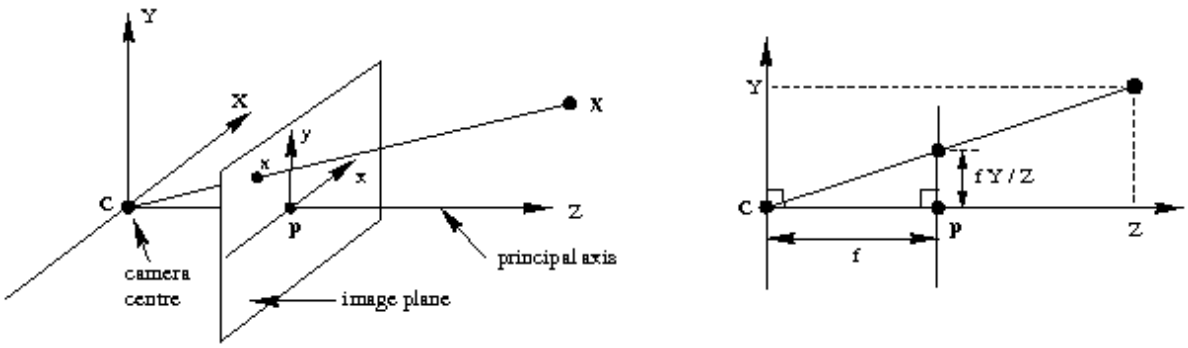
Flight camp part 3

In the part of the flight camp we will tell you some basics about images and how you can do some simple processing of the images. We assume ROS and as part of that OpenCV has been installed in your system which you should have if you managed to come this far.

Mapping the world onto an image

The world is 3D but the images that we capture by our camera are 2D projections of this world. This mapping between 3D and 2D is described by the camera model. We can use this model to tell where what we detected in the image is in the real world, what size it has there and it can also help us calculate where something in the 3D world will be mapped into the images. This is important for you in your project task and for many if not most mobile robot tasks involving cameras. For example, the ArUco marker detector that you used in **Flight camp Part 1 (https://kth.instructure.com/courses/17743/pages/flight-camp-part-1)** uses the camera model to calculate where the tag is in the frame of the camera. This allows you to tell where the drones if you know where the tag is in the world (given that there are no errors...) and it will allow you to tell where the objects are in the world assuming that you know where the drone is. Yes, here you should be thinking TF, TF, TF!

In this tutorial we will use the pin-hole camera model and it is sufficient to handle most use cases in computer vision applications unless you have a very high requirements on accuracy or a very wide field of view camera. The geometry described by the pin hole model is shown in the image below.



Lower case letters refer to coordinates in the image plane and upper case letters refer to coordinates in the 3D world (in the camera frame). We see that we can use similar triangles to derive a relationship between measured in pixels (in the image) and 3D coordinates in meters. Here f is the focal length which is typically expressed in pixels for convenience in computer vision, but can be converted to meter if you know the pixel size.

For example, if you know the position (X,Y,Z) of a point in 3D you can calculate the position in the image as x=f*X/Z and y=f*Y/Z.

Question: Given a position in the image, can you tell where in 3D the point is?

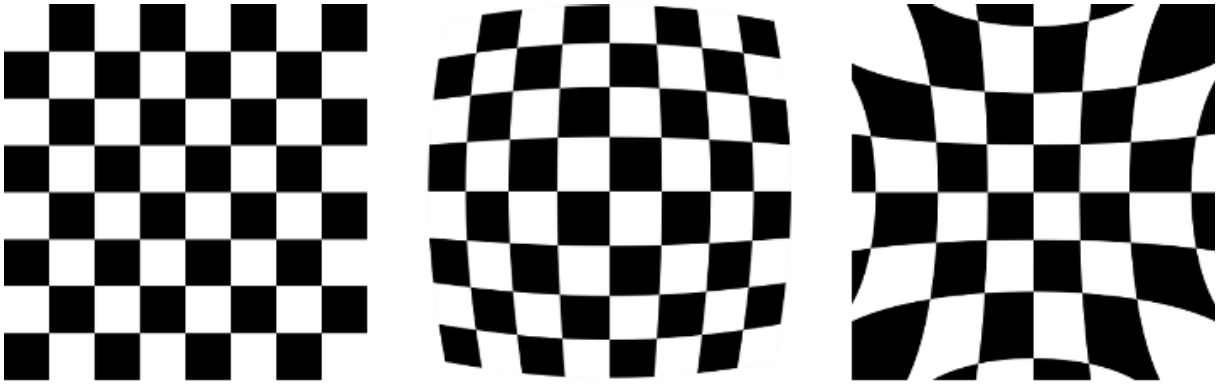
Often one models the focal length separately for the x and y direction, fx and fy. In addition to the focal length(s) there are the two parameters (cx,cy) needed to perform the mapping. These describe where the principal axis intersects the image plane, called the principal point. This is usually very close to the image

center. The so called camera matrix is defined as

C=[fx 0 cx; 0 fy cy; 0 0 1]; (where ";" refers to new line in the matrix)

Lens Distortion

If you look at an image from the camera that we use in the course you will find that the pin hole model as it is now does not describe it well by looking how straight lines in the world are mapped into the image. They are distorted. The image below shows two examples of distortion.



No distortion | Positive radial distortion (Barrel distortion) | Negative radial distortion (Pincushion distortion)

The distortion is caused by the lens and is captured by a distortion model, typically a polynomial, consisting of terms connected to radial and tangential components of distortion. The radial distortion is caused by the shape of lens and typically more visible for a wide angle lens. The tangential distortion is due to the lens is not parallel to the physical imaging plane.

For a more complete presentation of the camera model look at, for example,

http://www.cs.cmu.edu/~16385/s17/Slides/11.1_Camera_matrix.pdf (http://www.cs.cmu.edu/~16385/s17/Slides/11.1_Camera_matrix.pdf)

Camera models and ROS

To find the camera parameters we perform camera calibration. In this course you will be given a rough model of the camera that will take you quite far. The model is stored in a file in ~/.ros/camera_info and is used by the ROS image and computer vision nodes. An example is pasted below. Here we see that the parameters refer to an image of size 640x480. We see that the focal lengths are fx=231.250001 and fy=231.065552, i.e. quite close and that the principal point is (cx,cy)=(320.519378, 240.631482), i.e. quite close to the center of the image which would be (320,240).

```
image_width: 640
image_height: 480
camera_name: camera
camera_matrix:
  rows: 3
  cols: 3
  data: [231.250001, 0.000000, 320.519378, 0.000000, 231.065552, 240.631482, 0.000000, 0.000000, 1.00
0000]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
```

```
  cols: 5
  data: [0.061687, -0.049761, -0.008166, 0.004284, 0.000000]
rectification_matrix:
  rows: 3
  cols: 3
  data: [1.000000, 0.000000, 0.000000, 0.000000, 1.000000, 0.000000, 0.000000, 0.000000, 1.000000]
projection_matrix:
  rows: 3
  cols: 4
  data: [231.25, 0.000000, 322.360322, 0.000000, 0.000000, 231.06, 240.631, 0.000000, 0.000000, 0.000
000, 1.000000, 0.000000]
```

The documentation for the calibration file format can be found at **http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html (http://docs.ros.org/api/sensor_msgs/html/msg/CameraInfo.html)**

Let us take a look at what camera model is used in our simulator!

Make sure that you have started the simulator. Take a look at what topics are available by

```
rostopic list
```

This should give you a list like

```
robot@PJMSI:~/catkin_ws$ rostopic list
/cf1/camera/camera_info
/cf1/camera/image_raw
/cf1/camera/image_raw/compressed
/cf1/camera/image_raw/compressed/parameter_descriptions
/cf1/camera/image_raw/compressed/parameter_updates
/cf1/camera/image_raw/compressedDepth
/cf1/camera/image_raw/compressedDepth/parameter_descriptions
/cf1/camera/image_raw/compressedDepth/parameter_updates
/cf1/camera/image_raw/theora
/cf1/camera/image_raw/theora/parameter_descriptions
/cf1/camera/image_raw/theora/parameter_updates
/cf1/camera/parameter_descriptions
/cf1/camera/parameter_updates
/cf1/cmd_full_state
/cf1/cmd_hover
/cf1/cmd_position
/cf1/cmd_stop
/cf1/cmd_vel
/cf1/external_position
/cf1/pose
/cf1/rssi
/cf1/setpoint_pose
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/rosout
/rosout_agg
```

where we can see that the camera information is published in the topic /cf1/camera/camera_info. To look

at the model, we do

```
rostopic echo /cf1/camera/camera_info
```

and part of what you see would be

```
height: 480
width: 640
distortion_model: "plumb_bob"
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [245.54354677695392, 0.0, 320.5, 0.0, 245.54354677695392, 240.5, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [245.54354677695392, 0.0, 320.5, -0.0, 0.0, 245.54354677695392, 240.5, 0.0, 0.0, 0.0, 1.0, 0.0]
```

Question: How much distortion is modelled in the simulator in this case?

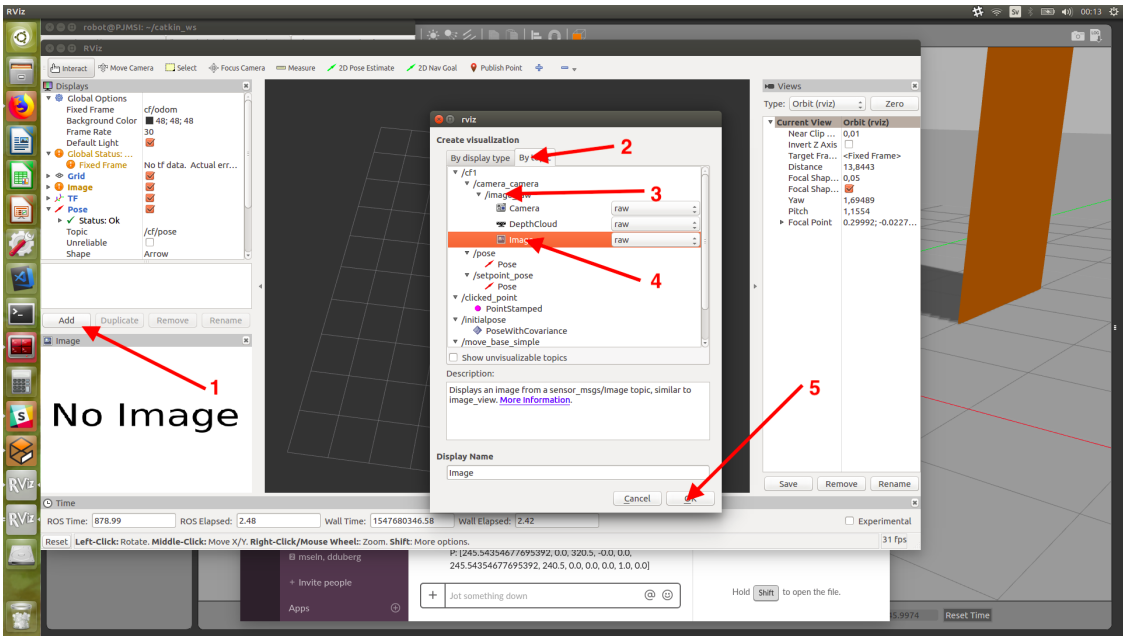
If you want more details on the camera model and how it is used in the image pipeline in ROS look at [http://wiki.ros.org/image_pipeline/CameraInfo (http://wiki.ros.org/image_pipeline/CameraInfo)](http://wiki.ros.org/image_pipeline/CameraInfo)

To those who would like to understand how the calibration is theoritically performed, check chapter 7 in book [Multiple View Geometry in Computer Vision (http://www.robots.ox.ac.uk/~vgg/hzbook/)](http://www.robots.ox.ac.uk/~vgg/hzbook/) for more details.

Image passing and processing using OpenCV and ROS

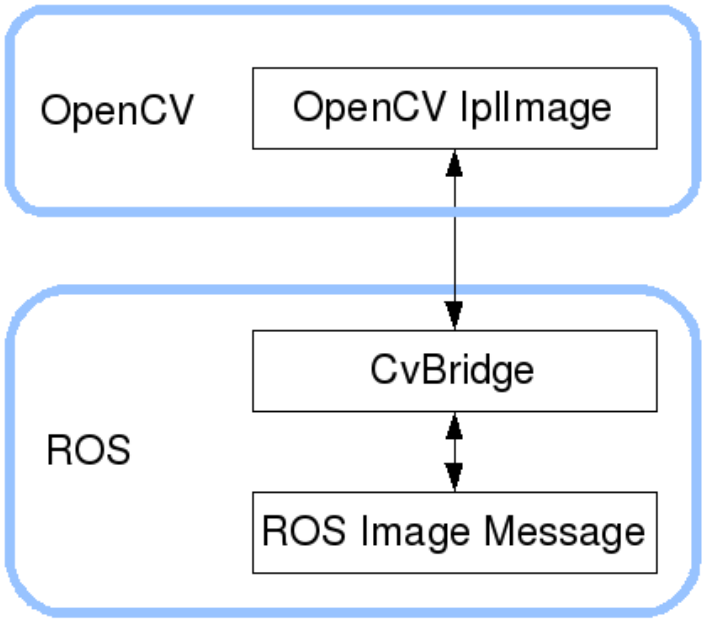
In Flight camp Part 1&2, you have familiarised yourself with basic ROS concepts. One of them was topics and messages. Images have their own encoding in ROS used when being communicated between nodes.

Make sure your simulator is running and start RViz. Now click the Add button, select the "By topic" tab in the window that pops up, make sure that the topic is expanded so that you see types inside there and select the image and then click OK.



You should now see a live image from the camera in a new subwindow in RViz.

Now, let us look at a simple node that subscribes to these images and does something simple with it. We will be using the computer vision package OpenCV which comes packed with many useful tools. To process the image with OpenCV, a package called `cv_bridge` is used to perform the conversion from ROS images to OpenCV images and vice versa. The images below illustrates this. For details, refer to the **cv_bridge documentation (http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython)**, from where this image was copied.



The aim in what follows is to give you a common example of getting an image into a program, do some processing and publishing the result of that processing as an image that someone else could use here. The starting point for our examples will be the **cv_bridge tutorial. (http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython)**

Practice what you did in part2 by creating a new package, this time called part3, and create a node called `imagecircle.py`.

Paste in the code below into that node

```
#!/usr/bin/env python
from __future__ import print_function

import roslib
import sys
import rospy
import cv2
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class image_converter:

  def __init__(self):
    self.image_pub = rospy.Publisher("/myresult", Image, queue_size=2)
```

```
    self.bridge = CvBridge()
    self.image_sub = rospy.Subscriber("/cf1/camera/image_raw", Image, self.callback)

  def callback(self,data):
    # Convert the image from OpenCV to ROS format
    try:
      cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
    except CvBridgeError as e:
      print(e)

    # Extract the dimension of the image
    (rows,cols,channels) = cv_image.shape

    # Assuming it is larger than 100x100 we draw a 100x100 circle at 50,50 with radius 50 into the im
age
    if cols > 100 and rows > 100 :
      cv2.circle(cv_image, (50,50), 50, 255)

    # Publish the image
    try:
      self.image_pub.publish(self.bridge.cv2_to_imgmsg(cv_image, "bgr8"))
    except CvBridgeError as e:
      print(e)

def main(args):
  rospy.init_node('imagecircle', anonymous=True)

  ic = image_converter()

  print("running...")
  try:
    rospy.spin()
  except KeyboardInterrupt:
    print("Shutting down")

  cv2.destroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)
```

Make sure the simulator and RViz are running, and you should the be able to run the new node which will add a circle to the image and publish this new image under the topic /myresult

```
rosrun part3 imagecircle.py
```

If you then go into RViz and look at the messages being displayed in the Display tab in the upper left corner you should find the image you added in the steps before. Open that message and change the topic from /cf1/camera/camera_image to /myresult which is the topic that is pubilshed by the program above. You can also add a new display window with the new image if you prefer.

Close the previous program if you did not already. Let us look at another program that uses OpenCV. This one will segment the image into pixels that are close to white and not and create a black and white image from that. Create a node called `colorseg.py` and paste the following code into it.

```
#!/usr/bin/env python
from __future__ import print_function

import roslib
```

```
import sys
import rospy
import cv2
import numpy as np
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class image_converter:

  def __init__(self):
    self.image_pub = rospy.Publisher("/myresult", Image, queue_size=2)

    self.bridge = CvBridge()
    self.image_sub = rospy.Subscriber("/cf1/camera/image_raw", Image, self.callback)

  def callback(self,data):
    # Convert the image from OpenCV to ROS format
    try:
      cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
    except CvBridgeError as e:
      print(e)

    # Convert BGR to HSV
    hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)

    # define range of the color we look for in the HSV space
    lower = np.array([0,0,250])
    upper = np.array([255,5,255])

    # Threshold the HSV image to get only the pixels in ranage
    mask = cv2.inRange(hsv, lower, upper)

    # Bitwise-AND mask and original image
    res = cv2.bitwise_and(cv_image, cv_image, mask= mask)

    # Publish the image
    try:
      self.image_pub.publish(self.bridge.cv2_to_imgmsg(res, "bgr8"))
    except CvBridgeError as e:
      print(e)

def main(args):
  rospy.init_node('colorseg', anonymous=True)

  ic = image_converter()

  print("running...")
  try:
    rospy.spin()
  except KeyboardInterrupt:
    print("Shutting down")

  cv2.destroyAllWindows()

if __name__ == '__main__':
    main(sys.argv)
```

Move the drone to a place in the simulator where you can see one of the ArUco markers and run the program with

```
rosrun part3 colorseg.py
```

This program publish its results on the same topic, /myresult, so you should be able to see its output immediately in RViz. Before jumping to the conclusion that color segmentation is the solution to all your problems know that color constancy is a myth. Talk to the people that took DD2425 and they will confirm this story.