EECS 376/476 Spring 2011

ROS and OpenCV

We will need to process electronic images to give our robots vision. For image processing, we will use "OpenCV"—a popular open-source library of computer-vision routines. (See ***opencv***.*willowgarage.com,* or other on-line repositories). A good text for learning OpenCV is available as a Safari e-book through Case's libraries. (see http://proquest.safaribooksonline.com/book/programming/opencv/9780596516130).

Starter code and data: OpenCV pre-dates ROS, and the integration of OpenCV with ROS is not entirely seamless. The example code prepared for you by Eric and Chad should provide a painless introduction. To get the starter/example code from Eric's repository, do the following:

- Navigate to: https://github.com/ericperko/eecs376_sandbox
- In the URL window, select the text and copy it.
- In a terminal window, cd to the desired destination directory for the code.
- type: git clone (and paste the URL you copied)

This will download a set of ROS and OpenCV code to get you started.

You will also need to download a set of data to analyze. This data resides on the server "Farnsworth." It is large, so you'll want to be hard-wire connected to Case's network when you download this (> 1GB) data. To get the data, use an ftp tool, such as Filezilla. Set the host to: farnsworth.case.edu. You can leave the username and password blank. On the server, navigate to: /external_drive/eecs376_spring2011_logs.

Drag over the file: eecs376_extrinsics_calibration_safari.bag.tar.gz to a destination directory on your workstation. You will need to unzip and expand this archive, but this should happen automatically when you try to open it within linux.

The example bag file "eecs376_extrinsics_calibration_safari.bag" contains 70 seconds of images captured at 15Hz (in addition to Lidar data). You can have ROS playback the bag file (as you have done before) with the command*: rosbag play --pause (name of bag file)*. The "pause" option will start up the playback suspended, and you can toggle the playback off and on, as desired, using the spacebar with the cursor in the terminal executing the playback. What is new this time is that the playback also publishes images. The information being published is indistinguishable from the image data that would be published by a ROS camera driver in real time on the actual robot.

Image display utility: While "rosbag play" will begin publishing the stored images, you cannot see them without a corresponding node that subscribes to the published images and displays them. A convenient utility for this is:

```
rosrun   image_view   image_view   image:=/front_camera/image_rect_color
```

This command will run a node that subscribes to the "rectified" images being published. The "rectified" images have been preprocessed with camera intrinsic parameters to remove fish-eye warping and to account for pixel aspect ratio. This is possible because the bag file also saves camera parameters (under the topic camera_info). The image_view utility allows you to view the captured snapshots as an equivalent movie. A useful feature is that if you click in the playback window, this will invoke capturing a frame and saving it to disk. It will be saved in the same directory from which *rosrun image_view* was invoked. Sequential saved images will be saved as files with sequential numbers. A limitation is that you cannot know the timestamp corresponding to images thus saved (so you cannot match them to corresponding Lidar scans). However, such captures would be useful for testing image-processing options.

Subscribing to images and OpenCV: The example code provided shows how to subscribe to published images. See the code in "demo_node.cpp". The compiled code can be run with the help of a corresponding launch file as: *roslaunch start_vision_demo.launch*

This demo uses 3 windows: one to run the node, another to display the input images, and the third to display the processed images. The example demo_node.cpp, lines 44-46, invoke functions normalizeColors(), blobfind() and findLines() (which may be optionally commented out). These functions perform operations on the input images, producing the output images. The source code for these functions is in the *file lib_demo.cpp,* in which OpenCV functions are called. The function demo_node.cpp illustrates interactions with ROS, whereas the functions in lib_demo.cpp illustrate how to use OpenCV functions to process images. OpenCV operations include: segmentation (blob finding) based on ranges in color space, erosion, dilation, a gradient (edge-finding) operation, and a Hough transform.

Note that image data types have some historical differences. The older C interface to OpenCV uses a datatype IplImage, whereas newer, C++ openCV code uses cv::Mat objects. Some data type conversions are necessary for compatibility with ROS. After processing images with OpenCV, the resulting images need to be converted to corresponding ROS message types.

Publishing images is conceptually similar to publishing examples we have used earlier. However, publishing images presents bandwidth challenges, and specialized handling is performed by code within the ROS package "image_transport", which has special cases of publish and subscribe. Subscribing to published images requires instantiating an image_transport object.

Camera Extrinsics: The rectified images are preprocessed to perform dewarping, based on the instrinsic camera parameters. However, the resulting images still show perspective effects, and pixel coordinates correspond to unknown world-space coordinates. To translate image-space coordinates into world-space coordinates requires knowledge of the extrinsic parameters. There are 6 extrinsic parameters, describing the 3 translations of the focal point relative to the robot's reference frame, and 3 rotations (direction of the optical axis and rotation of the image plane about the optical axis) that define how the camera is mounted to the robot. With knowledge of these extrinsic parameters, it is possible to compute the correspondence between pixel coordinates and world coordinates.

The example bag file provided has enough information to deduce the extrinsic parameters. The scenes include views of a bright-orange object at floor level. A metal beam extends vertically upward from this object, and this beam is sensed by the Lidar. By comparing image-space centroid coordinates of the distinctive orange blob to egocentric-space Lidar sensing of the vertical beam, it is possible to deduce the camera’s extrinsic parameters (and this will be an assignment).