EE106A Lab 4: Image Manipulation, Camera Calibration & AR Tags *

Fall 2018

Goals

By the end of this lab you should be able to:

- Use `roslaunch` to run multiple ROS nodes
- Be familiar with ROS webcam interfaces and OpenCV image display tools
- Use NumPy to calculate homography information
- Be familiar with AR Tags

Relevant Tutorials and Documentation:

- `roslaunch`: `http://wiki.ros.org/roslaunch`
- `usb_cam`: `http://wiki.ros.org/usb_cam`
- `image_view`: `http://wiki.ros.org/image_view`
- `cv_bridge`: `http://wiki.ros.org/cv_bridge`
- OpenCV: `http://docs.opencv.org/trunk/doc/py_tutorials/py_gui/py_image_display/py_image_display.html`
- NumPy: `http://www.sam.math.ethz.ch/~raoulb/teaching/PythonTutorial/intro_numpy.html`
- `ar_track_alvar`: `http://wiki.ros.org/ar_track_alvar`

Contents

*Developed by Austin Buchan and Aaron Bestick, Fall 2014. Concept modified from content by Nikhil Naikal and Victor Shia, Fall 2012.

1 Introduction

In this lab, we will learn how to interface with a simple webcam in ROS, read in images, convert them to OpenCV images, perform some basic image processing, compensate for the perspective change between the floor of the lab and the camera thereby calibrating it, and, lastly, use the camera calibration to compute distances on the floor just by taking an image. We will introduce the new ROS tool `roslaunch` to run multiple nodes with specified parameters and topic renaming. This tool will help to reduce the number of terminals you have open to run ROS applications.

To get started, **remove any** "`source /labx/devel/setup.bash`" **lines from your** `.bashrc` **file.** (Leave the line that sources the `baxter_ws`.) Download the `lab4_resources.zip` file from the bCourses site and unzip it in your `ros_workspaces` directory. Run `catkin_make` there, then modify your `~/.bashrc` file so that it sources the `lab4/devel/setup.bash` script as the last line. Only one catkin workspace can be in use at a given time (unless those workspaces are *overlaid*, which is what happens when we source the Baxter workspace before building our own; you can read about this at `http://wiki.ros.org/catkin/Tutorials/workspace_overlaying` if you are interested).

1.1 Roslaunch

Examine the file `run_cam.launch` in the `launch` directory of the package `lab4_cam`. This is an XML file that specifies several nodes for ROS to launch, with various parameters and topic renaming directions. Initially, several commands are commented out (anything between `<!-- -->`). Leave these for now. Ensure that a webcam is connected to your computer. Depending on which type you select, you may need to modify the parameters in the launch file. (The default is the Microsoft cameras.) Run this launch file using the command:

```
roslaunch lab4_cam run_cam.launch
```

Initially, all that the launch file does is start roscore and a node `usb_cam`. Verify that this node is publishing image information with `rostopic`.

Next run an instance of the `image_view` node with the following command:

```
rosrun image_view image_view image:=/usb_cam/image_raw
```

You should now see a window with the video stream from the webcam on top of the monitor. This command shows an example of renaming the topic "`image`" (which is what `image_view` subscribes to by default) to "`/usb_cam/image_raw`" (which is what `usb_cam` publishes to). Use `rqt_graph` to verify that these nodes are connected via a topic.

Now kill all running processes with a Ctrl-C command in each terminal. Edit `run_cam.launch` to uncomment the first block of code that deals with `image_view`. Save the file, and launch it again with the command above. This should produce the same behavior as running the `image_view` node with `rosrun`, with the addition that the window is now autosized. Is anything different about the `rqt_graph`?

1.2 Webcams, ROS, and OpenCV

Now we are ready to extract some useful information from the webcam stream. For the purposes of this lab, we want to be able to adjust the webcam to be aligned with the floor and select points on a still frame. To facilitate this, we've included a node `camera_srv.py` that provides a single image over a ROS service when the user presses enter. Examine `src/camera_srv.py`, `srv/ImageSrv.srv`, and the `CMakeLists.txt` in the `lab4_cam` package to see how this service is provided and how to make sure that the service definition is generated. We've provided a skeleton implementation of a node that uses the service in `image_process.py`. This node also handles a lot of conversions between ROS, OpenCV, and NumPy array information. The comments and documentation above should help you figure out what each of the lines does if you are curious.

To use these nodes, edit `run_cam.launch` once more to remove the comments around the `camera_srv.py` and `image_process.py` node tags. Now when you launch the file, you should see a prompt to press enter in the terminal after the `image_view` window pops up. After adjusting the camera to point where you want, you can hit

Figure 1: Sample image of floor plane.

enter in the terminal to capture a single frame and display it in a second window. This window will wait for you to click 4 times on the image, displaying the pixel coordinates of the click in the terminal window. Finally, this script will display homography information (that you will calculate later) in a third window until the user presses a key while the third window is selected. Because you have not yet implemented the homography calculation, a placeholder "calculation" is used that just displays a matrix of black dots in the upper left corner of the image. Run through this whole process of capturing, clicking, and displaying homography a few times to familiarize yourself with the flow. For the most stable results, you should kill these processes by pressing `Ctrl+C` in the terminal window where you ran `roslaunch`.

2 Floor Plane Calibration

We will now consider the problem of calibrating the camera pixels corresponding to the floor of the lab. The objective is to determine a projective transform $H \in \mathbb{R}^{3\times 3}$ that transforms the ground plane to camera pixels. This mapping is bijective and can also map pixels in the camera back to floor coordinates. For this part of the lab, you will need to capture an image like the one in Figure **??** that shows an empty rectangular region on the floor.

Let us consider a point on the lab floor given by $\tilde{X} = [x, y]^T \in \mathbb{R}^2$. In order to keep the transformations linear (instead of dealing with affine projections), we will use *homogeneous* coordinates, where we append a "1" to the coordinates of $\tilde{X}$ as

$$X = \begin{bmatrix} \tilde{X} \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \in \mathbb{R}^3.$$

Our goal is to determine the linear transform $H \in \mathbb{R}^{3\times 3}$ that maps the point X to the pixel $Y = [u \quad v \quad 1]^T \in \mathbb{R}^3$. This transform is called a *homography* and takes the form

$$H := \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix}. \tag{1}$$

Notice that the last element of the homography is 1. This means that only 8 parameters of the matrix need to be estimated. Once the matrix is estimated, the pixel coordinates $\tilde{Y} = (u, v)$ can be determined using the following equations:

$$u = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + 1}, \qquad v = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + 1}. \tag{2}$$

These 2 equations can be rewritten in linear form as

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -u\cdot x & -u\cdot y \\ 0 & 0 & 0 & x & y & 1 & -v\cdot x & -v\cdot y \end{bmatrix} \cdot \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}. \tag{3}$$

Since Eqn. **??** has 8 unknowns, in order to uniquely determine the unknowns, we will need $N \geq 4$ floor point $\leftrightarrow$ image pixel pairs. With these N points, the equation becomes

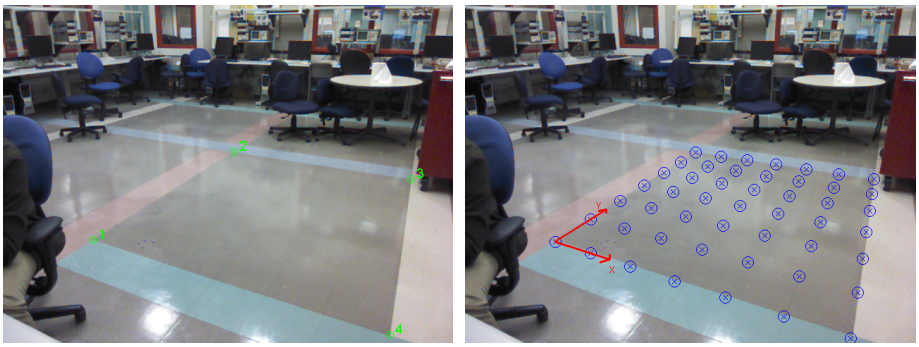
$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}, \tag{4}$$

where

$$\mathbf{A} = \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -u_1\cdot x_1 & -u_1\cdot y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_1\cdot x_1 & -v_1\cdot y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -u_2\cdot x_2 & -u_2\cdot y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -v_2\cdot x_2 & -v_2\cdot y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -u_3\cdot x_3 & -u_3\cdot y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -v_3\cdot x_3 & -v_3\cdot y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -u_4\cdot x_4 & -u_4\cdot y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -v_4\cdot x_4 & -v_4\cdot y_4 \\ & & & & \vdots & & & \end{bmatrix} \in \mathbb{R}^{2N\times 8}, \quad \mathbf{x} = \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \in \mathbb{R}^8, \quad \mathbf{b} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \\ \vdots \end{bmatrix} \in \mathbb{R}^{2N}. \tag{5}$$

Note: $[u, v]^T$ are pixel coordinates and $[x, y]^T$ are ground coordinates with respect to the origin that you have defined.

Modify `image_process.py` to compute the homography matrix between the floor plane and the camera plane. Define the $[x, y]^T$ floor coordinates of the first point you click to be $[0, 0]^T$. Calculate the $[x, y]^T$ coordinates of the other 3 points you click using the fact that the ground tiles are 30.48 cm (1ft) per side. (See Figure **??** for reference.) Modify the code to create and solve the linear equations above for the values of H. (You can use the `inv` function from the `np.linalg` module.) If you calculated the homography correctly, and if you selected a tile intersection as your origin, running this function should draw black dots on the intersections of the tiles, as in Figure **??**.



(a) (b)

Figure 2: (a) Corner selection for ground plane (b) Calibrated image with projected grid.

Checkpoint 1

At this point you should be able to:

- Show a picture of the floor with black dots at the intersections of the tiles
- Explain how homography works
- Explain your `create_homography` function

3 Mapping Pixels to Floor Coordinates

Now that we have computed the homography to map points from the floor coordinates to pixel coordinates, we will consider the inverse mapping from pixel coordinates $[u, v, 1]^T$ back to floor coordinates $[x, y, 1]^T$. This can be done by simply using the inverse of the homography matrix, H^{-1}. Let this matrix be

$$H^{-1} := Q = \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{bmatrix}. \tag{6}$$

With this inverse transform, the floor coordinates of any pixel can be computed as

$$x = \frac{q_{11}u + q_{12}v + q_{13}}{q_{31}u + q_{32}v + q_{33}}, \qquad y = \frac{q_{21}u + q_{22}v + q_{23}}{q_{31}u + q_{32}v + q_{33}}, \tag{7}$$

which is equivalent to computing x and y from $[\alpha x,\ \alpha y,\ \alpha]^T = Q \cdot [u,\ v,\ 1]^T$.

Modify your code to compute the distance between two points on the floor when you click on them in the image. Test your code by placing an object of known length on the floor and use your code to measure it. Try measuring the length of an object that does not lie in the floor plane. Are the measurements still accurate? Why or why not?

Figure 3: Length between selected points

Checkpoint 2

Get a TA to check your work. At this point you should be able to:

- Place an object of known length on the floor and measure it by clicking on the ends of the object in the picture from the webcam, recording the pixel points, and using the inverse homography to determine the length of the object (See Figure **??**. Compare the known length of the object with the length you measured using the inverse homography. Are they the same? (If the object is lying directly on the floor, they should be very close.)
- Measure an object of known length that doesn't lie in the plane of the floor. Compare the known length of this object with the length you measured using the inverse homography. Are they the same? Why or why not?

4 AR Tags

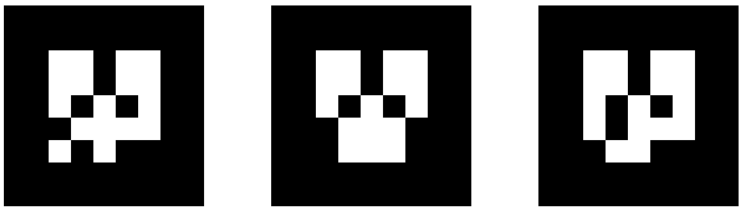


Figure 4: Example AR Tags

AR (Augmented Reality) Tags have been used to support augmented reality applications to track the 3D position of markers using camera images. An AR Tag is usually a square pattern printed on a flat surface, such as the patterns in Figure **??**. The corners of these tags are easy to identify from a single camera perspective, so that the homography to the tag surface can be computed automatically. The center of the tag also contains a unique pattern to identify multiple tags in an image. When the camera is calibrated and the size of the markers is known, the pose of the tag can be computed in real-world distance units. There are several ROS packages that can produce pose information from AR tags in an image; we will be using the `ar_track_alvar` [1] tutorial.

4.1 Webcam Tracking Setup

1. Download the package to the `src` directory of a ROS workspace with

   ```
   git clone https://github.com/ucb-ee106/ar_track_alvar.git
   ```

2. Download `artag_resources.zip` from the bCourses website, and unzip this to the `launch` directory of the `ar_track_alvar` package.

3. Update the `camera_info_url` parameter in `webcam_track.launch` to have the file path

   ```
   file://$(find ar_track_alvar)/launch/lifecam.yml
   ```

[1] `http://wiki.ros.org/ar_track_alvar`

(If you are using a Logitech camera, instead use `usb_cam.yml`.)

IMPORTANT NOTE: You need to leave the `file://` in front of the path to the yml file. The parameter is expecting a web URL, but the `file://` tells it to look in the local file system. The command `pwd` will print the full path to the current directory.

4. If any other parameters have changed, such as the name of the webcam, make sure they are consistent in the launch file (i.e., ensure that you are properly using either the Microsoft or the Logitech parameters).

5. Run `catkin_make` from the workspace (this may take a while).

6. Find or print some AR Tags. There should be a class set of 4 in Cory 111. Please only use these for testing and leave them unmodified so others can use them. The `ar_track_alvar` documentation has instructions for printing more tags that you can use in your project.

4.2 Visualizing results

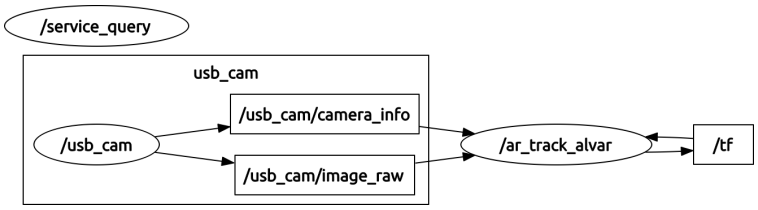


Figure 5: RQT Graph using AR Tags

Once the tracking package is installed, you can run tracking by launching `webcam_track.launch`. You should see topics `/visualization_marker` and `/ar_pose_marker` being published. They are only updated when a marker is visible, so you will need to have a marker in the field of view of the camera to get messages.

Running `rqt_graph` at this point should produce something similar to Figure **??**. As this graph shows, the tracking node also updates the `/tf` topic to have the positions of observed markers published in the TF Tree.

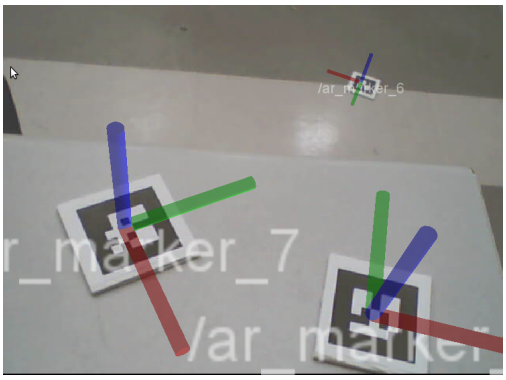


Figure 6: Tracking AR Tags with webcam

To get a sense of how this is all working, you can use RViz to overlay the tracked positions of markers with camera imagery. With the camera and tracking node running, start RViz with:

```
rosrun rviz rviz
```

From the Displays panel in RViz, add a "Camera" display. Set the Image Topic of the Camera Display to the appropriate topic (`/usb_cam/image_raw` for the starter project), and set the Global Options Fixed Frame to `usb_cam`.

(Note: you may need to place an AR tag in the field of view of the camera to cause the `usb_cam` frame to appear.) You should now see a separate docked window with the live feed of the webcam.

Finally, add a TF display to RViz. At this point, you should be able to hold up an AR Tag to the camera and see coordinate axes superimposed on the image of the tag in the camera display. Figure **??** shows several of these axes on tags using the lab webcams. Making the marker scale smaller and disabling the Show Arrows option can make the display more readable. This information is also displayed in the 3D view of RViz, which will help you debug spatial relationships of markers for your project.

Alternatively, you can display the AR Tag positions in RViz by adding a Marker Display to RViz. This will draw colored boxes representing the AR Tags.

Checkpoint 3

Get a TA to check your work. At this point you should be able to:

- Show that you can track the position and orientation of an AR tag

---