OpenCV for Robotics

Unit 2: Computer Vision Basics

Introduction

- Summary -

Estimated time to completion: **4-5 hours**

In this course, we will talk about the basics of computer vision, and how you can use it with a robot in ROS, in which we will emphasize the following topics:

- cv_bridge
- Color spaces and color filtering
- Edge detection and a brief introduction to convolutions
- Morphological transformations

- End of Summary -

Before Starting

In order to have all the content necessary to work with this course, you have to download this git (https://bitbucket.org/theconstructcore/opencv_for_robotics_images/src/master/). So, in a shell, `execute` the following lines.

In [ ]:
```
cd ~/catkin_ws/src
```

In [ ]:
```
git clone https://bitbucket.org/theconstructcore/opencv_for_robotics_images.git
```

Now you can see that you have a new folder in your IDE called **opencv_for_robotics_images** and it contains the images to do the exercises and some files, too.

NOW LET'S GO!

Computer vision in a nutshell

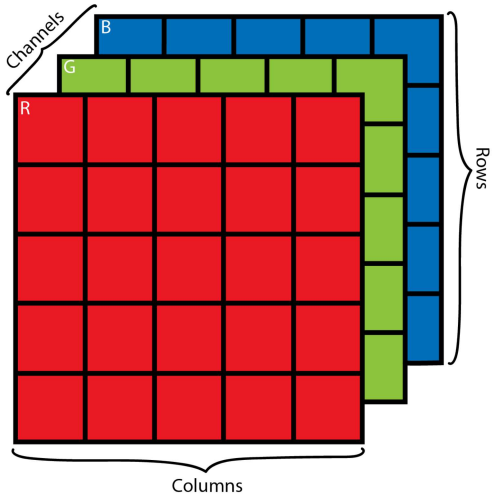
Computer vision, commonly abbreviated as CV, could be described as a field of study that allows a computer to analyze and have understanding of an uploaded digital image or group of images, such as videos.

The main idea of CV, along with robotics and other fields of study, is to improve tasks that could be exhaustive or repetitive for humans. In recent years, there have been many improvements with the invention of complex computer vision and deep learning systems, such as the well-known convolutional neural networks. These inventions shifted the point of view to solve many problems, such as facial recognition and medical images.

Images

First of all, we need to understand what exactly an image is. Colloquially, we could describe it as a visual representation of something that, by itself, is a set of many characteristics such as color, shapes, etc. For a computer, an image could be better described as a matrix, in which every value is considered a pixel, so when you are talking about a 1080p image resolution, you are referring to a specific 1080x1920 px matrix.

2.1 Color

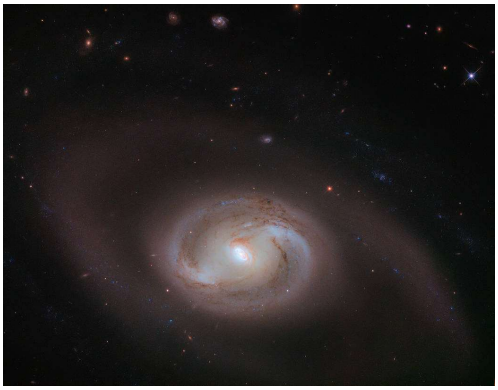


In the case of a colored image, we are talking about a three-dimensional matrix where each dimension corresponds to a specific color channel (Red, green, or blue). The dimensions of this matrix will be different for different color spaces, which we will discuss further in the course.

We can describe an image in many more complex ways, like the color construction that is mainly a result of the light over the object surface. When we have something black, it is actually the lack of light. The color formation will depend on the wavelength of the main components of white light.

If you like physics as much as I do, you will find this phenomenon interesting, where the color deformation can be seen: the stars. In many pictures of space, you can see that the rock formations that are far from us have a red color, while the closest ones have a blue color. This phenomenon was discovered by the North American astronomer Edwin Hubble in 1929. We know that space is in a state of constant expansion, so if space is deformed, the light that we receive from those stars will suffer from that expansion, too. As a consequence, the wavelength of the light will be higher and the color we perceive will have a red tone instead of a blue one.

This is an open source image from NASA. You can find it at [https://images.nasa.gov/ (https://images.nasa.gov/)](https://images.nasa.gov/)



We don't want to go much deeper on the color formations and theory of it. The main idea is so you can understand the basis of what we are going to work with for the rest of the course. It will be helpful if you want to do more profound research on this topic, which I consider really interesting.

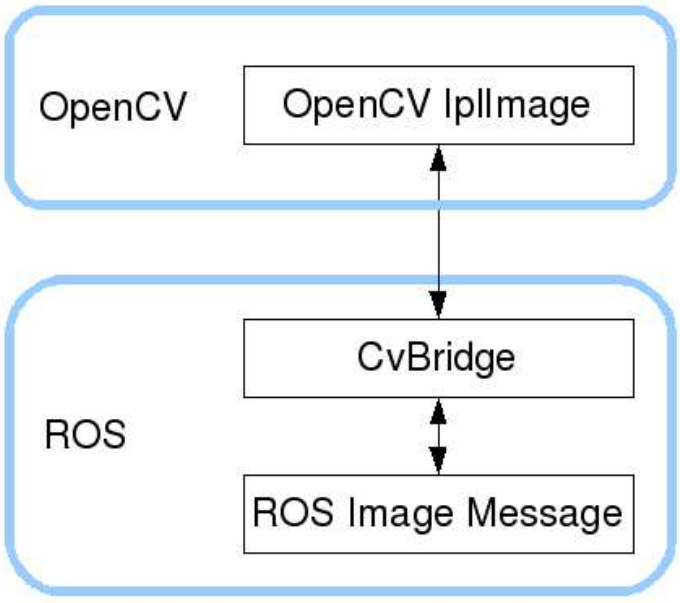
Going into practice!

Ok, so now that you have a brief introduction about what computer vision is, and a little background on image formation, it's time to describe one of the basic tasks of CV: color filtering. Color filtering is extracting from an image the information of a specific color. Before that, we will see some basic operations with opencv so you can get acquainted with this library, and understand the code ahead.

2.2 cv_bridge, the connection between opencv and ROS

First of all, you have to know that ROS passed images from its sensors using its own sensor_msgs/Image (http://docs.ros.org/api/sensor_msgs/html/msg/Image.html) message format, but sometimes you need to use this images with **OpenCV** in order to work properly with Computer vision algorithms.

Fortunately, we have **CvBridge**, which is a ROS library that provides an interface between ROS and OpenCV, converting ROS images to an Open CV format and vice versa, like this image. This image is taken from this wiki (http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython)



If you want more information about this, you can visit this wiki of ros (http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython)

As for how you will use it, well, you will see similar code like the following one.

```
1 from cv_bridge import CvBridge
2 bridge = CvBridge()
3 cv_image = bridge.imgmsg_to_cv2(image_message, desired_encoding='passthrough')
```

In **1**, you are importing CvBridge; in **2**, you are creating your object; and in **3**, you are converting imgmsg to cv2 in order to work with Open CV.

- Exercise 2.1 -

Look at this example and try to find these **three** parts.

In [ ]:

```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2


class ShowingImage(object):

    def __init__(self):

        self.image_sub = rospy.Subscriber("/camera/rgb/image_raw",Image,self.c
        self.bridge_object = CvBridge()

    def camera_callback(self,data):
        try:
            # We select bgr8 because its the OpenCV encoding by default
            cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding
        except CvBridgeError as e:
            print(e)

        cv2.imshow('image',cv_image)
        cv2.waitKey(0)



def main():
    showing_image_object = ShowingImage()
    rospy.init_node('line_following_node', anonymous=True)
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

... .. . **Time's up!**

- End of Exercise 2.1 -

Now that you know **cv_bridge**, let's try to work with some Open CV code. Let's go!

2.3 Loading and writing an image

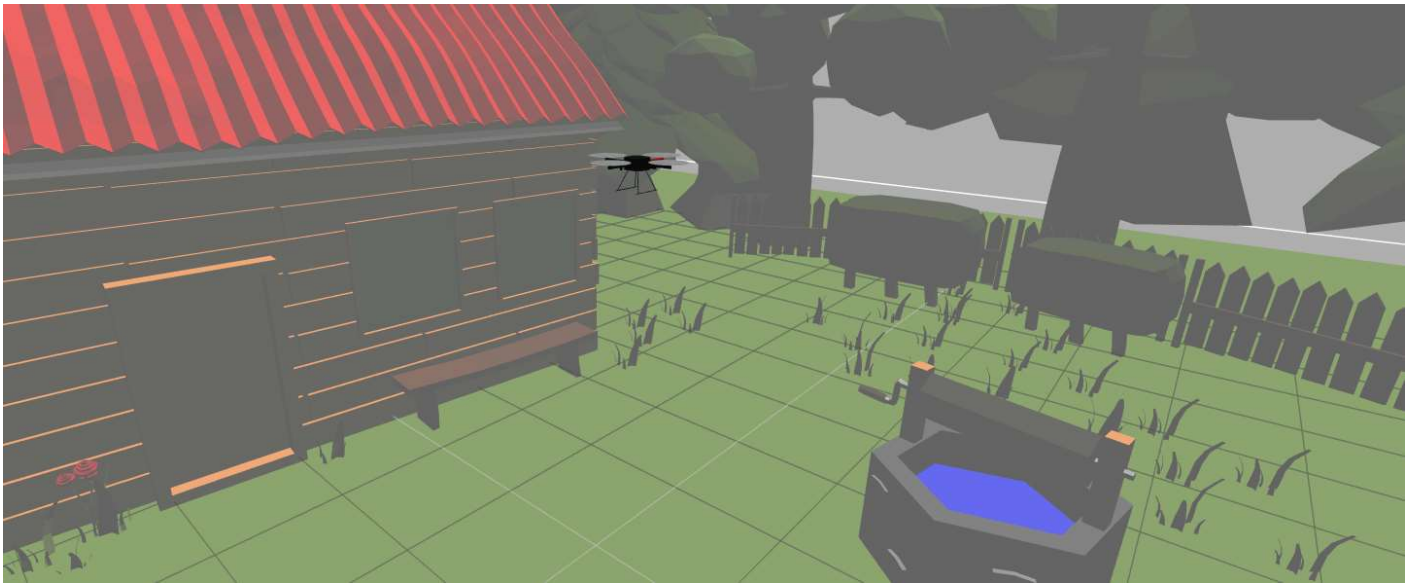
Opencv has some algorithms that permit work easily with images. The principal ones are **cv2.imread** , **cv2.imwrite**,and **cv2.imshow**. For example, here is a simple code in python that uses some of these algorithms. Let's see.

In [ ]:
```
#Import the Opencv Library
import cv2

#Read the image file
img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/C

#Display the image in a window
cv2.imshow('image', img)

#Save the image "img" in the current path
cv2.imwrite('image.jpg', img)


#The window will close after a key press
cv2.waitKey(0)
cv2.destroyAllWindows()
```

But, if we want to use it with ROS, how do they work? Let's have a look.

imread ,imshow and imwrite

As you can see, the drone is flying in the simulation.



- Exercise 2.2 -

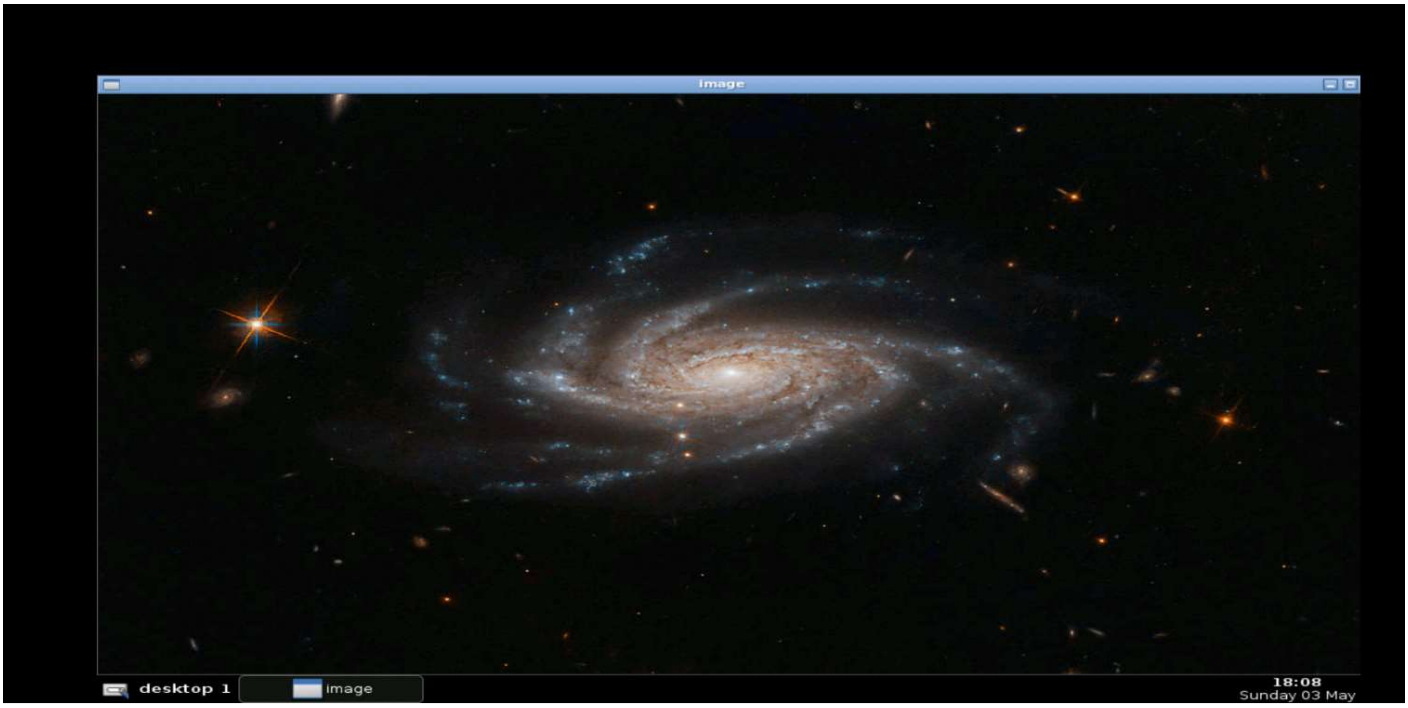
Based on the previous exercise and the previous python code, try to merge both and make a code that can read an image, write an image, and show an image following the next steps:

- a) Create a package in catkin_ws and call it **unit2** that depends on **rospy**.
- b) Inside *src* folder, create a file and call it **load_image.py**. In order to write the code, you can use **IDE** tool.
- c) The topic of the rgb camera of the robot is **"/camera/rgb/image_raw"**. Subscribe to this topic in order to get the data for the cv_bridge.
- d) This is the path of an image **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/Course_images/test_image_1.jpg'**. Use it to read this image and show it (You will see it in the **Graphical Tools**).
- e) Take a picture using the rgb camera of the robot and call it **drone_image.jpg**.
- f) Inside the unit2 package, create a folder and call it **launch**. Inside it, create a launch file and launch it.

- End of Exercise 2.2 -

- Expected Behavior for Exercise 2.2 -

You will see two important things. The first is a spectacular open source pictura from NASA in the **graphical tools**, and the second is that in the direction you launch this file, you will have a new image. Yes, you're right, it is a picture taken from your drone.



- End of Expected Behavior for Exercise 2.2 -

- Exercise 2.3 -

*Can you identify the parts of cv_bridge? Well, now it's your turn. Make a similar code, but instead of seeing the galaxy picture, show the picture taken from your drone, and show the image taken from your camera now. You will see the difference between them. So, follow these instructions:

- a) Create a new file inside the path **/catkin_ws/src/unit2/src** and call it **load_image2.py**.
- b) Based on the previous exercise, read the picture that the drone took and show it (Don't forget that you can write your code using the **IDE** tool).
- c) Show the image of the camera in real time and compare them. Remember that you will see it in the **Graphical Tools**.
- d) Create a launch file of this code, in the *launch* folder, in order to launch it.

We'll wait for you, take your time...

Come on! You can do it! JUST DO IT!

- End of Exercise 2.3 -

Great, your code should be similar to the next one. **If you haven't already done your code, don't be a cheater!**

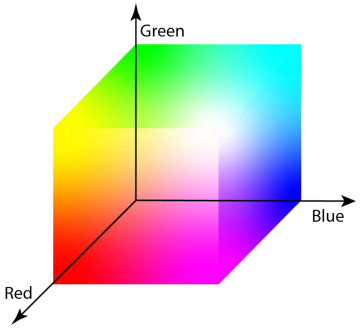
- Expected Behavior for Exercise 2.3 -

If the result looks like this, Congratulations! You took the first step using computer vision with ROS, but it is only the beginning.

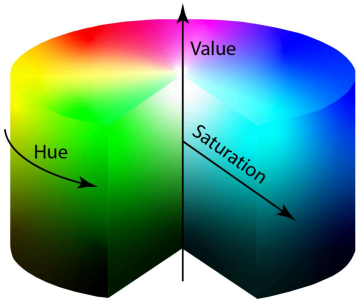
- End of Expected Behavior for Exercise 2.3 -

2.4 Space colors

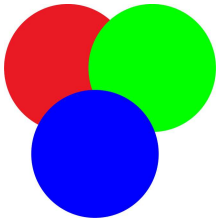
Before going into color filtering, you need to understand the concept of space colors. We are going to use it often during the course. Additionally, it will help you to experiment with different color spaces for different applications. A space color is no more than a three-dimensional model that tries to describe the human perception known as color, where the coordinates of the model will define a specific color. One of them that you may know is the RGB, where all the colors are created by mixing red, green, and blue (Python works with quite a different model of RGB, inverting the order of the colors, so the final model is BGR).



Just like we said at the beginning of the course, one of the main objectives is to detect colors in images. For this specific task, we will use a color space know as HSV (Hue Saturation Value) that is a closer model to how humans perceive colors. This is a non-linear model of RGB with cylindrical coordinates.

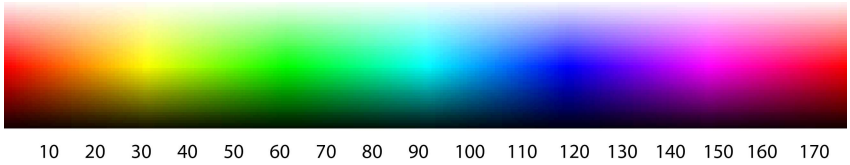


For the next exercise, we will apply a color filter to the next image. The main idea of this exercise is to pull apart each of the three colors.

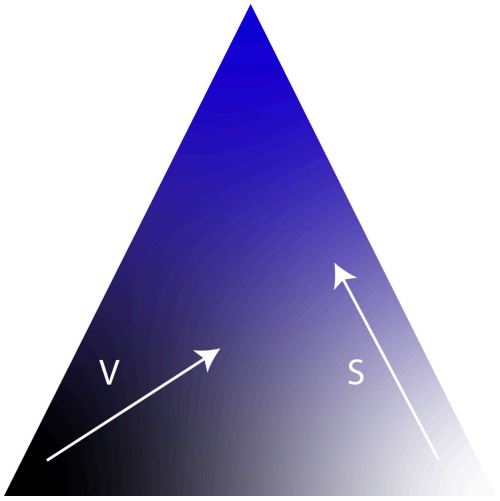


We use the HSV space colors because they make it easier to define the color we want. The color components themselves are defined by the Hue channel, having the entire chromatic spectrum present, compared to the RGB where we need all three channels to define a color.

For better comprehension of this part, we can make use of the following image. It is an approximation of how the colors are defined in the hue channel.



For example, if the color I'm looking for is blue, my Hue range should be between 110 and 120, or 100 and 130 for a wider range. So the value of my lower limit should look something like `min_blue = np.array([110,Smin,Vmin])` and the higher limit `max_blue = np.array([120,Smax,Vmax])` . In the case of Saturation and value, we can say that the lower the saturation, the closer to white, and the lower the value, the closer to black, as can be seen in the image below:

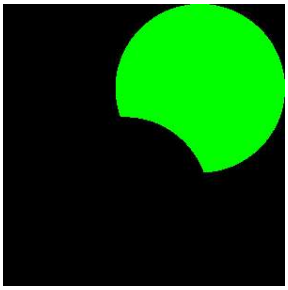


Color Filtering

Well, if you were working with just OpenCV, you would have something similar to the next code.

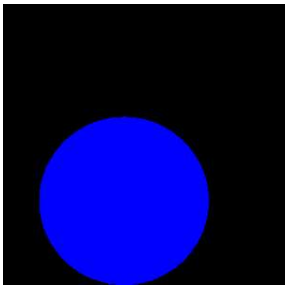
In [ ]:

```
import cv2

#Import the numpy library which will help with some matrix operations
import numpy as np

image = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2

#I resized the image so it can be easier to work with
image = cv2.resize(image,(300,300))

#Once we read the image we need to change the color space to HSV
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

#Hsv limits are defined
#here is where you define the range of the color you´re looking for
#each value of the vector corresponds to the H,S & V values respectively
min_green = np.array([50,220,220])
max_green = np.array([60,255,255])

min_red = np.array([170,220,220])
max_red = np.array([180,255,255])

min_blue = np.array([110,220,220])
max_blue = np.array([120,255,255])


#This is the actual color detection
#Here we will create a mask that contains only the colors defined in your limi
#This mask has only one dimention, so its black and white }
mask_g = cv2.inRange(hsv, min_green, max_green)
mask_r = cv2.inRange(hsv, min_red, max_red)
mask_b = cv2.inRange(hsv, min_blue, max_blue)

#We use the mask with the original image to get the colored post-processed ima
res_b = cv2.bitwise_and(image, image, mask= mask_b)
res_g = cv2.bitwise_and(image,image, mask= mask_g)
res_r = cv2.bitwise_and(image,image, mask= mask_r)

cv2.imshow('Green',res_g)
```



```
cv2.imshow('Red',res_r)
```

```
cv2.imshow('Blue',res_b)
```



```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

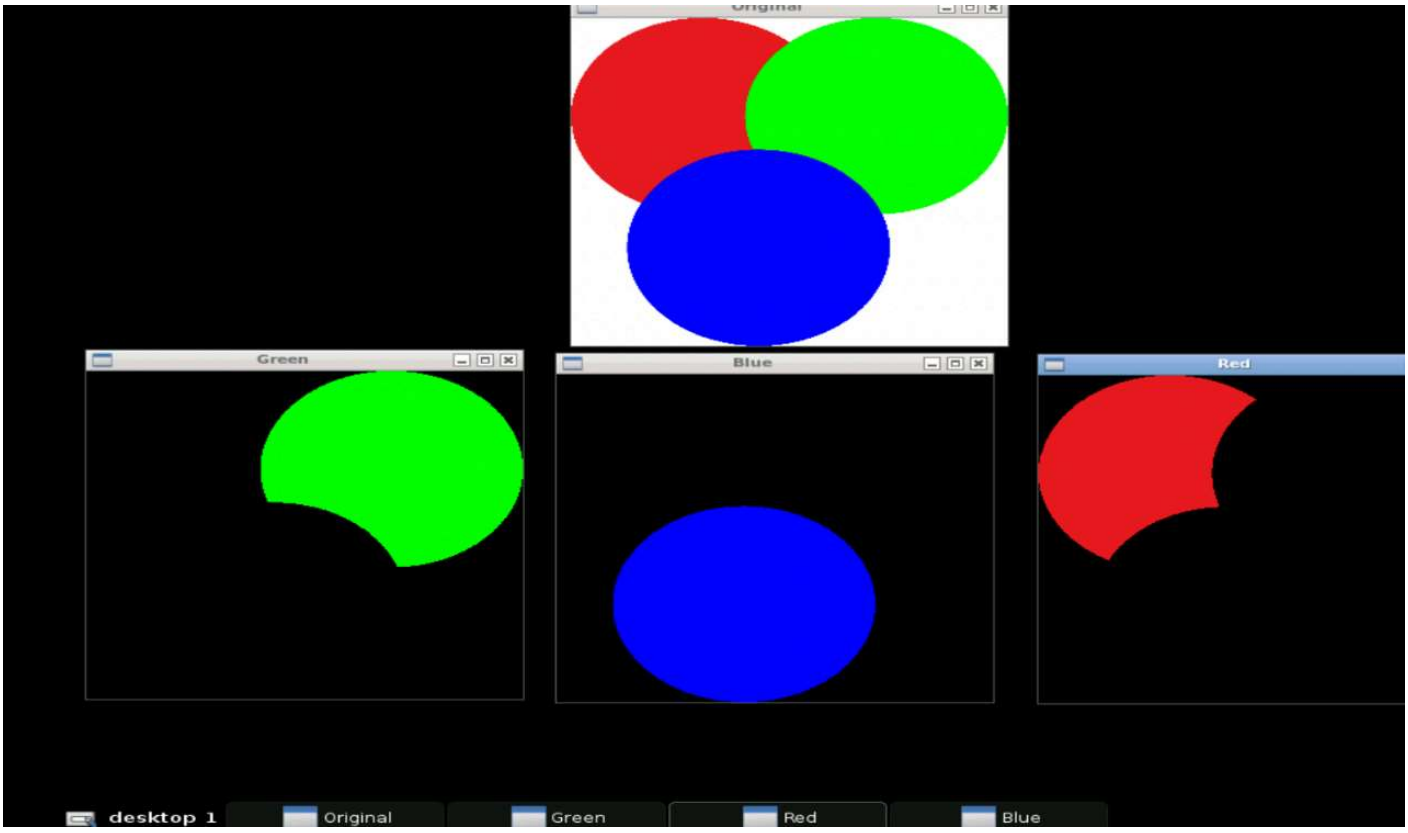
- Exercise 2.4 -

Now let's try to prove this with the architecture we use in ROS. Try this exercise:

- a) In the same package you did before, create a new file called **color_filter.py**.
- b) Try to merge the python code of this section to the architecture of **section 2.2**, the path of the image example is **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/Course_images/Filtering.png'**. Use the **IDE** Tool to write the code.
- c) Show all the results and the original image at the same time. (They will appear in the **Graphical Tools)**
- d) Create a launch file of this code, in the *launch* folder, in order to launch it.

- End of Exercise 2.4 -

- Expected Behavior for Exercise 2.4 -



- End of Expected Behavior for Exercise 2.4 -

Do you see it? You have already proven some filters of HSV. **GREAT JOB!** But you are not using the robot, so why do we use cv_bridge and everything? Because now it's your turn.

- Exercise 2.5 -

Make three different filters and show these 3 filters and the original image from the drone. We want to see in one just the grass, in another one just water, and in the other one just the roof of the house. Go ahead! YOU CAN DO IT!!

You can use the last code as a guide, in order to make the new filters. Just a tip, they are not the same green, red, and blue.

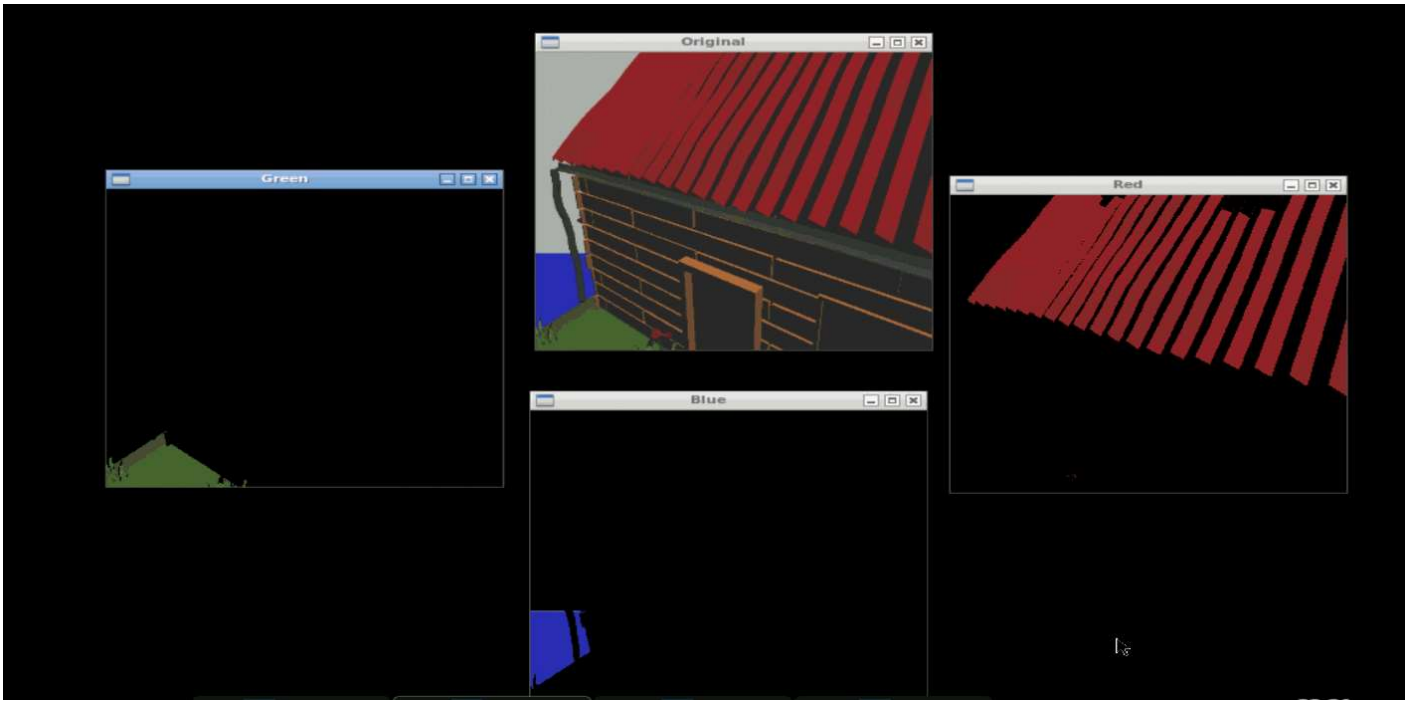
So follow these steps:

- a) In the same package you did before, create a new file called **color_filter2.py**.
- b) Create the code in order to show the 3 filters and the original image in real time. Use the IDE Tool to write the code and remember that the images will apper in the **Graphical Tool**.
- c) Create a launch file of this code, in the *launch* folder, in order to launch it.

- End of Exercise 2.5 -

- Expected Behavior for Exercise 2.5 -

You would have something similar to this! Have a look at this gif!



- End of Expected Behavior for Exercise 2.5 -

That's amazing! If you did it, Congratulations!!! But if you didn't, don't worry and keep trying. If you get stuck, you will find the code below, but see it as a last resort.

GREAT JOB!!! You are doing this very well. Keep learning! and go ahead!

2.5 Edge Detection

Edge detection in the image processing world is very important because it facilitates object recognition, region segmentation of images, and many other tasks. The edges are places on the image where an abrupt change in the levels of gray exist.

For this next chapter, we are going to work with edge detection. First, we will talk about the canny detector, which uses convolution masks and is based on the first derivative. Second is the sobel operator, which also works with convolutions. (It should be noted that the canny detector uses the sobel operator to get the first derivative in the horizontal and vertical direction for the gradient.)

Sobel Operator

The Sobel operator (https://www.researchgate.net/publication/239398674_An_Isotropic_3x3_Image_Gradient_Operator) is used in image processing, especially in edge detection algorithms. The operator calculates the intensity gradient (http://www.cs.umd.edu/~djacobs/CMSC426/ImageGradients.pdf) of an image in every píxel using the convolution function, and the result shows the intensity magnitude changes that could be considered edges. (The convolution is a mathematical operation that can be widely used in the signal processing as a filter - it transforms two functions into a third one, representing how much it changes the second function with respect to the first one.)

The convolution

Let's suppose we have a Matrix M with ***m*** rows by ***n*** columns:

$$M_{m,n} = \begin{bmatrix} M_{1,1} & \dots & M_{1,n} \\ \vdots & & \vdots \\ M_{m,1} & \dots & M_{m,n} \end{bmatrix}$$

And a second matrix (this should be a square Matrix) that we will call the kernel, with ***i*** rows and ***j*** columns:

$$k_{i,j} = \begin{bmatrix} k_{1,1} & \dots & k_{1,j} \\ \vdots & & \vdots \\ k_{i,1} & \dots & k_{i,j} \end{bmatrix}$$

So the convolution between the ***M*** matrix and the kernel ***k*** will be:

$$R_{a,b} = M_{m,n} * k_{i,j}$$

Which results in a third matrix ***R***:

$$R_{a,b} = \begin{bmatrix} R_{1,1} & \dots & R_{1,b} \\ \vdots & & \vdots \\ R_{a,1} & \dots & R_{a,b} \end{bmatrix}$$

Where the rows and columns will be defined by:

$$x = m - i + 1$$
$$y = n - j + 1$$

So the components of R will be given by:

$$R_{(a,b)} = \sum_{r=1}^{x} \sum_{s=1}^{y} (k_{(r,s)} \cdot M_{(a-1+r,b-1+s)})$$

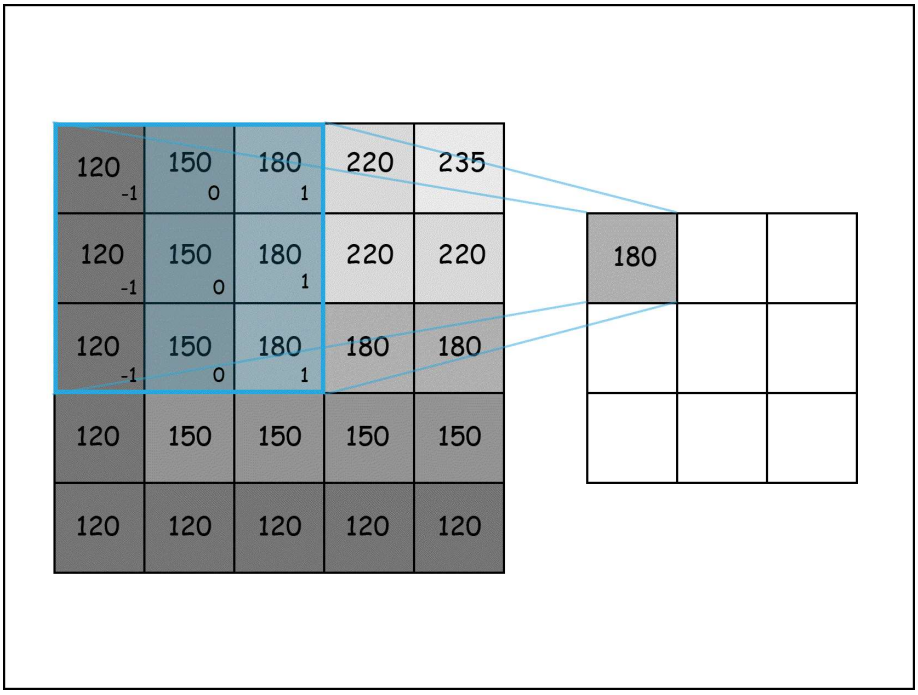
For a better understanding of this operation, let's work with a ***5x5*** Matrix and a ***3x3*** kernel:

120	150	180	220	235
120	150	180	220	220
120	150	180	180	180
120	150	150	150	150
120	120	120	120	120

*

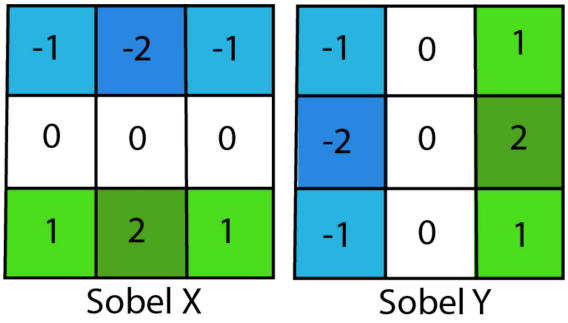
1	0	-1
1	0	-1
1	0	-1

The Convolution operation will be something like:



As can be seen in the gif above, the kernel travels around the image from left to right and top to bottom. The jump distance of the kernel is called stride, which is commonly set as 1px.

It is important that you understand that the convolution operation, the gradient calculated by the sobel operator in an image, is made by convolving the "Sobel Filters" all over the image. We have 2 kernels, for vertical and horizontal gradients:



As always, you will see now how the codes will be if they were using just Open CV and if it works in order to use the Sobel Operator.

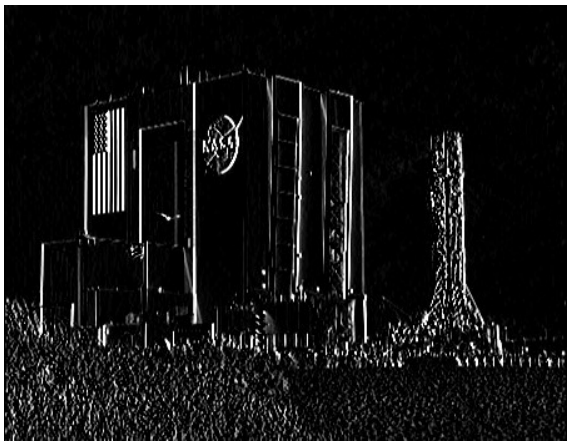
sobelA.py

In [ ]:

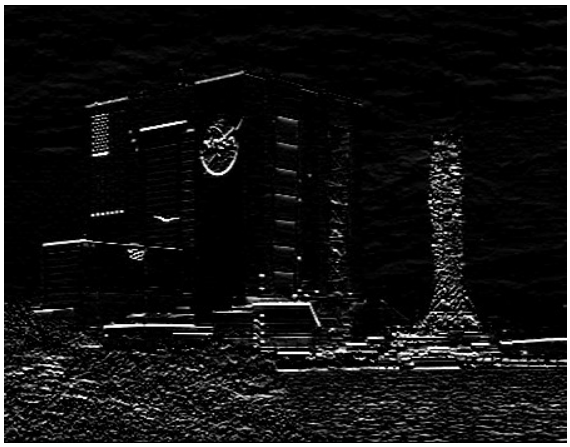
```
import cv2
import numpy as np

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/C

#Convert the image to gray scale so the gradient is better visible
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.resize(img,(450,350))

#Apply the horizontal sobel operator with a kernel size of 3
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=3)

#Apply the vertical sobel operator with a kernel size of 3
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=3)

cv2.imshow('Original',img)
```

This is an open source image from NASA. at https://images.nasa.gov/ (https://images.nasa.gov/)


```
cv2.imshow('sobelx',sobelx)
```



```
cv2.imshow('sobely',sobely)
```



```
cv2.waitKey(1)
cv2.destroyAllWindows()
```

- Exercise 2.6 -

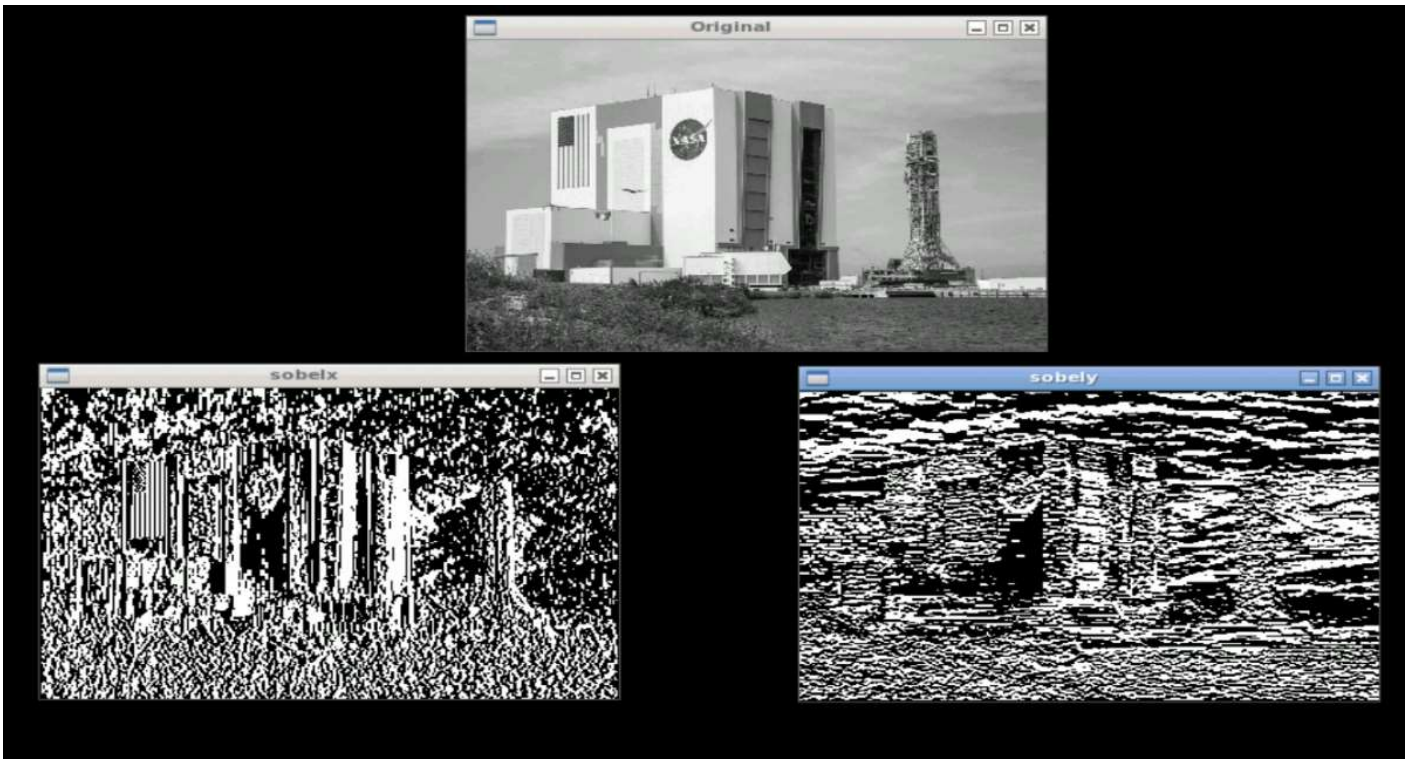
Now we will have the example using the architecture that we use in ROS. Try to do the following steps, the practice time is still running.

- a) In the same package you did before, create a new file called **sobela.py**.
- b) Try to merge the python code of this section to the architecture of **section 2.2**, the path of the image example is **'/home/user/opencv_for_robotics_images/Unit_2/Course_images/test_img_b.jpg'**. Use the IDE Tool to write the code and remember that the images will apper in the **Graphical Tools**.
- c) Create a launch file of this code, in the *launch* folder, in order to launch it and enjoy it.

- End of Exercise 2.6 -

You will see something similar to this picture:

- Expected Behavior for Exercise 2.6 -



- End of Expected Behavior for Exercise 2.6 -

- Exercise 2.7 -

But how do you think it will work with the image of our drone? Let's have a look, try to apply this algorithm to the image of the drone. What do you have to do? Well...

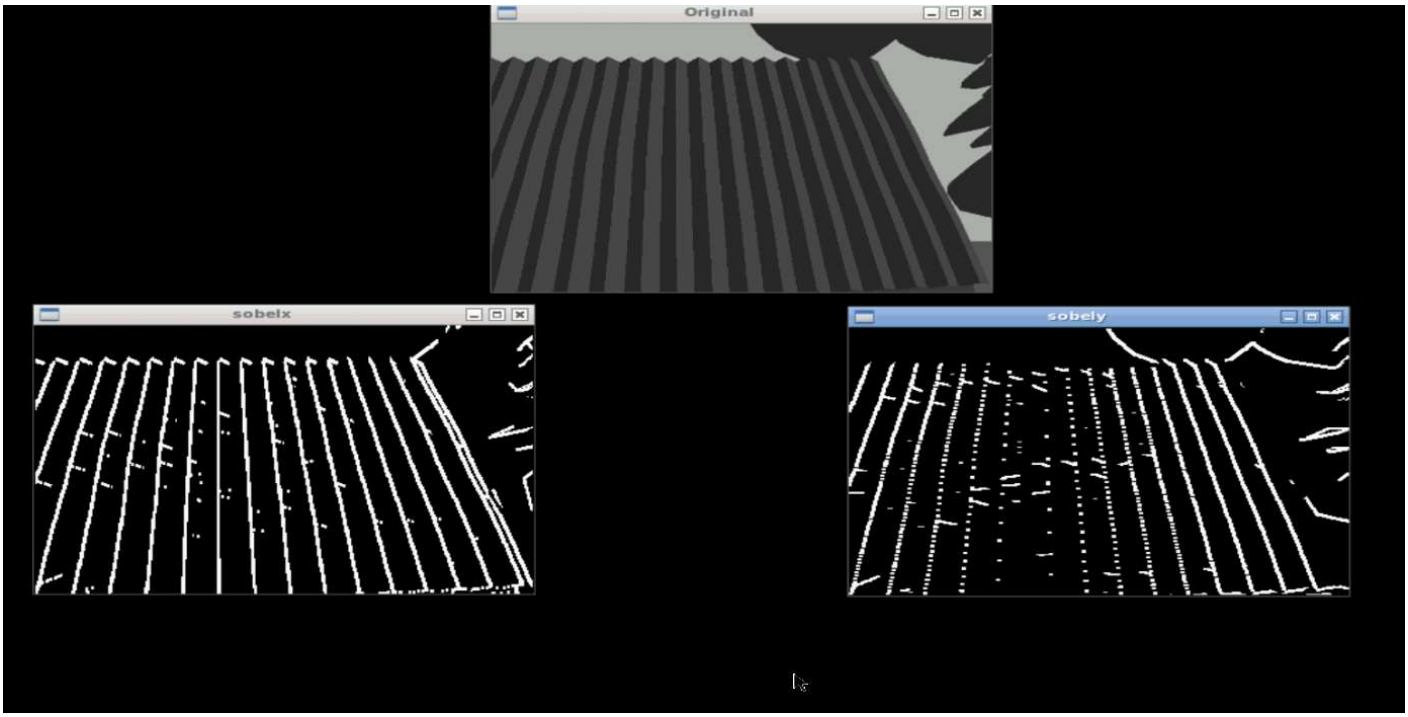
- a) In the same package you did before, create a new file called **sobela2.py**.
- b) Apply the horizontal sobel operator with a kernel size of 3 to the camera's image in real time.
- c) Apply the vertical sobel operator with a kernel size of 3 to the camera's image in real time.
- d) Show both results and the original image in gray scale.
- e) Create a launch file of this code, in the *launch* folder, in order to launch it.

Use the **IDE** Tool to write the code and remember that the images will apper in the **Graphical Tools**.

- End of Exercise 2.7 -

You should have something similar to this interesting gif!

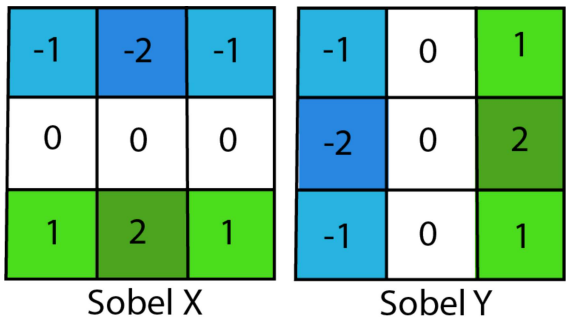
- Expected Behavior for Exercise 2.7 -



- End of Expected Behavior for Exercise 2.7 -

Here is the code of the exercise! But remember, don't be a cheater if you haven't already finished.

This is the method that is already built into Opencv. For a better understanding of the sobel operator, we can create our own sobel operators and use the convolution to extract the gradients, to find the same results. The Sobel kernels are the following, as we saw before:

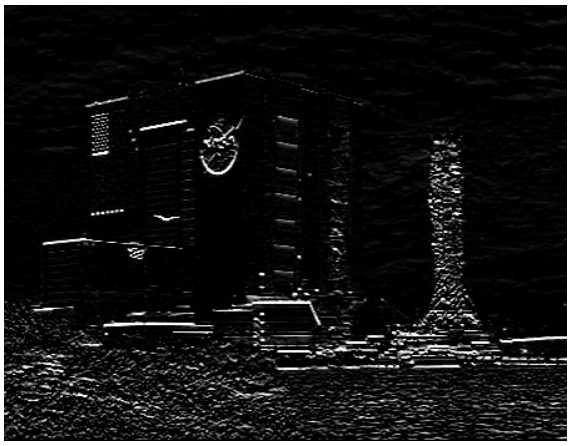


sobelB.py

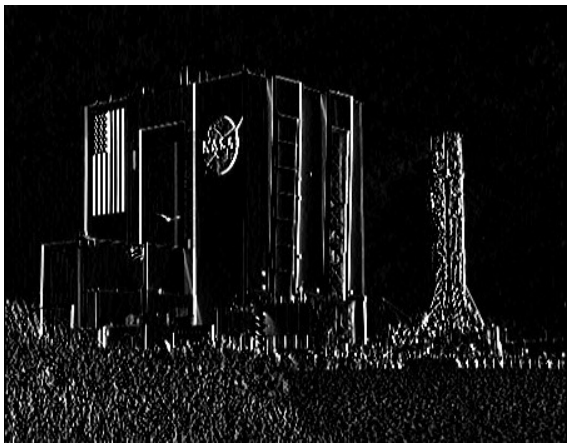
In [ ]:

```
import cv2
import numpy as np

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/C

img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
img = cv2.resize(img,(450,350))

#Here we define the sobel operators
#This are no more than a numpy matrix
kernel_x = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
kernel_y = np.array([[-1,0,1],[-2,-0,2],[-1,0,1]])

#This part is where the magic happens
#We convolve the image read with the kernels defined
x_conv = cv2.filter2D(img,-1,kernel_x)
y_conv = cv2.filter2D(img,-1,kernel_y)

cv2.imshow('Original',img)
```


```
cv2.imshow('sobelx',x_conv)
```



```
cv2.imshow('sobely',y_conv)
```



```
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

As you can see, the results are basically the same. The convolution is a method of filtering images that have been used in the last years for developing complex models of neural networks to work with images and video. This runs out of the idea that instead of a kernel of $3x3$, you can have many n dimensional kernels of mxm size, and their values are not fixed. There are variables that can be trained for any purpose. Under this idea, you could be able to train a filtering model that can detect almost anything you want. Pretty awesome, no?

Feel free to play and experiment with the above code. A good exercise for understanding can be changing the values of the kernels, and also adding more dimensions to the matrix to see what happens.

- Exercise 2.8 -

Let's challenge you. Here is the code using just Open CV. Try to create your own code - go ahead!

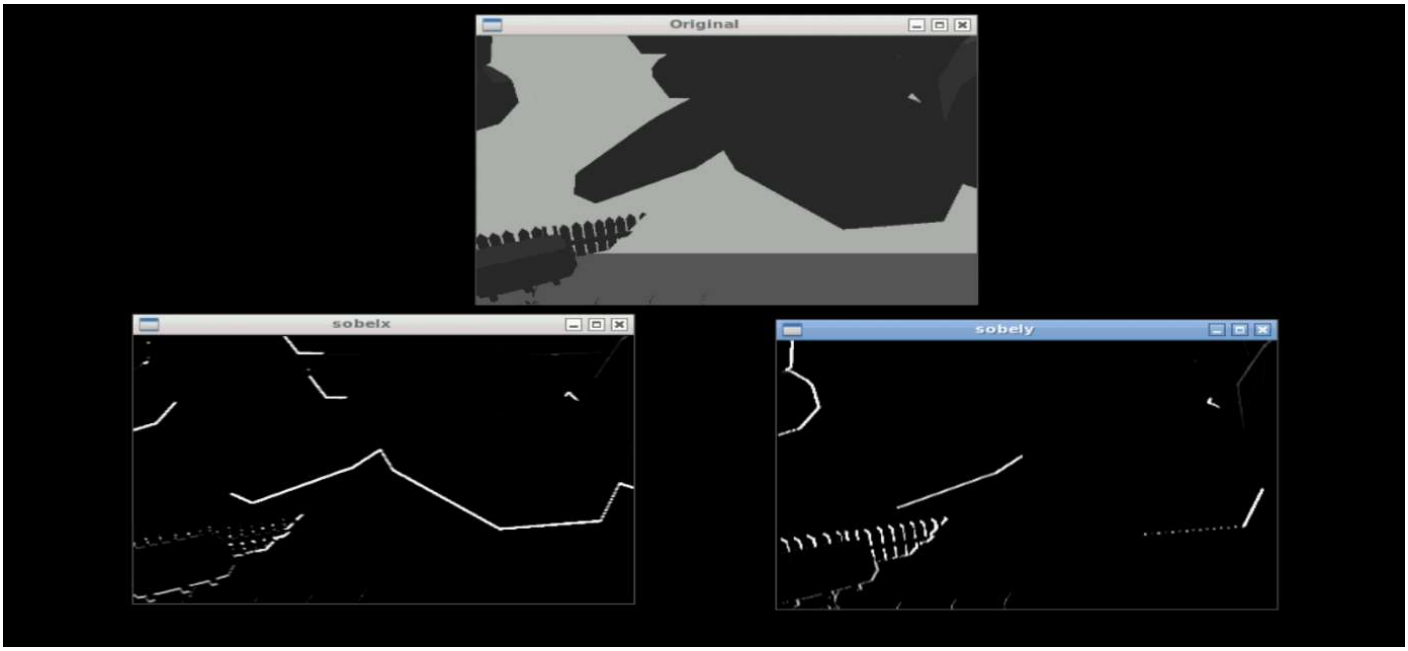
- a) In the same package you did before, create a new file called **sobelb.py.py**.
- b) Apply the horizontal sobel operator using the matrix of Sobel X to the camera's image in real time.
- c) Apply the vertical sobel operator using the matrix of Sobel Y to the camera's image in real time.
- d) Show both results and the original image in gray scale.
- e) Create a launch file of this code, in the *launch* folder, in order to launch it.

Use the **IDE** Tool to write the code and remember that the images will apper in the **Graphical Tools**.

- End of Exercise 2.8 -

You wil have something very similar to this gif:

- Expected Behavior for Exercise 2.8 -



- End of Expected Behavior for Exercise 2.8 -

Canny edge detection

For a better understanding of the canny edge detector, you can visit the Opencv Page (https://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html)

This is a multi-stage algorithm based on the sobel operator, described above. The first stage of the canny detector is Noise reduction, which is done by applying a Gaussian Filter, defined by:

$$G_{x,y} = \frac{1}{\sum_{i=1}^{x}\sum_{j=1}^{y} G_{i,j}} \begin{bmatrix} G_{1,1} & \dots & G_{1,y} \\ \vdots & & \\ G_{x,1} & & G_{x,y} \end{bmatrix}$$

where each value of the kernel is described by:

$$G_{i,j} = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

This step will smooth the image, like the algorithm is based on the derivative to get the gradient (Sobel). This is very sensitive to noise, so this step helps to decrease that sensitivity.

The second stage is acquiring the gradients, where the intensity (Magnitude) and orientation of the edges is calculated. The first part is done by obtaining the derivatives ***Ix*** and ***Iy***, which can be implemented by convolving the image with the horizontal and vertical Sobel kernels.

Once we have both derivatives, the magnitude ***G*** and orientation ***θ*** are defined by:

$$|G| = \sqrt{I_x^2 + I_y^2}$$

$$\theta_{(x,y)} = arctan(\frac{I_x}{I_y})$$

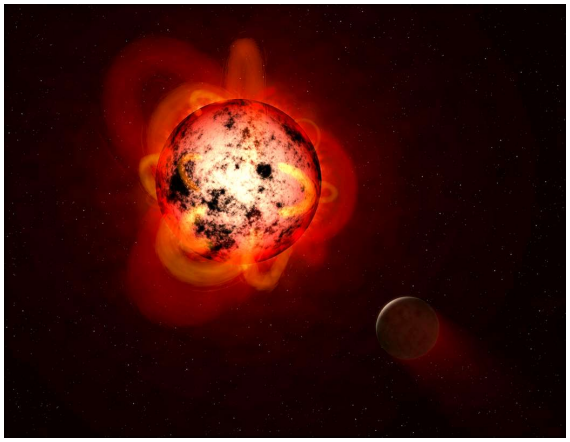
After obtaining this information, a stage of Non Maximal supression (https://arxiv.org/pdf/1705.02950.pdf) is applied, to eliminate pixels that could not be wanted. For this, every pixel is checked to see if it is a local maximum in its neighborhood in the direction of gradient. Finally, a stage of Hysteresis thresholding is applied, which decides which edges are really edges and which are not.

A nice explanation of the Hysteresis can be found here (https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123).

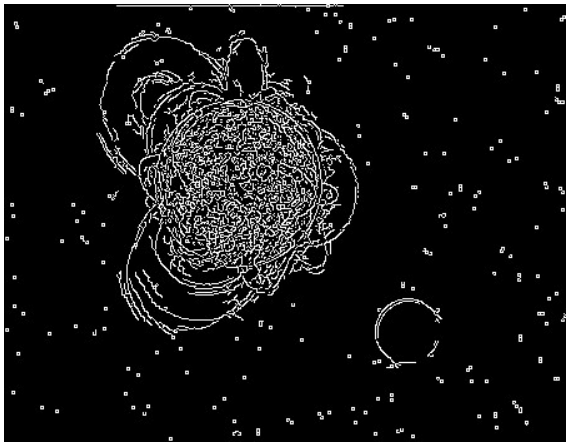
As always, we show you how the code would look if you are just working with Open CV. So, see what is going to happen with the image as an example:

In [ ]:
```
import cv2
import numpy as np

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/C
img = cv2.resize(img,(450,350))

#The canny detector uses two parameters appart from the image:
#The minimum and maximum intensity gradient
minV = 30
maxV = 100

edges = cv2.Canny(img,minV,maxV)
cv2.imshow('Original',img)
```

This is an open source image from NASA. You can find it at https://images.nasa.gov/ (https://images.nasa.gov/).



```
cv2.imshow('Edges',edges)
```



```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The `minV` and `maxV` are considered the limits of intensity gradient. This means that if the gradient intesity is lower than minV, this part of the image is considered a non-edge, so it will be discarded. If the value is higher than maxV, they are considered borders. Finally, those who are in between the limits will be considered edges or non-edges, depending on their connectivity.

- Exercise 2.9 -

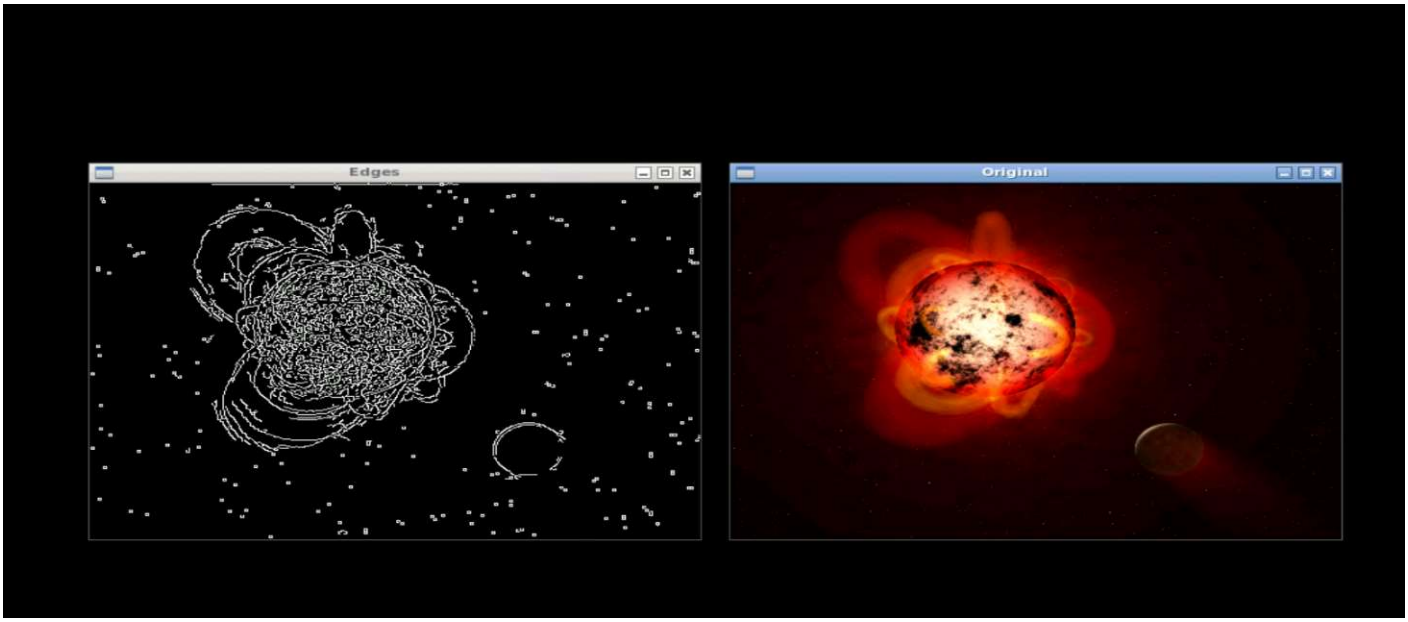
Well now, here is what you were waiting for: the code using the architecture with ROS!!! Come on! Go ahead and have a look!

Follow these steps:

- a) In the same package you did before, create a new file a called it **canny.py**.
- b) Try to merge the python code of this section to the architecture of **section 2.2**, the path of the image example is **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/Course_images/test_img.png'**. Use the IDE Tool to write the code and remember that the images will apper in the **Graphical Tools**.
- c) Create a launch file of this code, in the *launch* folder, in order to launch it.

- End of Exercise 2.9 -

- Expected Behavior for Exercise 2.9 -



- End of Expected Behavior for Exercise 2.9 -

- Exercise 2.10 -

That's so cool! But now it's your turn! Apply this code; we want to see you applying the canny algorithm to the images from your drone!

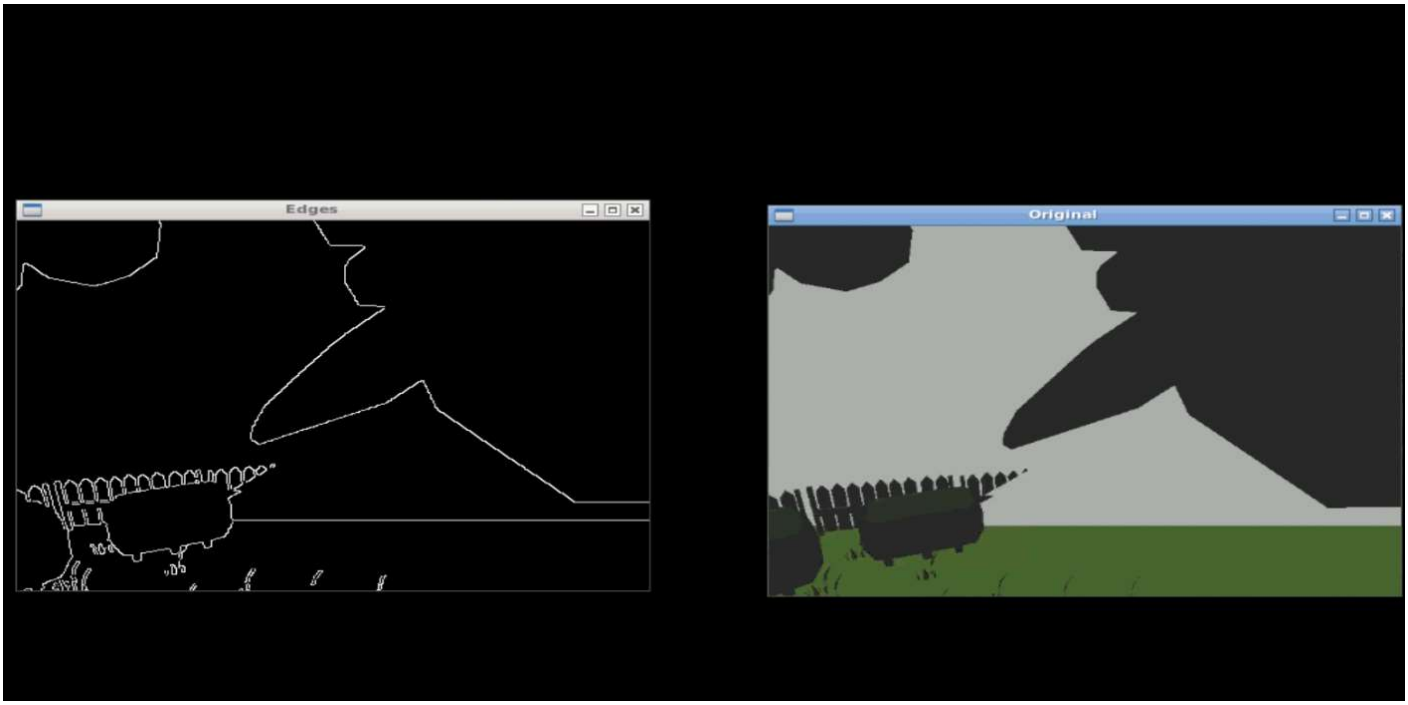
Try it! Go ahead! This way is the best way of learning! Try to do the next steps.

- a) In the same package you did before, create a new file called **canny2.py**.
- b) Apply the canny algorithm in order to see how it works with the image of the camera in real time. Show both images. Use the IDE Tool to write the code and remember that the images will apper in the **Graphical Tools**.
- c) Create a launch file of this code, in the *launch* folder, in order to launch it.

- End of Exercise 2.10 -

- Expected Behavior for Exercise 2.10 -

If you went the right way, you will see something like this amazing gif!



- End of Expected Behavior for Exercise 2.10 -

2.6 Morphological Transformations

Morphological transformations are, in my personal opinion, one of the most important operations in image processing, and they can be helpful with noise supression in images and other tasks. These are simple operations based on the image form commonly applied over a binary image. This works with a matrix kernel that can be, for example, a $5x5$ matrix of ones.

Four of the most common morphological transformations are:

- Erosion
- Dilation
- Opening
- Closing

More information about morphological transformations can be found at the [Opencv Page (https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html)](https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html).

This is the last part of this unit, but not the least important! We have the Open CV code here, and how it changes the images. Let's have a look, and then try it using the architecture of ROS, and finally applying it to the **cv_image**. Go ahead! Come on! It's almost done!

main.py

In [ ]:

```
import cv2
import numpy as np

#Read the image in grayscale
img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/C
img = cv2.resize(img,(450,450))

#Define a kernel for the erosion
kernel_a = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel_a,iterations = 1)

#Define a kernel for the dilation
kernel_b = np.ones((3,3),np.uint8)
dilation = cv2.dilate(img,kernel_b,iterations = 1)

#Define a kernel for the opening
kernel_c = np.ones((7,7),np.uint8)
opening = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel_c)

#Define a kernel for the closing
kernel_d = np.ones((7,7),np.uint8)
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel_d)

cv2.imshow('Original',img)
```

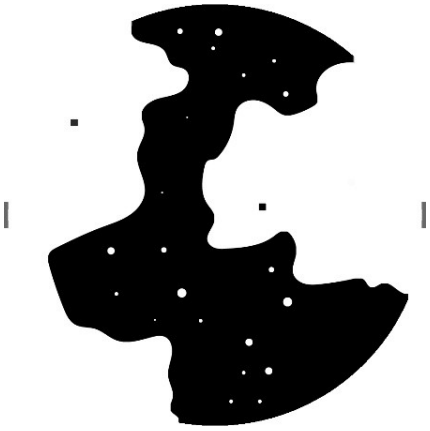
```
cv2.imshow('Erosion',erosion)
```

The erotion transformation provokes the blacks to be wider than the original image.

```
cv2.imshow('Dilation',dilation)
```

Unlike the erosion, the dilation provokes the whites to be wider, as you can see as the borders of the circle became thinner.

```
cv2.imshow('Opening',opening)
```



The opening and the closing are my favourites. They help to eliminate little dots that can be considerewd noise in the image. In the case of the opening, it takes little black dots as noise and supresses them.

```
cv2.imshow('Closing',closing)
```

The closing is similar to the opening, it works with white noise. As you can see, the inner white dots in the circle were almost eliminated from the image.

```
cv2.waitKey(0)
cv2.destroyAllWindows()
```

You can modify the parameters of the transformations to make the effect of them stronger or weaker. It will depend on the application you want.

- Exercise 2.11 -

Now create the same code but with the architecture that we use in ROS throughout this whole unit. But how? You could...

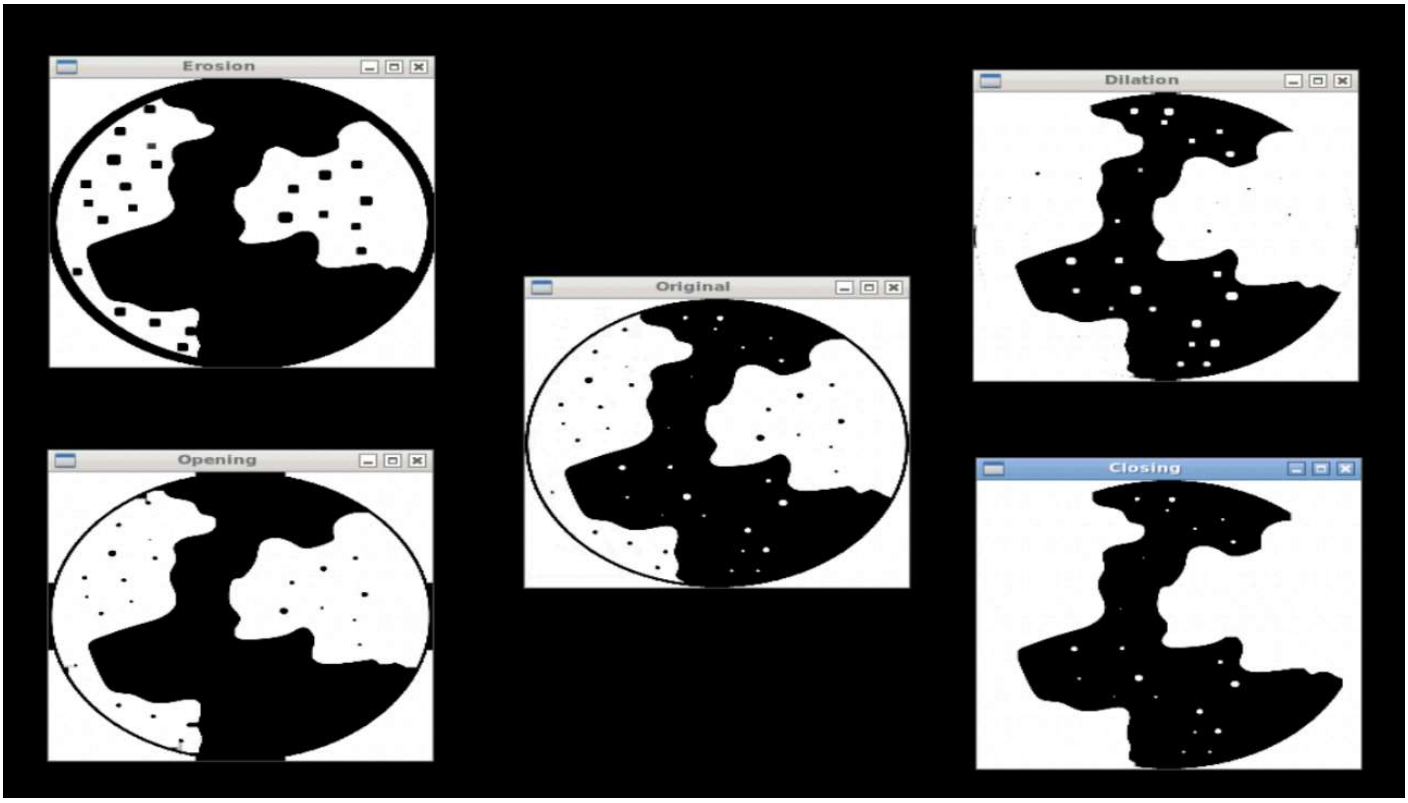
- a) In the same package you did before, create a new file called **transformations.py**.
- b) Try to merge the python code of this section to the architecture of **section 2.2**, the path of the image example is **'/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_2/Course_images/world.png'**.
- c) Show all the morphological filters and the original image at the same time.
- d) Create a launch file of this code, in the *launch* folder, in order to launch it.

Use the **IDE** Tool to write the code and remember that the images will apper in the **Graphical Tools**.

- End of Exercise 2.11 -

You will have something like this!

- Expected Behavior for Exercise 2.11 -



- End of Expected Behavior for Exercise 2.11 -

- Exercise 2.12 -

And finally, let's apply this to the *cv_image*. Come on now, it's your turn, go ahead!

WAIT! Let's do something interesting. Apply these algorithms to both axis of your sobel. Try it! You will find it so cool! So... just follow all these instructions:

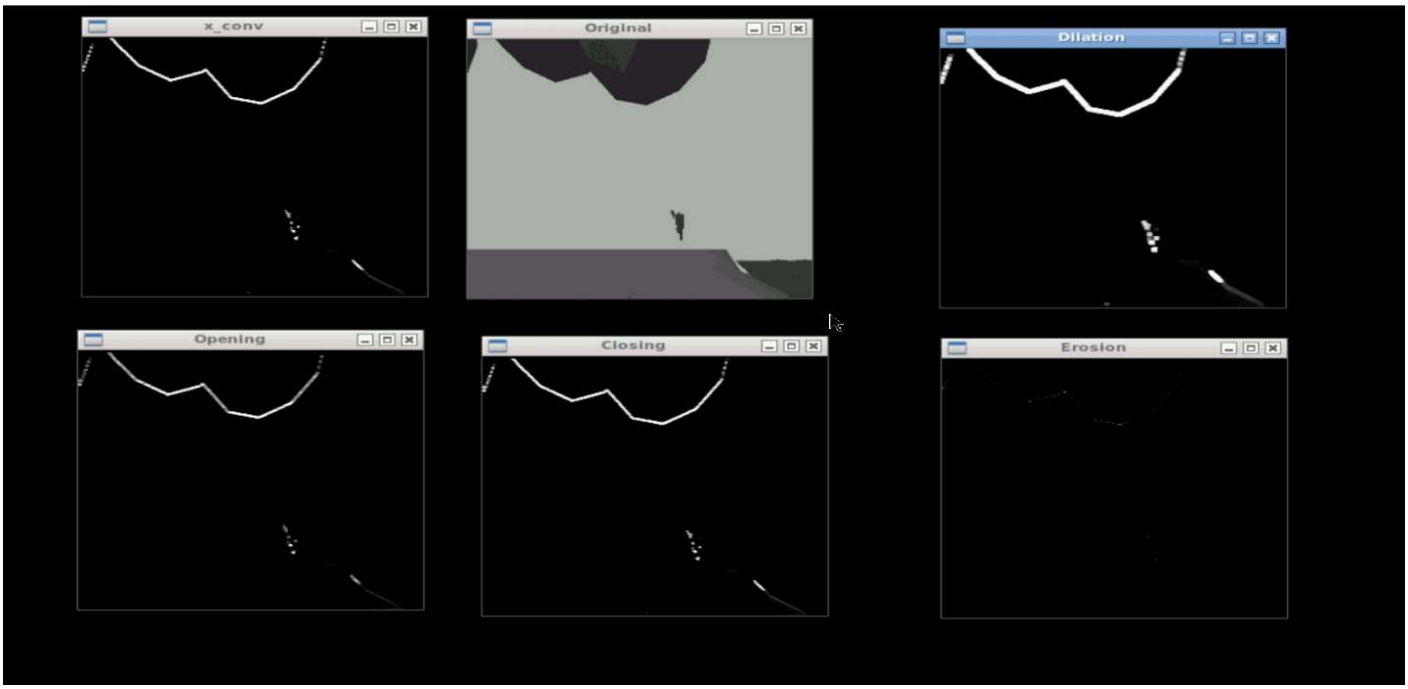
- a) In the same package you did before, create two new files called **transformationsx.py** and **transformationsy.py**.
- b) In the first one, apply all these filters to the results of using the horizontal sobel in the real time image of the drone.
- c) In the second one, apply all these filters to the result of using vertical sobel in the real time image of the drone.
- d) Show all the filters applied, the initial sobel image used, and the original image in gray scale in real time per each code.
- e) Create a launch file per each file and launch them, one at a time. Enjoy the results.

Use the **IDE** Tool to write the code and remember that the images will apper in the **Graphical Tools**.

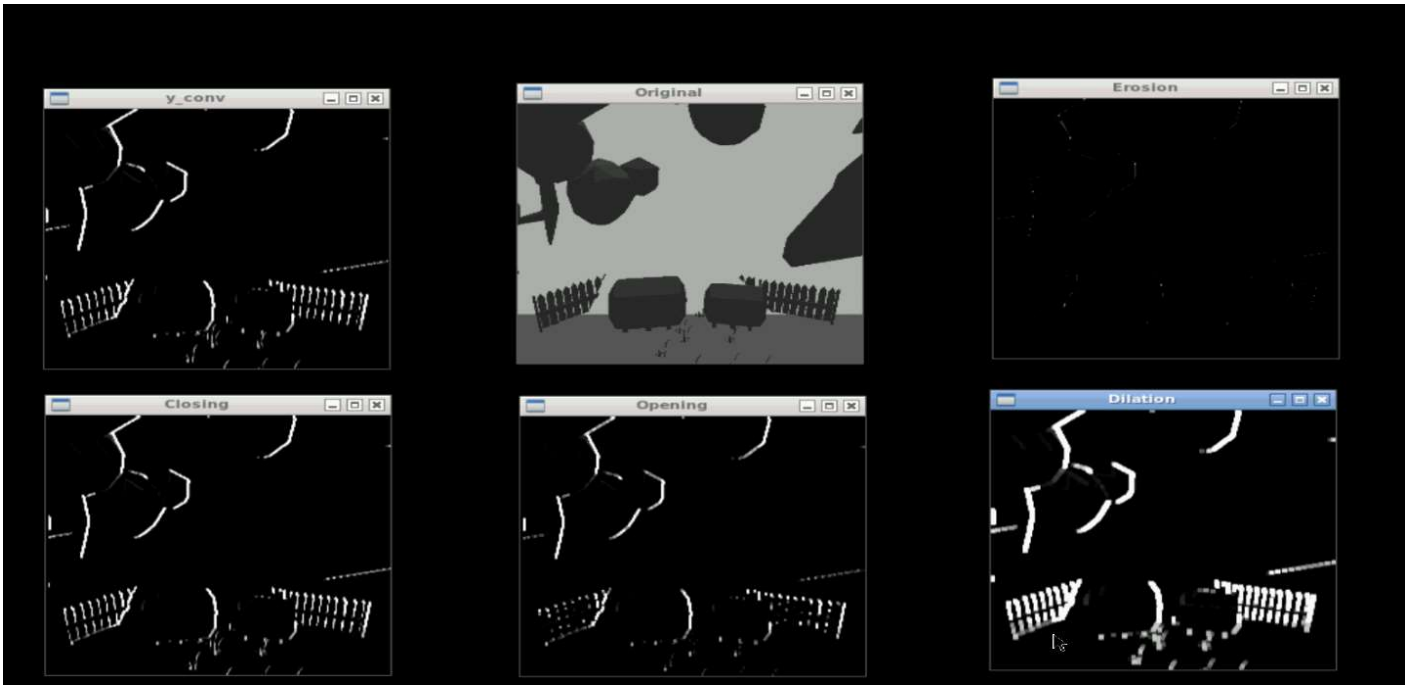
- End of Exercise 2.12 -

- Expected Behavior for Exercise 2.12 -

Look at how the x axis responds:



Now it's the y axis' turn:



- End of Expected Behavior for Exercise 2.12 -

Can you now differentiate each morphological transformation? We hope so! Congratulations! You have finished this unit! Let's go to the next one!

Don't worry if you want to see the codes for each axis. Here they are.

GOOD JOB! GO TO THE NEXT UNIT!