Rover Project Test Notebook

This notebook contains the functions from the lesson and provides the scaffolding you need to test out your mapping methods. The steps you need to complete in this notebook for the project are the following:

- First just run each of the cells in the notebook, examine the code and the results of each.
- Run the simulator in "Training Mode" and record some data. Note: the simulator may crash if you try to record a large (longer than a few minutes) dataset, but you don't need a ton of data, just some example images to work with.
- Change the data directory path (2 cells below) to be the directory where you saved data
- Test out the functions provided on your data
- Write new functions (or modify existing ones) to report and map out detections of obstacles and rock samples (yellow rocks)
- Populate the `process_image()` function with the appropriate steps/functions to go from a raw image to a worldmap.
- Run the cell that calls `process_image()` using `moviepy` functions to create video output
- Once you have mapping working, move on to modifying `perception.py` and `decision.py` to allow your rover to navigate and map in autonomous mode!

Note: If, at any point, you encounter frozen display windows or other confounding issues, you can always start again with a clean slate by going to the "Kernel" menu above and selecting "Restart & Clear Output".

Run the next cell to get code highlighting in the markdown cells.

Student

Document edited by Miguel A. Colmenares from Oct/08/2017

In [1]:
```
%%HTML
<style> code {background-color : orange !important;} </style>
```

In [2]:
```
%matplotlib inline
#%matplotlib qt # Choose %matplotlib qt to plot to an interactive win
dow (note it may show up behind your browser)
# Make some of the relevant imports
import cv2 # OpenCV for perspective transform
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import scipy.misc # For saving images as needed
import glob  # For reading in a list of images from a folder
import imageio
imageio.plugins.ffmpeg.download()
```

Quick Look at the Data

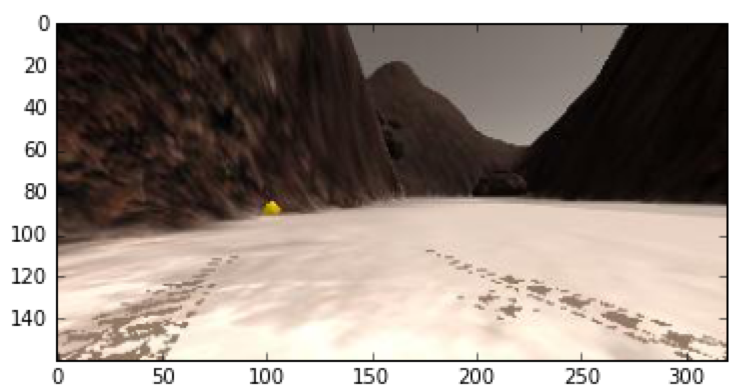
There's some example data provided in the `test_dataset` folder. This basic dataset is enough to get you up and running but if you want to hone your methods more carefully you should record some data of your own to sample various scenarios in the simulator.

Next, read in and display a random image from the `test_dataset` folder

In [9]:
```
path = '../test_dataset/IMG/*'
img_list = glob.glob(path)
print(len(img_list), "Imagenes")
# Grab a random image and display it
idx = np.random.randint(0, len(img_list)-1)
print("Seleccionada la imagen Nro", idx)
image = mpimg.imread(img_list[idx])
print("Direccion de la imagen", img_list[idx])
plt.imshow(image)
```

```
397 Imagenes
Seleccionada la imagen Nro 5
Direccion de la imagen ../test_dataset/IMG/robocam_2017_10_19_22_18_38_490.jpg
```

Out[9]: `<matplotlib.image.AxesImage at 0x7f747d1862b0>`



Calibration Data

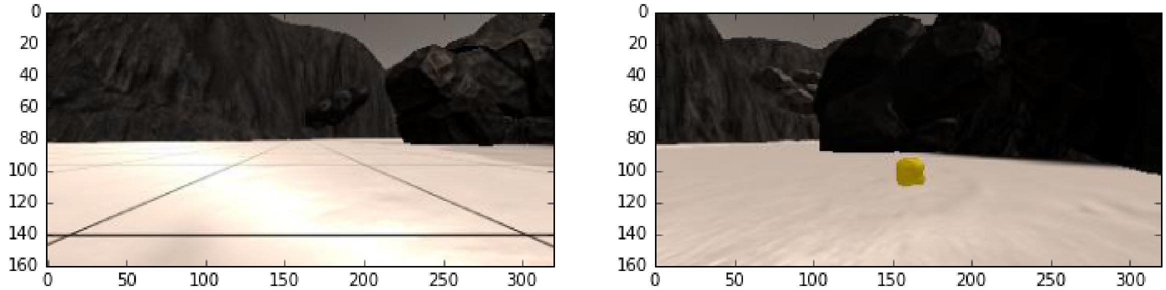
Read in and display example grid and rock sample calibration images. You'll use the grid for perspective transform and the rock image for creating a new color selection that identifies these samples of interest.

In [4]:
```
# In the simulator you can toggle on a grid on the ground for calibration
# You can also toggle on the rock samples with the 0 (zero) key.
# Here's an example of the grid and one of the rocks
example_grid = '../calibration_images/example_grid1.jpg'
example_rock = '../calibration_images/example_rock1.jpg'
grid_img = mpimg.imread(example_grid)
rock_img = mpimg.imread(example_rock)

fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(grid_img)
plt.subplot(122)
plt.imshow(rock_img)

print("Nro de dimensiones:",rock_img.ndim)
print("Nro de elementos por dimension:",rock_img.shape)
```

```
Nro de dimensiones: 3
Nro de elementos por dimension: (160, 320, 3)
```



Perspective Transform

Define the perspective transform function from the lesson and test it on an image.

In [5]:
```
# Define a function to perform a perspective transform
# I've used the example grid image above to choose source points for
 the
# grid cell in front of the rover (each grid cell is 1 square meter i
n the sim)
# Define a function to perform a perspective transform
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1],
img.shape[0]))# keep same size as input image
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shap
e[1], img.shape[0]))
    # La mascara se implementa para obtener imformacion valida de don
de se puede caminar
    # ya que, como se observa en la imagen, la tarnsformada de perspe
ctiva genera
    # unos destellos ubicados en una zona donde hay un ostaculo.
    # la mask conforma los 180 grados de vision del robot
```

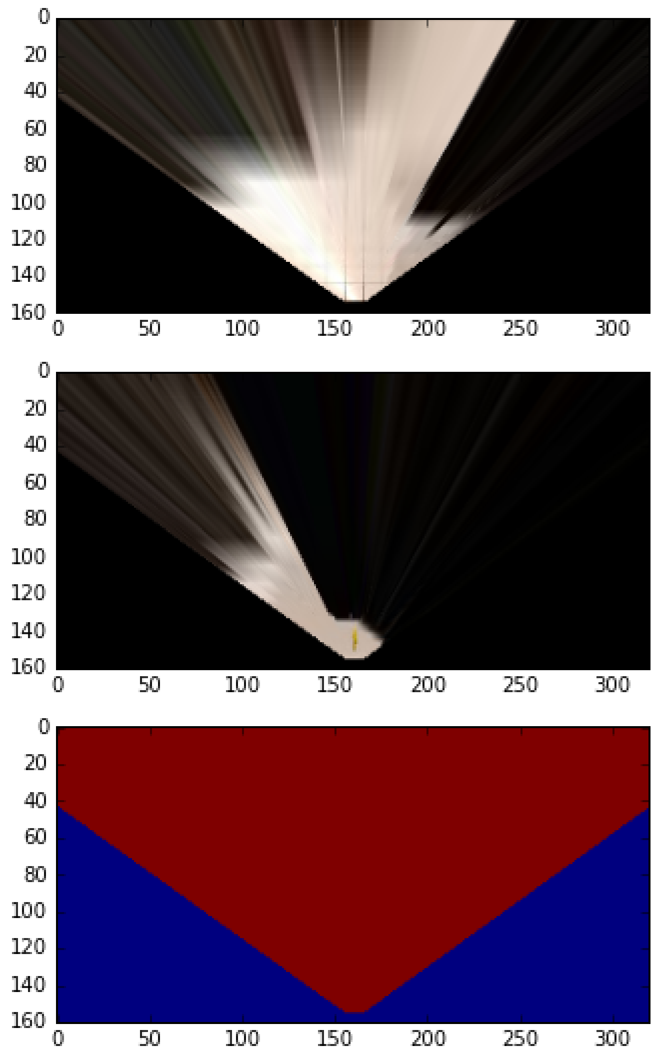
```
    return warped, mask


# Define calibration box in source (actual) and destination (desired)
 coordinates
# These source and destination points are defined to warp the image
# to a grid where each 10x10 pixel square represents 1 square meter
# The destination box will be 2*dst_size on each side
dst_size = 5
# Set a bottom offset to account for the fact that the bottom of the
 image
# is not the position of the rover but a bit in front of it
# this is just a rough guess, feel free to change it!
bottom_offset = 6
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
# Miro la imagen con la rejilla y con un programa ubico
# la posicion x e y de las esquinas del rectangulo. Para la fuente
print(source,"\n")
destination = np.float32([[image.shape[1]/2 - dst_size,
image.shape[0] - bottom_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - bott
om_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - 2*ds
t_size - bottom_offset],
                  [image.shape[1]/2 - dst_size, image.shape[0] - 2*ds
t_size - bottom_offset],
                  ])
print(destination)
warpedgrid, maskgrid = perspect_transform(grid_img, source, destinati
on)
warpedrock, maskrock = perspect_transform(rock_img, source, destinati
on)
fig = plt.figure(figsize=(12,9))
plt.subplot(321)
plt.imshow(warpedgrid)
plt.subplot(323)
plt.imshow(warpedrock)
plt.subplot(325)
plt.imshow(maskgrid)
#scipy.misc.imsave('../output/warped_example.jpg', warped)
```

```
[[  14.  140.]
 [ 301.  140.]
 [ 200.   96.]
 [ 118.   96.]]

[[ 155.  154.]
 [ 165.  154.]
 [ 165.  144.]
 [ 155.  144.]]
```

```
Out[5]: <matplotlib.image.AxesImage at 0x7f747d4e32b0>
```



Color Thresholding

Define the color thresholding function from the lesson and apply it to the warped image

TODO: Ultimately, you want your map to not just include navigable terrain but also obstacles and the positions of the rock samples you're searching for. Modify this function or write a new function that returns the pixel locations of obstacles (areas below the threshold) and rock samples (yellow rocks in calibration images), such that you can map these areas into world coordinates as well.
Hints and Suggestion:

- For obstacles you can just invert your color selection that you used to detect ground pixels, i.e., if you've decided that everything above the threshold is navigable terrain, then everthing below the threshold must be an obstacle!

- For rocks, think about imposing a lower and upper boundary in your color selection to be more specific about choosing colors. You can investigate the colors of the rocks (the RGB pixel values) in an interactive matplotlib window to get a feel for the appropriate threshold range (keep in mind you may want different ranges for each of R, G and B!). Feel free to get creative and even bring in functions from other libraries. Here's an example of color selection (http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html) using OpenCV.
- **Beware However:** if you start manipulating images with OpenCV, keep in mind that it defaults to `BGR` instead of `RGB` color space when reading/writing images, so things can get confusing.

In [6]:

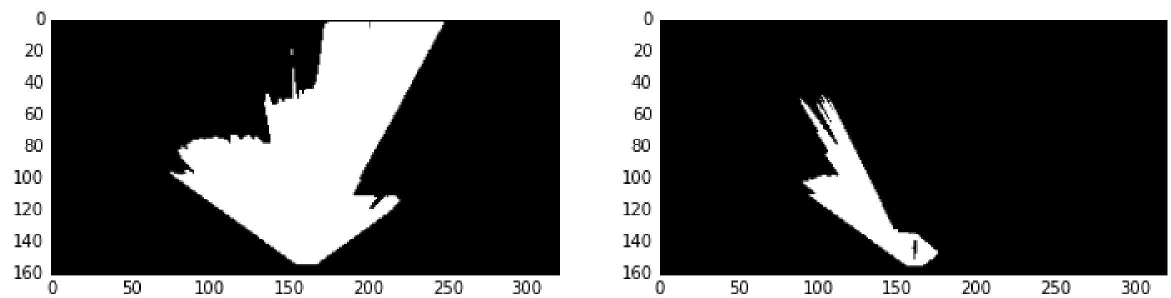
```
# Identify pixels above the threshold
# Threshold of RGB > 160 does a nice job of identifying ground pixels
 only
def color_thresh(img, rgb_thresh=(110, 110, 110)):
    # Create an array of zeros same xy size as img, but single channe
l
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in
 RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])

    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select

threshed_grid = color_thresh(warpedgrid)
threshed_rock = color_thresh(warpedrock)
#rockthreshed = color_thresh(rock_img)
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.imshow(threshed_grid, cmap='gray')
plt.subplot(122)
plt.imshow(threshed_rock, cmap='gray')
#plt.imshow(rockthreshed, cmap='gray')
#scipy.misc.imsave('../output/warped_threshed.jpg', threshed*255)
```

Out[6]:

```
<matplotlib.image.AxesImage at 0x7f747d5b7630>
```



Coordinate Transformations

Define the functions used to do coordinate transforms and apply them to an image.

In [7]:

```
# Define a function to convert from image coords to rover coords
def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position
 being at the
    # center bottom of the image.
```

```
    x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
    return x_pixel, y_pixel

# Define a function to convert to radial coords in rover space
def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dist, angles

# Define a function to map rover space pixels to world space
def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    yaw_rad = yaw * np.pi / 180
    xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix *
np.sin(yaw_rad))

    ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix *
np.cos(yaw_rad))
    # Return the result
    return xpix_rotated, ypix_rotated

def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated


# Define a function to apply rotation and translation (and clipping)
# Once you define the two functions above this function should work
def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
    # Apply rotation
    xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
    # Apply translation
    xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, yp
os, scale)
    # Perform rotation, translation and clipping all at once
    x_pix_world = np.clip(np.int_(xpix_tran), 0, world_size - 1)
    y_pix_world = np.clip(np.int_(ypix_tran), 0, world_size - 1)
    # Return the result
    return x_pix_world, y_pix_world

# Grab another random image
idx = np.random.randint(0, len(img_list)-1)
image = mpimg.imread(img_list[idx])
warped, mask = perspect_transform(image, source, destination)
threshed = color_thresh(warped)

# Calculate pixel values in rover-centric coords and distance/angle t
o all pixels
xpix, ypix = rover_coords(threshed)
```

```
dist, angles = to_polar_coords(xpix, ypix)
print(angles.shape,"\n")
print(dist, " \n", angles, "\n")
mean_dir = np.mean(angles)
print(mean_dir, "\n")
if -0.4 <= mean_dir <= 0.4:
    anglesnew = np.zeros_like(angles.shape)
    angnew = np.mean(anglesnew)
    print("Nuevo angulo: ", angnew)



# Do some plotting
fig = plt.figure(figsize=(12,9))
plt.subplot(221)
plt.imshow(image)
plt.subplot(222)
plt.imshow(warped)
plt.subplot(223)
plt.imshow(threshed, cmap='gray')
plt.subplot(224)
plt.plot(xpix, ypix, '.')
plt.ylim(-160, 160)
plt.xlim(0, 160)
arrow_length = 100
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
plt.arrow(0, 0, x_arrow, y_arrow, color='red', zorder=2, head_width=1
0, width=2)
```

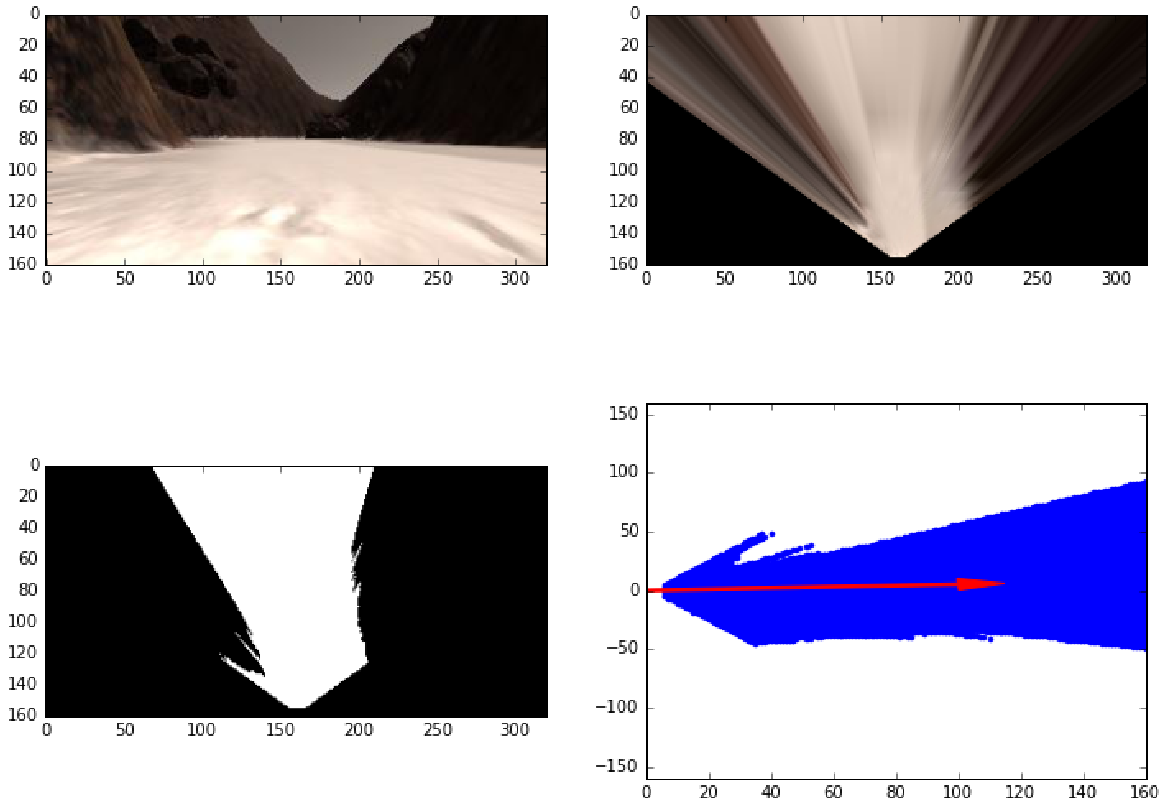
```
(13509,)

[ 185.0648535   184.56435192  184.06792225 ...,    6.70820393    7.21
110255
    7.81024968]
 [ 0.52651863  0.52183428  0.51712455 ..., -0.46364761 -0.5880026
 -0.69473828]

0.0514070222412

Nuevo angulo:  0.0
```

```
Out[7]: <matplotlib.patches.FancyArrow at 0x7f747d334a90>
```



Finding samples

Defines the function used to find yellow samples and apply them to an image.

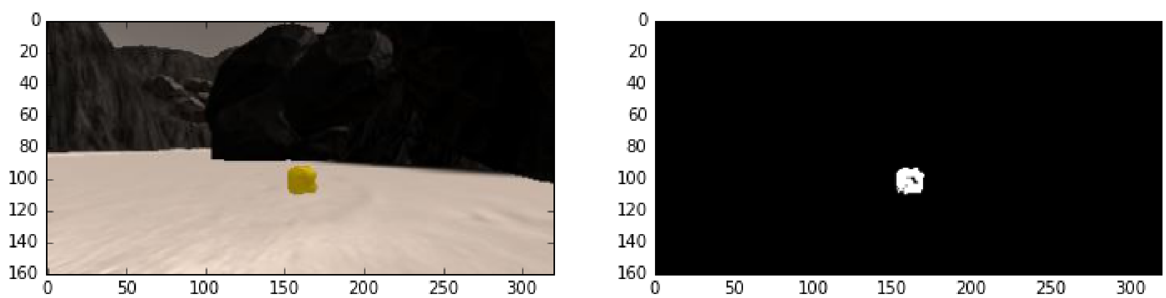
```
In [8]: def find_rocks (img, levels=(110,110,50)):
            rockpix = ((img[:,:,0] > levels[0]) \
                      & (img[:,:,1] > levels[1]) \
                      & (img[:,:,2] < levels[2]))
            # Esta seccion se explica diciendo que el amarillo se compone de
         juntar
            # el rojo y el verde y suprimiendo el azul por consiguiente
            # busco estar por encima (>) del rojo y del verde
            # y por debajo (<) del azul

            color_select = np.zeros_like(img[:,:,0])
            color_select[rockpix] = 1

            return color_select

        rock_map = find_rocks(rock_img)
        fig = plt.figure(figsize=(12,3))
        plt.subplot(121)
        plt.imshow(rock_img)
        plt.subplot(122)
        plt.imshow(rock_map, cmap='gray')
```

```
Out[8]: <matplotlib.image.AxesImage at 0x7f73b8e417f0>
```



Read in saved data and ground truth map of the world

The next cell is all setup to read your saved data into a `pandas` dataframe. Here you'll also read in a "ground truth" map of the world, where white pixels (pixel value = 1) represent navigable terrain.

After that, we'll define a class to store telemetry data and pathnames to images. When you instantiate this class (`data = Databucket()`) you'll have a global variable called `data` that you can refer to for telemetry and map data within the `process_image()` function in the following cell.

In [9]:
```
# Import pandas and read in csv file as a dataframe
import pandas as pd
# Change the path below to your data directory
# If you are in a locale (e.g., Europe) that uses ',' as the decimal
 separator
# change the '.' to ','
df = pd.read_csv('../test_dataset/robot_log.csv', delimiter=';', deci
mal='.')
csv_img_list = df["Path"].tolist() # Create list of image pathnames
# Read in ground truth map and create a 3-channel image with it
ground_truth = mpimg.imread('../calibration_images/map_bw.png')
ground_truth_3d = np.dstack((ground_truth*0, ground_truth*255, ground
_truth*0)).astype(np.float)

# Creating a class to be the data container
# Will read in saved data from csv file and populate this object
# Worldmap is instantiated as 200 x 200 grids corresponding
# to a 200m x 200m space (same size as the ground truth map: 200 x 20
0 pixels)
# This encompasses the full range of output position values in x and
 y from the sim
class Databucket():
    def __init__(self):
        self.images = csv_img_list
        self.xpos = df["X_Position"].values
        self.ypos = df["Y_Position"].values
        self.yaw = df["Yaw"].values
        self.count = 0 # This will be a running index
        self.worldmap = np.zeros((200, 200, 3)).astype(np.float)
        self.ground_truth = ground_truth_3d # Ground truth worldmap

# Instantiate a Databucket().. this will be a global variable/object
# that you can refer to in the process_image() function below
data = Databucket()
```

Write a function to process stored images

Modify the `process_image()` function below by adding in the perception step processes (functions defined above) to perform image analysis and mapping. The following cell is all set up to use this `process_image()` function in conjunction with the `moviepy` video processing package to create a video from the images you saved taking data in the simulator.

In short, you will be passing individual images into `process_image()` and building up an image called `output_image` that will be stored as one frame of video. You can make a mosaic of the various steps of your analysis process and add text as you like (example provided below).

To start with, you can simply run the next three cells to see what happens, but then go ahead and modify them such that the output video demonstrates your mapping process. Feel free to get creative!

In [10]:

```
# Define a function to pass stored images to
# reading rover position and yaw angle from csv file
# This function will be used by moviepy to create an output video
```

```
def process_image(img):
    # Example of how to use the Databucket() object defined above
    # to print the current x, y and yaw values
    # print(data.xpos[data.count], data.ypos[data.count], data.yaw[da
ta.count])

    # TODO:
    # 1) Define source and destination points for perspective transfo
rm
    # 2) Apply perspective transform
    # 3) Apply color threshold to identify navigable terrain/obstacle
s/rock samples
    # 4) Convert thresholded image pixel values to rover-centric coor
ds
    # 5) Convert rover-centric pixel values to world coords

    # Perspective Transform and mask
    warped, mask = perspect_transform(image, source, destination)
    # Apply color threshold to identify navigable terrain/obstacles/r
ock samples
    threshed = color_thresh(warped)
    # Se crea un mapa de obstaculos que es el valor absoluto de (thre
shed-1) * mask
    obs_map = np.absolute(np.float32(threshed)-1) * mask
    # Calculate pixel values in rover-centric coords and distance/ang
le to all pixels
    xpix, ypix = rover_coords(threshed)
    # Convert rover-centric pixel values to world coords
    world_size = data.worldmap.shape[0]
    scale = 2 * dst_size
    xpos = data.xpos[data.count]
    ypos = data.ypos[data.count]
    yaw = data.yaw[data.count]
    x_world, y_world = pix_to_world(xpix, ypix, xpos, ypos, yaw, worl
d_size, scale)
    obsxpix, obsypix = rover_coords(obs_map)
    obs_x_world, obs_y_world = pix_to_world(obsxpix, obsypix, xpos, y
pos, yaw, world_size, scale)

    # 6) Update Rover worldmap (to be displayed on right side of scre
en)
        # Example: data.worldmap[obstacle_y_world, obstacle_x_world,
 0] += 1
        #          data.worldmap[rock_y_world, rock_x_world, 1] += 1
        #          data.worldmap[navigable_y_world, navigable_x_worl
d, 2] += 1
    data.worldmap[y_world, x_world, 2] = 255
    data.worldmap[obs_y_world, obs_x_world, 0] = 255
    nav_pix = data.worldmap[:,:,2] > 0
    data.worldmap[nav_pix,0] = 0

    # See if we can find some rock
    rock_map = find_rocks (warped, levels=(110,110,50))
    if rock_map.any():
        rock_x, rock_y = rover_coords(rock_map)
        rock_x_world, rock_y_world= pix_to_world(rock_x, rock_y,
xpos, ypos, yaw, world_size, scale)
```

```
        data.worldmap[rock_y_world, rock_x_world, :] = 255

    # 7) Make a mosaic image, below is some example code
        # First create a blank image (can be whatever shape you like)
    output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], i
mg.shape[1]*2, 3))
        # Next you can populate regions of the image with various out
put
        # Here I'm putting the original image in the upper left hand
 corner
    output_image[0:img.shape[0], 0:img.shape[1]] = img

        # Let's create more images to add to the mosaic, first a warp
ed image
    #warped = perspect_transform(img, source, destination)
        # Add the warped image in the upper right hand corner
    output_image[0:img.shape[0], img.shape[1]:] = warped

        # Overlay worldmap with ground truth map
    map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth, 0.
5, 0)
        # Flip map overlay so y-axis points upward and add to output_
image
    output_image[img.shape[0]:, 0:data.worldmap.shape[1]] =
np.flipud(map_add)


        # Then putting some text over the image
    cv2.putText(output_image,"Populate this image with your analyses
 to make a video!", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    if data.count < len(data.images) - 1:
        data.count += 1 # Keep track of the index in the Databucket()

    return output_image
```

Make a video from processed image data

Use the moviepy (https://zulko.github.io/moviepy/) library to process images and create a video.