project2

July 1, 2020

In [4]:
```
################Segment 1
import numpy as np
import cv2
import pickle
import glob
import matplotlib.pyplot as plt
%matplotlib qt



# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d points in real world space
imgpoints = [] # 2d points in image plane.

# Make a list of calibration images
images = glob.glob('./camera_cal/calibration*.jpg')

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

    # Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (9,6),None)

    # If found, add object points, image points
    if ret == True:
        print('working on', fname)
        objpoints.append(objp)
        imgpoints.append(corners)

        # Draw and display the corners
        cv2.drawChessboardCorners(img, (9,6), corners, ret)
```

```
            write_name = './camera_cal/corners_found' + str(idx+1) + '.jpg'
            #write_name = './test_imagess/binary' + str(idx+1) + '.jpg'
            cv2.imwrite(write_name, img)
    #load image for reference
    img = cv2.imread('./camera_cal/calibration1.jpg')
    img_size = (img.shape[0], img.shape[1])
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size ,  Non
    dist_pickle = {}
    dist_pickle["mtx"] = mtx
    dist_pickle["dist"] = dist
    pickle.dump(dist_pickle, open("./camera_cal/calibration_pickle.b" ,"wb" ))
```

```
working on ./camera_cal/calibration9.jpg
working on ./camera_cal/calibration15.jpg
working on ./camera_cal/calibration10.jpg
working on ./camera_cal/calibration12.jpg
working on ./camera_cal/calibration17.jpg
working on ./camera_cal/calibration7.jpg
working on ./camera_cal/calibration20.jpg
working on ./camera_cal/calibration19.jpg
working on ./camera_cal/calibration8.jpg
working on ./camera_cal/calibration11.jpg
working on ./camera_cal/calibration14.jpg
working on ./camera_cal/calibration13.jpg
working on ./camera_cal/calibration18.jpg
working on ./camera_cal/calibration16.jpg
working on ./camera_cal/calibration6.jpg
working on ./camera_cal/calibration3.jpg
working on ./camera_cal/calibration2.jpg
```

In [5]:
```
################Segment 2
import numpy as np
import cv2
import pickle
import glob
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
#from tracker import tracker



dist_pickle = pickle.load(open("./camera_cal/calibration_pickle.b" ,"rb" ))
mtx = dist_pickle["mtx"]
dist = dist_pickle["dist"]

###########################
```

```python
# Implement Sliding Windows and Fit a Polynomial



# Load our image
#binary_warped = mpimg.imread('warped_example.jpg')




def find_lane_pixels(binary_warped):
    # Take a histogram of the bottom half of the image
    histogram = np.sum(binary_warped[binary_warped.shape[0]//2:,:], axis=0)
    # Create an output image to draw on and visualize the result
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))
    # Find the peak of the left and right halves of the histogram
    # These will be the starting point for the left and right lines
    midpoint = np.int(histogram.shape[0]//2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # HYPERPARAMETERS
    # Choose the number of sliding windows
    nwindows = 9
    # Set the width of the windows +/- margin
    margin = 100
    # Set minimum number of pixels found to recenter window
    minpix = 50

    # Set height of windows - based on nwindows above and image shape
    window_height = np.int(binary_warped.shape[0]//nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])
    # Current positions to be updated later for each window in nwindows
    leftx_current = leftx_base
    rightx_current = rightx_base

    # Create empty lists to receive left and right lane pixel indices
    left_lane_inds = []
    right_lane_inds = []

    # Step through the windows one by one
    for window in range(nwindows):
        # Identify window boundaries in x and y (and right and left)
        win_y_low = binary_warped.shape[0] - (window+1)*window_height
        win_y_high = binary_warped.shape[0] - window*window_height
```

```python
        win_xleft_low = leftx_current - margin
        win_xleft_high = leftx_current + margin
        win_xright_low = rightx_current - margin
        win_xright_high = rightx_current + margin

        # Draw the windows on the visualization image
        cv2.rectangle(out_img,(win_xleft_low,win_y_low),
        (win_xleft_high,win_y_high),(0,255,0), 2)
        cv2.rectangle(out_img,(win_xright_low,win_y_low),
        (win_xright_high,win_y_high),(0,255,0), 2)

        # Identify the nonzero pixels in x and y within the window #
        good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
        (nonzerox >= win_xleft_low) &  (nonzerox < win_xleft_high)).nonzero()[0]
        good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
        (nonzerox >= win_xright_low) &  (nonzerox < win_xright_high)).nonzero()[0]

        # Append these indices to the lists
        left_lane_inds.append(good_left_inds)
        right_lane_inds.append(good_right_inds)

        # If you found > minpix pixels, recenter next window on their mean position
        if len(good_left_inds) > minpix:
            leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
        if len(good_right_inds) > minpix:
            rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

    # Concatenate the arrays of indices (previously was a list of lists of pixels)
    try:
        left_lane_inds = np.concatenate(left_lane_inds)
        right_lane_inds = np.concatenate(right_lane_inds)
    except ValueError:
        # Avoids an error if the above is not implemented fully
        pass

    # Extract left and right line pixel positions
    leftx = nonzerox[left_lane_inds]
    lefty = nonzeroy[left_lane_inds]
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]

    return leftx, lefty, rightx, righty, out_img


def fit_polynomial(binary_warped):
    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(binary_warped)
    # Fit a second order polynomial to each using `np.polyfit`
```

```
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0] )
    try:
        left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
        right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
    except TypeError:
        # Avoids an error if `left` and `right_fit` are still none or incorrect
        print('The function failed to fit a line!')
        left_fitx = 1*ploty**2 + 1*ploty
        right_fitx = 1*ploty**2 + 1*ploty
    ## Visualization ##
    # Colors in the left and right lane regions
    out_img[lefty, leftx] = [255, 0, 0]
    out_img[righty, rightx] = [0, 0, 255]

    # Plots the left and right polynomials on the lane lines
    #plt.plot(left_fitx, ploty, color='yellow')
    #plt.plot(right_fitx, ploty, color='yellow')

    return out_img, left_fitx, right_fitx, ploty, left_fit, right_fit
##########################################################################################
def fit_poly(img_shape, leftx, lefty, rightx, righty):
     ### TO-DO: Fit a second order polynomial to each with np.polyfit() ###
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)
    # Generate x and y values for plotting
    ploty = np.linspace(0, img_shape[0]-1, img_shape[0])
    ### TO-DO: Calc both polynomials using ploty, left_fit and right_fit ###
    left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
    right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]

    return left_fitx, right_fitx, ploty

def search_around_poly(binary_warped, left_fit, right_fit):
    # HYPERPARAMETER
    # Choose the width of the margin around the previous polynomial to search
    # The quiz grader expects 100 here, but feel free to tune on your own!
    margin = 100

    # Grab activated pixels
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])
    nonzerox = np.array(nonzero[1])

    ### TO-DO: Set the area of search based on activated x-values ###
    ### within the +/- margin of our polynomial function ###
```

```
    ### Hint: consider the window areas for the similarly named variables ###
    ### in the previous quiz, but change the windows to our new search area ###
    left_lane_inds = ((nonzerox > (left_fit[0]*(nonzeroy**2) + left_fit[1]*nonzeroy +
                    left_fit[2] - margin)) & (nonzerox < (left_fit[0]*(nonzeroy**2) +
                    left_fit[1]*nonzeroy + left_fit[2] + margin)))
    right_lane_inds = ((nonzerox > (right_fit[0]*(nonzeroy**2) + right_fit[1]*nonzeroy +
                    right_fit[2] - margin)) & (nonzerox < (right_fit[0]*(nonzeroy**2) +
                    right_fit[1]*nonzeroy + right_fit[2] + margin)))

    # Again, extract left and right line pixel positions
    leftx = nonzerox[left_lane_inds]
    lefty = nonzeroy[left_lane_inds]
    rightx = nonzerox[right_lane_inds]
    righty = nonzeroy[right_lane_inds]

    # Fit new polynomials
    left_fitx, right_fitx, ploty = fit_poly(binary_warped.shape, leftx, lefty, rightx, r

    ## Visualization ##
    # Create an image to draw on and an image to show the selection window
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))*255
    window_img = np.zeros_like(out_img)
    # Color in left and right line pixels
    out_img[nonzeroy[left_lane_inds], nonzerox[left_lane_inds]] = [255, 0, 0]
    out_img[nonzeroy[right_lane_inds], nonzerox[right_lane_inds]] = [0, 0, 255]
    #######################################plt.imshow(out_img)
    # Generate a polygon to illustrate the search window area
    # And recast the x and y points into usable format for cv2.fillPoly()
    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx-margin, ploty]))])
    left_line_window2 = np.array([np.flipud(np.transpose(np.vstack([left_fitx+margin,
                              ploty])))])
    left_line_pts = np.hstack((left_line_window1, left_line_window2))
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx-margin, ploty]))])
    right_line_window2 = np.array([np.flipud(np.transpose(np.vstack([right_fitx+margin,
                              ploty])))])
    right_line_pts = np.hstack((right_line_window1, right_line_window2))

    # Draw the lane onto the warped blank image
    cv2.fillPoly(window_img, np.int_([left_line_pts]), (0,255, 0))
    cv2.fillPoly(window_img, np.int_([right_line_pts]), (0,255, 0))
    result = cv2.addWeighted(out_img, 1, window_img, 0.3, 0)
    #plt.imshow(result)
    # Plot the polynomial lines onto the image
    ## End visualization steps ##

    return result

########################################################################################
```

```python
def color_threshold(image, sthresh=(0, 255), vthresh=(0, 255)):
    img = np.copy(image)

    hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
    s_channel = hls[:,:,2]
    s_binary = np.zeros_like(s_channel)
    s_binary[(s_channel >= sthresh[0]) & (s_channel <= sthresh[1])] = 1

    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    v_channel = hsv[:,:,2]
    v_binary = np.zeros_like(v_channel)
    v_binary[(v_channel >= vthresh[0]) & (v_channel <= vthresh[1])] = 1

    output = np.zeros_like(s_channel)
    output[(s_binary == 1) & (v_binary == 1)] = 1
    return output



def abs_sobel_thresh(img, orient='x', thresh_min=0, thresh_max=255):
    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # 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))
        plt.imshow(cv2.Sobel(gray, cv2.CV_64F, 1, 0))
    if orient == 'y':
        abs_sobel = np.absolute(cv2.Sobel(gray, cv2.CV_64F, 0, 1))
    # Rescale back to 8 bit integer
    scaled_sobel = np.uint8(255*abs_sobel/np.max(abs_sobel))
    # Create a copy and apply the threshold
    binary_output = np.zeros_like(scaled_sobel)
    # Here I'm using inclusive (>=, <=) thresholds, but exclusive is ok too
    binary_output[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

    # Return the result
    return binary_output

def mag_thresh(img, sobel_kernel=3, mag_thresh=(0, 255)):
    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # Take both Sobel x and y gradients
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
```

```python
    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 a binary image of ones where threshold is met, zeros otherwise
    binary_output = np.zeros_like(gradmag)
    binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1

    # Return the binary image
    return binary_output


def dir_threshold(img, sobel_kernel=3, thresh=(0, np.pi/2)):
    # Grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # Calculate the 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)
    # Take the absolute value of the gradient direction,
    # apply a threshold, and create a binary image result
    absgraddir = np.arctan2(np.absolute(sobely), np.absolute(sobelx))
    binary_output =  np.zeros_like(absgraddir)
    binary_output[(absgraddir >= thresh[0]) & (absgraddir <= thresh[1])] = 1

    # Return the binary image
    return binary_output


idx =0
images = glob.glob('./test_imagess/test*.jpg')
for idx , fname in enumerate(images):
    img = cv2.imread(fname)
    img = cv2.undistort(img, mtx, dist, None, mtx)
    undist = img
    write_name = './test_imagess/calibrated' + str(idx+1) + '.jpg'
    cv2.imwrite(write_name, img)
    processImage =np.zeros_like(img[:, :, 0])
    gradx = abs_sobel_thresh(img, orient='x', thresh_min=12, thresh_max=255)
    grady = abs_sobel_thresh(img, orient='y', thresh_min=25, thresh_max=255)
    c_binary = color_threshold(img, sthresh=(100, 255), vthresh=(50, 255))
    processImage[((gradx ==1) & (grady ==1) | (c_binary ==1))] = 255
    result1 = processImage
    write_name = './test_imagess/binary' + str(idx+1) + '.jpg'
    cv2.imwrite(write_name, result1)
    img_size = (img.shape[1], img.shape[0])
    src = np.float32([[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],[((img_size[0] / 6
```

```
[(img_size[0] * 5 / 6) + 60, img_size[1]],[(img_size[0] / 2 + 55), img_size[1] / 2 +
dst = np.float32([[(img_size[0] / 4), 0],[(img_size[0] / 4), img_size[1]],[(img_size
[(img_size[0] * 3 / 4), 0]])
M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)
warped1 = cv2.warpPerspective(img, M, img_size, flags = cv2.INTER_LINEAR)
result2 = warped1
write_name = './test_imagess/color_warped' + str(idx+1) + '.jpg'
cv2.imwrite(write_name, result2)
warped2 = cv2.warpPerspective(processImage, M, img_size, flags = cv2.INTER_LINEAR)
result3 = warped2
write_name = './test_imagess/binary_warped' + str(idx+1) + '.jpg'
cv2.imwrite(write_name, result3)
binary_warped = result3

out_img, left_fitx, right_fitx, ploty, left_fit, right_fit = fit_polynomial(binary_w

write_name = './test_imagess/binary_warped_marked' + str(idx+1) + '.jpg'
cv2.imwrite(write_name, out_img)




#############################################
ym_per_pix = 30/720 # meters per pixel in y dimension
xm_per_pix = 3.7/700 # meters per pixel in x dimension
# Fit a second order polynomial to each using `np.polyfit`
left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)   #left_fit_cr =
right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2) #right_fit_cr
#print('left_fit_cr is:', left_fit_cr)
#print('right_fit_cr is:', right_fit_cr)
y_eval = np.max(ploty)
# Calculation of R_curve (radius of curvature)
left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5
right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**
#print(left_curverad, 'm', right_curverad, 'm')

# Create an image to draw the lines on
warp_zero = np.zeros_like(binary_warped).astype(np.uint8)
color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
# Recast the x and y points into usable format for cv2.fillPoly()
pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
pts = np.hstack((pts_left, pts_right))
# Draw the lane onto the warped blank image
cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))
# Warp the blank back to original image space using inverse perspective matrix (Minv
########newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image.shap
```

```
        newwarp = cv2.warpPerspective(color_warp, Minv, (1280, 720))
        # Combine the result with the original image
        #undist = mpimg.imread('test_imagess/calibrated2.jpg')
        result4 = cv2.addWeighted(undist, 1, newwarp, 0.3, 0)
        ### positioning lines back into real world on alredy calibrated images:
        write_name = './test_imagess/calibrated_marked' + str(idx+1) + '.jpg'
        cv2.imwrite(write_name, result4)

        ######################################
        camera_center = (left_fitx[-1] + right_fitx[-1])/2
        center_diff = (camera_center-binary_warped.shape[1]/2)*xm_per_pix

        # font
        string1 = str(round(right_curverad+2.5))
        string2 = str(round(center_diff,2))
        font = cv2.FONT_HERSHEY_SIMPLEX

        # org
        org = (150, 100)
        org2 = (80, 200)

        # fontScale
        fontScale = 1

        # Blue color in BGR
        color = (0, 0, 255)

        # Line thickness of 2 px
        thickness = 2

        # Using cv2.putText() method
        result5 = cv2.putText(result4, 'RADIUS OF CURVATURE IS: '+ string1 + ' METERS', org,
        result6 = cv2.putText(result5, 'VEHICLE IS ' + string2 + ' METERS AWAY FROM CENTER L
        #################################################



        ###########################################
        write_name = './test_imagess/calibrated_marked_texted' + str(idx+1) + '.jpg'
        cv2.imwrite(write_name, result6)
        #######################################

        result11 = search_around_poly(binary_warped, left_fit, right_fit)
        write_name = './test_imagess/binary_warped_fitpoly' + str(idx+1) + '.jpg'
        cv2.imwrite(write_name, result11)
        ###########################################
```

```
In [ ]: # segment 3
```

```
import numpy as np
import cv2
import pickle
import glob
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

dist_pickle = pickle.load(open("./camera_cal/calibration_pickle.b" ,"rb" ))
mtx = dist_pickle["mtx"]
dist = dist_pickle["dist"]

def process_image(image):
    img = cv2.undistort(image, mtx, dist, None, mtx)
    undist = img   # Calibrate the image (Undistort the image)

    processImage =np.zeros_like(img[:, :, 0])
    gradx = abs_sobel_thresh(img, orient='x', thresh_min=12, thresh_max=255)
    grady = abs_sobel_thresh(img, orient='y', thresh_min=25, thresh_max=255)
    c_binary = color_threshold(img, sthresh=(100, 255), vthresh=(50, 255))
    processImage[((gradx ==1) & (grady ==1) | (c_binary ==1))] = 255
    result1 = processImage  # Make a binary image

    img_size = (img.shape[1], img.shape[0])
    src = np.float32([[(img_size[0] / 2) - 55, img_size[1] / 2 + 100],[((img_size[0] / 6
    [(img_size[0] * 5 / 6) + 60, img_size[1]],[(img_size[0] / 2 + 55), img_size[1] / 2 +
    dst = np.float32([[(img_size[0] / 4), 0],[(img_size[0] / 4), img_size[1]],[(img_size
    [(img_size[0] * 3 / 4), 0]])
    M = cv2.getPerspectiveTransform(src, dst)
    Minv = cv2.getPerspectiveTransform(dst, src)
    binary_warped = cv2.warpPerspective(processImage, M, img_size, flags = cv2.INTER_LIN
    result2 = binary_warped   # Make a binary warped image (binary eagle eye image)

    out_img, left_fitx, right_fitx, ploty, left_curverad, right_curverad = fit_polynomia

    warp_zero = np.zeros_like(binary_warped).astype(np.uint8)
    color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
    # Recast the x and y points into usable format for cv2.fillPoly()
    pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
    pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
    pts = np.hstack((pts_left, pts_right))
    # Draw the lane onto the warped blank image
    cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))
    # Warp the blank back to original image space using inverse perspective matrix (Minv
    ########newwarp = cv2.warpPerspective(color_warp, Minv, (image.shape[1], image.shap
    newwarp = cv2.warpPerspective(color_warp, Minv, (1280, 720))
    # Combine the result with the original image
    #undist = mpimg.imread('test_imagess/calibrated2.jpg')
    result = cv2.addWeighted(undist, 1, newwarp, 0.3, 0)
```

```
        ### positioning lines back into real world on alredy calibrated images:
        ##############################

        ym_per_pix = 30/720 # meters per pixel in y dimension
        xm_per_pix = 3.7/700 # meters per pixel in x dimension
        # Fit a second order polynomial to each using `np.polyfit`
        left_fit_cr = np.polyfit(ploty*ym_per_pix, left_fitx*xm_per_pix, 2)   #left_fit_cr =
        right_fit_cr = np.polyfit(ploty*ym_per_pix, right_fitx*xm_per_pix, 2) #right_fit_cr
        #print('left_fit_cr is:', left_fit_cr)
        #print('right_fit_cr is:', right_fit_cr)
        y_eval = np.max(ploty)
        # Calculation of R_curve (radius of curvature)
        left_curverad = ((1 + (2*left_fit_cr[0]*y_eval*ym_per_pix + left_fit_cr[1])**2)**1.5
        right_curverad = ((1 + (2*right_fit_cr[0]*y_eval*ym_per_pix + right_fit_cr[1])**2)**
        #print(left_curverad, 'm', right_curverad, 'm')

        camera_center = (left_fitx[-1] + right_fitx[-1])/2
        center_diff = (camera_center-binary_warped.shape[1]/2)*xm_per_pix

        # font

        string1 = str(round(left_curverad-2.5))
        string2 = str(round(center_diff,2))
        font = cv2.FONT_HERSHEY_SIMPLEX

        # org
        org = (150, 100)
        org2 = (80, 200)

        # fontScale
        fontScale = 1

        # Blue color in BGR
        color = (0, 0, 255)

        # Line thickness of 2 px
        thickness = 2

        # Using cv2.putText() method
        result5 = cv2.putText(result, 'RADIUS OF CURVATURE IS: '+ string1 + ' METERS', org,
        result6 = cv2.putText(result5, 'VEHICLE IS ' + string2 + ' METERS AWAY FROM CENTER L
        ##############################
        return result6
```

In [ ]:
```
#segment 4
# Import everything needed to edit/save/watch video clips

from moviepy.editor import VideoFileClip
```

```
from IPython.display import HTML

white_output = 'test_videos_output/project_video.mp4'
clip1 = VideoFileClip('test_videos/project_video.mp4')
white_clip = clip1.fl_image(process_image)
%time white_clip.write_videofile(white_output, audio=False)
```

```
In [ ]: HTML("""
<video width="960" height="540" controls>
  <source src="{0}">
</video>
""".format(white_output))
```