Representation of images

ME 416 - Prof. Tron

Thursday 2nd May, 2019

1 Images

Images can be seen as a mapping from pixel coordinates to colors. Formally, we can define an image I as a function of two discrete independent variables i,j (row and column of a pixel) to a vector $I[i, j]$. Each entry of the vector $I[i, j]$ is called a *channel* of the image, and the dimension d of the vector is equal to the *number of channels* in the image. Typical dimensions are $d = 1$ for a grayscale image, and $d = 3$ for a color image.

★★The indexes i, j for a pixel can be thought of as coordinates in the image frame. However, there are two important considerations to be made:

1) The j index represents the column of a pixel, i.e., the x (horizontal) coordinate, while i represents the row of a pixel, i.e., the y (vertical) coordinate; as a result, indexes $[i, j]$ are in the opposite order of how they should appear when considered as (x, y) coordinates. This is particularly important when working with OpenCV and NumPy arrays (see http://wiki.bu.edu/roboclass/index.php/OpenCV_quick_reference for reference).

2) The direction of the i index, or y coordinate, is "upside down" (it increases when moving from top to bottom). This is consistent with the typical convention used for images.

★★★Curiosity: higher values of d (ranging from 4 to 256) are used for *hyperspectral* images, where each channel captures the amount of light around in a different frequency band (e.g., infrared in addition to the typical red, green, blue channels).

1.1 Color spaces

While a physical color is represented in a computer with three numbers (the three entries in the vector $I[i, j]$), the way those three numbers should be interpreted to reconstruct the color can be subject to different standards. A standard is commonly referred to as a *color space*.

★★Note: A color space can be intepreted as a choice of coordinates in the space of colors, and changing color spaces. The situation is similar to what is discussed for rigid body transformations (TODO: insert internal link), where physical points are akin to physical colors, and color spaces are akin to a choice of the coordinate system.The main difference is that rigid body transformations are affine (i.e., linear with some offset), while color space transformations are generally non-linear.

Common color spaces are identified with acronyms representing the meaning of each number:

RGB : red, green and blue components with (integer) numbers ranging from 0 to 255. This is the typical way actual hardware devices (screens, graphic cards) represent colors.

HSV : hue (an angle on the color wheel), saturation (with 0 being gray, and 1 being a "full" color) and value (with zero being black, and 1 being the "full" color). TODO: insert images from Wikipedia. In practice, this is one of the best color spaces for detecting a specific color independently from the illumination conditions (e.g., light/shadow).

CIELAB : one value is "lightness" (dark/bright), the other two represent balances between opposing pairs of colors: green-red and blue-yellow. This model is specifically designed for representing the colors that can be percieved with the human vision system.

2 Representation in ROS and OpenCV

At a fundamental level, images are stored in a computer as a three-dimensional array, where each entry `a[i,j,k]` represents the value of the k-th channel of $I[i,j]$. Unfortunately, when working with ROS there are at least two ways to represent an image that one might have to deal with.

`sensor_msgs/Image` and `sensor_msgs/CompressedImage` this is the ROS message type for transferring images over topics.

NumPy arrays NumPy is a Python module dedicated to general mathematical processing on arrays, see the corresponding wiki page for details.

If you want to do image processing, there are two common options:

OpenCV OpenCV this is a general-purpose computer vision library, containing implementations of a large number of basic computer vision operations and functions to implement simple GUIs.

Direct operations One can operate directly on the images treated as NumPy arrays (this is best for reshaping, extracting patches, repeating or concatenating images).

As you might have noticed, it is not common to perform image processing directly on ROS messages. Hence, you will need to convert between the two format every time you want to receive/send images over topics and process them. Luckily, this is facilitated by the ROS package `cv_bridge` (see the section below for an introduction on how to use it).

★★★: due to performance reasons (memory alignment), images might not be stored contiguously in memory. TODO: discussion of *stride*

3 ROS `image_view` package

The best way to visualize a stream of images traveling on a topic in ROS is to use the image_view package. This package is typically used by running the homonym node from the package and passing the topic to visualize as an argument, as in the following example (which is run from the command line):

```
rosrun image_view image_view image:=image_topic
```

where `image_topic` is the name of the image topic (e.g., `/raspicam_node/image`).

Note: on ROSBot, you will need to open the VNC/Remote Desktop connection and export the `DISPLAY` enviromental variable correctly to see the images.

★★in a launch file, you can select which image topic to see with the image_view node by remapping the topic. See the snipped in the file below for an example.

```
<node pkg="image_view" name="raw_image_view" type="image_view" output="screen">
  <param name="image_transport" value="compressed"/>
  <remap from="image" to="/raspicam_node/image"/>
</node>
```

4 ROS cv_bridge package

The ROS `cv_bridge` package provides facilities for converting ROS `sensor_msgs/Image` into OpenCV in Python. For using it, you need to import the correct packages, create a bridge object, and then use the methods of the object to perform conversions.

4.1 Imports

To use `cv_bridge`, you need to import both OpenCV (`cv2`) and `cv_bridge` (or at least the `CvBridge` class, as shown below).

```
import cv2
from cv_bridge import CvBridge
```

4.2 Bridge object

Every conversion is done through the methods of a bridge object. The object can be created as follows:

```
bridge = CvBridge()
```

You should create only one bridge object during the lifetime of the node. You can perform any number of conversions using the same bridge object.

4.3 Message to image conversions

The following line converts a `sensor_msgs/Image` object into an OpenCV image.

```
cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
```

The string `"bgr8"` indicates the color space of the OpenCV image.

4.4 Image to message conversions

The following line converts an OpenCV image into a sensor_msgs/Image object.

```
bridge.cv2_to_imgmsg(cv_image, "bgr8")
```

Again, the string `"bgr8"` indicates the color space of the OpenCV image.