 Configure and Test the QuickCam

1. If you have not already done so, plug in the camera to the netbook's USB port. The QuickCam should be auto-detected as `/dev/video0`.
2. Verify that the required modules are loaded by running `lsusb | grep Logitech` at the shell prompt. You should see something resembling the following output:

   ```
   Bus 001 Device 004: ID 046d:0804 Logitech, Inc. Webcam C250
   ```

   If you do not see the above module list, wait a few moments and try again. If this doesn't work, try unplugging and reinserting the camera plug. If you have any problems please ask a TA.
3. Please see Appendix A at the end of this lab handout to finish configuring the camera.
4. Open a terminal window or tab on the netbook for each of the following commands:

   ```
   roscore
   rosjavarun uorc_listener Listener __name:=listen
   rosjavarun uorc_publisher Publisher __name:=publisher
   rosjavarun odometry odometry.Odometry __name:=odom
   export GSCAM_CONFIG="v4l2src device=/dev/video0 ! video/x-raw-rgb,
      width=160, height=120 ! ffmpegcolorspace" ; rosrun gscam gscam
   rosrun image_transport republish compressed in:=/gscam/image_raw raw out:=/rss/video
   rosrun image_view image_view image:=/rss/video
   rosjavarun lab4 VisualServo.VisionGUI
   ```

   Note: Ignore the I2C errors after running the second command.
5. Verify that the video is properly displayed and then close the VisionGUI by typing `Ctrl-C` in the terminal window.

6 Run from Eclipse (optional)

You may wish to use the Eclipse IDE and debugger while developing your code.

1. Run `eclipse` to start the eclipse IDE.
2. Close any projects that are currently open in your workspace.
3. Go to File | Import | Existing Projects into Workspace and choose the lab4 directory as the "root directory".
4. Press "Finish".
5. Go to Run | Run configurations... Select "Java Applications" and press the "New" toolbar button.
6. Set the "Project" to be `lab4` and the "Main class" to be `org.ros.RosRun`.
7. On the "Arguments" tab, set the "Program arguments" to be `VisualServo.VisionGUI __name:=gui`.
8. Press "Apply" and then "Run" to execute the program.

7 Running Code Remotely

> *I say we take off and nuke the site from orbit. It's the only way to be sure.*
> — Ripley, Aliens

You may find it easier to code on your workstation, rather than the netbook. ROS is a distributed environment, and nodes can be run on multiple processors or computers. Alternatively, you can use SSH to display windows on the netbook on your workstation. We will experiment with both methods.

7.1 SSH forwarding

We can develop and code remotely by tunnelling over SSH. From the workstation, execute:

```
ssh -X rss-student@<IP of netbook>
```

If you open an editor, e.g., `emacs`, the window will be displayed on the local workstation. Note that graphic-intensive programs may be slower. For example, video displays will have much lower frame rates.

7.2 Distributed ROS nodes

SSH forwarding can be slow as it essentially forwards all graphics and user input over the network. A preferred method is to transmit only the information required by our robot. In order to do this, we will (1) setup hostnames and environment variables, (2) checkout and build your group's code on the workstation, and (3) learn to use roslaunch.

1. On the workstation, do `sudo emacs /etc/hosts` and add two lines:

   ```
   XX.XX.XX.XX workstation
   YY.YY.YY.YY netbook
   ```

   where `XX.XX.XX.XX` and `YY.YY.YY.YY` are the IP addresses of your workstation and netbook, respectively. Because the netbooks and workstations have static IPs, these will not change.
2. Make the same additions to `/etc/hosts` on the netbook.
3. Commit your group's code on the netbook. Then, check the code out on the workstation in `~\RSS-I-group`. Refer to previous labs if needed.
4. On the workstation, use `rosmake [package]` to build all the code. Refer to Section 4 for specifics.
5. On the netbook, press `Ctrl+C` to stop all the processes started in Section 5. Close eclipse.

You now have a complete copy of your code on the workstation and on the netbook. We will now use `roslaunch` to start some of the processes on the workstation and some on the netbook. When roslaunch starts a node on a remote machine, it SSH's into the remote machine and starts the node. This means that it executes the node compiled on the remote machine, **not** the node compiled on the local machine. Note that roslaunch does not take the binary from your local machine and run it on the remote machine (some other distributed process environments, however, do this).

The `uorc_listener`, `uorc_publisher`, `odometry`, and `gscam` are run on the netbook. You will probably not have need to change this code. Instead, you will edit classes on the workstation and run them on the workstation.

1. From the workstation, SSH into the robot and start `roscore`. Do this with:

   ```
   ssh -X rss-student@netbook
   export ROS_MASTER_URI=http://netbook:11311
   export ROS_HOSTNAME=netbook
   roscore
   ```
2. Examine the file `config.launch` in the launch directory of lab4. If you have changed your `rss-student` password from the default, update the appropriate field.
3. On the workstation, run:

   ```
   export ROS_MASTER_URI=http://netbook:11311
   export ROS_HOSTNAME=workstation
   roslaunch lab4 lab4.launch
   ```
4. Because you will need to export `ROS_MASTER_URI` and `ROS_HOSTNAME` every time you start a shell, we suggest you add them to your `~\.bashrc` on the netbook and workstation (note that `ROS_HOSTNAME` is different for the workstation and netbook). If you run nodes from Eclipse, you will need to set the environment variables in the run configuration (go to Run | Run Configurations... | Environment and create `ROS_MASTER_URI` and `ROS_HOSTNAME`).

5. After several seconds, the VisionGUI should be displayed. From this GUI, you can drive the robot and view the output of the video. Review `VisionGUI.java` for the key bindings to drive the robot.
6. In an editor, examine the file `lab4.launch`. Be prepared to explain each line to the TA.
7. On the workstation, run `rostopic list` to view the published messages. Use what you learned from the `lab4.launch` file to set `ROS_MASTER_URI` and `ROS_HOSTNAME` appropriately.
8. Use `rxgraph` to generate a complete graph showing the connections between all nodes. Make sure all nodes are being displayed; it should look similar to Figure 1. Upload this image to the wiki and write a short description (2-3 sentences) of each node's function.

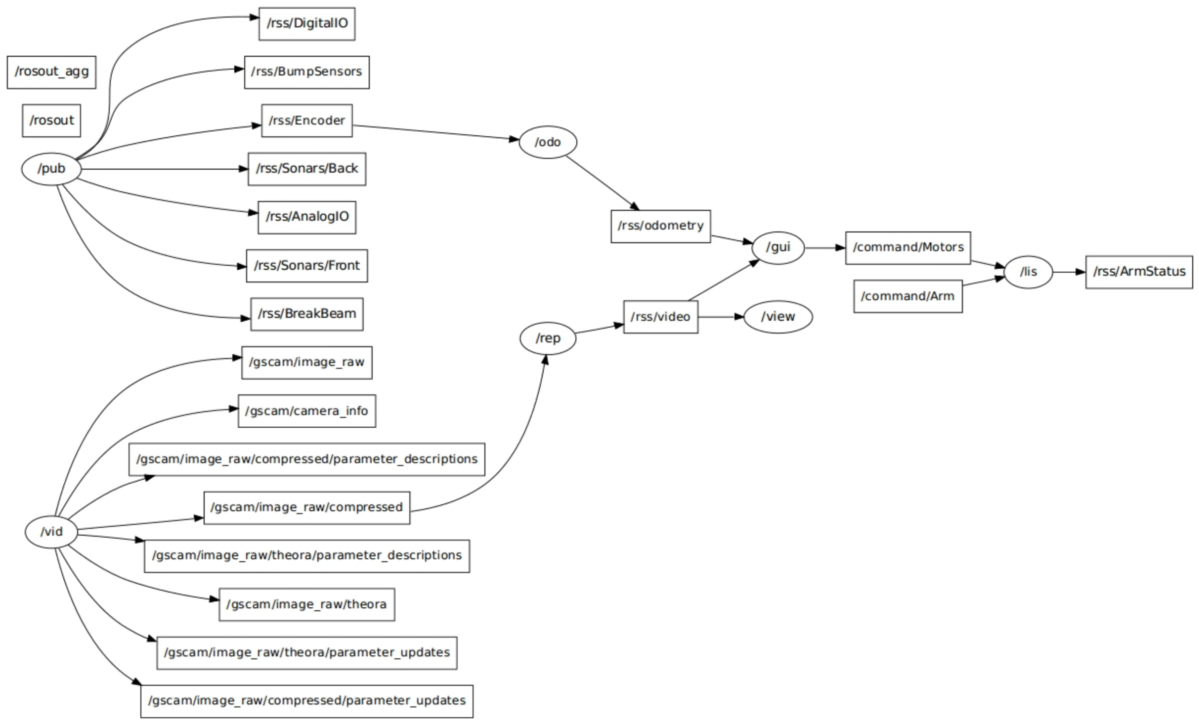

Figure 1: Example output of ROS nodes and topics

9. Stop `roscore` and `Ctrl+C` in the roslaunch window. You should restart roscore whenever you make a change to your source code. Also, note that the code run on the netbook must be compiled from the netbook using rosmake (and likewise for the workstation).

Deliverables: Upload the node graph and write node summaries on the wiki.

Checkpoint 1 (2/27): Demonstrate to a TA that you can drive the robot remotely and view the video. Be prepared to discuss the launch files. What purposes do the `/rep` and `/view` nodes serve?

8 Target Pixel Characterization

Your goal in this part of the lab is to write a method that detects, reasonably robustly, whether or not the colored target ball is being held in the camera's field of view. Read through this entire section before doing any coding or experiments.

In this section you will implement your blob segmentation algorithm in the BlobTracking class. After completing the segmentation algorithm you will add code to this class that computes the desired translational and rotational velocities for the robot to track the blob. This will be used by `VisualServo.java`. The method

```
public void apply(Image src, Image dest)
```

takes an input image from the camera handler, and returns a modified image to be displayed in VisionGUI.

As you develop your code, you can start the distributed processes by running:

```
roslaunch lab4 servo.launch
```

This will launch the GUI and video on the workstation and the μOrc controllers on the netbook. Remember, you also need to start a `roscore` on the netbook. You can edit your code on the workstation using Eclipse or your favorite editor. If you use the command line to edit, you can simply run `ant` in the lab4 directory to recompile (and avoid the time consuming `rosmake`), as long as you do not change any dependencies.

Also, feel free to edit the .launch files to aid your debugging. Roslaunch redirects stderr and stdout to the log files, which can be somewhat cumbersome. You may want to modify the .launch files so you can start some of the nodes from the command line and directly examine the stderr and stdout.

You will confront significant image *noise* in this lab. Think about how you might combat noise through manipulation of your source image data, i.e., by some operation on each image as soon as it is received from the camera.

- RGB Value Stability
  1. Write code within `BlobTracking` to print the RGB pixel values reported by your camera when it is, and is not, imaging your target ball. (You may want to print only a few values near the center of the image for each video frame your camera captures, and/or print an average value over some central region.) *Record on the Wiki the sensed RGB values for the ball under a variety of lighting conditions:* near the lab windows in bright and cloudy daylight; away from the windows; with the lab lights on, and the ball fully illuminated; and with the ball in a shadow (for example, due to a large occluder placed between the ball and any significant light sources).
  2. We have provided a template (`Histogram.java`) for producing a histogram of the R, G and B values of an image. If the frame is W pixels wide, the histogram method creates a 2-D array with W buckets and three entries per bucket. These histograms tabulate, for each (discretized) value, the number of pixels with that value. This will help you gain intuition into which pixel representation you should use, and how you should design your BlobPixel classifier so that it is reasonably robust.

     A realtime histogram can be created by applying the method `Histogram.getHistogram()` to all of the images you display. Your job is to complete the `makeHistogram()` method which tallies the RGB values in the image. Do this, then enjoy your "real-time" histogram display in the VisionGUI window!

     The displayed histogram is fairly primitive; it just stacks the channels on top of each other. For extra credit you can improve the displayed histogram. For instance you can make it photoshop-like and blend the channels together. Or you can write the bins with the max values out to the display image (for easy calibration). You will need to modify `overlayHistogram()` to do this.
  3. For a frame in which the ball **is not present**, examine the histograms (one each for the R, G and B values) of all pixels in that frame. *Use* `printscreen` *to capture the VisionGUI window; post your screenshot to the wiki.*
  4. For a frame in which the ball **is present**, and occupies between a third and half of the frame area, *capture and examine the same set of histograms as above.*
  5. *Discuss the following on the Wiki:*

     How stable are the sensed RGB values as lighting conditions vary? As the camera-ball distance changes? At the apparent center of the ball versus near its limb? At the image center versus near its edge? What average RGB value and tolerance would include all of these sensed values as "positive" detection events? Would a detector using this average and tolerance be sufficiently discriminative to enable successful ball-following behavior in the lab? Why or why not?

- HSB Color Space
  1. The `Image.Pixel` class has the ability to convert RGB pixel values to the HSB (Hue, Saturation, Brightness) color space, which represents color stimuli by the hue or pure color of the stimulus, the saturation or

amount of black (white) in the stimulus, and the brightness or (roughly) equivalent monochrome intensity of the stimulus. Note that the hue value wraps from 0.0 to 1.0. Adapt `makeHistogram` such that it pays attention to the `hsbHistogram` variable. If it is true the histogram should be computed in terms of HSB, if it is false it should compute the RGB histogram. Now repeat your experiments above, characterizing the range of HSB values at pixels imaging your target ball under the same variety of lighting conditions, standoff distances, and on-axis/off-axis viewing conditions.

2. For a frame in which the ball **is not** present, *capture and examine the histograms of H, S and B values in that frame.*
3. For a frame in which the ball **is** present, and occupies between a third and half of the frame, *capture and examine the same set of histograms as above.*
4. *Discuss the following on the Wiki:* How stable are the sensed HSB values? On which of the HSB values would you base an improved ball detector? What value and tolerance would you use to achieve a reasonably high true-positive rate without unreasonably large numbers of false-positives?

- BlobPixel Method
  1. Using the detection criteria you arrived at above, write a method in `BlobTracking` that classifies each pixel in the image as a target blob (ball) pixel or not. Write code to modify the contents of the output image message so that blob pixels are converted to a pure, saturated color (e.g., for a red ball, Red=255, Green=0, Blue=0; for a yellow ball Red=255, Green=255, Blue=0), and non-blob pixels are converted to grey-scale pixels with luminance equal to the average of their RGB values (e.g., Red=Green=Blue=(R+G+B)/3). Be careful not to overwrite the RGB values as you compute the new ones! This technique preserves the general appearance of the scene while highlighting the detected element.
  2. We have included a Gaussian blur filter in `GaussianBlur.java`, which you may choose to utilize for preprocessing. Additionally, you can set various system parameters (thresholds, etc.) at runtime by setting parameters on blobTrack in the VisualServo class.

- BlobPresent Method
  1. Now write a method in `BlobTracking` that uses your pixel-level classification to determine whether the ball is present in the overall image. (You can use simple pixel counting as the basis for your decision; we will see more sophisticated methods later in RSS, and you are welcome to experiment with such methods in this lab if you wish. We have included `ConnectedComponents.java` if you wish to try segmenting with connected components.)
  2. Compute the ball area, in pixels, as well as the ball centroid, in pixel coordinates. Next, modify the output image message to highlight these pixels, for example by drawing their bounding box or using any other method you wish. Finally write code to indicate the ball centroid with an 8×8 pixel cross in some complementary color with vertical post and horizontal arms.

Deliverables: Your Wiki materials should include brief responses to the questions raised above. Also include screen shots created as you worked through the items above, with captions, specifically:

- *Four histograms (RGB, ball present and not present; HSB, ball present and not present).*
- *The VisionGUI window when no ball is present, showing the unmodified camera image.*
- *The VisionGUI window when the ball is present, showing the camera image with the highlighted ball pixels and indicated centroid.*

Checkpoint 2 (3/4): Demonstrate to a TA or LA your VisionGUI with a modified camera image. Be prepared to discuss the differences between RGB and HSB values.

9 Visual Servoing

I don't tell you how to tell me what to do, so don't tell me how to do what you tell me to do.
— Bender, Futurama

You are now poised to implement your robot's ball-following behavior.

- Calibrate Your Camera

  Write the output of the centroid and area of the detected ball to `System.out` or to a file. Now hold your target ball at a variety of known distances and headings from the robot origin (take into account the difference between robot and camera origins). For each image you know the ball's actual size, and a "Fix" (range and bearing) on the ball with respect to the camera. Use this information to derive a function `blobFix` that, given the output of `blobPresent`, reports the estimated range (in meters) and bearing (in signed radians, with zero signifying straight ahead) to the ball target.

- Visual Servo to the Ball

  Your robot should attempt to maintain a pose that faces the ball target and is 0.5m away (i.e., heading = 0 radians, range = 0.5m). You may use your code or the staff solutions from previous labs. If you choose to write your own controller, you may choose to structure it either as a velocity controller (i.e., command speed and direction) or as a pose controller (i.e., command body-relative position and heading). Formulate two error terms, one for each of the range and heading errors. Write a controller that drives both errors to zero by commanding the robot with appropriate motion commands.

  Once you have calculated the desired robot velocities, create a ROS publication named `command/Motors` of type `rss_msgs\MotionMsg`. This can be done in VisualServo.java in the `run()` method. ROS documentation for rosjava is currently quite limited; however, you may find the C++ reference of use: http://www.ros.org/wiki/roscpp/Overview/Publishers%20and%20Subscribers
  and there are some examples of publication in rosjava here:
  https://rosjava.googlecode.com/hg/rosjava_tutorial_pubsub/src/main/java/org/ros/tutorials/pubsub/Talker.java

Deliverables: Briefly describe on the wiki the performance of your visual servo algorithm. What is its characteristic response time? Does it oscillate? Is it susceptible to noise? Does it have reasonably low false negative (i.e., missed target) and false positive (i.e., hallucinated target) detection rates?

10 Run Fully Autonomously

Until now, you have been using a develop and debug cycle that includes a VisionGUI process, allowing you to see the image frames captured by the robot, the outputs of its pixel classifiers and ball detectors, etc.

For this final part of the Lab, we ask you A) not to run VisionGUI, and B) to run VisualServo on your netbook. Now hold your ball target up in the robot's field of view. Everything should work just as before, probably with a lower-latency control loop. Congratulations! You have realized a fully autonomous, high-level, responsive behavior on your robot.

Deliverables: Post a description of your robot's operation. Does anything about its behavior differ from the previous run? Include in your briefing a demo video of your robot running fully autonomously. It should show a human moving the ball in front of the robot, and the robot servoing toward and away from the ball.

11 Assumptions and Failure Modes

One critical part of any engineering endeavor is a clear statement of your design assumptions, operating assumptions, and failure modes:

- The assumptions that you made in designing your solution, i.e. any assumptions you made about the functional components (including sensors) within your system.
- Your assumptions about the operating conditions under which those components will function as designed and as desired.

- The failure modes for your artifact, in other words, a clear statement of the conditions under which your system will not produce useful behavior.

Making an explicit statement about assumptions and operating conditions is useful for several reasons. First, it helps delineate which aspects of some overlying problem have been solved by the current effort, which is useful both for you and any others who wish to continue to improve the system's performance. Second, it "keeps you honest" in the sense that an explicit statement of failure modes is a clear acknowledgement that one does not believe that one's system has solved every problem in the world. You might be surprised at the high degree to which such statements of acknowledgement are appreciated by your peers, for example, at technical conferences. Third, just as a general matter of scientific and engineering practice, it is difficult to make well-defined progress in a given problem area unless the people working in that area agree upon, and apply, qualitative and quantitative assessment metrics for their efforts.

In light of these general admonitions to good engineering practice, let us now turn to your specific efforts for this lab. In writing code for each of the lab parts you made certain operational assumptions about how the robot works, both at the hardware and software level. *In your Wiki materials, discuss the ways in which your operational assumptions can be made to fail in practice. What assumptions underlie your hardware and software? What assumptions does your software, architecture or implementation make about how the hardware works? Under what conditions will your system function as desired? Under what conditions is it likely to fail? For each assumption or failure mode, briefly discuss how you would generalize your implementation to either remove the assumption or to decrease the likelihood of the failure mode's occurrence.*

Time Accounting and Self-Assessment

After preparing your wiki materials, return to the Time and Assessment pages. Tally your total individual effort there, including time spent writing up your responses. Answer the "Self Assessment" questions again, post-Lab. Add the date and time of your post-lab responses.

Conclusion

The completed code and wiki materials (with linked images and videos) are due by **3pm on Wednesday, March 6**. Commit your software into your group's repository and link from your team's top-level page to your wiki materials for the lab.

As usual, be prepared to demonstrate your robot's behavior at the start of Lab on the due date.

This concludes the VisualServo Lab.

A Changing Camera Settings

1. Start the V4L2 configuration utility by running `guvcview /dev/video0`
2. Experiment with the settings to find what works best for you. We have found the following to be useful:
   - Uncheck the box for "White Balance Temperature, Auto"
   - Set "Backlight Compensation" to 0
   - Set "Exposure, Auto" to "Manual Mode"
   - Uncheck "Exposure, Auto Priority"
   - Move the "Exposure, Absolute" Slider until the image looks good. (Try this even if you think it already looks good!)
3. Close the configuration utility and run `v4l2ctrl -d /dev/video0 -s video_settings.txt` to save the current settings to the file. Later, after rebooting or unplugging the camera, you can reload these same settings with `v4l2ctrl -d /dev/video0 -l video_settings.txt`.
4. Add `video_settings.txt` to your SVN repository.

Note: Once you have set up your camera, you should be able to make multiple runs in the same environmental conditions. However, if the lighting changes (i.e. on a different day), or if your code is not recognizing colors properly, you may need to reset the camera settings. To do this, follow the steps above. (Extra credit: make your detector robust to widely-changing lighting conditions.)