Ridecell: Camera and LIDAR Calibration and Visualization in ROS

By Munir Jojo-Verge (June 2018)

Assignment Description

This assignment is given to test your skills in ROS, PCL, OpenCV etc.

There are 2 tasks to perform:

- Task 1: Calculate (using code/script) the camera calibration, and use it to rectify the image as shown here http://wiki.ros.org/image_proc (http://wiki.ros.org/image_proc)
- Task 2: Calculate (using code/script) translation and rotation offset between camera and lidar, and wire static transform accordingly and show overlay in rviz.

Submit videos of screen or pictures and code (as zip files or github link)

Link to ROS Bag file http://gofile.me/6qNOh/5XdKNtJ5n (http://gofile.me/6qNOh/5XdKNtJ5n)

The checkboard pattern used 5x7 inside corners and size of each square 5cm

Goals

The goals / steps of this project are the following:

- Inspect & play the bag file
- Compute the camera calibration matrix and distortion coefficients given:
  - The ROS bag, and
  - a set of images (in this case extracted from the bag)
- If time permits, compare the 2 calibration values and proof that both methos should be "good" (as in less than 5% difference).
- Apply a distortion correction to raw images: Create a "corrected" ROS bag
-

ROS Nano-introduction

ROS provides a powerful build and package management system called Catkin. A Catkin workspace is essentially a directory where Catkin packages are built, modified and installed.

Typically when you're developing a ROS based robot or project, you will be working out of a single workspace.

This singular workspace will hold a wide variety of Catkin packages.

All ROS software components are organized into and distributed as Catkin packages. Similar to workspaces, Catkin packages are nothing more than directories containing a variety of resources which, when considered together constitute some sort of useful module.

Catkin packages may contain source code for nodes,useful scripts, configuration files and more.

We will start by creating a new catkin workspace, and getting all necessary packages, solving all dependencies, and in general getting everything ready for this assignment.

My Virtual Machine wasn't ready for a 3.2G ROS bag so I had to extend the physical and logical drives and partitions and spend some time getting all ready to work.

Our "workspace" and all the assignment files will be located on the "ridecell" folder (catkin worksapce) on:

In [ ]:
```
cd "/media/robond/e2507505-dfde-40e2-9c5d-a7ecc505e0f0/ridecell"
```

this folder was initialized as our catkin workspace using the following command.

```
$ catkin_init_workspace
```

and built with

```
$ catkin_make
```

The entire workspace structure looks like:

In [ ]:
```
!ls
```

A ROS system usually consists of many running nodes.

Running all of the nodes by hand though can be torturous.

This is where the roslaunch command comes to save the day.

Roslaunch allows you to:

- launch multiple nodes with one simple command,
- set default parameters in the pram server,
- automatically respond processes that have died and
- much more.

To use roslaunch, you must first make sure that your **work space has been built and sourced.**

```
$ source devel/setup.bash
```

With our workspace built and sourced we can now start solving this task by creating the ncessary srcripts and launching all the necessary nodes.

Inspecting & Playing the bag file

What does the bag file contain?

In [5]:
```
!rosbag info 2016-11-22-14-32-13_test.bag
```

```
path:        2016-11-22-14-32-13_test.bag
version:     2.0
duration:    1:53s (113s)
start:       Nov 22 2016 14:32:14.41 (1479853934.41)
end:         Nov 22 2016 14:34:07.88 (1479854047.88)
size:        3.1 GB
messages:    5975
compression: none [1233/1233 chunks]
types:       sensor_msgs/CameraInfo  [c9a58c1b0b154e0e6da7578cb991d214]
             sensor_msgs/Image       [060021388200f6f0f447d0fcd9c64743]
             sensor_msgs/PointCloud2 [1158d486dd51d683ce2f1be655c3c181]
topics:      /sensors/camera/camera_info   2500 msgs    : sensor_msgs/Camer
aInfo
             /sensors/camera/image_color   1206 msgs    : sensor_msgs/Image
             /sensors/velodyne_points      2269 msgs    : sensor_msgs/Point
Cloud2
```

If you run sucessfully this command you should get something like:

```
path:        2016-11-22-14-32-13_test.bag
version:     2.0
duration:    1:53s (113s)
start:       Nov 22 2016 14:32:14.41 (1479853934.41)
end:         Nov 22 2016 14:34:07.88 (1479854047.88)
size:        3.1 GB
messages:    5975
compression: none [1233/1233 chunks]
types:       sensor_msgs/CameraInfo  [c9a58c1b0b154e0e6da7578cb991d214]
             sensor_msgs/Image       [060021388200f6f0f447d0fcd9c64743]
             sensor_msgs/PointCloud2 [1158d486dd51d683ce2f1be655c3c181]
topics:      /sensors/camera/camera_info   2500 msgs    : sensor_msgs/CameraInfo
             /sensors/camera/image_color   1206 msgs    : sensor_msgs/Image
             /sensors/velodyne_points      2269 msgs    : sensor_msgs/PointCloud2
```

To play the video we can use the "play" argument as follow:

In [ ]:
```
!rosbag play 2016-11-22-14-32-13_test.bag
```

In [ ]:
```
!rosbag play -r 0.5 2016-11-22-14-32-13_test.bag
```

Task 1: Camera Calibration (`cameracalibrator.py`)

The first step will be to read in calibration images of a chessboard. During my Self-Driving Car Nanodegree lectures, it was recommeded to use at least 20 images to get a reliable calibration. Since I didn't get a hold of the ros bag file inmediatelly, I used a different set of images for illustration & research purposes, althogh the distortion correction was performed with the calibration obtained from the data file provided. My own set of chessboard images is located on "myChessboard" folder and each chessboard image has nine by six inside corners to detect.

After I got the ros bag, the first step was to inspect it and see what did it contain.

The camera calibration can be done in 2 different ways:

Note: As mentioned on the assignment description, the checker board pattern used 5 x 7 inside corners and size of each square 5 cm.

Calibration through a Video (ROS bag)

The following series of commands will "play" the bag file (run on one terminal) and run the "camera_calibrarion" ROS node on a separare terminal to collect enough images to cover the X, Y, Size, and Skew parameter spaces needed for correction. I tried to play the bag file at different speeds (1, 0.5 and 0.2 of the normal speed) to see if I could collect more images while going slower and therefore improving the correction values. The results were exactly the same. I collected 23 images.

To learn how to use "camera_calibrarion" the perfect tutorial is ROS Wiki (http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration), which I relied on heaviy to develop this task.

suncessful instalation of "camera_calibrarion" required, on my setup, the installation of ROS kinetic (http://wiki.ros.org/kinetic/Installation/Ubuntu (http://wiki.ros.org/kinetic/Installation/Ubuntu)).

After the entire Ros Kinetic library was installed I proceded to install and compile the "camera_calibration" dependecies.

This method, as mentioned before requires 2 terminals: One playing the bag and another one capturing and gathering the calibration parameters.

Here's the last set of commands needed to perform this Calibration:

```
$ rosdep install camera_calibration

$ rosmake camera_calibration

$ rosbag play -r 0.5 2016-11-22-14-32-13_test.bag
```

and on a separete terminal you shuould run:

```
$ rosrun camera_calibration cameracalibrator.py --size=5x7 --square=0.050 image:=
/sensors/camera/image_color camera:=/sensors/camera/camera_info  --no-service-che
ck
```

As the tutorial clearly states, as the video plays and the checkerboard moves around you will see three bars on the calibration sidebar increase in length. When the CALIBRATE button lights, you have enough data for calibration and can click CALIBRATE to see the results. After running the entire bag and pressing "Calibrate" you will see the calibration results in the terminal and the calibrated image in the calibration window.

When you click on the "Save" button after a succesfull calibration, the data (calibration data and images used for calibration) will be written to /**tmp**/**calibrationdata.tar.gz**. Below, when using the second method, we will see that the calibration get's aslo saven in the same place and with the same file name.

The Results

```
('D = ', [-0.20046456284402592, 0.06947530966095249, 0.003302010137310338, 0.0002
1698698103442295, 0.0])
('K = ', [485.07003816979477, 0.0, 457.19389875599717, 0.0, 485.4215104101991, 36
5.2938207194185, 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 = ', [427.07855224609375, 0.0, 461.2431237551573, 0.0, 0.0, 433.6468505859375
, 369.92239138540754, 0.0, 0.0, 0.0, 1.0, 0.0])
None
#oST version 5.0 parameters

[image]

width
964

height
724

[narrow_stereo]

camera matrix
485.070038 0.000000 457.193899
0.000000 485.421510 365.293821
0.000000 0.000000 1.000000

distortion
-0.200465 0.069475 0.003302 0.000217 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

projection
427.078552 0.000000 461.243124 0.000000
0.000000 433.646851 369.922391 0.000000
0.000000 0.000000 1.000000 0.000000
```

Let's move /**tmp**/**calibrationdata.tar.gz** into 'ridecell/Results'

Let's now open the file and extract the "ost.yalm" and for clarity let's rename this file "calibrationdata1.yalm"

Calibration through a set of images

To perform a calibration using a set of images there are 2 steps:

- Extract and store images from the ros bag video that contain the chessboard in a variaty of locations
- Run the calibrator through the images in the same fashion we did run it directly over the bag file.

Extracting images from the video

One of the best ways to do this is to use `image_view` & right click to save screenshot on the desired spots. To do that we have to, in a similar fashion as before, run 2 terminals: one playing the ROS bag and the the other one running `image_view` node as follow:

$ rosrun image_view image_view image:=/sensors/camera/image_color

A great resource for this is:

https://coderwall.com/p/qewf6g/how-to-extract-images-from-a-rosbag-file-and-convert-them-to-video (https://coderwall.com/p/qewf6g/how-to-extract-images-from-a-rosbag-file-and-convert-them-to-video)

Once we capture at least 20 "good" images, we can proceed to the next step. I captured 30 images located on /cal_images

Run the calibrator

The script that will go through all 30 images and use them to obtain the camera calibration parameters is located in:

/ridecell/scripts

and it's called "calibrate_using_imgs.py"

```
import cv2
from camera_calibration.calibrator import MonoCalibrator, ChessboardInfo

numImages = 30

images = [ cv2.imread( 'cal_images/frame{:04d}.jpg'.format( i ) ) for i in range(
numImages ) ]

board = ChessboardInfo()
board.n_cols = 7
board.n_rows = 5
board.dim = 0.050

mc = MonoCalibrator( [ board ], cv2.CALIB_FIX_K3 )
mc.cal( images )
print( mc.as_message() )

mc.do_save()
```

On a terminal, navigate to **"/ridecell/scripts"** and execute it. Make sure the folder "cal_images" exists and contains 30 images.

```
$ python calibrate_using_imgs.py
```

You should get the following result

```
header:
  seq: 0
  stamp:
    secs: 0
    nsecs:         0
  frame_id: ''
height: 724
width: 964
```

Apply a distortion correction

Adding calibration information to bag files

To apply a distortion correction over the ROS bag, we can use "change_camera_info.py" included as part of the "bag_tools" http://wiki.ros.org/bag_tools (http://wiki.ros.org/bag_tools)

It turns out that installing "bag_tools" it's been souranded by issues since the package is broken as it lacks the executables that need to be compiled from c++. So you need to build and install it from source [https://github.com/srv/srv_tools/tree/kinetic/bag_tools (https://github.com/srv/srv_tools/tree/kinetic/bag_tools)], which worked out without problems.

By looking at the tutorial, we noticed that we are only interested in "change_camera.py". For this reason and for clarity and modularity, I decided to copy the latest "change_camera.py" on **"ridecell/scripts"**

For presentation purposes and since this script is short and simple to understand I decided to show you below the entire script:

In [ ]:
```
#!/usr/bin/python
"""
Copyright (c) 2012,
Systems, Robotics and Vision Group
University of the Balearican Islands
All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of Systems, Robotics and Vision Group, University of
      the Balearican Islands nor the names of its contributors may be used
to
      endorse or promote products derived from this software without specif
ic
      prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLI
ED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES
;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF TH
IS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
"""


PKG = 'bag_tools' # this package name

import roslib; roslib.load_manifest(PKG)
import rospy
import rosbag
import os
import sys
import argparse
import yaml
import sensor_msgs.msg

def change_camera_info(inbag,outbag,replacements):
  rospy.loginfo('      Processing input bagfile: %s', inbag)
  rospy.loginfo('     Writing to output bagfile: %s', outbag)
  # parse the replacements
  maps = {}
  for k, v in replacements.items():
    rospy.loginfo('Changing topic %s to contain following info (header will
not be changed):\n%s',k,v)

  outbag = rosbag.Bag(outbag,'w')
  for topic, msg, t in rosbag.Bag(inbag,'r').read_messages():
    if topic in replacements:
      new_msg = replacements[topic]
      new_msg.header = msg.header
      msg = new_msg
    outbag.write(topic, msg, t)
```

Now, on your terminal you can run the script with the following parameters:

Note: Make sure you go to /ridecell/scripts

Format: change_camera_info(inbag, outbag, calibrationdata)

```
$ python change_camera_info.py ../2016-11-22-14-32-13_test.bag ../2016-11-22-14-3
2-13_test.task1.bag /sensors/camera/camera_info=../Results/calibrationdata.yaml
```

In [ ]:
```
!python change_camera_info.py ../2016-11-22-14-32-13_test.bag ../2016-11-22
-14-32-13_test.task1.bag /sensors/camera/camera_info=../Results/calibration
data.yaml
```

Once it finish running the rectification over the entire ROS bag you should have an output like:

```
[INFO] [1529788406.779433]:       Processing input bagfile: ../2016-11-22-14-32-1
3_test.bag
[INFO] [1529788406.779660]:      Writing to output bagfile: ../2016-11-22-14-32-1
3_test.task1.bag
[INFO] [1529788406.780132]: Changing topic /sensors/camera/camera_info to contain
following info (header will not be changed):
header:
  seq: 0
  stamp:
    secs: 0
    nsecs:         0
  frame_id: ''
height: 724
width: 964
distortion_model: "plumb_bob"
D: [-0.196038, 0.0624, 0.002179, 0.000358, 0.0]
K: [485.763466, 0.0, 457.00902, 0.0, 485.242603, 369.066006, 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: [419.118439, 0.0, 460.511129, 0.0, 0.0, 432.627686, 372.659509, 0.0, 0.0, 0.0,
1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
[INFO] [1529788670.096924]: Closing output bagfile and exit...
```

and you should have a new ROS bag **"2016-11-22-14-32-13_test.task1.bag"**

To check if everything looks ok, you can opt for playing the new ROS bag.

Rectifiying the images

All the way to this point we have managed to find the "calibration" parameters from the video recorded (given to us as a ROS bag) and add/change these calibration parameters to the ROS bag. But we haven't rectified the images yet.

To do so, we need to continue looking at [http://wiki.ros.org/image_proc (http://wiki.ros.org/image_proc)] and especifically to `image_proc` nodelets

The main idea behind the following process is to:

- play the bag file with the "raw" images,
- rectify them and
- save the result in a seprate video.

The `.launch` file to do this will contain the following script:

```
<launch>
    <node name="rosbag" pkg="rosbag" type="play" args="../2016-11-22-14-32-13_tes
t.task1.bag"/>
    <node name="image_proc" pkg="image_proc" type="image_proc" respawn="false" ns
="/sensors/camera">
        <remap from="image_raw" to="image_color"/>
    </node>
    <node name="rect_video_recorder" pkg="image_view" type="video_recorder" respa
wn="false">
        <remap from="image" to="/sensors/camera/image_rect_color"/>
    </node>
</launch>
```

In general, processes launched with roslaunch have a working directory in $ROS_HOME (default ~/.ros) so we need to make sure to pass a **full path** to the bag file for it to be able to find the bag file.

By default, `video_recorder` creates `output.avi` in `/home/ros/.ros` and that will take care of our last bullent point above. After running this launch file, the resulting `output.avi` was moved to the `/results/videos` directory and rename it as `rectified.avi`.

The result after executing this command is:

```
roslaunch task1-cameracalibrator-recordvideo.launch
... logging to /home/robond/.ros/log/f210a1d4-7725-11e8-9fd4-000c294d9802/roslaun
ch-udacity-12906.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://root:36987/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.13

NODES
  /sensors/camera/
```

Task 2: Camera to LIDAR Offset Calculation

The Steps:

The process to solve this task is:

- To create a ROS package that holds all the scripts to run the trasformations (Translations and rotations).
- Use `scipy.optimize.minimize` function to find the optimal translation and rotation between the camera frame and LIDAR frame. This function will take a CostFunction representing both, the rotation and translation errors/difference and try to minimize these errors by chosing and optimal rotation angle and translation parameters.
- Create a composite OPTICAL-LIDAR image.

1. Creating a new ROS Package and the .launch files required

First we created a "ridecell_pkg" to hold scripts used to run the offeset calculation. To use them, first add `ridell_pkg` folder to `ROS_PACKAGE_PATH`

```
$ export ROS_PACKAGE_PATH=/media/robond/e2507505-dfde-40e2-9c5d-a7ecc505e0f0/ride
cell/src/ridecell_pkg:$ROS_PACKAGE_PATH
```

In this folder we will create 'ridecell_pkg/launch' folder to hold all the launch files.

The first launch file needed is to run 'lidar_camera_offset.py' which is the core of this task since it will calculate the minimum angle rotation and minimum translation required to "fit" both "images" that previously needed to be: 1) put on the same reference frame 2) converted from 3D into 2D.(A pin hole camera model was used to project the rotated 3D points into image coordinates)

```
$ roslaunch launch/task2-cameralidar-offset.launch
```

```
<launch>
    <node name="rosbag" pkg="rosbag" type="play" args="/media/robond/e2507505-dfd
e-40e2-9c5d-a7ecc505e0f0/ridecell/2016-11-22-14-32-13_test.task1.bag"/>
    <node name="lidar_camera_offset" pkg="ridecell_pkg" type="lidar_camera_offset
.py" args="/media/robond/e2507505-dfde-40e2-9c5d-a7ecc505e0f0/ridecell/data/lidar
_camera_calibration_data.json /media/robond/e2507505-dfde-40e2-9c5d-a7ecc505e0f0/
ridecell/cal_images/lidar_offset_frame.jpg /media/robond/e2507505-dfde-40e2-9c5d-
a7ecc505e0f0/ridecell/Results/Images/lidar_offset_output.jpg " output="screen">
        <remap from="camera" to="/sensors/camera/camera_info"/>
    </node>
</launch>
```

The python script `ridecell/src/ridecell_pkg/lidar_camera_offset.py` requires a `.json` file containing point correspondences between 3D Points and 2D image coordinates. The point correspondences used to generate the results below can be found in `data/lidar_camera_calibration_data.json`. Optional parameters can be included to generate an image using the expected and generated image coordinates for the provided 3D points.

```
{
    "points": [
        [ 1.568, 0.159, -0.082, 1.0 ], // top left corner of grid
        [ 1.733, 0.194, -0.403, 1.0 ], // bottom left corner of grid
        [ 1.595, -0.375, -0.378, 1.0 ], // bottom right corner of grid
        [ 1.542, -0.379, -0.083, 1.0 ], // top right corner of grid
        [ 1.729, -0.173, 0.152, 1.0 ], // middle of face
        [ 3.276, 0.876, -0.178, 1.0 ] // corner of static object
    ]
```

APPENDIX

Some work done before I got the ROS bag and other image correction studies done in the past.

In [ ]:
```
import os
import pickle
import cv2
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import glob
%matplotlib qt

from scipy.signal import find_peaks_cwt
%matplotlib inline

# Read in and made a list of the calibrartion images provided
images = glob.glob('../myGoProCalibration/GOPR0*.jpg')
#images = glob.glob('../camera_cal/calibration*.jpg')
NumCalibrationImages = len(images)

if NumCalibrationImages > 0 :
    print('Number of calibrarion images: ', NumCalibrationImages)
    print(' ******* For the sake of the exercise lets print them all  *****
** ')
    plt.figure(figsize=(30,200))
    for i in range(NumCalibrationImages):
        img = cv2.imread(images[i])
        plt.subplot(NumCalibrationImages,4,i+1)
        plt.xticks([])
        plt.yticks([])
        plt.imshow(img)
        plt.title(images[i])
        #plt.show()
        #plt.title("Chessboard image without the corners detected")
        #plt.show()
else:
    print('No calibration images were found!!!')
```

We will start finding and plotting the inside corners of a randomly chosen chessboard image using the OpenCV function cv2.findChessboardCorners() which will need to be fed by Gray scale images. Therefore we will convert the image to grayscale first using the appropriate conversion (from RGB -> GRAY or from BGR -> GRAY depending on the format that we read the image.

In [ ]:
```
# Set the number of inside corners
#_____________________________________________
nx = 8 # The number of inside corners in x direction
ny = 6 # The number of inside corners in y direction
#_____________________________________________

# For the sake of this test I'll load image 1 which is "pretty" to show onc
e the corners have been found
img = cv2.imread(images[3])

# Convert to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Find the chessboard corners
ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

# If corners were found, draw corners on the image.
if ret == True:
    print('Num corners found: ', len(corners))

    # Visualize Origianl before we draw the corners
    f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
    ax1.imshow(img)
    ax1.set_title('Original Image', fontsize=15)

    # Draw and display the corners
    cv2.drawChessboardCorners(img, (nx, ny), corners, ret)

    ax2.imshow(img)
    ax2.set_title('Chessboard image with corners', fontsize=15)

else:
    print('No corners were found!!!')
```

We will map the coordinates of the corners in the 2D image (image points) to the 3D coordinates of the real and undistorted chessboard corners (Object points). We will start setting up 2 empty arrays that will hold all these points, image points and object points for all our 20 images. The the real world, the object points of a chessboard are all equally separated and flat. For simplicity, we will assuming the chessboard is fixed on the (x, y) plane at z=0, such that the object points are the same for each calibration image. Therefore, we will create a template of these object points for one image/board and add it as the object points for all the images we read and process. The next step would be to do the same for all the calibration images so we can feed this values to the OpenCV calibration function.

In [ ]:
```
# Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space for each and every image we
will process. Since they are all images of the same chessboard, we will hav
e all exact same values
imgpoints = [] # 2d points in image plane.

# prepare the object points template for all images of the same chessboard:
(0,0,0), (1,0,0), (2,0,0) ....,(7,5,0)
objp = np.zeros((nx*ny,3), np.float32)
objp[:,:2] = np.mgrid[0:nx, 0:ny].T.reshape(-1,2)

# Step through the list and search for chessboard corners
for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (nx,ny), None)

    # If corners were found, add object points (ALWAYS THE SAME) and the im
age points
    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)
    else:
        print('No corners were found on image: ', fname)
```

Let's now use the output objpoints and imgpoints to compute the camera calibration and distortion coefficients using the cv2.calibrateCamera() function.

In [ ]:
```
# Perform the camera calibration given object points and image points
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, im
g.shape[0:2], None, None)
```

2. Image Distortion Correction

We will applied this distortion correction to the a test image using the cv2.undistort() function and show the result

In [ ]:
```
test_img = cv2.imread('../myGoProCalibration/test_image.jpg')
dst = cv2.undistort(test_img, mtx, dist, None, mtx)
cv2.imwrite('../myGoProCalibration/test_image_undist.jpg',dst)

# Save the camera calibration result for later use (we won't worry about rv
ecs / tvecs)
dist_pickle = {}
dist_pickle["mtx"] = mtx
dist_pickle["dist"] = dist
pickle.dump( dist_pickle, open( "camera_calibration.p", "wb" ) )

# Visualize undistortion
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.imshow(test_img)
ax1.set_title('Original Image', fontsize=15)
ax2.imshow(dst)
ax2.set_title('Undistorted Image', fontsize=15)
```

3. Color and Gradient Trasnformations

In the following scripts we will define the methods reviewed in the lectures (color transforms, gradients, etc.,) to create a thresholded binary image that will be used to enhance lane detection under different conditions.

We will present the effects of each method separately by tuning interactively the photos provided for this purpose with the objective of narrowing down the threshold ranges that produce the best outcome for this application.

After testing and tuning each trasformation we will cobine all of them over single images and observe the effects. Would the different transformations cobiened help each other to produce a better outcome or would they, somehow, interfer and counteract each other?

The trasformation methods that will be studied are:

- Gaussian Blurring to reduce noises
- Sobel Operator: This is the gradient (or conceptually the difference in grayscale intensity - value- between neigbour pixels:
  - Absolute value of the gradient on the x-direction or y-direction
  - Magnitude of the Gradient as a combination of the gradient in both directions
  - Direction of the Gradient as a combination of the gradient in both directions. (We are interested, mostly, in semi-vertical lines for lane detection)
- Binary Noise Reduction: We will explore OpenCV filter function "cv2.filter2D" to filter out color tones
- HLS Color Threshold: Using the HLS color space, we will explore the positive effect in lane detection of the S Channel.

In [ ]:
```
# Define a function to threshold (binary) a specific channel (you pass, for
example, the s channel = hls[:,:,2] and the theshold values)
def binary_thresh(img_ch, thresh=(0, 100)):
    binary_output = np.zeros_like(img_ch)
    binary_output[(img_ch > thresh[0]) & (img_ch <= thresh[1])] = 1
    # Return the binary image
    return binary_output

# Define a function to threshold (just therhold and not convert to binary)
a specific channel (you pass, for example, the s channel = hls[:,:,2] and t
he theshold values)
def color_thresh(img_ch, thresh=(0, 100)):
    binary_output = binary_thresh(img_ch, thresh)
    filtered_img = binary_output * img_ch
    # Return a color image
    return filtered_img

# Define a function that applies Gaussian smoothing bluring to and image (1
to 3 channles)
def gaussian_blur(img, kernel_size):
    """Applies a Gaussian Noise kernel"""
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

# Define a function that takes an image (alredy converted into grayscale -
to avoid not applying the right conversion -,
# gradient orientation (x or y), the sobel kernel (max 31, min 3, only odd
numbers) and threshold (min, max values).
def abs_sobel_thresh(gray, orient='x', sobel_kernel=3, thresh=(0, 255)):
    # Apply x or y gradient with the OpenCV Sobel() function
    # and take the absolute value
    if orient == 'x':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sob
el_kernel))
    if orient == 'y':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sob
el_kernel))
    # Rescale back to 8 bit integer
    scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))

    # Create the binaty filtered image
    binary_output = binary_thresh(scaled_sobel, thresh=thresh)

    # Return the result
    return binary_output

# Define a function to return the magnitude of the gradient
# for a given sobel kernel size and threshold values.
# as before, the img passed should be already in grayscale to avoid not app
lying the right conversion
# This is exactly the same as cv2.laplace but we can specify the kernel in
this case
def mag_thresh(gray, sobel_kernel=3, thresh=(0, 255)):
    # Take both Sobel x and y gradients
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
    # Calculate the gradient magnitude
    gradmag = np.sqrt(sobelx**2 + sobely**2)
    # Rescale to 8 bit
    scale_factor = np.max(gradmag)/255
    gradmag = (gradmag/scale_factor).astype(np.uint8)

    # Create the binaty filtered image
    binary_output = binary_thresh(gradmag, thresh=thresh)
```

Now that we have defined the main techniques described in the lectures is time to use the various aspects of the **gradient measurements (x, y, magnitude, and direction)** and also the **Color Transforms** to isolate lane-line pixels. We will research how we can combine thresholds of the x and y gradients, the overall gradient magnitude, and the gradient direction; as well as the HLS and color thresholds to focus on pixels that are likely to be part of the lane lines.

We will start with first with just the **gradient measurements (x, y, magnitude, direction)**

Sobel Threshold and Color Threshold Tuning

The entire Sobel set of funtions defined above are based on "gray" images since they calculate gradients. But what is really a grayscale image but some sort of averaging of the "color" channels we use! If we are using RGB format, each color pixel is described by a triple (R, G, B) of intensities for red, green, and blue, and the conversion to "gray" could use different ways of "averaging" these channels. The most common methods are:

- The lightness method averages the most prominent and least prominent colors: (max(R, G, B) + min(R, G, B)) / 2.
- The average method simply averages the values: (R + G + B) / 3.
- The luminosity method is a more sophisticated version of the average method. It also averages the values, but it forms a weighted average to account for human perception. We're more sensitive to green than other colors, so green is weighted most heavily. The formula for luminosity is 0.21 R + 0.72 G + 0.07 B.

But what if we stack all the **"relevant"** (I will address this later) color spaces channels, instead of just the R,G and B, and apply a simple average or some other mathematical averaging method to obtain our own grayscale image? OR what if we chose these **"relevant"** channels and apply sobel function individually to them instead of averaging them. All these questions will be addressed next.

What are the **"relevant"** channels for this application?

From the lectures we know that the R and G channels are the most useful channels on the RGB stack to detect white and yellow lines, although they might lack in performance under different light & brightness conditions. R and G values get lower under shadow and don't consistenly recognize the lane lines under extreme brightness. We also know that on the "HLS" color space, the H and the S channels stay fairly consistent in shadow or excesive brighness and we should be able to detect different lane lines (usually yellow and white) more reliably than in RGB color space. This section is meant to investigate the combination of all these color channels and possibly others like the YUV that when applying the Sobel suit of functions described above, could produce the best outcome in different conditions (different test images)

---> NOTE <---

While developing the ".py" files for the video generation, I came across this paper:

ROBUST AND REAL TIME DETECTION OF CURVY LANES (CURVES) WITH DESIRED SLOPES FOR DRIVING ASSISTANCE AND AUTONOMOUS VEHICLES by Amartansh Dubey and K. M. Bhurchandi

On this paper, the authors argue that one of the biggest hurdles for new autonomous vehicles is to detect curvy lanes, multiple lanes and lanes with a lot of discontinuity and noise. This paper presents very efficient and advanced algorithm for detecting curves having desired slopes (especially for detecting curvy lanes in real time) and detection of curves (lanes) with a lot of noise, discontinuity and disturbances. Overall aim is to develop robust method for this task which is applicable even in adverse conditions. They insist that even in some of most the famous and useful libraries like OpenCV and Matlab, there is no function available for detecting curves having desired slopes , shapes, discontinuities. Only few predefined shapes like circle, ellipse, etc, can be detected using presently available functions. They argue also that the proposed algorithm can not only detect curves with discontinuity, noise, desired slope but also it can perform shadow and illumination correction and detect/ differentiate between different curves.

How?, you may be wondering

In this algorithm, two very small Hough lines are taken on the curve, then weighted centroids of these Hough lines are calculated.

I would have loved to try to replicate their results here but I really don't have the time!! :-(

Interactive Tool

After some research we descovered a great suit of interactive tools that will allows as to try different configurations (active channels, ranges/thresholds, and sobel processing) in a fast and efficient way to try to extract yellow and white lane-lines under a multitude of light and road conditions.

In [ ]:
```
from ipywidgets import widgets, interactive, FloatSlider, IntSlider, IntRan
geSlider, FloatRangeSlider, RadioButtons, Select

def combineAll(image_idx, use_sobelXY, sobel_kernel, sobelX_thresh, sobelY_
thresh, use_MagDir_thresh, mag_thresh_range, dir_thresh_range, R,R_thresh,
G,B, H,L,S, S_thresh, Y,U,V, blur):
    # Assign the image from the already loaded images
    RGB_img = RGB_images[image_idx]
    HLS_img = cv2.cvtColor(RGB_img, cv2.COLOR_RGB2HLS)
    YUV_img = cv2.cvtColor(RGB_img, cv2.COLOR_RGB2YUV)
    YUV_img = 255 - YUV_img

    num_ch = sum([R,G,B,H,L,S,Y,U,V])
    if num_ch == 0:
        raise ValueError('You have to select at least one color channel')

    # This will be the image (width and height) with all the selected chann
els stacked in layers
    img_stacked = np.zeros((*RGB_img.shape[:-1], num_ch))
    ch_layer = 0 # <- at least one color channel. This is the first layer

    # Stacking RGB channels as selected
    if R:
        ch_filtered = color_thresh(RGB_img[:,:,0],R_thresh)
        img_stacked[:,:,ch_layer] = ch_filtered
        ch_layer += 1
    if G:
        img_stacked[:,:,ch_layer] = RGB_img[:,:,1]
        ch_layer += 1
    if B:
        img_stacked[:,:,ch_layer] = RGB_img[:,:,2]
        ch_layer += 1

    # Stacking HLS channels as selected
    if H:
        img_stacked[:,:,ch_layer] = HLS_img[:,:,0]
        ch_layer += 1
    if L:
        img_stacked[:,:,ch_layer] = HLS_img[:,:,1]
        ch_layer += 1
    if S:
        ch_filtered = color_thresh(HLS_img[:,:,2],S_thresh)
        img_stacked[:,:,ch_layer] = ch_filtered
        ch_layer += 1

    # Stacking YUV channels as selected
    if Y:
        img_stacked[:,:,ch_layer] = YUV_img[:,:,0]
        ch_layer += 1
    if U:
        img_stacked[:,:,ch_layer] = YUV_img[:,:,1]
        ch_layer += 1
    if V:
        img_stacked[:,:,ch_layer] = YUV_img[:,:,2]
        ch_layer += 1


    # Grayscale image needed for Sobel
    #gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # For simplicity let's take just the average of all the channel values.
the HLS values are normalized between 0 255 also
    gray = np.mean(img_stacked,2).astype(np.float32)/255
```

Let's test this!!

In [ ]:
```
# Read in and made a list of the calibrartion images provided
images_paths = glob.glob('../test_images/test*.jpg')
RGB_images = []
# Step through the list and search for chessboard corners
for fname in images_paths:
    RGB_images.append(mpimg.imread(fname))

print('We have loaded', len(RGB_images))
print('Image shape:',RGB_images[0].shape)

# Parameters to feed the interactive tool
#(image_idx, use_sobelXY, sobel_kernel, sobelX_thresh, sobelY_thresh, use_M
agDir_thresh, mag_thresh_range, dir_thresh_range,
# R,R_thresh, G,B, H,L,S, S_thresh, Y,U,V, blur):

interactive(combineAll,
            image_idx = IntSlider(min=1, max=len(images)-1, step=1, value=1
2),
            use_sobelXY = True,
            sobel_kernel=IntSlider(min=1, max=31, step=2, value=31),
            sobelX_thresh=IntRangeSlider(min=0, max=255, step=1,value=[5, 1
00]),
            sobelY_thresh=IntRangeSlider(min=0, max=255, step=1,value=[0, 2
55]),
            use_MagDir_thresh = True,
            mag_thresh_range=IntRangeSlider(min=0, max=255, step=1,value=[5
0, 200]),
            dir_thresh_range=FloatRangeSlider(min=0, max=np.pi / 2, step=0.
01,value=[0.65, 1.05]),
            R=False,
            R_thresh=IntRangeSlider(min=0, max=255, step=1,value=[200, 255]
),
            G=False, B=False, H=False, L=False,
            S=True,
            S_thresh=IntRangeSlider(min=0, max=255, step=1,value=[170, 255]
),
            Y=True, U=True, V=False,
            blur=IntSlider(min=1, max=37, step=2, value=1))
```

Color and Gradient Trasnformations Summary

After playing for a while we can conclude that:

- There is **NO one single combination** that works perfectly for all scenarios.
- Since we are very focused on detecting 2 main types of lane lines (yellow and white) under as many different conditions as we can, the two approaches that came to mind are:
  - Create specific functions to extract/detect yellow and white lines on images under as many possible conditions as possible. Something very similar was done on assignment 1: Lane detection, that I called colorFilter().
  - Create a tool that can pre-process the image and, some how, classifiy what set of thresholds (color and gradient) suit better for those conditions and apply them to hopefully extract the correct lines with a higher probability. As we saw above, shadows, brightness, even night images can have clearly different ranges that work best in each case.
- Another approach is not lookig at absolute values for ranges but x% of the channel values. For instance, in shadow areas a yellow line might be actually almos gray..but it can be considered lighter/highlight compared to the neighbours. We know the Red channel is good for detecting yellow and white and we also played with the S channel. We can research in this path further.

We will have to explore these possibilities deeper to decide what to use for the final video lane detection problem

In [ ]:
```
# Let's start with Option 1 (This is an exact copy from Assignment one)
'''
input:
   image: in any format (RGB, HLS, YUV, or 1 single channel)
   colorBounderies: Example in RGB.. a truple with the lower range and anot
her truple with the higher range
   colorBoundaries = [
       ([174, 131, 0], [255, 255, 255])
       ]
Output:
    filtered image bitwise masked, i.e. grayscale
'''
def colorFilter(image, colorBoundaries, blur=1):
    img = gaussian_blur(image, blur)
    # loop over the boundaries
    for (lower, upper) in colorBoundaries:
        # create NumPy arrays from the boundaries
        lower = np.array(lower, dtype = "uint8")
        upper = np.array(upper, dtype = "uint8")

        # find the colors within the specified boundaries and apply
        # the mask
        mask = cv2.inRange(image, lower, upper)
        #output = cv2.bitwise_and(image, image, mask = mask)

        return mask


def colorFilterInteractive(image_idx, img_format='RGB', Ch1=(174, 255), Ch2
=(131, 255), Ch3=(0, 255), blur=1):
        # Assign the image from the already loaded images
        Original = RGB_images[image_idx]
        img = Original

        # let's load the best channel bounderies for each image type we hav
e found so far
        if img_format == 'RGB':
            img = Original
            #Ch1=(174, 255)
            #Ch2=(131, 255)
            #Ch3=(0, 255)
        else:
            if img_format == 'HLS':
                img = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
                #Ch1=(0, 25)
                #Ch2=(100, 255)
                #Ch3=(150, 255)
            else:
                if img_format == 'YUV':
                    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
                    img = 255-img
                else:
                    raise ValueError('You have to select an Image format fr
om the list!')

        img = gaussian_blur(img, blur)

        lower = np.array((Ch1[0], Ch2[0], Ch3[0]), dtype = "uint8")
        upper =  np.array((Ch1[1], Ch2[1], Ch3[1]), dtype = "uint8")

        # find the colors within the specified boundaries and apply
        # the mask
        mask = cv2.inRange(img, lower, upper)
```

To test this function and find the right boundery values for yellow and white, let's use again the interactive tool to facilitate this task

In [ ]:
```
from ipywidgets import widgets, interactive, FloatSlider, IntSlider, IntRan
geSlider, FloatRangeSlider, RadioButtons, Select

# We should have the images already loaded from the cells above, but just t
o let us test this cell in isolation, let's
# reload the images

# Read in and made a list of the test images provided
images_paths = glob.glob('../test_images/test*.jpg')
RGB_images = []
# Step through the list and search for chessboard corners
for fname in images_paths:
    RGB_images.append(mpimg.imread(fname))

print('We have loaded', len(RGB_images))
print('Image shape:',RGB_images[0].shape)

# Parameters to feed the interactive tool
# (image_idx, img_format='RGB', Ch1=(174, 255), Ch2=(131, 255), Ch3=(0, 255
)):
interactive(colorFilterInteractive,
            image_idx = IntSlider(min=1, max=len(images)-1, step=1, value=1
),
            img_format = Select(
            options=['RGB', 'HLS', 'YUV'],
            value='HLS',
            description='Image Format:',
            disabled=False),
            Ch1=IntRangeSlider(min=0, max=255, step=1,value=[18, 40]),
            Ch2=IntRangeSlider(min=0, max=255, step=1,value=[45, 255]),
            Ch3=IntRangeSlider(min=0, max=255, step=1,value=[150, 255]),
            blur=IntSlider(min=1, max=37, step=2, value=1))
```

After some playing around we have found that HLS is definitely more robust and the thresholds found for consistent yellow lane detection are:

- Ch1 = H = (18, 40)
- Ch2 = L = (45, 255)
- Ch3 = S = (150, 255) and (45, 160)

For shadow in front of the car:

- Ch1 = H = (110, 140)
- Ch2 = L = (0, 70)
- Ch3 = S = (0, 30)

For the white lanes:

- Ch1 = H = (0, 40)
- Ch2 = L = (100, 255)
- Ch3 = S = (150, 255)

For Yellow lanes YUV worked fine on this range, but really bad in very dark shadowed areas.

- Ch1 = Y = (0, 255)
- Ch2 = U = (0, 255)
- Ch3 = V = (144, 255)

Too much time consumed!

Let's try to combine both again as we did above

In [ ]:
```
WhiteYellowColorBoundaries = [
       ([15, 45, 150], [40, 255, 255]),
       ([110, 0, 0], [140, 70, 255]),
       ([0, 100, 150], [40, 255, 255])
    ]
#WhiteYellowColorBoundaries = [([0, 45, 150], [40, 255, 255])]

# Read in and made a list of the test images provided
images_paths = glob.glob('../test_images/test*.jpg')
# Step through the list and search for chessboard corners
for fname in images_paths:
    Org = mpimg.imread(fname)
    img = cv2.cvtColor(Org, cv2.COLOR_RGB2HLS)
    WhiteYellow_Img = colorFilter(img, WhiteYellowColorBoundaries)
    # Visualize
    f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
    ax1.imshow(Org)
    ax1.set_title('Original Image', fontsize=15)

    ax2.imshow(WhiteYellow_Img, cmap='gray')
    ax2.set_title('Processed Image', fontsize=15)
```

These results are not bad, but they could definitely be better.

Let's now try the last bullet point mentioned above... and try to extract "highlights" in different channels by thresholding a certain percent of the values in that channnel.

In [ ]:
```
import numpy

def extract_highlights(img, per=99.9):
    """
    Generates an image mask selecting highlights.
    Input Parameters:
        img: image with pixels in range 0-255
        per: percentile for highlight selection. default=99.9

    :return: Highlight 255 not highlight 0
    """
    p = int(np.percentile(img, per) - 30)
    mask = cv2.inRange(img, p, 255)
    ##output = cv2.bitwise_and(img, img, mask = mask)
    return mask

def extract_highlightsInteractive(image_idx, Ch, Percent, NegativeImg=False
):
    rgb = RGB_images[image_idx]
    yuv = cv2.cvtColor(rgb, cv2.COLOR_RGB2YUV)
    yuv = 255 - yuv
    hls = cv2.cvtColor(rgb, cv2.COLOR_RGB2HLS)

    if Ch=='R':
        Img_Ch = rgb[:,:,0]
    if Ch=='G':
        Img_Ch = rgb[:,:,1]
    if Ch=='B':
        Img_Ch = rgb[:,:,2]

    if Ch=='Y':
        Img_Ch = yuv[:,:,0]
    if Ch=='U':
        Img_Ch = yuv[:,:,1]
    if Ch=='V':
        Img_Ch = yuv[:,:,2]

    if Ch=='H':
        Img_Ch = yuv[:,:,0]
    if Ch=='L':
        Img_Ch = yuv[:,:,1]
    if Ch=='S':
        Img_Ch = yuv[:,:,2]

    Highlights = extract_highlights(img=Img_Ch, per=Percent)

    if NegativeImg:
        Highlights = numpy.invert(Highlights)

    # Visualize
    f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
    ax1.imshow(rgb)
    ax1.set_title('Original Image', fontsize=15)

    ax2.imshow(Highlights, cmap='gray')
    ax2.set_title('Highlights Detected', fontsize=15)
```

Let's test the highlight extraction idea

In [ ]:
```
# Read in and made a list of the calibrartion images provided
images_paths = glob.glob('../test_images/test*.jpg')
RGB_images = []
# Step through the list and search for chessboard corners
for fname in images_paths:
    RGB_images.append(mpimg.imread(fname))

print('We have loaded', len(RGB_images))
print('Image shape:',RGB_images[0].shape)

# Parameters to feed the interactive tool
#(image_idx, use_sobelXY, sobel_kernel, sobelX_thresh, sobelY_thresh, use_M
agDir_thresh, mag_thresh_range, dir_thresh_range,
# R,R_thresh, G,B, H,L,S, S_thresh, Y,U,V, blur):

interactive(extract_highlightsInteractive,
            image_idx = IntSlider(min=1, max=len(images)-1, step=1, value=1
1),
            Ch = RadioButtons(
                    options=['R', 'G', 'B', 'H', 'L','S','Y','U','V'],
                    value='R',
                    description='Channel:',
                    disabled=False
            ),
            Percent=FloatSlider(min=0, max=100, step=0.01,value=99.0),
            NegativeImg = False,
            )
```

The Highlight extraction works remarkably well on the R channel at 99.0 in images with good light. The S channel prove also to highlght the yellow line on the shaded area on image 4 but it overwhelms the output on image 3, for instance, under the dark bridge. It's much better to output a dark/black image so we can use a different channel or different range of therehold than overwhelming the output (almost all white).

After looking at these results we conclude that the appropriate way to succesfully extract the lane lines is to combine different extraction/filter thresholds and join their specific "powers" by bitwise OR them at the end. Let's try that.

In [ ]:
```
YellowBoundary = [([20, 50, 150], [40, 255, 255])]
WhiteBoundary = [([175, 150, 200], [255, 255, 255])]

# Let's load the test images
images_paths = glob.glob('../test_images/test*.jpg')
# Step through the list and search for chessboard corners
for fname in images_paths:
    Org = mpimg.imread(fname)
    hls = cv2.cvtColor(Org, cv2.COLOR_RGB2HLS)
    White_Highlights = colorFilter(Org, WhiteBoundary)
    Yellow_Highlights = colorFilter(hls, YellowBoundary)
    Highlights = extract_highlights(img=Org[:,:,0], per=99.0)

    out = np.zeros(Org.shape[:-1], dtype=np.uint8)

    out[:, :][((White_Highlights==255) | (Yellow_Highlights==255) | (Highli
ghts==255))] = 1


    # Visualize
    f, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(1, 5, figsize=(20,10))
    ax1.imshow(Org)
    ax1.set_title('Original Image', fontsize=15)

    ax2.imshow(White_Highlights, cmap='gray')
    ax2.set_title('White Extracted Image', fontsize=15)

    ax3.imshow(Yellow_Highlights, cmap='gray')
    ax3.set_title('Yellow Extracted Image', fontsize=15)

    ax4.imshow(Highlights, cmap='gray')
    ax4.set_title('Highlights', fontsize=15)

    ax5.imshow(out, cmap='gray')
    ax5.set_title('Combinied Image', fontsize=15)
```

As you can imagine, we tested many possible combinations and finally decided to extract white-ish pixels using RGB, extract the yellow-ish pixels using HLS color space and finally use a "highlight extractor" filtering pixels above a certain percent on the Red Channel. The results are good enough for now. Also note that YUV (specifically Y and U channels) produce a very good output when using sobel fubcions. Therefore, the idea is to use RGB to extract white, HLS to extract yellow, Highligh extraction and R,S,Y,U to apply sobel.

4. Perspective Trasnformations: Bird's-Eye View

As soon as we think about perspective trasformation, the first challenge that comes to mind is how to chose the source and the destination points in a semi-automated way or use constant values so there is less human intervention in this process. The intuition for this whole process is:

- Camera Calibration
- Distortion Correction
- Perspective Trasnformation

Since the images used to calibrate the camera have not been taken with the same camera than the images that we are using for testing the road, we will use just Perspective Transformation on those, but we will show the whole process on the chessboard ones. On those ones, it's very easy to choose the Source points since we have a funcion that will give as the inner corners detected (as we saw above) and we can use the 4 most outer corners as our Source points. The destination points, as we saw on the lectures, will be arbitrarily choosen to be a nice fit for displaying our warped result.

Let's begin by defining the function we will use for the chessboard images

In [ ]:
```
# Define a function that takes an image, number of x and y inner corner poi
nts,
# camera matrix and distortion coefficients from above
def warpImg(img, nx, ny, mtx, dist):
    # Use the OpenCV undistort() function to remove distortion
    undist = cv2.undistort(img, mtx, dist, None, mtx)
    # Convert undistorted image to grayscale
    gray = cv2.cvtColor(undist, cv2.COLOR_BGR2GRAY)
    # Search for corners in the grayscaled image
    ret, corners = cv2.findChessboardCorners(gray, (nx, ny), None)

    if ret == True:
        # If we found corners, draw them! (just for fun)
        cv2.drawChessboardCorners(undist, (nx, ny), corners, ret)
        # Choose offset from image corners to plot detected corners
        # This should be chosen to present the result at the proper aspect
ratio
        # My choice of 100 pixels is not exact, but close enough for our pu
rpose here
        offset = 100 # offset for dst points
        # Grab the image shape
        img_size = (gray.shape[1], gray.shape[0])

        # For source points I'm grabbing the outer four detected corners
        src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-
nx]])
        # For destination points, I'm arbitrarily choosing some points to b
e
        # a nice fit for displaying our warped result
        # again, not exact, but close enough for our purposes
        dst = np.float32([[offset, offset], [img_size[0]-offset, offset],
                                     [img_size[0]-offset, img_size[1]-offse
t],
                                     [offset, img_size[1]-offset]])
        # Given src and dst points, calculate the perspective transform mat
rix
        M = cv2.getPerspectiveTransform(src, dst)
        # Warp the image using OpenCV warpPerspective()
        warped = cv2.warpPerspective(undist, M, img_size)

        # Return the resulting image and matrix
        return warped, M
    else:
        return img, 0
```

In [ ]:
```
# Read in the saved camera matrix and distortion coefficients
# These are the arrays you calculated using cv2.calibrateCamera()
dist_pickle = pickle.load( open( "camera_calibration.p", "rb" ) )
mtx = dist_pickle["mtx"]
dist = dist_pickle["dist"]

# Let's load again the chessboard images
# Read in and made a list of the calibrartion images provided
images_paths = glob.glob('../myGoProCalibration/GOPR0*.jpg')
#images = glob.glob('../camera_cal/calibration*.jpg')
NumCalibrationImages = len(images_paths)


for fname in images_paths:
    img = mpimg.imread(fname)

    top_down, perspective_M = warpImg(img, nx, ny, mtx, dist)
    if np.all(perspective_M) != 0:
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(24, 9))
        f.tight_layout()
        ax1.imshow(img)
        ax1.set_title('Original Image', fontsize=50)
        ax2.imshow(top_down)
        ax2.set_title('Undistorted and Warped Image', fontsize=50)
        plt.subplots_adjust(left=0., right=1, top=0.9, bottom=0.)
```

Perspective Transformations on Road images

Next, as we mentioned before, we will perform a perspective transformation on the road test images. We will not correct for distortions since we don't have checkboards images taken with the same camera.

In [ ]:
```
# Let's start defineing the Class that will hold the trasformations
class perspective:
    # Define the Properties and the Constructor
    def __init__(self, src, dst):
        self.src = src
        self.dst = dst
        self.M = cv2.getPerspectiveTransform(src, dst)
        self.M_inv = cv2.getPerspectiveTransform(dst, src)

    # Methods
    def warp(self, img):
        img_size = (img.shape[1], img.shape[0])
        return cv2.warpPerspective(img, self.M, img_size, flags=cv2.INTER_L
INEAR)

    def inv_warp(self, img):
        img_size = (img.shape[1], img.shape[0])
        return cv2.warpPerspective(img, self.M_inv, img_size, flags=cv2.INT
ER_LINEAR)
```

After defining this simple but powerful class, we will define a function that will use it on every image on the Test set to...test it. We will keep using the useful "interactive" tool again for visual convinience. One of the key elements on this trasformation is, obviously, the selection of the source and destination poins. This part, if keept simpel, could be very similar to "region of interest" in assignment 1. In our case, we can assume that the camera will be always located facing forward on the car (usually on the top of the car or on the rear-view mirror). So we can take 2 points from the bottom of the image at the same Y (height) to avoid the hood, and a few pixels from each side to cover a wide area. For the next two points we can select them, in the same fashion as before, at teh same height (Y) from the top - we will play with this number to avoid the sky - and we will select the X values to follow an inverted V shape that will fit our perspective.

In [ ]:
```
'''

Top Values:     X[0]    X[1]
    Y[0] >_________/_________\_________
                  /           \
                 /             \
                /               \
               /                 \
              /                   \
             /                     \
    Y[1] >__/_____________________\____
Bot Values X[0]                   X[1]

'''
def birdsEyeView(image_idx, offset, Ys, topXs, botXs):
    SRC = np.float32([
    (botXs[0], Ys[1]),
    (botXs[1], Ys[1]),
    (topXs[0], Ys[0]),
    (topXs[1], Ys[0])])

    DST = np.float32([
        (SRC[0][0] + offset, SRC[0][1]),
        (SRC[-1][0] - offset, SRC[0][1]),
        (SRC[0][0] + offset, 0),
        (SRC[-1][0] - offset, 0)])


    aPerpespective = perspective(SRC, DST)

    # Assign the image from the already loaded images on RBG_images
    Original = RGB_images[image_idx]

    # Let's take a look at hte birds-eye view over the original (RGB) image
    biersEyeView_Org_Img = aPerpespective.warp(Original)

    # let's apply the threshold for the color spaces
    YellowBoundary = [([20, 50, 150], [40, 255, 255])]
    WhiteBoundary = [([175, 150, 200], [255, 255, 255])]

    img = cv2.cvtColor(Original, cv2.COLOR_RGB2HLS)
    White_Img_Binary = colorFilter(Original, WhiteBoundary)
    Yellow_Img_Binary = colorFilter(img, YellowBoundary)
    output = cv2.bitwise_or(White_Img_Binary, Yellow_Img_Binary)

    biersEyeView_Thr_Img = aPerpespective.warp(output)


    # Visualize
    f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 9))
    f.tight_layout()
    #f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
    ax1.imshow(Original)
    ax1.set_title('Original Image', fontsize=15)

    ax2.imshow(biersEyeView_Org_Img)
    ax2.set_title('Birds-Eye (Org) Image', fontsize=15)

    ax3.imshow(biersEyeView_Thr_Img, cmap='gray')
    ax3.set_title('Birds-Eye (Thr) Image', fontsize=15)
```

In [ ]:
```
# Let's load again the test images
images_paths = glob.glob('../test_images/test*.jpg')
RGB_images = []
# Step through the list and search for chessboard corners
for fname in images_paths:
    RGB_images.append(mpimg.imread(fname))

print('We have loaded', len(RGB_images))
print('Image shape:',RGB_images[0].shape)

interactive(birdsEyeView,
            image_idx = IntSlider(min=1, max=len(RGB_images)-1, step=1, val
ue=5),
            offset=IntSlider(min=0, max=1280/2, step=1,value=175),
            Ys=IntRangeSlider(min=0, max=720, step=1,value=[450, 675]),
            topXs=IntRangeSlider(min=0, max=1280, step=1,value=[540, 740]),
            botXs=IntRangeSlider(min=0, max=1280, step=1,value=[132, 1147])
)
```

Let's now verify that the perspective transformation was working as expected by drawing the src and dst points onto a test image and its warped counterpart to verify that the lines appear parallel in the warped image. **First** we will take a look at what we did on Assignment 1 and we will see how good/bad it performs on the test images (curves)

In [ ]:
```
def weighted_img(img, initial_img, α=0.8, β=1., λ=0.):
    """
    `img` is the output of the hough_lines(), An image with lines drawn on
it.
    Should be a blank image (all black) with lines drawn on it.

    `initial_img` should be the image before any processing.

    The result image is computed as follows:

    initial_img * α + img * β + λ
    NOTE: initial_img and img must be the same shape!
    """
    return cv2.addWeighted(initial_img, α, img, β, λ)

def InterpolateLanes(lines, imgShape, order):

    # Arrays where we will store the points(X,Y) for each lane to be fitted
    x_LeftLane = []
    y_LeftLane = []

    x_RightLane = []
    y_RightLane = []

    for line in lines:
        for x1,y1,x2,y2 in line:
            # Since we can't use all the points to Interpolate/Extrapolate
            # We first put together all the points (x,y) that belong to eac
h lane looking at their slope
            if (x2-x1) != 0:
                slope = ((y2-y1)/(x2-x1))
                # left lane
                if slope < 0:
                    x_LeftLane.append(x1)
                    y_LeftLane.append(y1)

                    x_LeftLane.append(x2)
                    y_LeftLane.append(y2)
                else:
                    if slope > 0:
                        x_RightLane.append(x1)
                        y_RightLane.append(y1)

                        x_RightLane.append(x2)
                        y_RightLane.append(y2)


    # Interpolate the Left Lane
    # 1) calculate polynomial (Not necesarly has to be all the time a line)
    z_LeftLane = np.polyfit(x_LeftLane, y_LeftLane, order)
    f_LeftLane = np.poly1d(z_LeftLane)
    # Where does this lane start
    x_LeftLaneStart = min(x_LeftLane)
    # Where does this lane finish
    x_LeftLaneEnd = max(x_LeftLane)


    # Interpolate the Right Lane
    # 1) calculate polynomial (Not necesarly has to be all the time a line)
    z_RightLane = np.polyfit(x_RightLane, y_RightLane, order)
    f_RightLane = np.poly1d(z_RightLane)
    # Where does this lane start
    x_RightLaneStart = min(x_RightLane)
```

In [ ]:
```
# Let's load again the test images
images_paths = glob.glob('../test_images/test*.jpg')
RGB_images = []
# Step through the list and search for chessboard corners
for fname in images_paths:
    RGB_images.append(mpimg.imread(fname))

print('We have loaded', len(RGB_images))
print('Image shape:',RGB_images[0].shape)

#try:
interactive(laneDraw,
            image_idx = IntSlider(min=1, max=len(RGB_images)-1, step=1, val
ue=1))
#except:
#    pass # <- This will allow is to jump to another picture in case we had
an error
```

Above is what we did on our asignment 1 which, as you can see it's not really that good for curved roads.

Let's use the new technique tought in class.

Line Finding Method using peaks in a Histogram

In [ ]:
```
import numpy as np
image_idx = 5
img = RGB_images[image_idx]

#Get the Birds-eye View
BEV_Org_img, BEV_Thr_img = birdsEyeView(img, offset=136, Ys=(475,720), topX
s=(540,720), botXs=(132, 1147))


histogram = np.sum(BEV_Thr_img[BEV_Thr_img.shape[0]/2:,:], axis=0)
plt.plot(histogram)

# Visualize
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.imshow(img)
ax1.set_title('Original Image', fontsize=15)

ax2.imshow(BEV_Org_img)
ax2.set_title('Birds Eye View (Org) Image', fontsize=15)

ax3.imshow(BEV_Thr_img, cmap='gray')
ax3.set_title('Birds Eye View (Thr) Image', fontsize=15)
```

For code organization and readibility I decided to move to ".py" files.

The Code files submited in conjuction with this notebook are:

- **CameraCalibration.py**: Defines the camera class and all its methods to obtain:
  - 1) Camera Matrix used for perfective
  - 2) distortion coefficients
  - 3) rotation vectors
  - 4) Translation vectors
- **ImageProcessingUtils.py**: This file defines ALL the functions described on this notebook (sobel, color thereholding, use Histograms for line fitting, and others to support the lane detection
- **LaneDetector.py**: This file defines the LaneDetector Class and all the methods to support the Lane detetion discussed in class.
  - 1) IsLane: Checks if two lines are likely to form a lane by comparing the curvature and distance. Basically are they parallel and if so, is the distance between them a reasonable "Lane size"
  - 2) Draw lane overlay and curvature information, etc..
- **Line.py**: This file defines the Line Class that will try to "fit" a line found on the image. I investigated and tried several features to improve the performance of "fitting". Definetely this needs more work but is a good start.
- **PerspectiveTrasformer.py**: This file defines the Perspective Class (properties and methods) exactly as we did above.
- **VideoProcessing.py**:This file defines main running function to produce the video outputs we are required to present for this assignment. Goes through all the original videos and proccess them to overlay the detected lanes frame-by-frame

4. Describe how (and identify where in your code) you identified lane-line pixels and fit their positions with a polynomial?

The entire process on how we identified lane-line pixels and fit their positions starts on **"LaneFinder.py"**. Under the class definition "LaneFinder" you'll see a method called **"process_frame(self, frame)"** where everthing starts. We begin by making a copy of the original image and "undistort" the image. We follow by applying all the image processing techniques we learned (shown above) using **generate_lane_mask(frame, v_cutoff=400)**. After we "warp" the image and start the lane detection using **histogram_lane_detection(...)**. These 2 functions (and all other image processing ones) are defined on **ImageProcessingUtils.py**. After we have collected the coordinates for all the pixels that we extracted and we beleive they might belong to a lane line, we proceed by fiting them in a line and perform some checks to asses the likelyhood of beeing actualy part of the lane lines. This fitting and checking happens on **"LaneLine.py"**. in this file and under the class definition "LaneLine" you'll see a method called **"update(self, x, y)"**. This method tries to fit, check and compare lines from previous frames to increase the confidence in our "finder" results.

5. Describe how (and identify where in your code) you calculated the radius of curvature of the lane and the position of the vehicle with respect to center.

The radious of curvature is calculated usign the following equation (that can be easily derived) obtained from "http://mathworld.wolfram.com/RadiusofCurvature.html (http://mathworld.wolfram.com/RadiusofCurvature.html)"

Radious Of Curvature Equation

The function that performs this calculation is on **LaneLine.py** and it's called **calc_curvature(curve)** . This function gets fed with a collection of points that represent the center of the lane (take a look at line 225 on **LaneFinder.py**). We take these points and do the conversion/scaling from pixels to meters and we fit the curve to a typical second order equation: y(x) = Ax^2 + Bx + C. The derivative of this curve is y'(x) = 2Ax + B and the second derivative y''(x) = 2A. After deriving the coeficients A and B (using np.polyfit) we calculate the RoC.

As an addition, we tried to calculate the value for a confortable speed during a curve using a paper that identifies the threshold value of comfort for lateral accelerations ona vehicle as being 1.8 m/s2, with medium comfort and discomfort levels of 3.6 m/s2 and 5 m/s2, respectively "W. J. Cheng, Study on the evaluation method of highway alignment comfortableness [M.S. thesis], Hebei University of Technology, Tianjin, China, 2007." The process is very simple. The radial acceleration equation (also very easy to derive) is:

Radial Acceleration

So, having defined a therehold for a confortable radial acceleration and knowing the radious of curvature on the curve, we can easily proceed to calculate the desirible speed of the vehicle while taking the turn. Since the project video and the challenge video are mostly on a straight road, this feature has not too much value at this point. The function that performs this calculation is on **LaneLine.py** and it's called **calc_desiredSpeed(roc)**.

You can find how the drawing overlay over the original image and the "Adding" of the these information is performed at the end of the above mentioned **"process_frame(self, frame)"** on the LaneFinder class (LineFinder.py).

6. Provide an example image of your result plotted back down onto the road such that the lane area is identified clearly.

Project Video 1

Project Video 2

Challenge Video

Harder Challenge Video 1

Harder Challenge Video 2

Harder Challenge Video 3

Conclusion

Produced 3 videos:

- **project_video_MunirJojoVerge.mp4**: Good and acceptable results
- **challenge_video_MunirJojoVerge.mp4**: Good and acceptable results
- **harder_challenge_video_MunirJojoVerge.mp4**: Good try. Don't trust this algorithm!! :-)

Challenges:

All the challenges facing this assignment were discussed in each section, but here's the summary for those of you without the time to go through it.

- Camera Calibration: After testing udacity images, we did NOT find inner corners on calibration images 1, 4 and 5. We decided to try a different set of images for illustration purposes, although the distortion correction on the road images were performed with the calibration obtained from udacity images.
- Color and Gradient Transformations:
  - Sobel Operator: For the 3 different sobel functions (Absolute, Magnitude and Direction) we found challenging to determine what "gray-scale" would produce the best outcomes. We tried isolated channels from different color spaces and also averaging (as the usual gray-scale) the number of channels used. This created an incredibly wide range of choices and made it difficult to assess the quality of the outputs (how good or bad the sobel operator was doing in comparison with other combination of channels or other sobel operators) and very time consuming. The ipython "interactive" tool proved to be very useful for this task.
  - Color: After playing for a while we can conclude that:
    - There is NO one single combination that works perfectly for all scenarios. It seems that the right approach must be a dynamic change (almost like a feedback loop that adjust the color thresholding depending on light conditions and speed)
    - The research and testing over the 3 main color spaces (RGB, HLS and YUV) proved to be very challenging due to the fact that we really don't have a strict way to evaluate "how good" they perform when it comes to detect the lane lines. We should focus on standardizing this evaluation as well as using some sort of automatic/smart technique to explore all possible combinations of color spaces and thresholds to find the optimal one for this application. A CNN comes to mind with a large set of images where the labels might be the 2nd order coefficients of the lane lines (maybe??)
    - Exploring other techniques that don't rely that much on color thresholding is probably a good idea. While exploring this path I found a paper exactly for that purpose. The details of this paper (authors, title, etc..) were presented above.
- Perspective Transformations: On this topic, the main challenge was to decide the source and destination points. The reason for this challenge comes due to the fact that in the Hard Challenge video, the lane lines are not always where we would like them to be to be detected easily. Changing this source window proved to get much better results. From this improvement, we can conclude that a dynamic selection of this window, based probably on IMU data (speed and angular values and rates) could be used to improve dynamically the prediction.
- Besides the previous points, the rest of the assignment challenges can be all included in the "programmatic" pack. How to do "this" on python - type of issue.

(Note: I used "We" in most on this notebook, but I'm the only one working on this. Just to make it specifically clear)