# MOOC Autonomous Mobile Robots Week 3

## Week 3 - Vision

Vision is perhaps the most powerful sense in humans, providing a huge amount of information about the environment, and enabling rich and intelligent interaction.

In this module, we present the fundamentals of computer vision and image processing for a locomotion task: the robot will follow a line on the ground using a camera.

- Acquiring Images (Acquiring%20Images.ipynb)
- Image Processing (Image%20Processing.ipynb)
- Line Detection (Line%20Detection.ipynb)
- Line Following (Line%20Following.ipynb)
- Line Following with Obstacle Avoidance (Line%20Following%20Obstacle.ipynb)

This module is based on Chapter 12 (Follow-Bot) of **Programming Robots with ROS** by Morgan Quigley, Brian Gerkey, and William D. Smart (http://wiki.ros.org/Books/Programming_Robots_with_ROS).

---

**Try-a-Bot: an open source guide for robot programming**

Developed by:

Universitat Jaume I Robotic Intelligence Lab (http://robinlab.uji.es)

Sponsored by:

IEEE Robotics & Automation Society (http://www.ieee-ras.org) CYBERBOTICS professional mobile robot simulation (http://www.cyberbotics.com) The Construct Just Simulate (http://www.theconstructsim.com)

## Acquiring Images


Image acquisition is the first stage of any vision system. It consists of the action of retrieving an image from some source (usually a camera device) and storing it in the computer for further processing.

In this course, a Kinect sensor is mounted on the Pioneer robot. This is a camera device that captures not only color but also *depth*. However, we are only going to use the color information.

In this notebook, you will move the robot around and learn how to capture and display an image.

First, as usual, we will initialize the robot.

In [ ]:
```
import packages.initialization
import pioneer3dx as p3dx
p3dx.init()
```

Next, we need to import the plotting libraries for displaying the images.

In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
# REMINDER: this cell may take some seconds to execute the first time
```

The motion GUI widget allows you to move the robot around.

In [ ]:
```
import motion_widget
```

### Tilting

The Kinect sensor features a motorized tilt mechanism, which is capable of tilting the sensor up to 27º either up or down (approximately 0.47 radians).

The next GUI widget controls the tilt angle of the simulated Kinect.

In [ ]:
```
import tilt_widget
```

### Acquisition and Display

Finally, the image is automatically stored in a variable that can be passed to the image plot function:

In [ ]:
```
plt.imshow(p3dx.image);
# Click here and press Shift+Enter to refresh the image
```

The image is stored as a numpy array (http://www.scipy-lectures.org/intro/numpy/array_object.html), which is very similar to a Matlab/Octave array.

For example, its dimensions can be obtained with:

In [ ]:
```
p3dx.image.shape
```

This result indicates that the image consists of 100 rows and 150 columns of RGBA (https://en.wikipedia.org/wiki/RGBA_color_space) pixels.

Next: Image Processing (Image%20Processing.ipynb)

## Image Processing

(http://opencv.org/) In this module, we will use OpenCV (http://opencv.org/) in Python to process the images coming through the camera from the simulated Pioneer 3DX.

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision.

In our task, the goal is to detect the location of the target line and follow it around the course. There are many strategies that can be used for that purpose, whose complexity increases with variability and noise. In our case, we are just going to consider an optimally painted, optimally illuminated bright cyan line.

The strategy will be to filter a block of rows of the image by color and drive the robot toward the center of the pixels that pass the color filter.

First, we initialize the robot, launch the widgets, and display the camera image.

In [ ]:
```
import packages.initialization
import pioneer3dx as p3dx
p3dx.init()
```

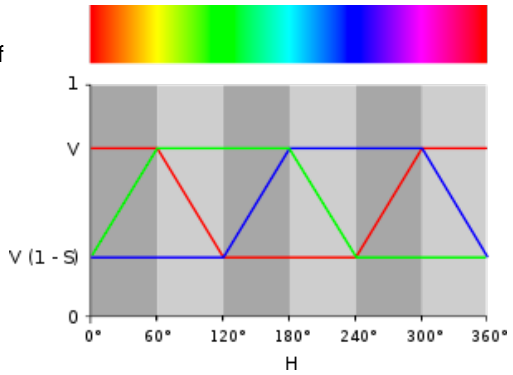
In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [ ]:
```
import motion_widget
```

In [ ]:
```
import tilt_widget
```

In [ ]:
```
plt.imshow(p3dx.image);
```

### Color Filtering




(https://en.wikipedia.org/wiki/HSL_and_HSV) The first idea would be to find the red, green, blue (RGB) values of a cyan image pixel and filter for nearby RGB values. Unfortunately, filtering on RGB values turns out to be a poor way to find a particular color in an image, since the raw values are a function of the overall brightness as well as the color of the object. Slightly different lighting conditions would result in the filter failing to perform as intended.

Instead, a better technique for filtering by color is to transform RGB images into hue, saturation, value (HSV) (https://en.wikipedia.org/wiki/HSL_and_HSV) images. The HSV image separates the RGB components into hue (color), saturation (color intensity), and value (brightness). Once the image is in this form, we can then apply a threshold for hues near cyan to obtain a *binary image* in which pixels are either true (meaning they pass the filter) or false (they do not pass the filter).

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

In [ ]:
```
hsv = cv2.cvtColor(p3dx.image, cv2.COLOR_RGB2HSV)
```

The cyan color (http://www.colorhexa.com/00ffff) has a hue angle of 180 degrees (of 360), a saturation of 100% and a value of 100%. However, since OpenCV uses a different scale (H: 0 - 180, S: 0 - 255, V: 0 - 255), the cyan hue angle will be 90 units.

In real lighting conditions, colors are not defined by single values, but by intervals, so we will use an interval of $\pm 10$ units around the central value.

Since the illumination is not extremely bright, the thresholds for saturation and value are set to 100.

In [ ]:
```
lower_cyan = numpy.array([80, 100, 100])
upper_cyan = numpy.array([90, 255, 255])
```

The mask is computed by the OpenCV function `inRange` (http://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#inrange).

In [ ]:
```
mask = cv2.inRange(hsv, lower_cyan, upper_cyan)
```

In [ ]:
```
plt.imshow(mask,cmap='gray');
```

Next: Line Detection (Line%20Detection.ipynb)

## Line Detection


The result of image processing was a binary image, or mask, with the pixels that belong to the line, i.e. the cyan-colored pixels.

For driving the robot, we need to compute some value relating the position of the robot to the line in the ground. In this task, it is sufficient to keep the line centered in the image. For more complex tasks, there are algorithms that compute the geometrical parameters of the line image, and its 3D reconstruction in real space.

Our method is far simpler: we consider the line as a blob in the image, whose *image moments* (http://aishack.in/tutorials/image-moments/) can be computed, particularly its *centroid* (https://en.wikipedia.org/wiki/Image_moment#Central_moments). That information will be used for later driving the robot appropriately.

#### Image Acquisition

In [ ]:
```
import packages.initialization
import pioneer3dx as p3dx
p3dx.init()
```

In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

In [ ]:
```
import motion_widget
```

In [ ]:
```
import tilt_widget
```

In [ ]:
```
plt.imshow(p3dx.image);
```

#### Image Processing

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

In [ ]:
```
hsv = cv2.cvtColor(p3dx.image, cv2.COLOR_RGB2HSV)
lower_cyan = numpy.array([80, 100, 100])
upper_cyan = numpy.array([100, 255, 255])
mask = cv2.inRange(hsv, lower_cyan, upper_cyan)
plt.imshow(mask,cmap='gray');
```

#### Computing the Centroid

We use here some heuristics: first, the Kinect should tilt down for observing the line close to the robot, not far away; second, we will only consider the bottom part of the line for computing the image moments; doing so will prevent the robot to turn before it actually arrives to the curve. In practice, we will set all the pixels to black (zeros) for the lines between 0 and 80.

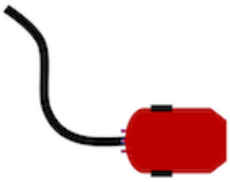
In [ ]:
```
mask[0:80, 0:150] = 0
plt.imshow(mask,cmap='gray');
```

Finally, we compute the moments, the centroid (https://en.wikipedia.org/wiki/Image_moment#Central_moments), and display the original image, with a red circle at the position of the centroid of the computed blob. If the result is correct, the circle should be centered on the bottom of the cyan line.

In [ ]:
```
M = cv2.moments(mask)
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])
plt.imshow(p3dx.image)
axes = plt.gca()
axes.add_artist(plt.Circle((cx,cy),10,color='r'));
```

Next: Line Following (Line%20Following.ipynb)

# Line Following



Up to now, we have worked up a line detection algorithm. Now that this scheme is up an running, we can move on to the task of driving the robot such that the line stays near the center of the camera image.

We propose to use a *proportional* controller (https://en.wikipedia.org/wiki/Proportional_control), which means that a linear scaling of an error drives the control output. In this case, the error signal is the distance between the center of the image and the center of the line that we are trying to follow. The control output is the steering (angular velocity) of the robot.

In [1]:
```
import packages.initialization
import pioneer3dx as p3dx
p3dx.init()
```

In [2]:
```
import cv2
import numpy
```

## Image processing

Fill in the necessary code in the following function, which computes the centroid of the line of the image passed as an argument, as explained in the previous notebook.

In [4]:
```
def line_centroid(image):
    ...
    return cx, cy
```

We need the code for the motion of the robot with the given linear and angular velocities, as in previous modules.

In [5]:
```
def move(V_robot,w_robot):
    r = 0.1953 / 2
    L = 0.33
    w_r = (2 * V_robot + L * w_robot) / (2*r)
    w_l = (2 * V_robot - L * w_robot) / (2*r)
    p3dx.move(w_l, w_r)
```

## Main loop

This is the main control loop. The error should be computed as:

$$err = C_x - \frac{width}{2}$$

where $C_x$ is the $x$-coordinate of the centroid, and $width$ is the width of the image.

The linear velocity is constant, e.g. $2m/s$ and the angular velocity $\omega$ is computed as:

$$\omega = -K_p err$$

where $K_p$ is the gain of the proportional controller, which can be set to $0.01$.

In [ ]:

```
p3dx.tilt(-0.47) # tilt down the Kinect
try:
    width = ...
    while True:
        cx, cy = line_centroid(p3dx.image)
        err = ...
        linear = ..
        angular = ...
        move(linear, angular)
except KeyboardInterrupt:
    move(0,0)
```

Next: Line Following with Obstacle Avoidance (Line%20Following%20Obstacle.ipynb)

## Line Following with Obstacle Avoidance

The final task of this week is a combination of the line following, obstacle detection, and wall following behaviors.

The robot should follow the line until an obstacle is detected in its path. Then, the robot will turn right and follow the wall at its right until the line is detected again, and it will resume the line following behavior.

Please watch the following demo video:

In [ ]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('Jd1jpt3pgc8')
```

This is the most complex task that we have programmed so far, thus it is a nice candidate for developing with the so-called **top-down** approach (https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design). With this methodology, we start with a high-level algorithm, and break it down into its components:

```
repeat forever
    follow line until an obstacle is detected
    get close to the wall
    follow wall until a line is detected
    get close to the line
```

### Initialization

First, we need to import all the required modules.

In [ ]:
```
import packages.initialization
import pioneer3dx as p3dx
p3dx.init()
import cv2
import numpy
```

### Component functions

The first function must return `True` if an obstacle is detected in front of the robot, or `False` otherwise.

In [ ]:
```
def is_obstacle_detected():
    ...
```

The second function is the line following behavior as seen in previous notebooks during this week.

In [ ]:
```
def follow_line():
    print('Following the line')
    while not is_obstacle_detected():
        ...
    print('Obstacle detected')
```

The next function was developed in the previous week: the robot turns until it is approximately parallel to the wall.

In [ ]:
```
def getWall():
    ...
```

The next function is checked during the wall following behavior: it must return `True` when the line is again detected, or `False` otherwise.

In [ ]:
```
def is_line_detected():
    ...
```

Next, we reuse the wall following behavior from previous week.

In [ ]:
```
def follow_wall():
    print('Following the wall')
    while not is_line_detected():
        ...
    print('Line detected')
```

Finally, a function is needed for turning the robot slightly until it is approximately parallel to the line again.

In [ ]:
```
def getLine():
    ...
```

Some additional lower-level functions are required (guess which ones?).

You can define them in the next empty cell.

In [ ]:
```
# Lower-level functions
...
```

### Main loop

The main loop looks very similar to the proposed algorithm:

In [ ]:
```
p3dx.tilt(-0.47)
try:
    while True:
        follow_line()
        getWall()
        follow_wall()
        getLine()
except KeyboardInterrupt:
    move(0,0)
```

Did it work? Congratulations, you have completed the task of this week!

**Try-a-Bot: an open source guide for robot programming**

Developed by:

(http://robinlab.uji.es)

Sponsored by:

(http://www.ieee-ras.org) (http://www.cyberbotics.com) (http://www.theconstructsim.com)

Follow us:

(https://www.facebook.com/RobotProgrammingNetwork)

(https://www.youtube.com/user/robotprogrammingnet)