CS5670 Homework 1

- *Naveen Parthasarathy - nlp49@cornell.edu*
- *Yanbo Li - yl2556@cornell.edu*

Summary

- We explore the basics of image filtering and enhancement in Task 1
- In Task 2, we explore color quantization with k-means on RGB and LAB color spaces and compared the outputs to the original image
- In Task 3, we study edge detection using Sobel Filter, Canny Edge Detection and Gaussian-Laplace filtering
- We also plotted and analyzed precision-recall curves and F-measure to evaluate each of the above edge detectors

Task 1: Image filtering and enhancement

Question 1

In [43]:

```
x = np.array([[4, 1, 6, 1, 3],
              [3, 2, 7, 7, 2],
              [2, 5, 7, 3, 7],
              [1, 4, 7, 1, 3],
              [0, 1, 6, 4, 4]])
k = (1/9.0)*np.array([[1, 1, 1],
                      [1, 1, 1],
                      [1, 1, 1]])
print "Output:"
signal.convolve2d(x, k, boundary='symm', mode='same')
```

Output:

Out[43]:

```
array([[ 2.88888889,  3.77777778,  3.55555556,  4.        ,  2.77777
778],
       [ 2.88888889,  4.11111111,  4.33333333,  4.77777778,  3.88888
889],
       [ 2.55555556,  4.22222222,  4.77777778,  4.88888889,  3.88888
889],
       [ 1.77777778,  3.66666667,  4.22222222,  4.66666667,  4.
],
       [ 0.88888889,  2.88888889,  3.77777778,  4.33333333,  3.44444
444]])
```

Question 2

In [44]:

```
print "Output:"
scipy.ndimage.filters.median_filter(x, 3)
```

Output:

Out[44]:

```
array([[3, 4, 2, 3, 3],
       [3, 4, 5, 6, 3],
       [2, 4, 5, 7, 3],
       [1, 4, 4, 4, 4],
       [1, 1, 4, 4, 4]])
```

It is seen that the values in the mean filter output are all close to 4 in terms of value. While for the median filter output, there is still some variation in the values as it ranges from 4 to 7. Hence the median filter preserves the edges in the image better. Also, it is seen that all the numbers in the output are still whole numbers after convolving with the median filter.

Question 3

In [39]:

```
import math

sobx = np.array([[1, 0, -1],
                 [2, 0, -2],
                 [1, 0, -1]])
soby = -1*np.transpose(sobx)

df_dx = signal.convolve2d(x, sobx, boundary='symm', mode='same')
df_dy = signal.convolve2d(x, soby, boundary='symm', mode='same')
df_dx_center = df_dx[2][2]
df_dy_center = df_dy[2][2]
gradient_magnitude_ctr = np.sqrt(np.square(df_dx_center)+np.square(df_dy_center)
)
gradient_direction_ctr = math.degrees(math.atan(df_dy_center/df_dx_center*1.0))

print "Gradient magnitude at center:", gradient_magnitude_ctr
print "Gradient direction at center (in degrees):", gradient_direction_ctr
```

```
Gradient magnitude at center: 4.472135955
Gradient direction at center (in degrees): -63.4349488229
```

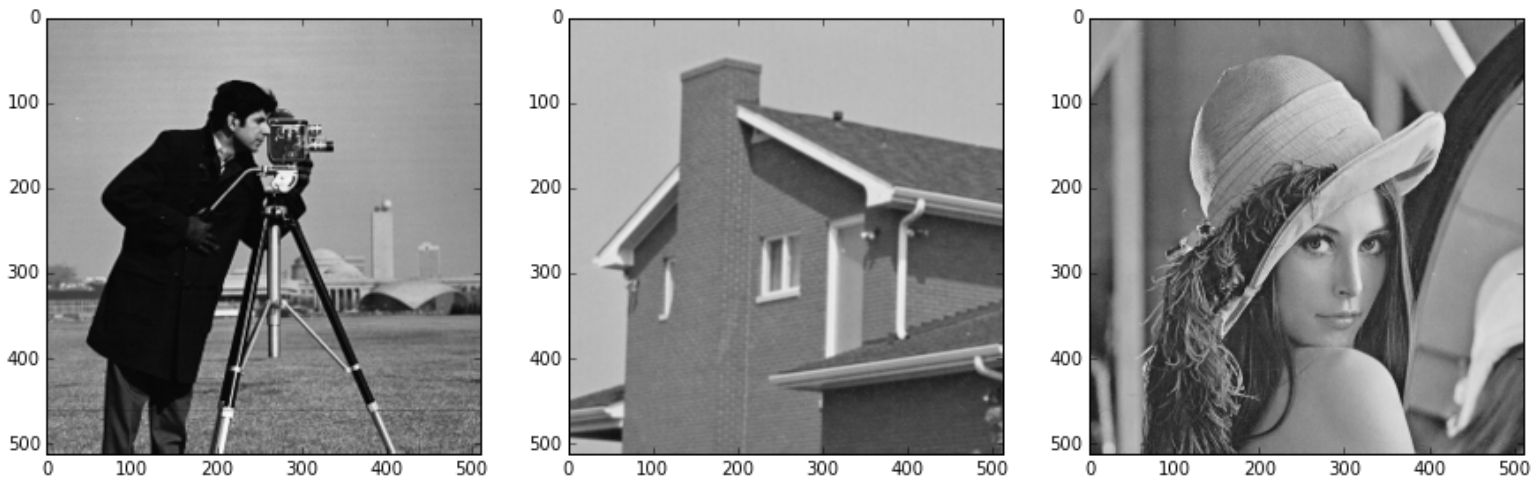
Question 4

First we display the 3 images after reading them into files

In [566]:

```
def display(img, colormap=cm.Greys):
    plt.imshow(img, cmap = colormap)
    plt.show()

def display3(img1, img2, img3, colormap=cm.Greys):
    f, axes = plt.subplots(ncols=3, figsize=(15,15))
    axes[0].imshow(img1, cmap = colormap)
    axes[1].imshow(img2, cmap = colormap)
    axes[2].imshow(img3, cmap = colormap)
    plt.show()

img1 = misc.imread("./Images/Q1/cameraman.jpg")
img2 = misc.imread("./Images/Q1/house.jpg")
img3 = misc.imread("./Images/Q1/lena.jpg")

print "We display the original images using the display method first"

display3(img1,img2,img3)
```

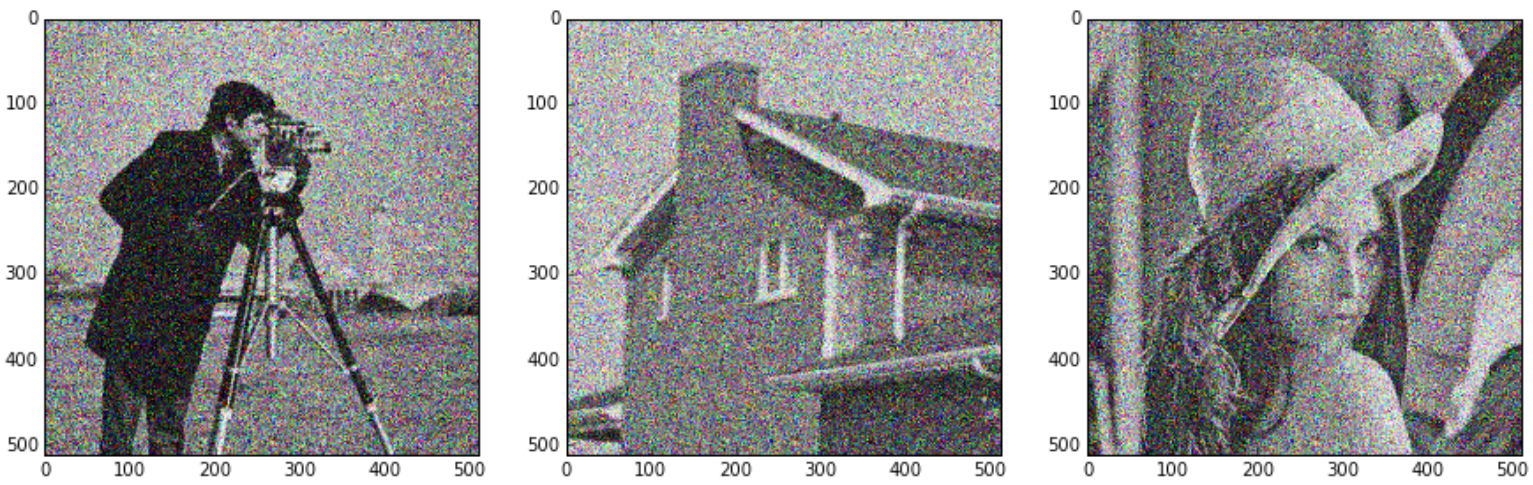
We display the original images using the display method first



In [342]:

```
from skimage import color
import skimage
# import random .

def add_noise(img,varn=0.1):
    return skimage.util.random_noise(img, mode='gaussian', seed=0, clip=True, va
r=varn)

img1_noise = add_noise(img1)
img2_noise = add_noise(img2)
img3_noise = add_noise(img3)

print "We display the noisy images"

display3(img1_noise,img2_noise,img3_noise)
```

We display the noisy images

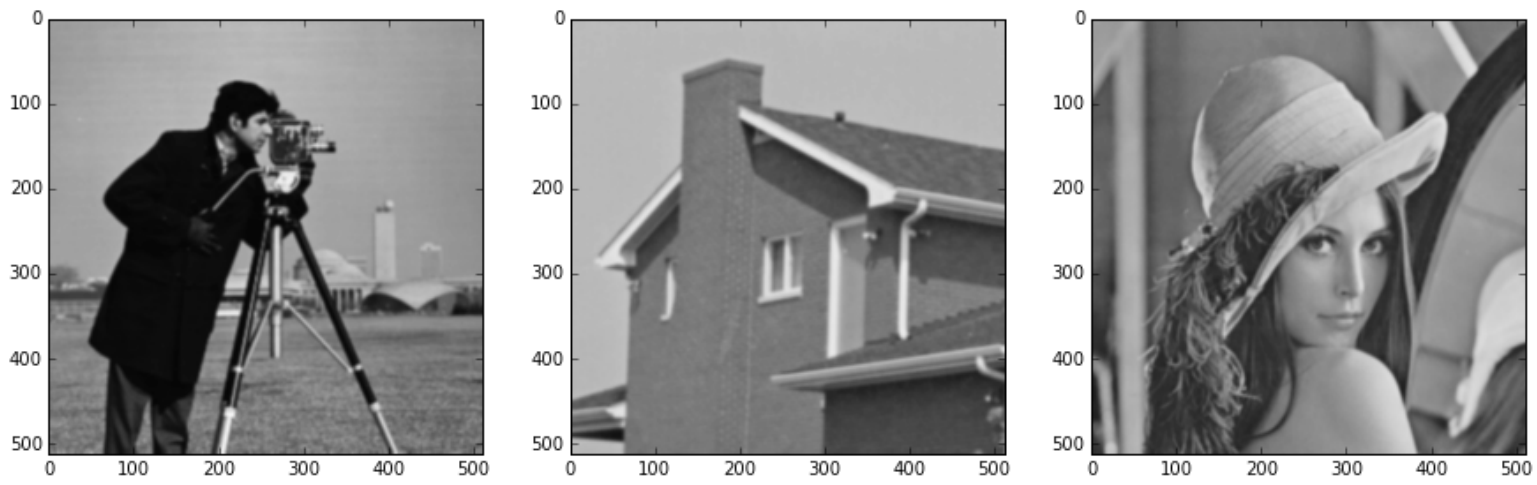


Part a: For a filter that only takes the pixel distance into account, we require a filter which has all values at the same distance from the center having the same value. An example of this is the gaussian filter. We use the gaussian filter for the given problem with sigma=1.5. For smaller values of sigma, too much noise is still present. While for larger values of sigma, there is too much blur.

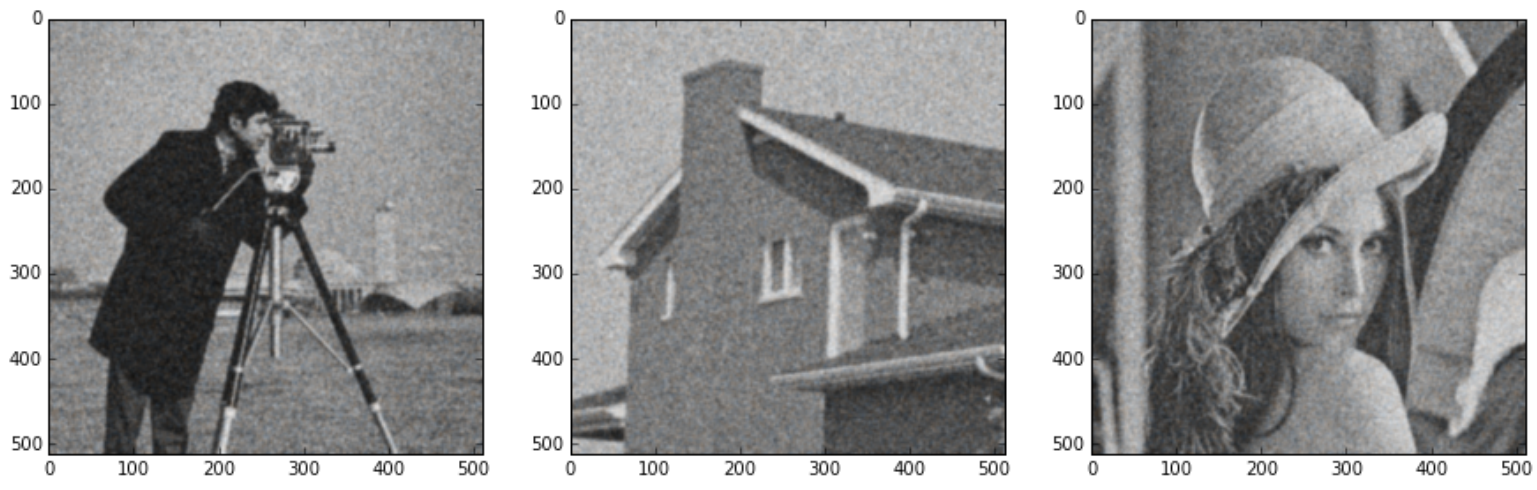
In [343]:

```
sigma = 1.5
img1_distfiltered = scipy.ndimage.filters.gaussian_filter(img1, sigma)
img2_distfiltered = scipy.ndimage.filters.gaussian_filter(img2, sigma)
img3_distfiltered = scipy.ndimage.filters.gaussian_filter(img3, sigma)
print "\nFilter non-noisy images with gaussian with sigma =",sigma
display3(img1_distfiltered,img2_distfiltered,img3_distfiltered)

img1_noisy_distfiltered = scipy.ndimage.filters.gaussian_filter(img1_noise, sigm
a)
img2_noisy_distfiltered = scipy.ndimage.filters.gaussian_filter(img2_noise, sigm
a)
img3_noisy_distfiltered = scipy.ndimage.filters.gaussian_filter(img3_noise, sigm
a)
print "\nFilter noisy images with Gaussian with sigma =",sigma
display3(img1_noisy_distfiltered,img2_noisy_distfiltered,img3_noisy_distfiltered
)
```

```
Filter non-noisy images with gaussian with sigma = 1.5
```




```
Filter noisy images with Gaussian with sigma = 1.5
```

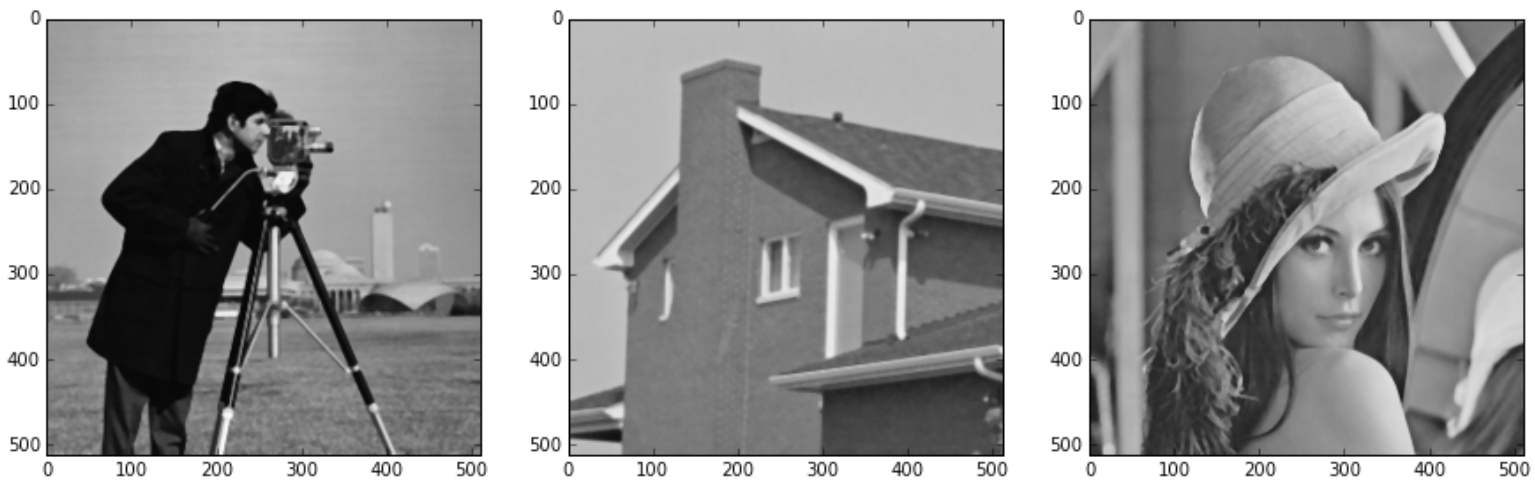


Part b: For a filter that only takes the pixel value into account, we can use the median filter. This looks at all neighbours and sorts them and selects the mean. Hence the position doesn't matter. For the given problem, we use a kernel size of 5x5. For smaller kernel sizes, there is too much noise. For higher kernel sizes, the blurring becomes too significant and the computation time increases signficantly

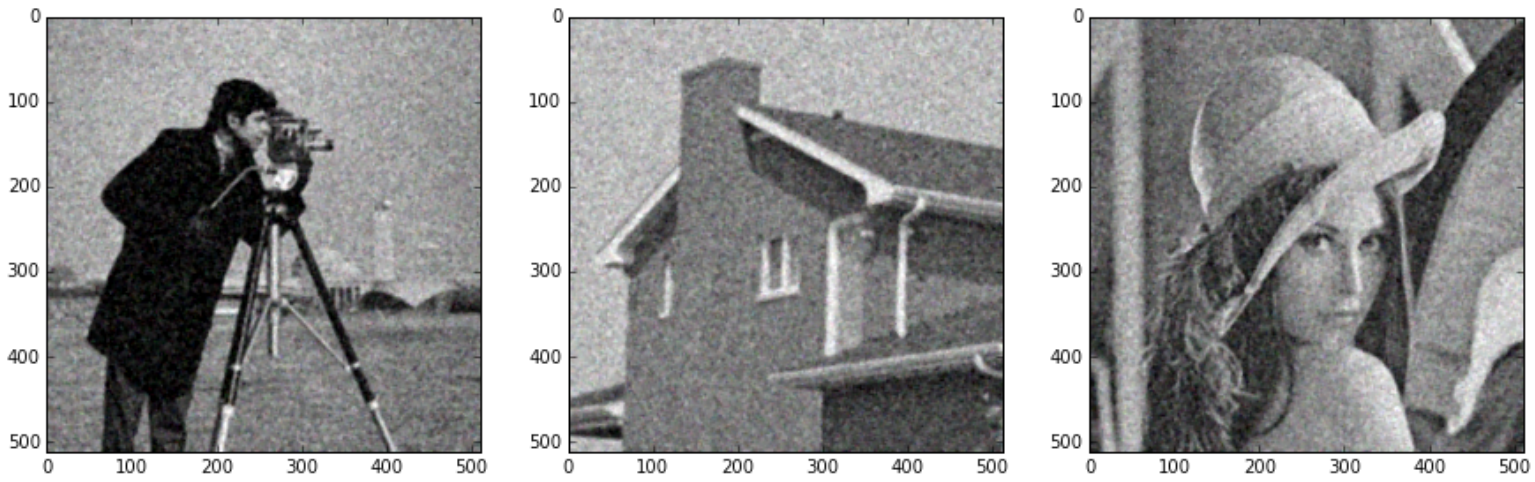
In [363]:

```
kernel_size = 5
img1_valfiltered = scipy.ndimage.filters.median_filter(img1, kernel_size)
img2_valfiltered = scipy.ndimage.filters.median_filter(img2, kernel_size)
img3_valfiltered = scipy.ndimage.filters.median_filter(img3, kernel_size)
print "\nFilter non-noisy images with median filter"
display3(img1_valfiltered,img2_valfiltered,img3_valfiltered)

img1_noisy_valfiltered = scipy.ndimage.filters.median_filter(img1_noise, kernel_
size)
img2_noisy_valfiltered = scipy.ndimage.filters.median_filter(img2_noise, kernel_
size)
img3_noisy_valfiltered = scipy.ndimage.filters.median_filter(img3_noise, kernel_
size)
print "\nFilter noisy images with median filter"
display3(img1_noisy_valfiltered,img2_noisy_valfiltered,img3_noisy_valfiltered)
```

```
Filter non-noisy images with median filter
```




```
Filter noisy images with median filter
```

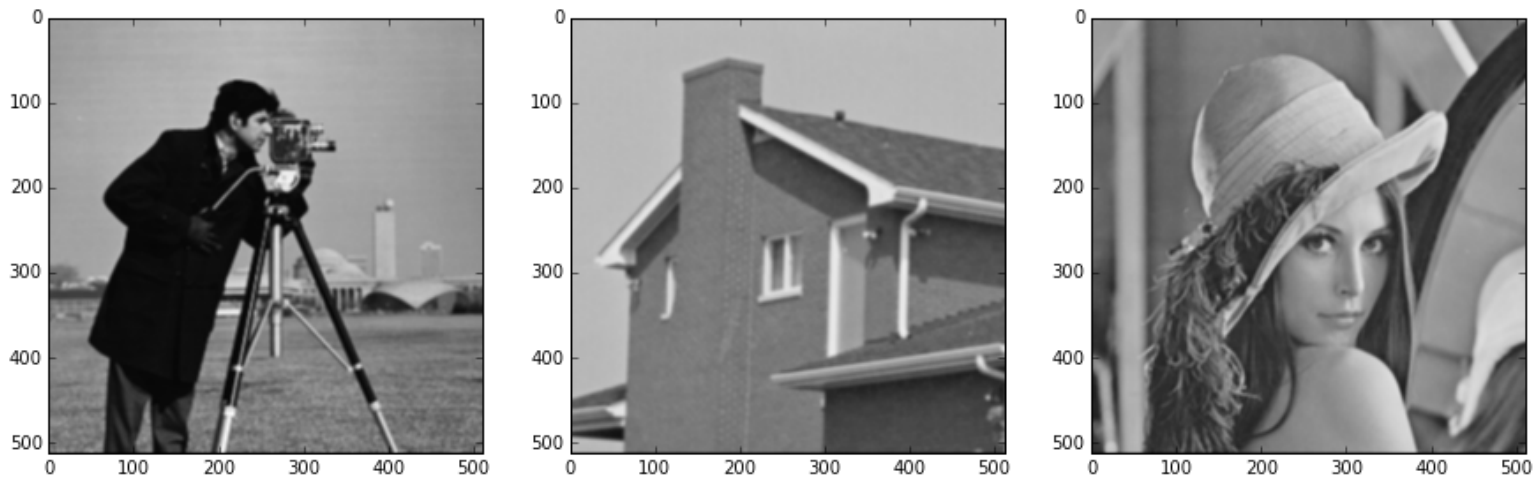


Part c: For a filter that takes into account both the pixel value and the pixel distance, we can use the bilateral filter, which is an edge-preserving and noise reducing filter. For the sigma range, we use a value of 5. For sigma spatial, we use a value of 2. Increasing sigma spatial beyond this value only makes the image more blurry without removing more noise. Lower sigma range does not filter out enough noise. While increasing sigma range produces diminishing improvements.

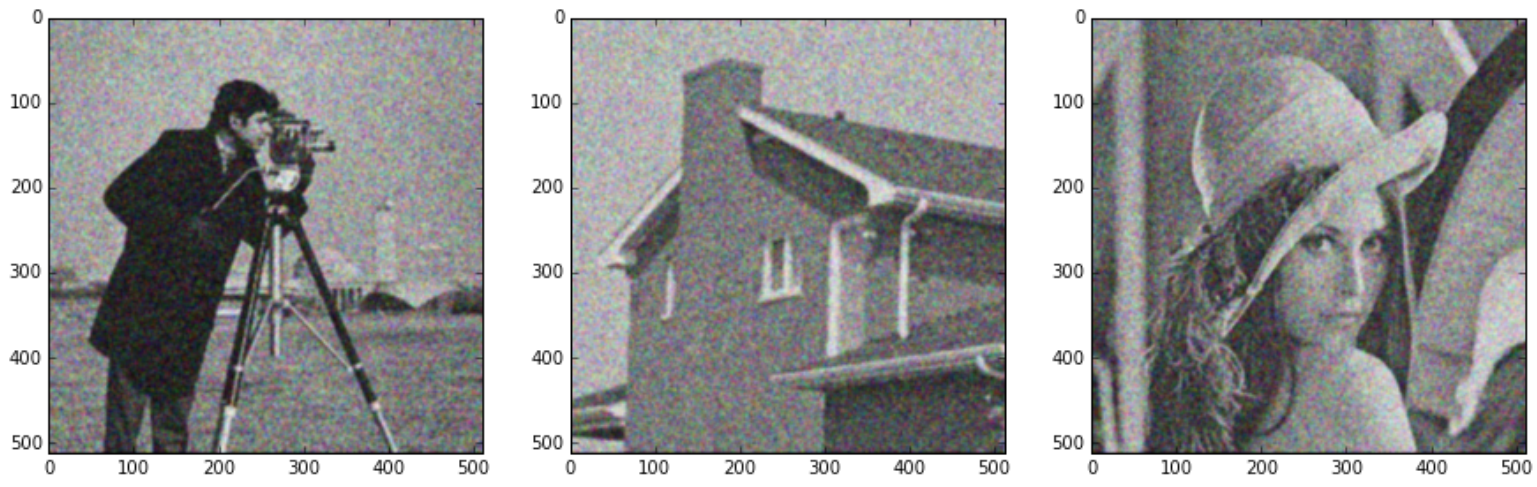
In [357]:

```
from skimage.restoration import denoise_bilateral

sigma_rng = 0.5
sigma_sptl = 5

img1_distvalfiltered = denoise_bilateral(img1, sigma_range=sigma_rng, sigma_spat
ial=sigma_sptl)
img2_distvalfiltered = denoise_bilateral(img2, sigma_range=sigma_rng, sigma_spat
ial=sigma_sptl)
img3_distvalfiltered = denoise_bilateral(img3, sigma_range=sigma_rng, sigma_spat
ial=sigma_sptl)
print "\nFilter non-noisy images with bilateral filter"
display3(img1_distvalfiltered,img2_distvalfiltered,img3_distvalfiltered)

img1_noisy_distvalfiltered = denoise_bilateral(img1_noise, sigma_range=sigma_rng
, sigma_spatial=sigma_sptl)
img2_noisy_distvalfiltered = denoise_bilateral(img2_noise, sigma_range=sigma_rng
, sigma_spatial=sigma_sptl)
img3_noisy_distvalfiltered = denoise_bilateral(img3_noise, sigma_range=sigma_rng
, sigma_spatial=sigma_sptl)
print "\nFilter noisy images with bilateral filter"
display3(img1_noisy_distvalfiltered,img2_noisy_distvalfiltered,img3_noisy_distva
lfiltered)
```

```
Filter non-noisy images with bilateral filter
```



```
Filter noisy images with bilateral filter
```



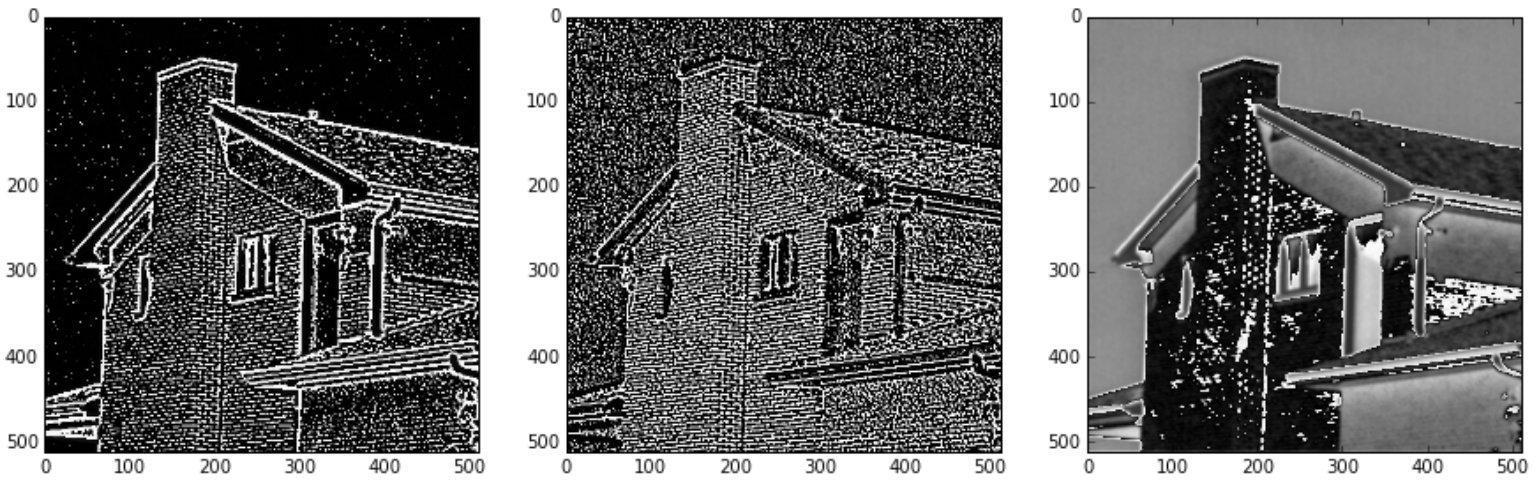
Based on the results above, it is seen that the gaussian filter is better at removing noise, but the edges are more blurred. This can be seen in the first figure below (difference of the original image and the filtered image) where the edges are more prominent in the difference, implying they were removed by the filter.

While for the median filter, it is less effective at reducing noise, but preserves the edges better. This can be seen from the second figure below where the edges are less prominent than for the gaussian filter, implying they were retained by the filter. But we can also observe more noise generated by the filter itself in the image.

In the case of the bilateral filter, there is a good compromise between noise reduction and edge preservation. Noise is removed from the uniform areas but there is some averaging. The edges are present, but not as thick as the gaussian filter.

In [364]:

```
display3((img2-img2_distfiltered),(img2-img2_valfiltered),(img2-img2_distvalfiltered))
```



Question 5

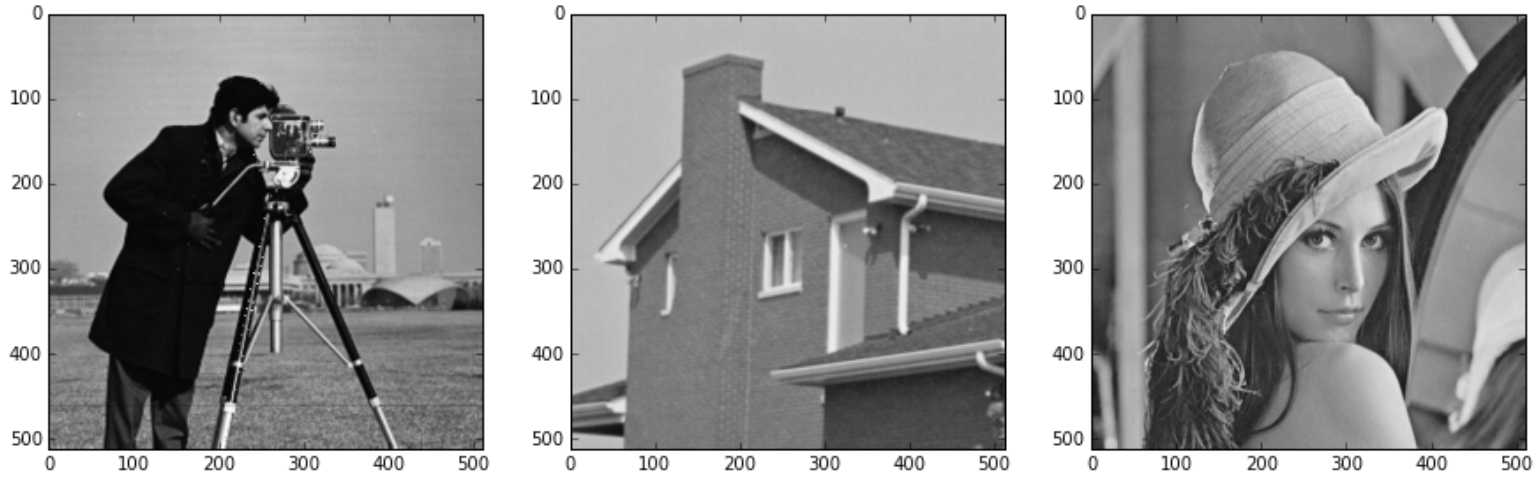
In [366]:

```
def blur(img, g_sigma, n=2):
    return ndimage.filters.gaussian_filter(img, g_sigma)

def f_unsharp(img, g_sigma, n=2):
    g_star_img = ndimage.filters.gaussian_filter(img, g_sigma)
    unsharp_img = n*img - g_star_img
    return unsharp_img
```

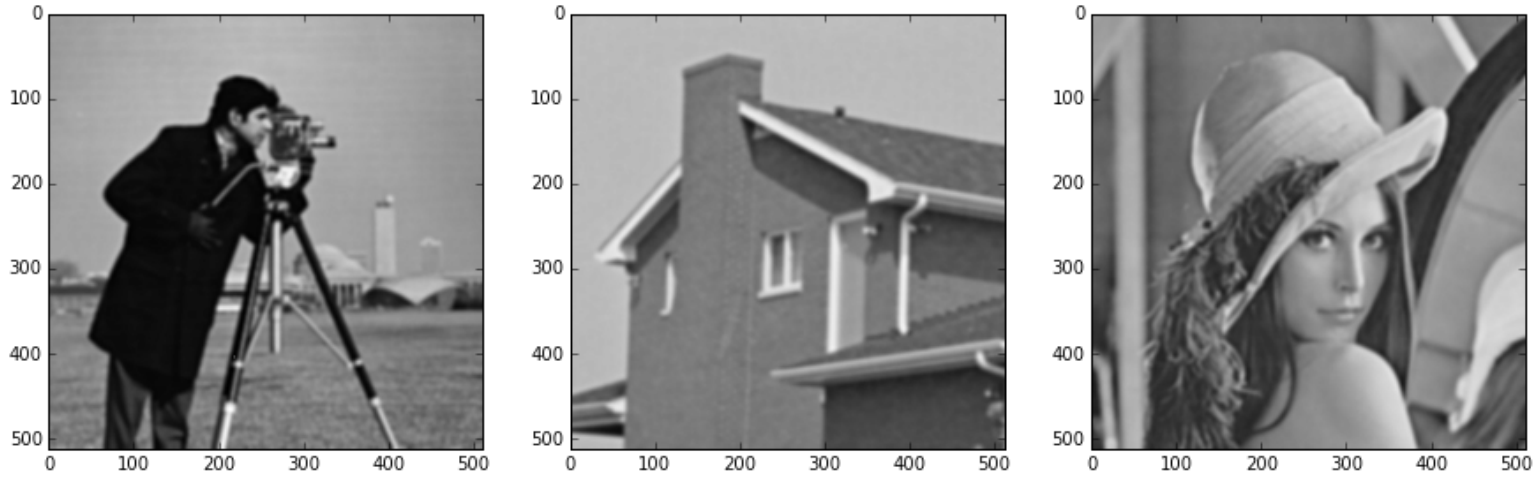
In [370]:

```
# first blur the images
img1_blur_0_75 = blur(img1, 0.75)
img1_blur_2_5 = blur(img1, 2.5)
img2_blur_0_75 = blur(img2, 0.75)
img2_blur_2_5 = blur(img2, 2.5)
img3_blur_0_75 = blur(img3, 0.75)
img3_blur_2_5 = blur(img3, 2.5)

print "Set A: f_unsharped images and blur images with sigma = 0.75"
display3(f_unsharp(img1_blur_0_75, 0.75),f_unsharp(img2_blur_0_75, 0.75),f_unsha
rp(img3_blur_0_75, 0.75))
print "\nSet B: f_unsharped images and blur images with sigma = 2.5"
display3(f_unsharp(img1_blur_2_5, 2.5),f_unsharp(img2_blur_2_5, 2.5),f_unsharp(i
mg3_blur_2_5, 2.5))
print "Set C: f_unsharped images with sigma = 2.5 and blur images with sigma = 0
.75"
display3(f_unsharp(img1_blur_0_75, 2.5),f_unsharp(img2_blur_0_75, 2.5),f_unsharp
(img3_blur_0_75, 2.5))
print "\nSet D: f_unsharped images with sigma = 0.75 and blur images with sigma
= 2.5"
display3(f_unsharp(img1_blur_2_5, 0.75),f_unsharp(img2_blur_2_5, 0.75),f_unsharp
(img3_blur_2_5, 0.75))
```

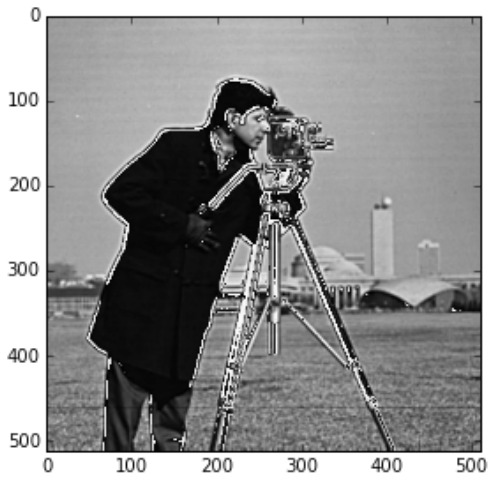
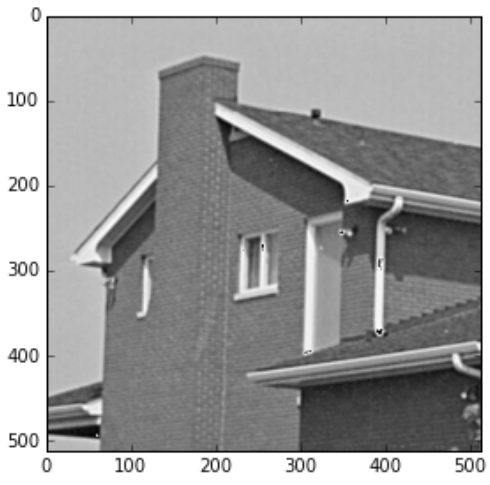
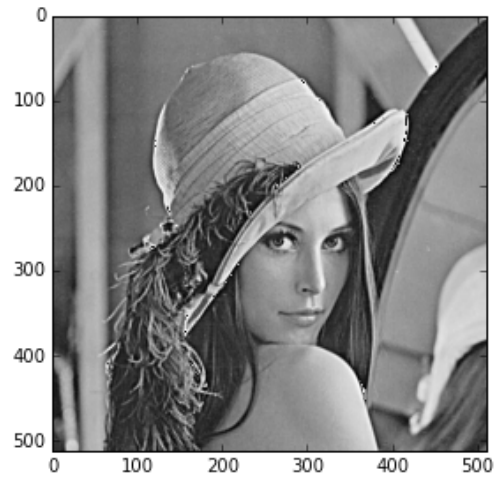
```
Set A: f_unsharped images and blur images with sigma = 0.75
```



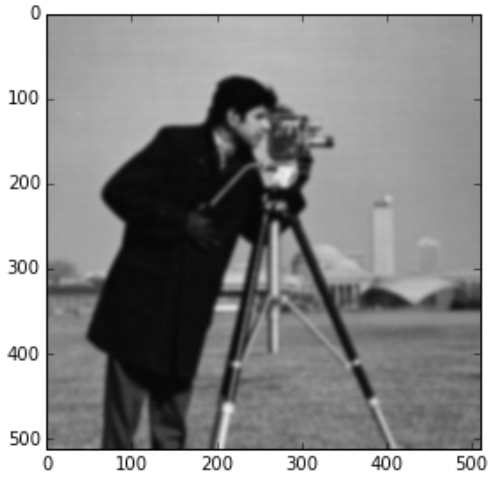
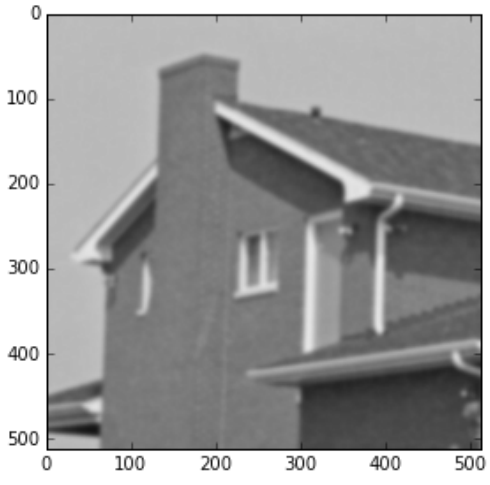
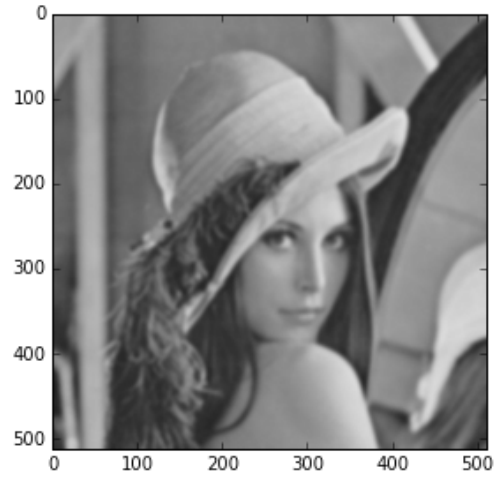
```
Set B: f_unsharped images and blur images with sigma = 2.5
```



```
Set C: f_unsharped images with sigma = 2.5 and blur images with sigm
a = 0.75
```

Set D: f_unsharped images with sigma = 0.75 and blur images with sig
ma = 2.5

In Set A above (sigma = 0.75 for both blur and unsharp mask), we notice that the images are sharpened and clear.

In Set B and Set D where the sigma for blur is 2.5, the blur effect dominates and the unsharp masked image is still blur.

In Set C on the other hand, where the blur is with sigma = 0.75 and the unsharp mask is with sigma = 2.5, we notice a halo effect around the edges.

To explain what happens, we can rewrite the f_unsharp method as:

```
f_unsharp(x,y) = 2f(x,y) - g(x,y)*f(x,y)

= f(x,y) + (f(x,y) - g(x,y)*f(x,y))
```

So, based on the above equation, we are basically adding to the image the term within the bracket.

`g(x,y)*f(x,y)` produces a blurred image with the high frequencies filtered out.

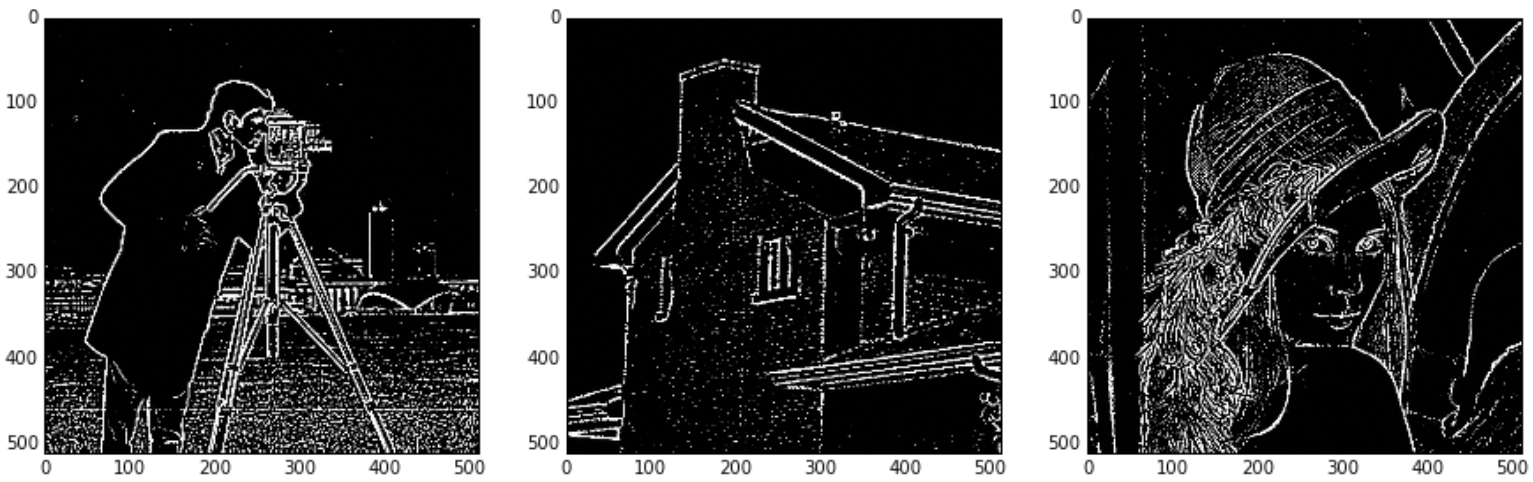
`(f(x,y) - g(x,y)*f(x,y))` then removes all the low frequency components from `f(x,y)` leaving only the area around the edges highlighted. The thickness of the edge depends on the sigma used. The higher the sigma, the higher the edge thickness. This is demonstrated by the images below.

For, thin edges, when it is added back to original image, it produces a sharpening effecting as seen in set A above. For thick edges, when it is added back to the original image, it can produce the halo effect seen in Set C.

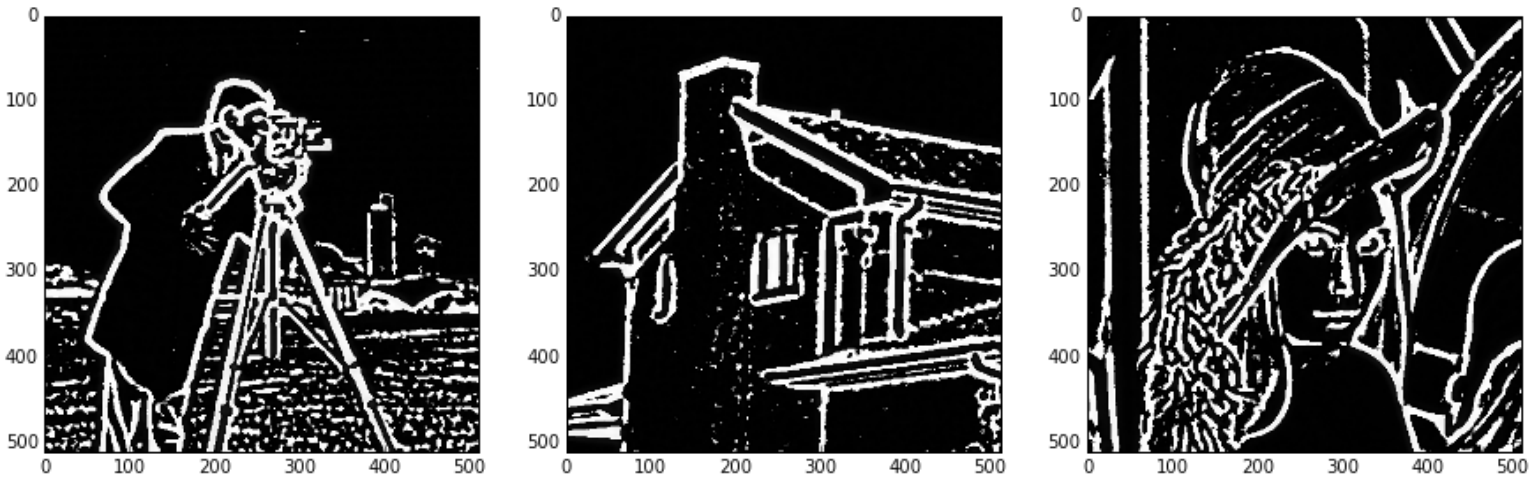
In [375]:

```
print "Edges when sigma = 0.75"
display3(f_unsharp(img1_blur_0_75, 0.75, 1),f_unsharp(img2_blur_0_75, 0.75, 1),f
_unsharp(img3_blur_0_75, 0.75, 1))
print "\nEdges when sigma = 2.5"
display3(f_unsharp(img1_blur_2_5, 2.5, 1),f_unsharp(img2_blur_2_5, 2.5, 1),f_uns
harp(img3_blur_2_5, 2.5, 1))
```

```
Edges when sigma = 0.75
```



```
Edges when sigma = 2.5
```

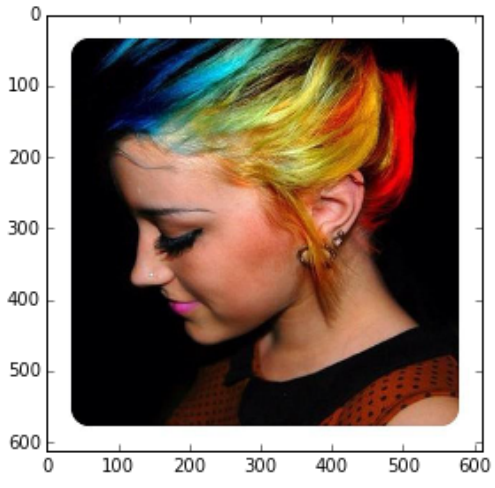


Task 2: Color quantization with k-means

We first read in the images

In [405]:

```
cimg1 = np.array(misc.imread("./Images/Q2/colorful1.jpg"))/255.0
cimg2 = np.array(misc.imread("./Images/Q2/colorful2.jpg"))/255.0
cimg3 = np.array(misc.imread("./Images/Q2/colorful3.jpg"))/255.0
display3(cimg1, cimg2, cimg3)
```

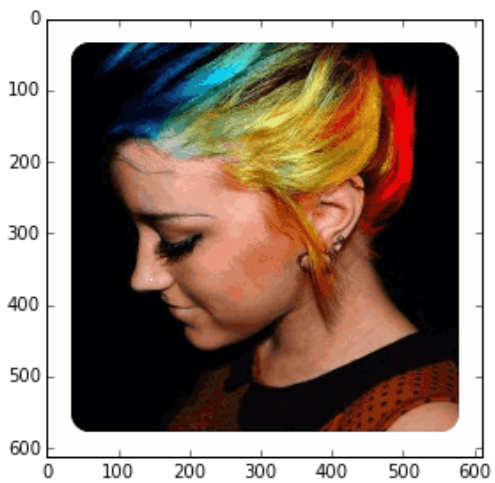


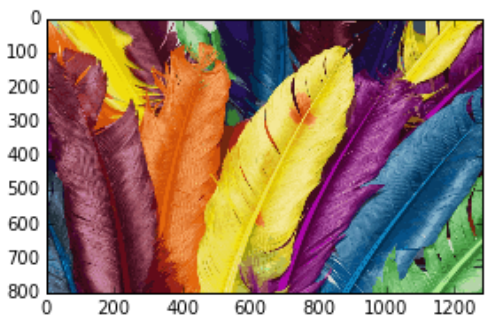
Question 1

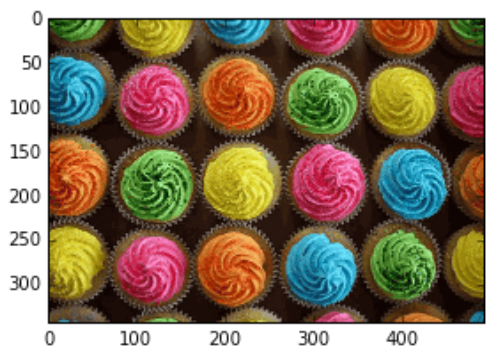
In [519]:

```
# gets as input an RGB image, quantizes the colors in the image via k-means,
# and then replaces the color at each pixel with the color of its quantized vers
ion
# k is the number of clusters
def quantize_kmeans(img, k):
    # Convert from 8 bits integer encoding to floats and divide by 255
    img = np.array(img, dtype=np.float64)
    w, h = img.shape[0], img.shape[1]
    img_flat = np.reshape(img, (w*h, 3))
    img_flat_sample = shuffle(img_flat, random_state=0)[:5000]
    kmeans = KMeans(n_clusters=k, random_state=0).fit(img_flat_sample)
    labels = kmeans.predict(img_flat)
    quantized_rgbs = kmeans.cluster_centers_
    img_quantized = np.zeros(img.shape)
    for i in range(w):
        for j in range(h):
            img_quantized[i][j] = quantized_rgbs[labels[i*h+j]]
    return img_quantized


cimg1_rgbq = quantize_kmeans(cimg1, 64)
cimg2_rgbq = quantize_kmeans(cimg2, 64)
cimg3_rgbq = quantize_kmeans(cimg3, 64)

print "Quantized RGB images"
display3(cimg1_rgbq, cimg2_rgbq, cimg3_rgbq)
```

```
Quantized RGB images
```





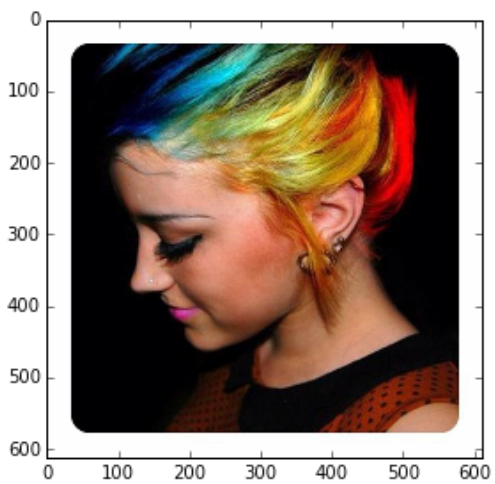


Question 2

In [520]:

```
def quantize_lab(rgb, k, ret_lab_img = False):
    w, h = rgb.shape[0], rgb.shape[1]
    lab = color.rgb2lab(rgb)
    lab_flat = np.reshape(lab[:,:,0], (w*h))
    lab_flat_sample = shuffle(lab_flat, random_state=0)[:5000]
    kmeans = KMeans(n_clusters=k, random_state=0).fit(lab_flat_sample.reshape(-1
, 1))
    labels = kmeans.predict(lab_flat.reshape(-1, 1))
    lab_cluster_centers = kmeans.cluster_centers_

    lab_quantized = np.array(lab, copy=True)

    for i in range(w):
        for j in range(h):
            lab_quantized[i][j][0] = lab_cluster_centers[labels[i*h+j]]

    # return lab image
    if ret_lab_img:
        return lab_quantized

    rgb_quantized = color.lab2rgb(lab_quantized)
    return rgb_quantized

cimg1_labq = quantize_lab(cimg1, 64)
cimg2_labq = quantize_lab(cimg2, 64)
cimg3_labq = quantize_lab(cimg3, 64)

print "Images with L-channel quantized and converted to RGB"
display3(cimg1_labq, cimg2_labq, cimg3_labq)
```

```
Images with L-channel quantized and converted to RGB
```



Question 3

In [512]:

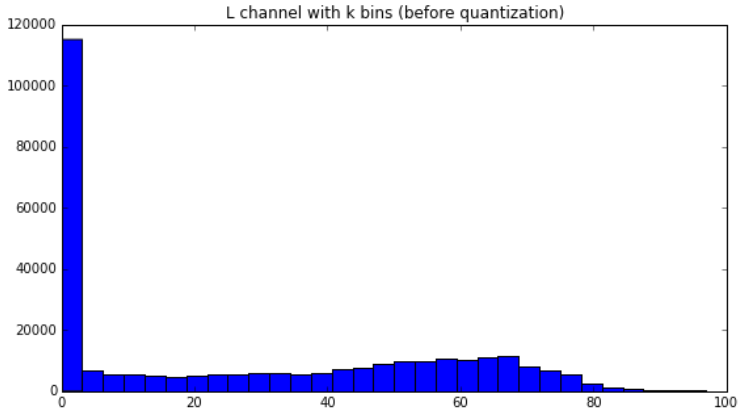
```
def get_ssd(rgb1, rgb2):
    return np.sum((rgb1[:,:,0:3] - rgb2[:,:,0:3]) ** 2)

print "\nSSD between rgb quantized images and original"
print "Image 1:", get_ssd(cimg1_rgbq, cimg1), "Image 2:", get_ssd(cimg2_rgbq, ci
mg2), "Image 3:", get_ssd(cimg3_rgbq, cimg3)

print "\nSSD between images with quantized L channel original"
print "Image 1:", get_ssd(cimg1_labq, cimg1), "Image 2:", get_ssd(cimg2_labq, ci
mg2), "Image 3:", get_ssd(cimg3_labq, cimg3)
```

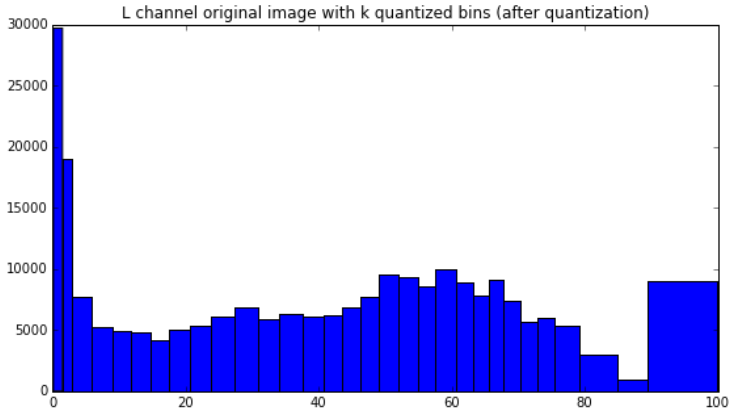
```
SSD between rgb quantized images and original
Image 1: 2542.38368783 Image 2: 15241.2397718 Image 3: 2592.46925678

SSD between images with quantized L channel original
Image 1: 214.887018274 Image 2: 1122.5882833 Image 3: 163.256690896
```

Question 4

In [513]:

```
def plot_histogram(rgb, k):
    w, h = rgb.shape[0], rgb.shape[1]
    lab = color.rgb2lab(rgb)
    l = lab[:,:,0].reshape(w*h,)

    f, axes = plt.subplots(ncols=2, figsize=(20,5))

    # plot l channel with k bins
    minval = np.min(l)
    maxval = np.max(l)
    axes[0].set_title("L channel with k bins (before quantization)")
    axes[0].hist(l,bins=np.arange(minval,maxval,(1.0*(maxval-minval)/k)))

    # plot l channel of original image with k quantized L channel bins
    lab_img = quantize_lab(rgb, k, True)
    axes[1].set_title("L channel original image with k quantized bins (after qua
ntization)")
    axes[1].hist(l,bins=np.sort(np.unique(lab_img[:,:,0].reshape(w*h,))))

    plt.show()

plot_histogram(cimg1, 32)
```





Question 5

In [515]:

```
def analyze_image(cimg):
    print "\nOriginal Image:"
    display(cimg)

    for k in [16, 64]:
        print "\n\nANALYSIS FOR K =", k

        # RGB quantization
        cimg_rgbq = quantize_kmeans(cimg, k)
        print "\nRGB quantized image"
        display(cimg_rgbq)

        # SSD of RGB quantized image wrt original image
        print "SSD between RGB quantized image and original:",get_ssd(cimg_rgbq,
cimg)

        # LAB L-Channel quantization
        cimg_labq = quantize_lab(cimg, k)
        print "\nLAB L-Channel quantized image"
        display(cimg_labq)

        # SSD of LAB L-Channel quantized image wrt original image
        print "SSD between LAB L-Channel quantized image and original:",get_ssd(
cimg_labq, cimg)

        # Histogram for k
        print "\nHistogram for original and quantized image"
        plot_histogram(cimg, k)
```

IMAGE 2

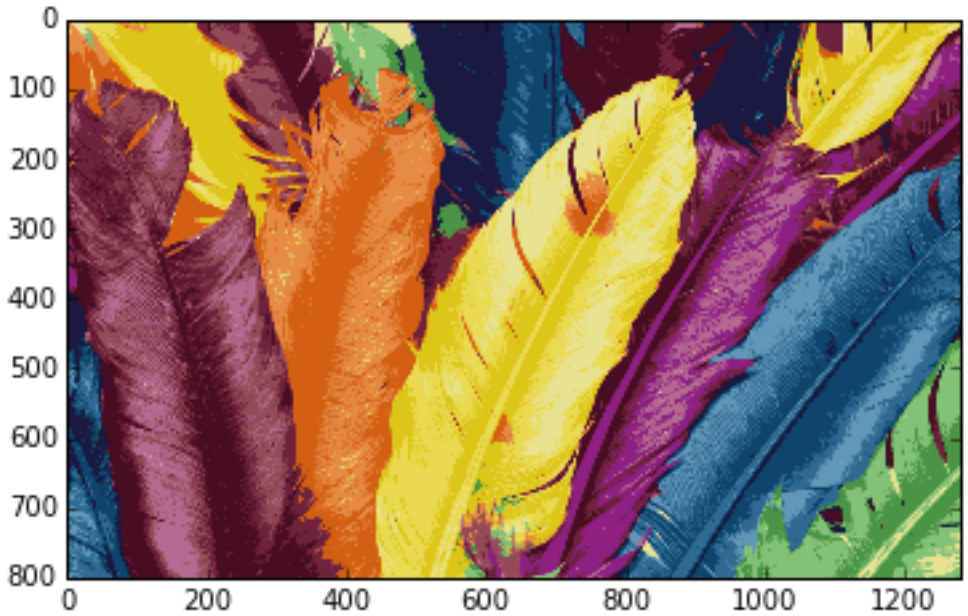
In [516]:

```
analyze_image(cimg2)
```

```
Original Image:
```

ANALYSIS FOR K = 16

RGB quantized image

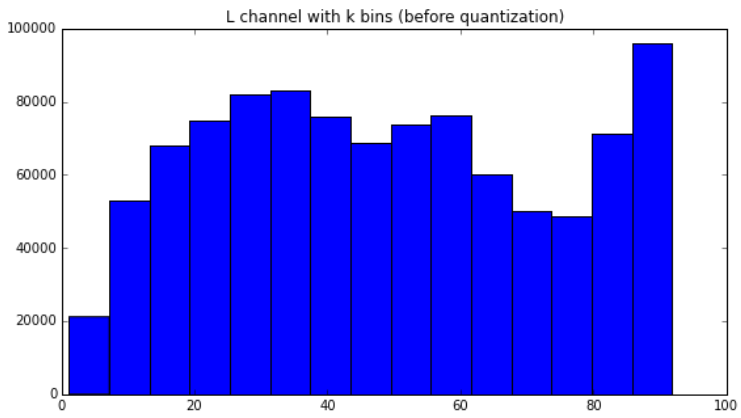


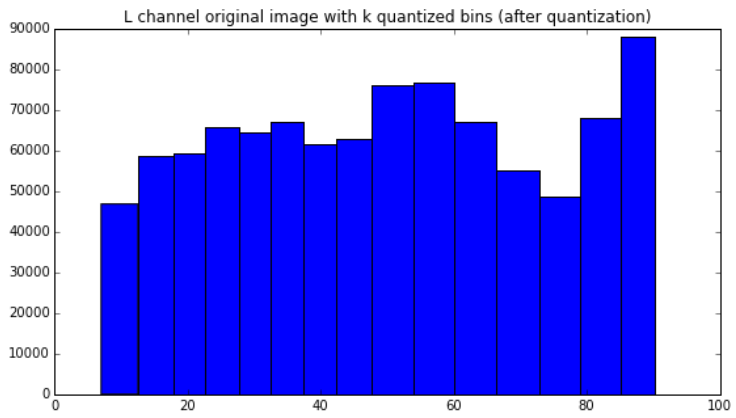
SSD between RGB quantized image and original: 15241.2397718

LAB L-Channel quantized image

SSD between LAB L-Channel quantized image and original: 1122.5882833

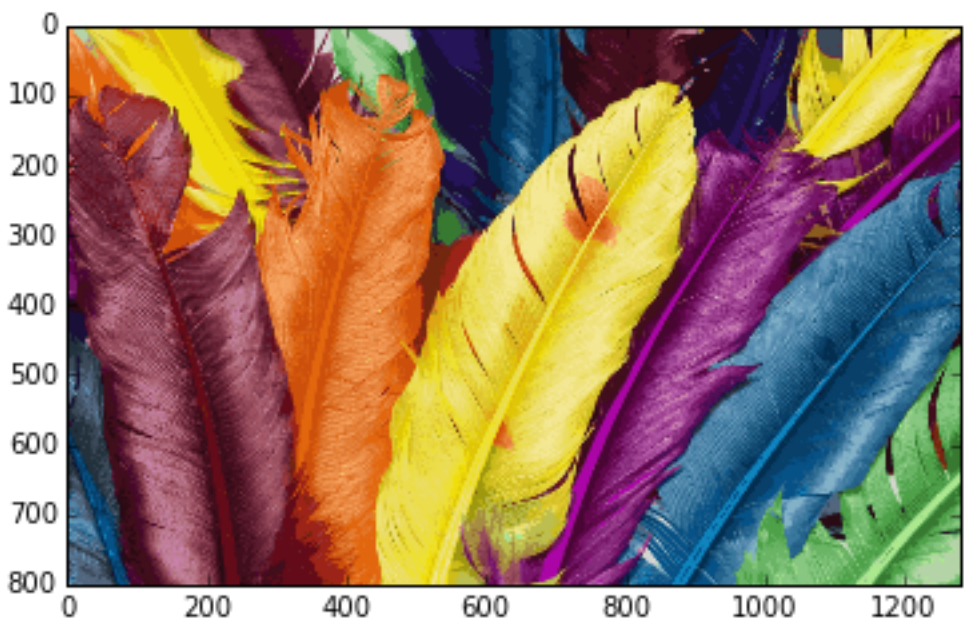
Histogram for original and quantized image





ANALYSIS FOR K = 64

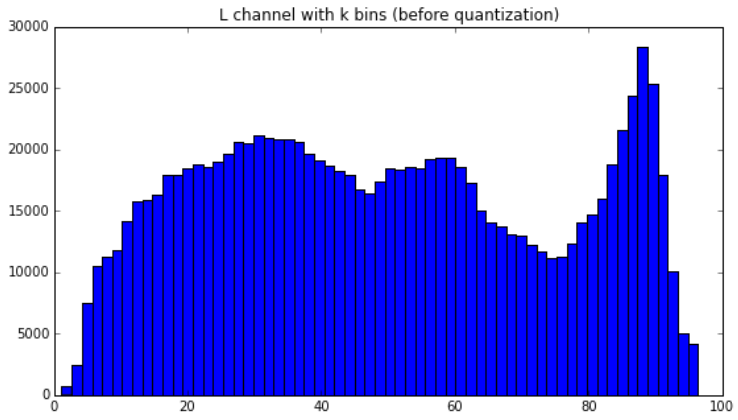
RGB quantized image



SSD between RGB quantized image and original: 4795.0050225

LAB L-Channel quantized image

```
SSD between LAB L-Channel quantized image and original: 76.259719044
6

Histogram for original and quantized image
```



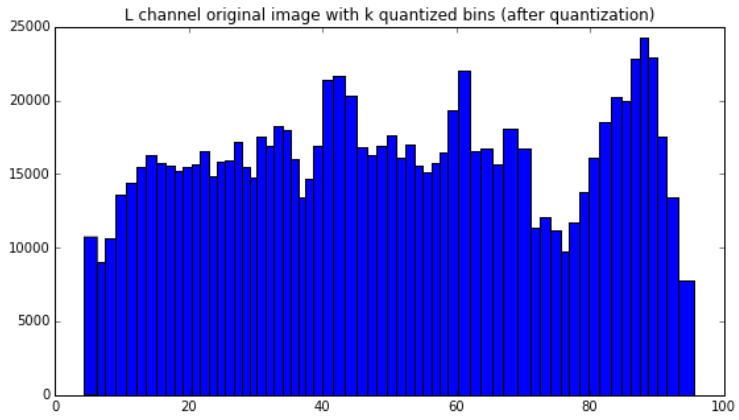


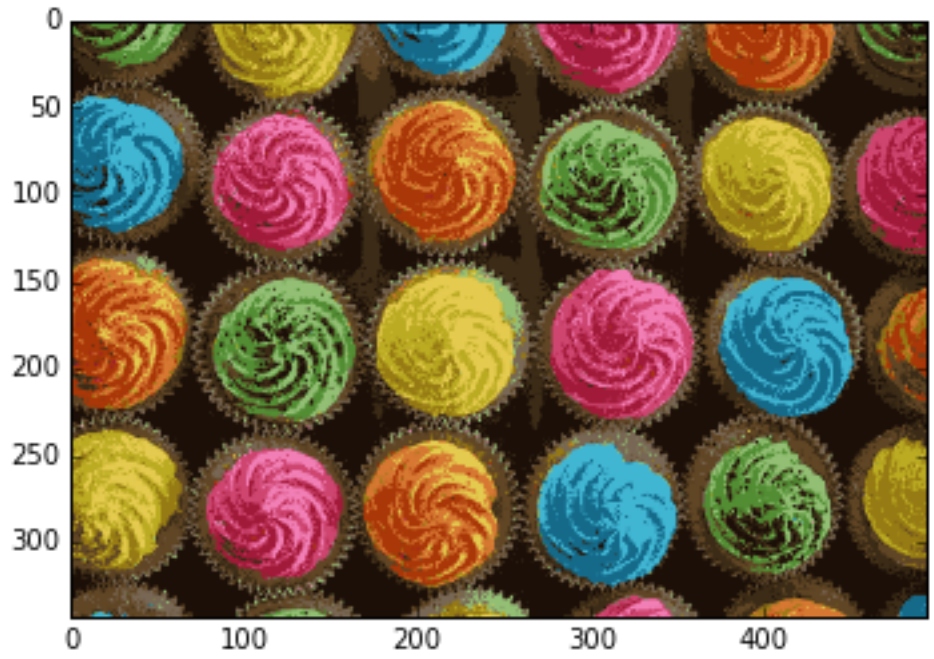
IMAGE 3

```
In [517]:
```

```
analyze_image(cimg3)
```

```
Original Image:
```


```
ANALYSIS FOR K = 16

RGB quantized image
```

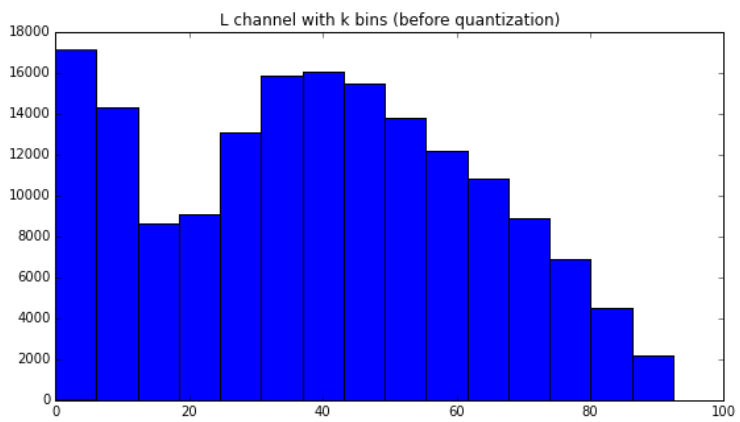


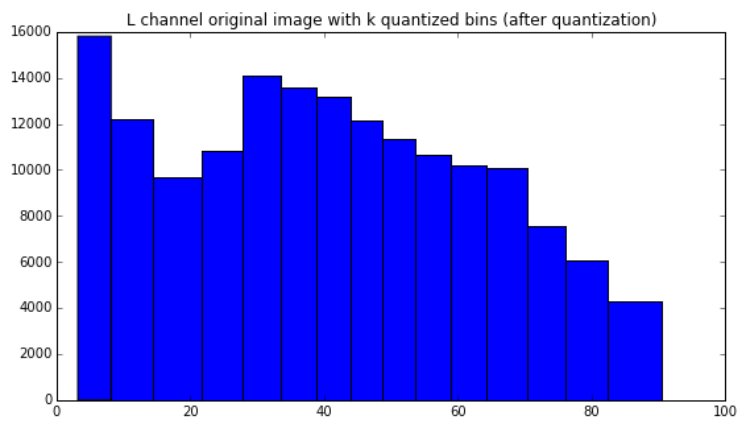
SSD between RGB quantized image and original: 2592.46925678

LAB L-Channel quantized image

SSD between LAB L-Channel quantized image and original: 163.25669089
6

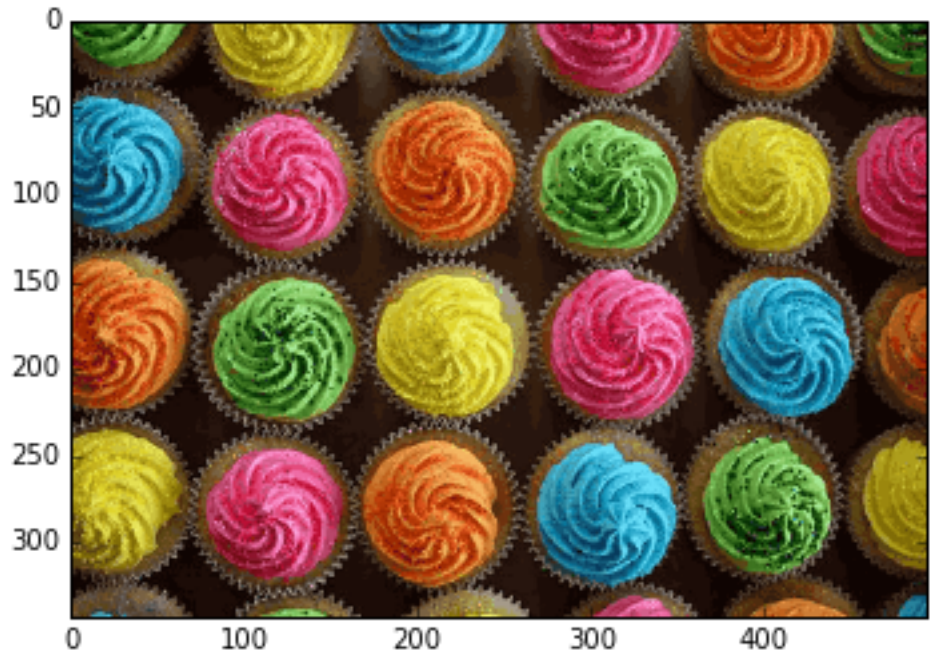
Histogram for original and quantized image





ANALYSIS FOR K = 64

RGB quantized image

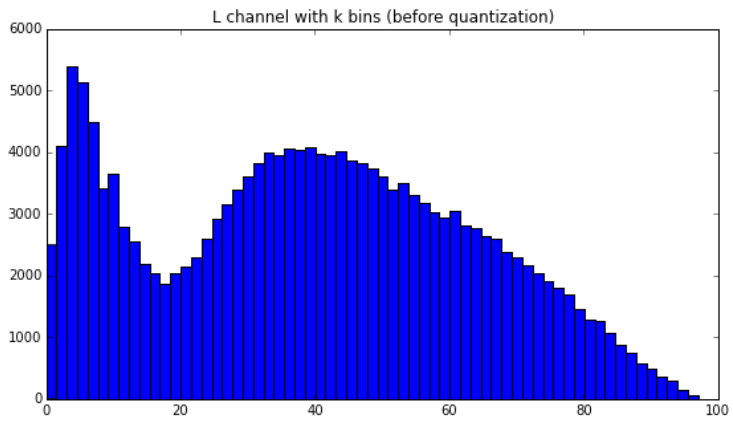


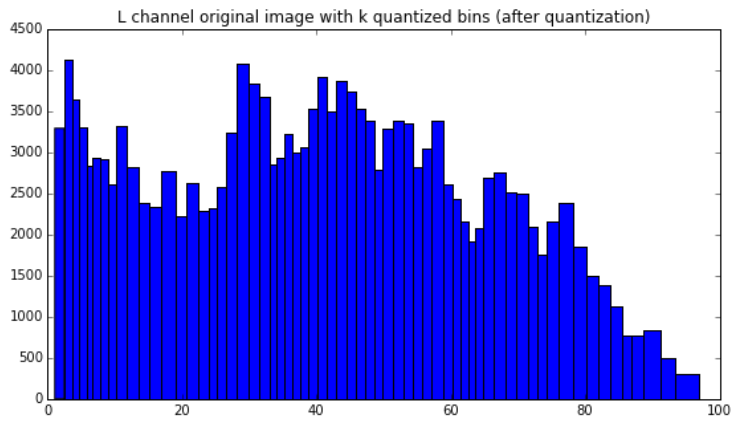

```
SSD between RGB quantized image and original: 807.936217374
```

```
LAB L-Channel quantized image
```


```
SSD between LAB L-Channel quantized image and original: 11.256391520
2
```

```
Histogram for original and quantized image
```





ANALYSIS OF RESULTS

Explain the results you get. What causes the differences before and after quantization?

We quantize the RGB values for the images first, display the quantized image, then calculate the SSD of the quantized image with the original. Finally, we plot a histogram of the L channel values of original image wrt a linear bin sequence and with bins chosen from the K-Means cluster centers.

We find that the quantized image for both RGB and L-Channel approximates the original image fairly well even while taking far lesser space. It is thus similar to image compression. We also see the L-Channel quantized images approximate the original image much more accurately than the RGB quantization. This is also confirmed by the reported SSD values. The SSD values for the quantized image is significantly smaller (~10x smaller) for the case of L-Channel quantized image when compared to RGB quantized image. This result can be due to the fact that in the case of L-Channel quantization, the A and B Channels remain continuous and only the L-Channel is quantized resulting in a closer approximation of the original image.

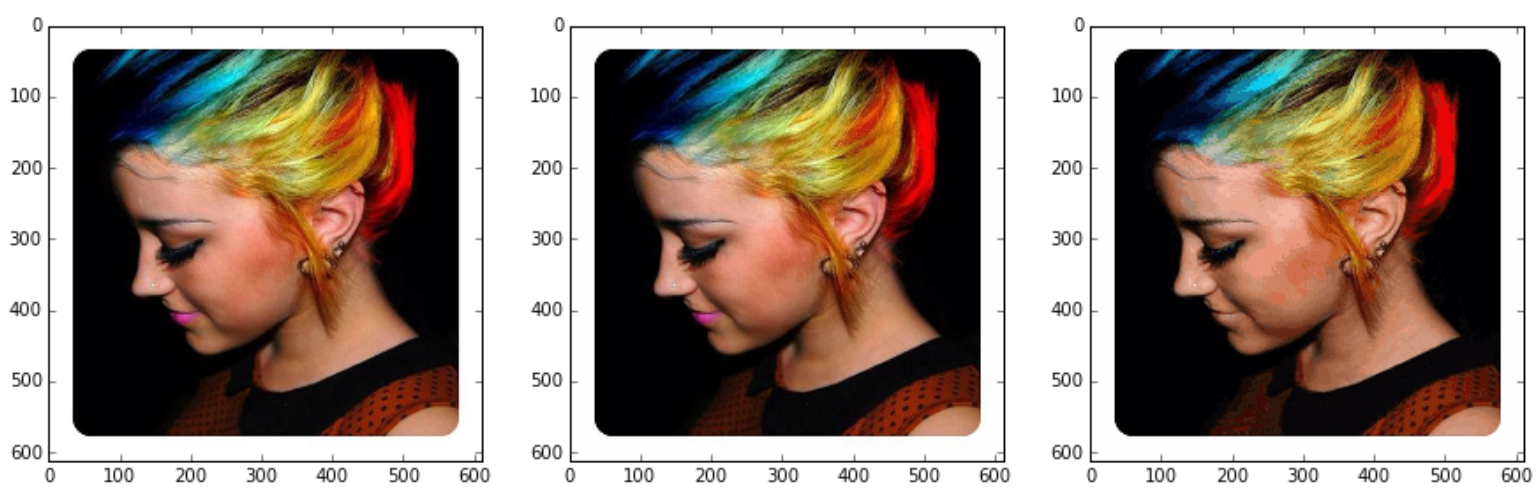
We notice that after quantization, the images seem to have patches which are particularly visible in the RGB image. These patches represent the clusters we get from the K-Means algorithm.

We notice from the histograms that the L-Channel values for quantized image appear to have a leveling effect across the cluster centers. The L-Channel quantized histograms are flatter with more uniform distribution of the L values and with lower peak L values than in the case of the unquantized image's histogram. This results in the LAB quantized image having a higher overall luminosity than the original image producing a color enhancement effect. This can be seen easily when comparing the original image, L-Channel quantized image and the RGB quantized image side by side as shown below. Also, the girl's hair appears smoother and shinier in the case of the L-Channel quantized image. Finally, when we look at the girl's lips, we notice that the color is accurately preserved in the LAB quantization, but it is totally misrepresented by the RGB quantization. This happens because small shifts in certain regions of the RGB color space can produce very different final colors unlike the case of the LAB space

In [524]:

```
print "L-R: Original Image, L-Channel Quantized Image, RGB Quantized Image"
display3(cimg1, cimg1_labq, cimg1_rgbq)
```

```
L-R: Original Image, L-Channel Quantized Image, RGB Quantized Image
```



What is the effect of changing k?

As k increases, the quantized image better approximates the original image and the SSD between the quantized image and the original (both for RGB quantization and L-Channel quantization) reduces.

Do you get the same results for every execution of your program?

We expect to get the same results for every execution of the program as the random_state has been set to 0 in all of the methods that use random values. However, if we did not set the random_state to a fixed value, we will not get the same results for every execution of our program

Task 3: Edge detection

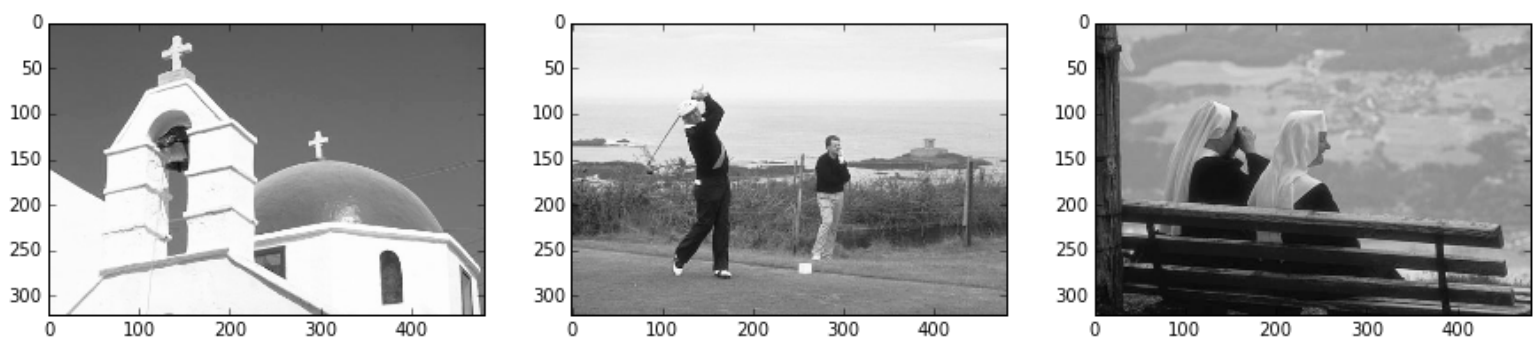
Part A

First we read in and display the images

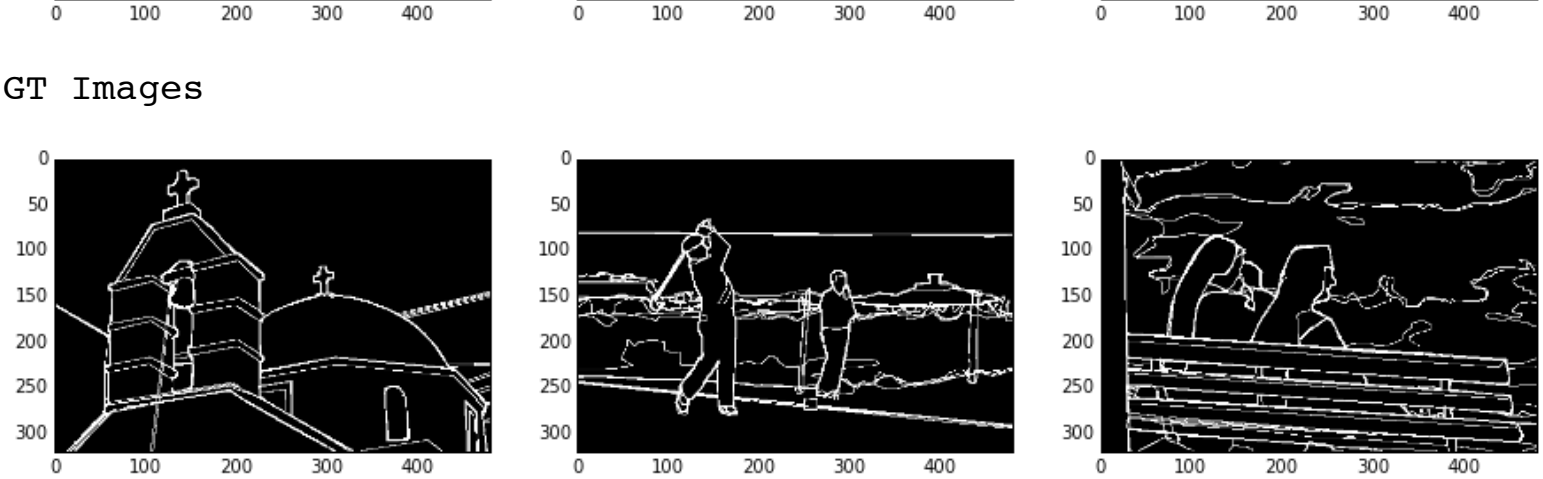
In [556]:

```
eimg1 = misc.imread('./Images/Q3/Church.jpg',0)
eimg1gt = misc.imread('./Images/Q3/Church_GT.bmp',0)
eimg2 = misc.imread('./Images/Q3/Golf.jpg',0)
eimg2gt = misc.imread('./Images/Q3/Golf_GT.bmp',0)
eimg3 = misc.imread('./Images/Q3/Nuns.jpg',0)
eimg3gt = misc.imread('./Images/Q3/Nuns_GT.bmp',0)

print "\nOriginal Images"
display3(eimg1,eimg2,eimg3, cm.Greys_r)
print "GT Images"
display3(eimg1gt,eimg2gt,eimg3gt, cm.Greys_r)
```

Original Images



GT Images

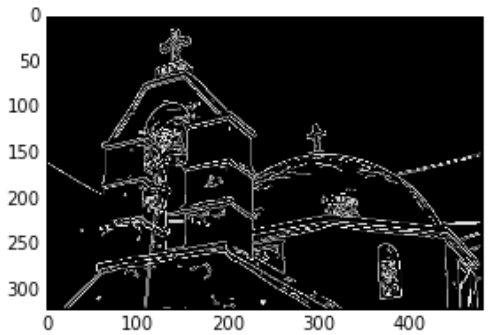
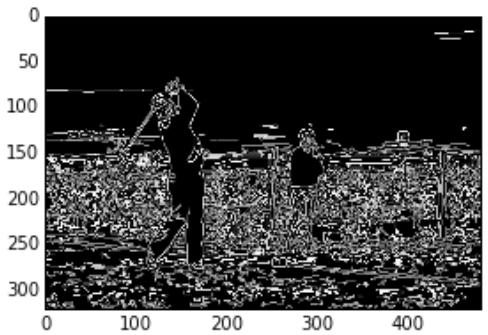
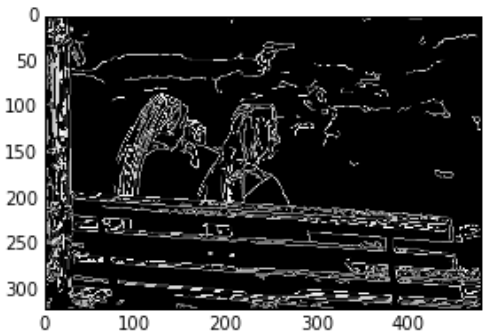


Canny Edge Detection

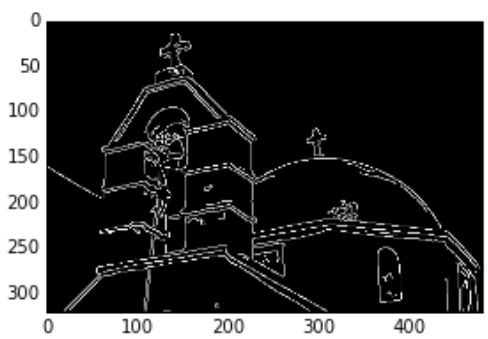
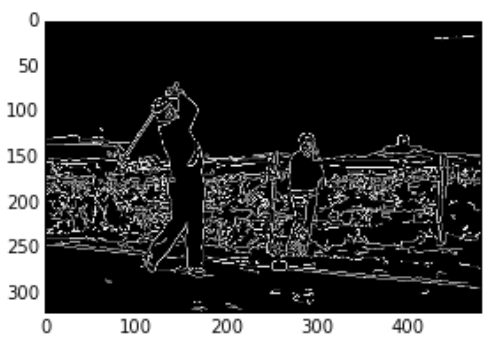
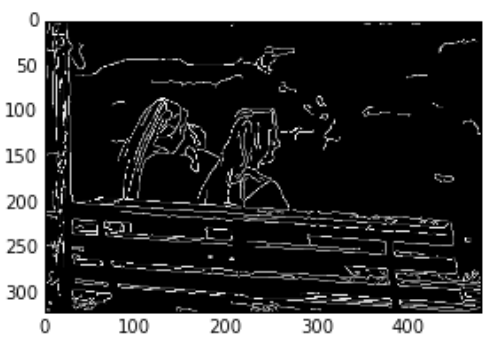
In [579]:

```
for sigma in np.arange(1,7):
    print "Sigma: ", sigma/2.0
    display3(canny(eimg1, sigma=sigma/2.0),canny(eimg2, sigma=sigma/2.0),canny(e
img3, sigma=sigma/2.0), cm.Greys_r)
```

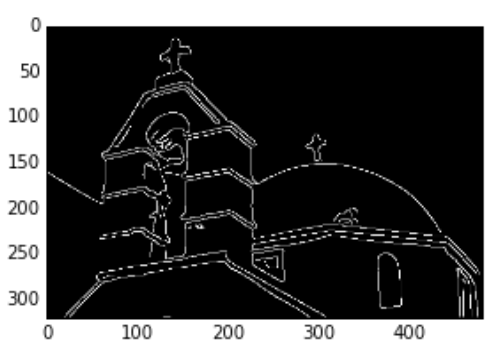
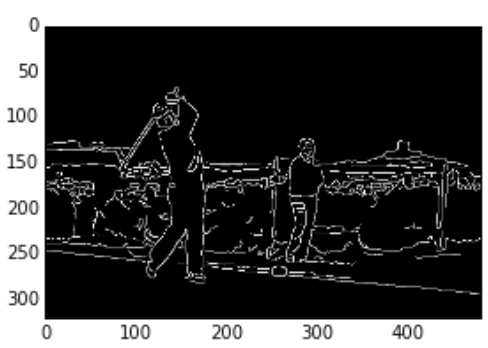
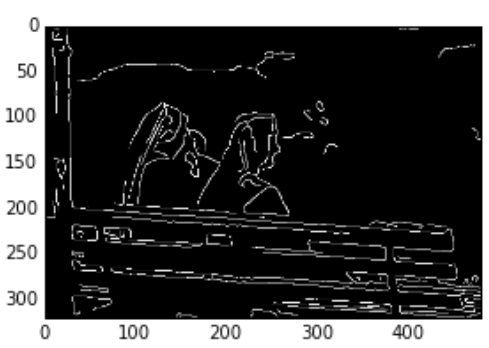
```
Sigma:  0.5
```

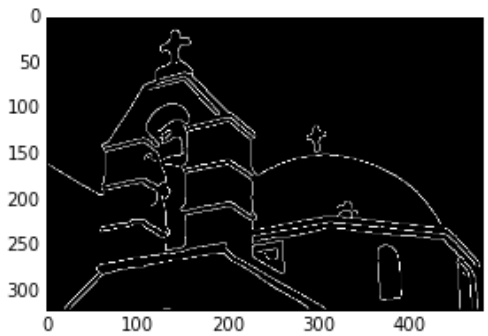
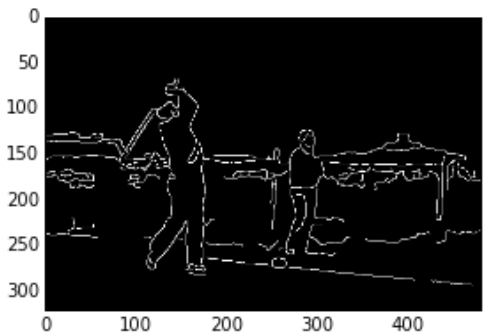
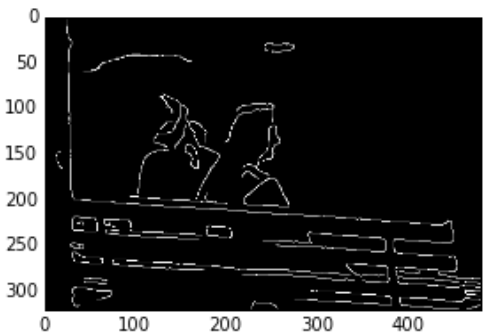
Sigma: 1.0

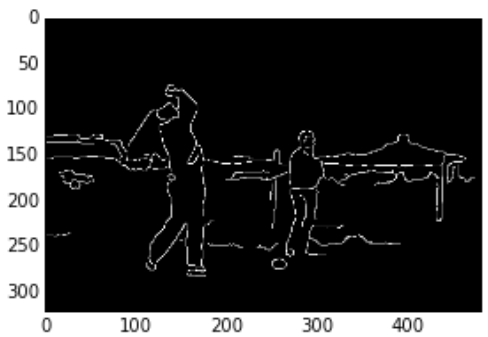
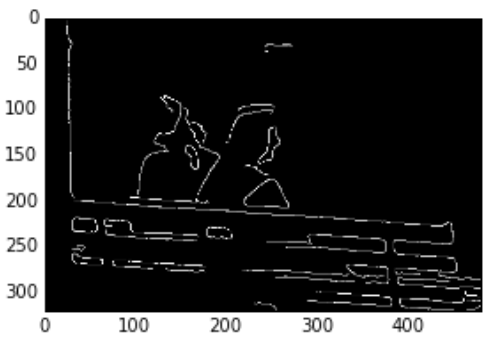
Sigma: 1.5

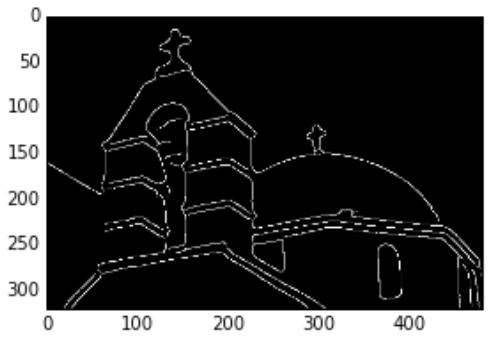
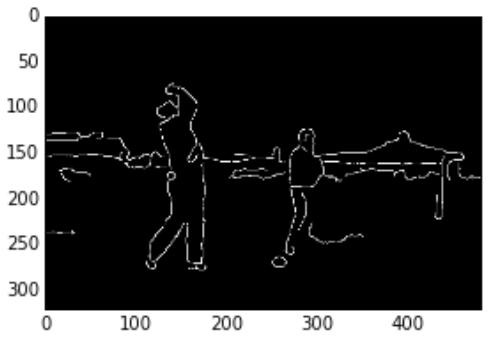
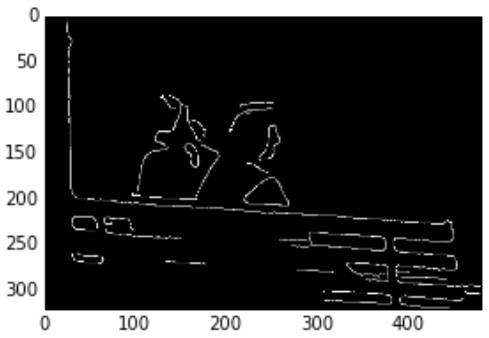
Sigma: 2.0

Sigma: 2.5

Sigma: 3.0

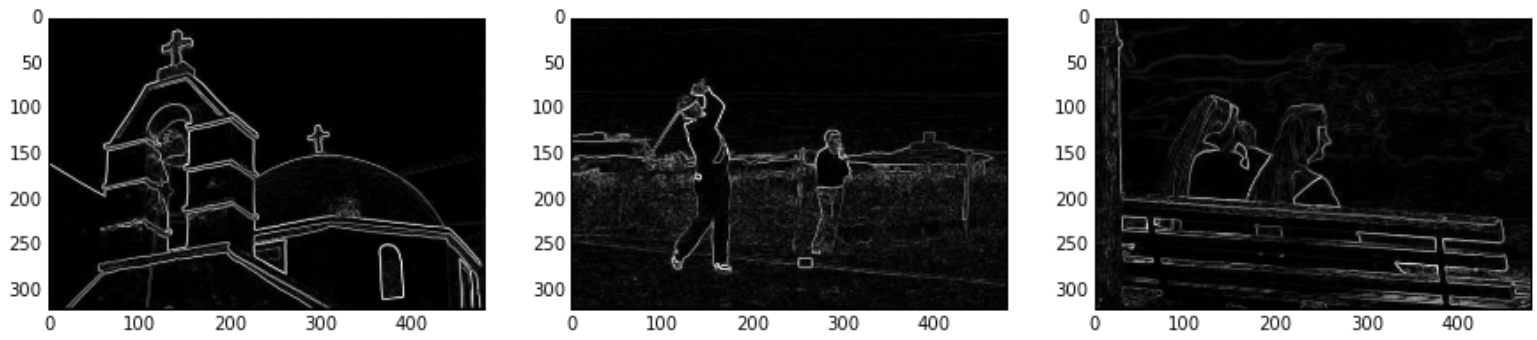
Based on the above results, 1.5 is found to be the best parameter value for sigma

Sobel Filtering

In [594]:

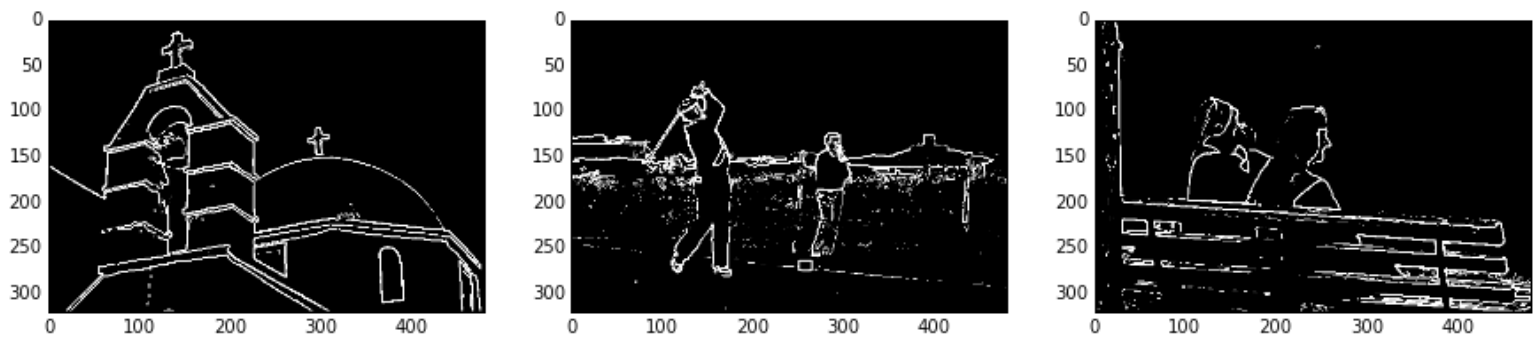
```
from skimage.filters import sobel, threshold_otsu

def global_threshold(eimg):
    threshold = threshold_otsu(eimg)
    eimg_threshold = eimg > threshold
    return eimg_threshold, threshold

eimg1_sobel = sobel(eimg1)
eimg2_sobel = sobel(eimg2)
eimg3_sobel = sobel(eimg3)

eimg1_sobel_thresholded, otsu1 = global_threshold(eimg1_sobel)
eimg2_sobel_thresholded, otsu2 = global_threshold(eimg2_sobel)
eimg3_sobel_thresholded, otsu3 = global_threshold(eimg3_sobel)

print "Sobel Filter output without thresholding"
display3(eimg1_sobel, eimg2_sobel, eimg3_sobel, cm.Greys_r)

print "Otsu thresholds for: \nimage1 =", otsu1, "\nimage2 =", otsu2, "\nimage3 =
", otsu3

print "\nSobel Filter output with global thresholding using otsu method"
display3(eimg1_sobel_thresholded, eimg2_sobel_thresholded, eimg3_sobel_threshold
ed, cm.Greys_r)
```

Sobel Filter output without thresholding



```
Otsu thresholds for:
image1 = 0.137356169245
image2 = 0.109153804057
image3 = 0.0889513681665
```

Sobel Filter output with global thresholding using otsu method



We use otsu thresholding to arrive at the most suitable threshold. This is image specific and we get the following thresholds for each image:

image1 threshold = 0.137356169245

image2 threshold = 0.109153804057

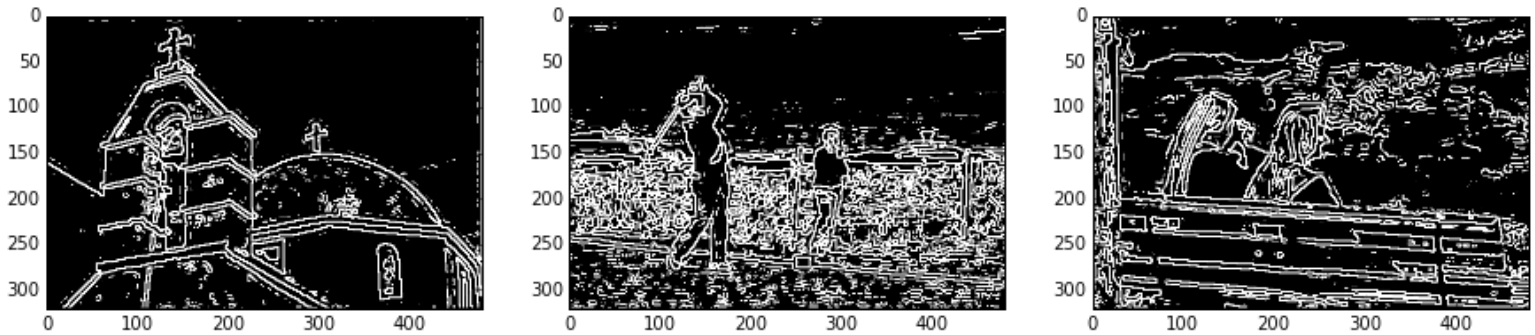
image3 threshold = 0.0889513681665

Gaussian-Laplace Filtering

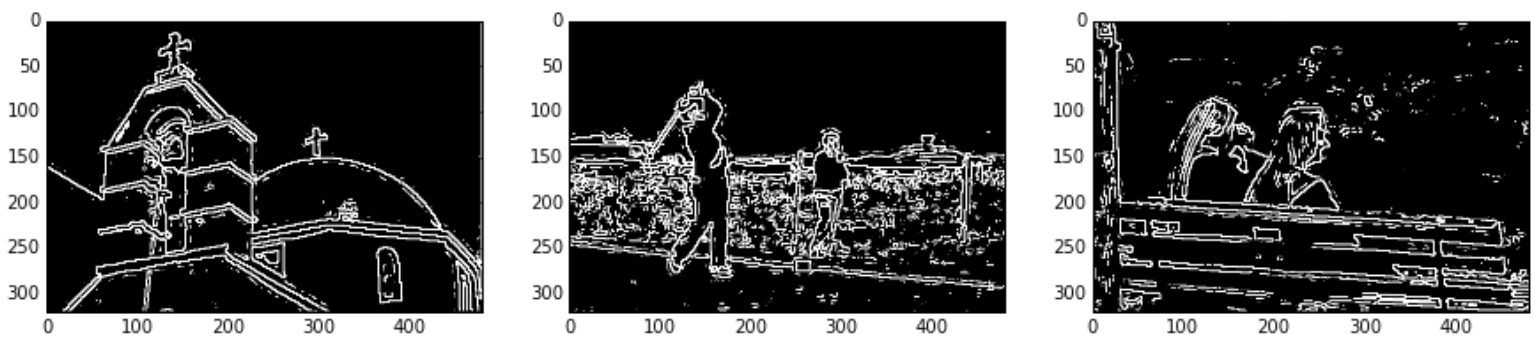
In [609]:

```
# Reference: http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.nd
image.filters.gaussian_laplace.html#scipy.ndimage.filters.gaussian_laplace

def laplace_of_gaussian(img, sigma, threshold):
    log_img = ndimage.gaussian_laplace(img, sigma)
    log_edge_img = scipy.zeros(log_img.shape)
    w, h = log_edge_img.shape[1], log_edge_img.shape[0]
    for i in range(1, h - 1):
        for j in range(1, w - 1):
            # we take a 3x3 patch to check for zero crossings in the middle
            # we do this as the discretized values may or may not have zeros
            # and we need to detect change in signs across the patch instead
            patch = log_img[i-1:i+2, j-1:j+2]
            p = log_img[i, j]
            max_p = patch.max()
            min_p = patch.min()
            if (p > 0):
                is_zero_crossing = True if min_p < 0 else False
            else:
                is_zero_crossing = True if max_p > 0 else False
            if ((max_p - min_p) > threshold) and is_zero_crossing:
                log_edge_img[i, j] = 1
    return log_edge_img


img1_32 = np.asarray(eimg1, dtype="float32" )
img2_32 = np.asarray(eimg2, dtype="float32" )
img3_32 = np.asarray(eimg3, dtype="float32" )

# threshold 0-30
def plot_laplace_of_gaussian(sigma,threshold):
    print "Gaussian-Laplace filtering, sigma =",sigma,", threshold =", threshold
    out1 = laplace_of_gaussian(img1_32,sigma,threshold)
    out2 = laplace_of_gaussian(img2_32,sigma,threshold)
    out3 = laplace_of_gaussian(img3_32,sigma,threshold)
    display3(out1,out2,out3,cm.Greys_r)
    return out1,out2,out3

plot_laplace_of_gaussian(2,1)
plot_laplace_of_gaussian(2,2)
plot_laplace_of_gaussian(2,3)
plot_laplace_of_gaussian(3,1)
plot_laplace_of_gaussian(3,2)
plot_laplace_of_gaussian(4,1)
plot_laplace_of_gaussian(4,2)
print ""
```

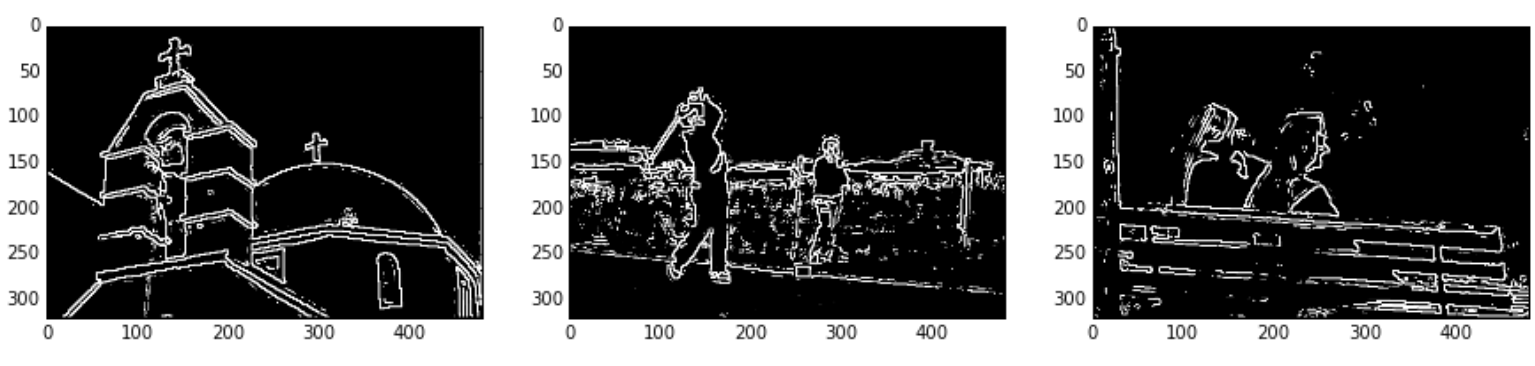
```
Gaussian-Laplace filtering, sigma = 2 , threshold = 1
```



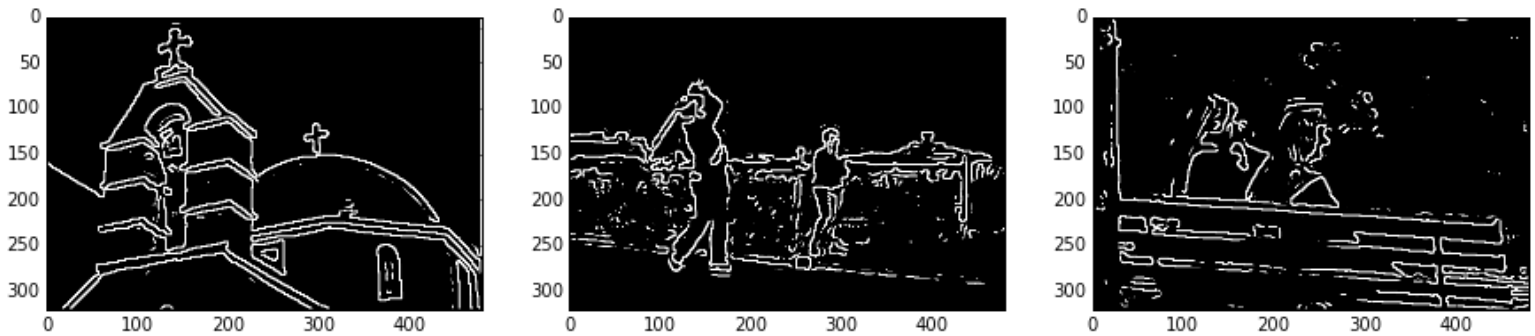
Gaussian-Laplace filtering, sigma = 2 , threshold = 2



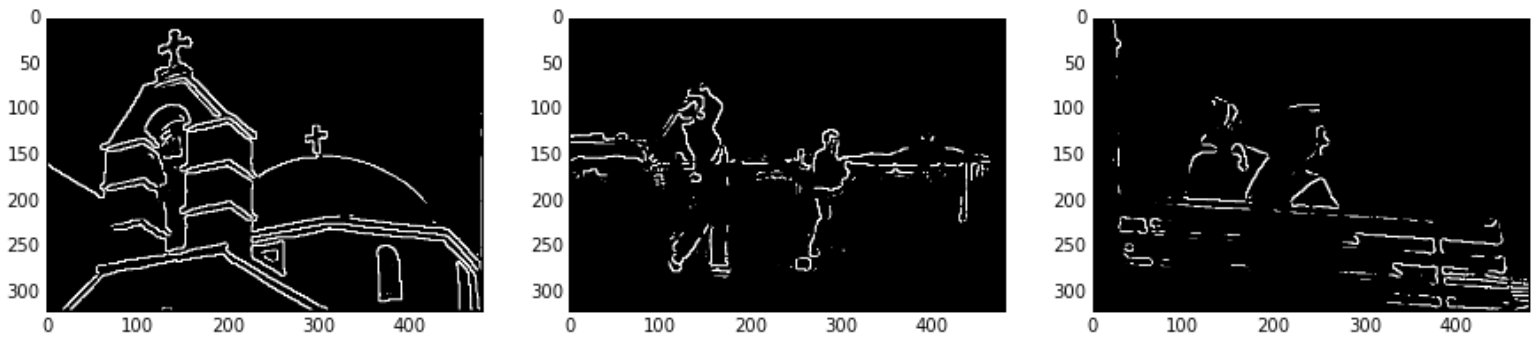
Gaussian-Laplace filtering, sigma = 2 , threshold = 3



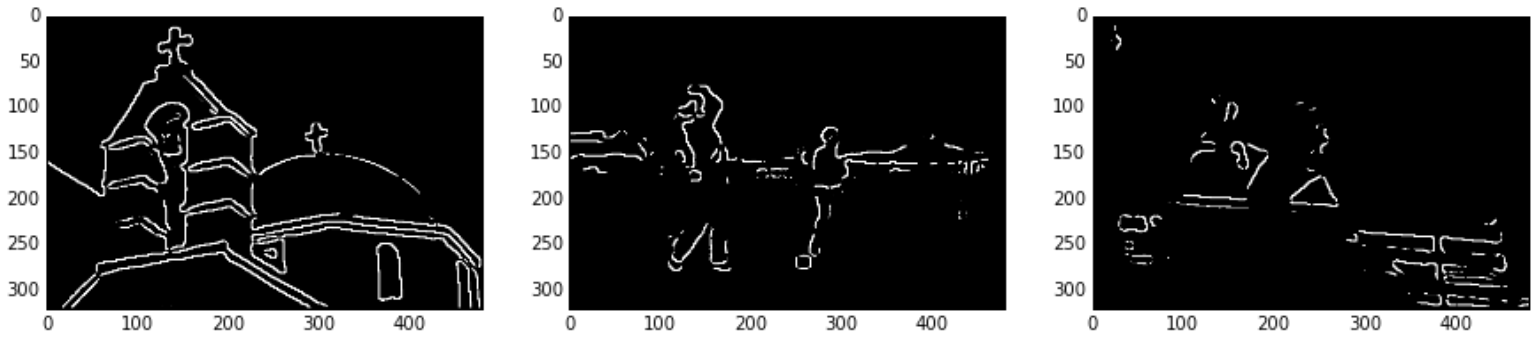
Gaussian-Laplace filtering, sigma = 3 , threshold = 1



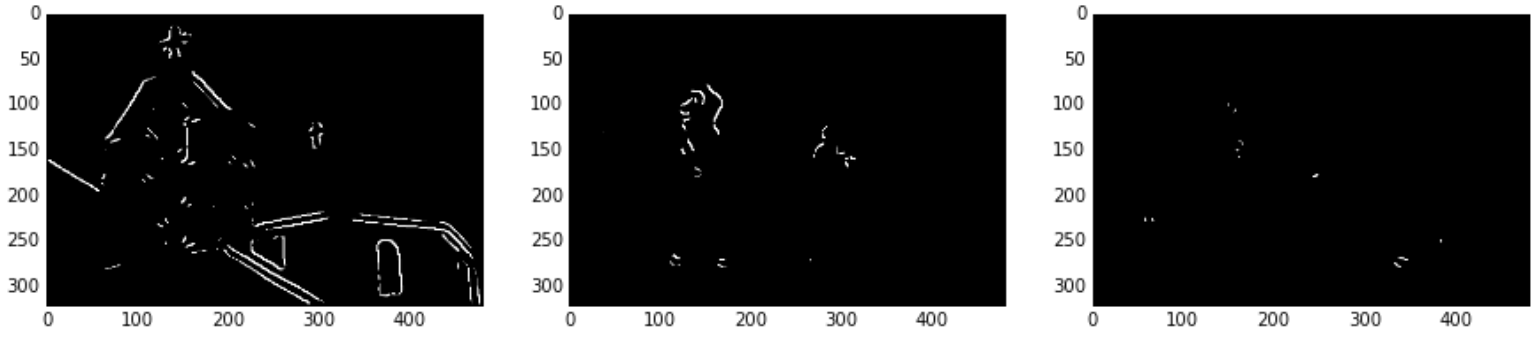
Gaussian-Laplace filtering, sigma = 3 , threshold = 2



Gaussian-Laplace filtering, sigma = 4 , threshold = 1



Gaussian-Laplace filtering, sigma = 4 , threshold = 2



Based on the results above, we find that the optimal parameter values are as follows:

sigma = 3 and **threshold = 1**

1. Explain in a few sentences the way of operation of each of the three detectors.

Canny Edge Detection: Here, we first smooth the image with a Gaussian filter to remove noise which corresponds to high frequency signals. Then, we get the gradient along the horizontal and vertical axes of the denoised image. Note, this can be done in one step by convolving the image with the Derivative-of-Gradient operator because of the associative property of filter convolution. After this, we suppress non-maxima values along the direction of the gradient (non-maximum suppression). Next, we perform thresholding on the gradient values. Finally, we try to get rid of edges that are not long and are not connected to any other edges as these are likely to be noise.

Sobel Filtering: We convolve the image with the Sobel masks in the X and Y directions. Then threshold the gradient magnitude with a suitable value of threshold setting all values above as 1 and all values below as 0. We can additionally do non-maximum suppression to reduce edge-thickness.

Gaussian-Laplace Filtering: First, we smooth the image with a Gaussian filter to remove noise. Then we apply the Laplacian operator (in both X and Y direction) to the smoothed image. This can be done in a single step by convolving the Laplacian of the Gaussian (LoG) with the image. As the Laplacian is effectively the second derivative, the edge maxima will be represented by 0s in the LoG image. We next have to detect the zero crossings where LoG the value goes from -threshold/2 to +threshold/2 or vice versa and set these to 1 and all other places to 0. These 1s will then correspond to the edges.

2. Explain the threshold parameter and the sigma parameter of the detectors. What happens when you increase/decrease each of the parameters? Which parameter values work best?

Canny Edge Detection: Here, sigma is the standard deviation of the Gaussian used to smooth the image prior to applying the gradient operator. Increasing sigma removes an increasing amount of high frequency components from the image. Since the edges are also high-frequency portions of the image, increasing the sigma beyond an optimal value starts removing the edges from the image as well. This is observed in the output for canny edge detector above. Also, 1.5 is found to be the best parameter value for sigma.

Sobel Filtering: The threshold is applied to gradient magnitude by setting all values above the threshold as 1 and all values below as 0. The lower the threshold, the thicker the edges and the higher the noise that will be included in the edges. In our implementation, we use Otsu Thresholding, which performs clustering on the image gradient to identify suitable value of threshold. This identifies a different threshold value for each image. The values we determined are: image1 threshold = 0.137356169245 image2 threshold = 0.109153804057 image3 threshold = 0.0889513681665

Gaussian-Laplace Filtering: Here, sigma is again the standard deviation of the Gaussian used to smooth the image prior to applying the gradient operator. Increasing sigma removes an increasing amount of high frequency components from the image. Since the edges are also high-frequency portions of the image, increasing the sigma beyond an optimal value starts removing the edges from the image as well.

The threshold is minimum jump in the Laplace of Gaussian (LoG) of image's values at a potential zero-crossing we find. Increasing the value of the threshold will result in a higher number of edge detections and beyond an optimal point, will start including a lot of noise as well.

Based on our experiments, the optimal values for sigma and threshold are sigma = 3 and threshold = 1

3. Run the Gaussian-Laplace method with threshold = 0. The edges are supposed to be closed curves. Explain why this happens.

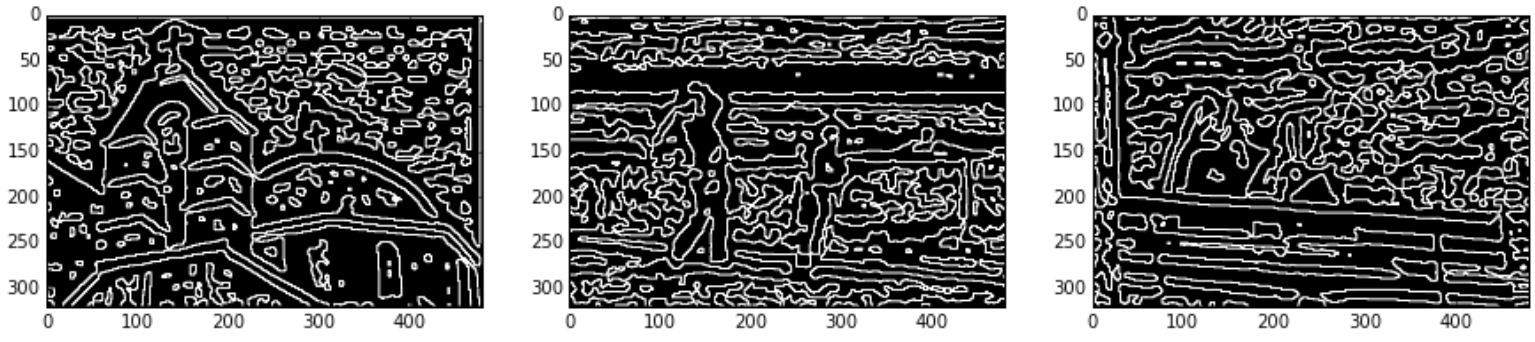
Below, we run the Gaussian-Laplace (LoG) method with threshold = 0 and find that the edges form closed curves. When we use zero threshold, it means we do not have any minimum value between a zero crossing's next element and it's previous element along the LoG gradient direction. The zero crossings always lie on closed contours. The reason why this happens can be imagined by thinking of the image as a topographical map. The zero-crossings are simply the points with an altitude of zero. These will then lie on closed contours.

Since all of the zero crossings are included in the output for threshold = 0, the final output has only closed curves (except near the image edges where it may not be closed).

In [612]:

```
plot_laplace_of_gaussian(4,0)
print ""
```

```
Gaussian-Laplace filtering, sigma = 4 , threshold = 0
```



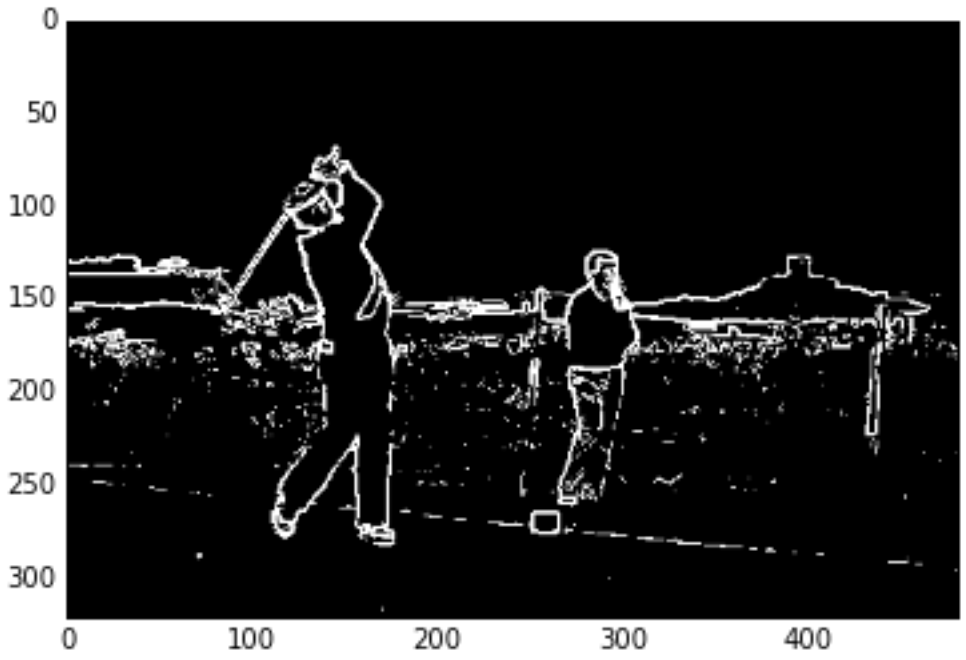
4. Invent your own edge detection method. It can be a variation of one of the tested ones or some other detector you came up with. Please do not copy something from the literature. We would like you to identify a limitation of the detectors you tested above and design your detector such that it aims to solve that limitation. A good idea with a short clear explanation is what would please us most. Demonstrate the results visually.

One problem we note with the sobel filter is the presence of a lot of noise in the detected edges for the image with the golfers in it (image 2). One way to solve this problem will be to apply a median filter to the sobel gradient image to remove tiny white regions in the background before thresholding it. As can be seen below, applying the median filter does remove the noisy edge detections in the grass and bushes behind the golfers to a certain extent

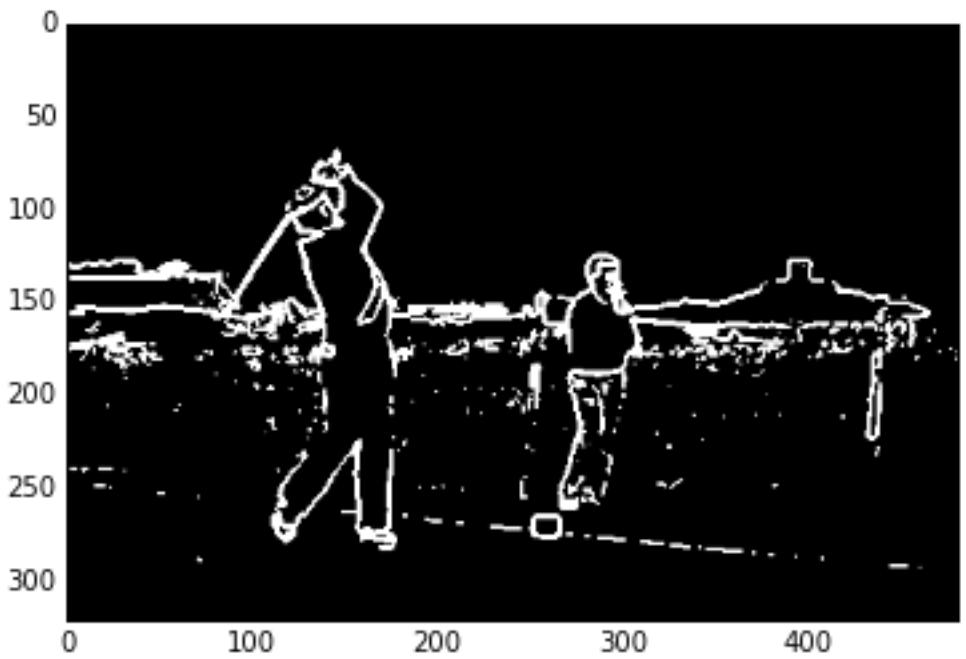
In [632]:

```
img_with_noise, otsu_temp = global_threshold(eimg2_sobel)
print "Prior to removing noisy edge detections in the grass"
display(img_with_noise, cm.Greys_r)

mfimg = scipy.ndimage.filters.median_filter(eimg2_sobel, 2)
imgb, otsub = global_threshold(mfimg)
print "After removing noisy edge detections in the grass using median filter"
display(imgb, cm.Greys_r)
```

Prior to removing noisy edge detections in the grass



After removing noisy edge detections in the grass using median filte
r



In [ ]:

```
# Imports

%matplotlib inline
import numpy as np
import scipy
from scipy import signal
from scipy import misc
from matplotlib import pylab as plt
import matplotlib.cm as cm
import cv2
from scipy import ndimage
from scipy import misc
from sklearn.utils import shuffle
from sklearn import cross_validation
from sklearn.cluster import KMeans
from skimage.feature import canny
```

Part B

Question 1

Precision in image edge detection means how many detected edges are acturally edge, while recall meatures how many edges are being detected. So having a high precision meaning mostly correct, but cannot say if the detector missed may edges. And vice versa. Therefore both precision and recall are important when evaluating an edge detector.

Question 2

We used the precision_recall_fscore_support function from sklearn.metrics to calculate the precision, recall and F score at the same time. As for input, we used np.ravel to make both GT image and detected edge image into a binary list as true value and predicted value perspectively.

Another problem is that, each method have different threshold. For example, Canny threshold ranges from 0 to 1000, Sobel is from 0 to 0.6, LoG (Laplace of Gaussian) is from 0 to 30. I normalized the threshold by dividing each range into 10 points, so that each method per image have 10 data points.

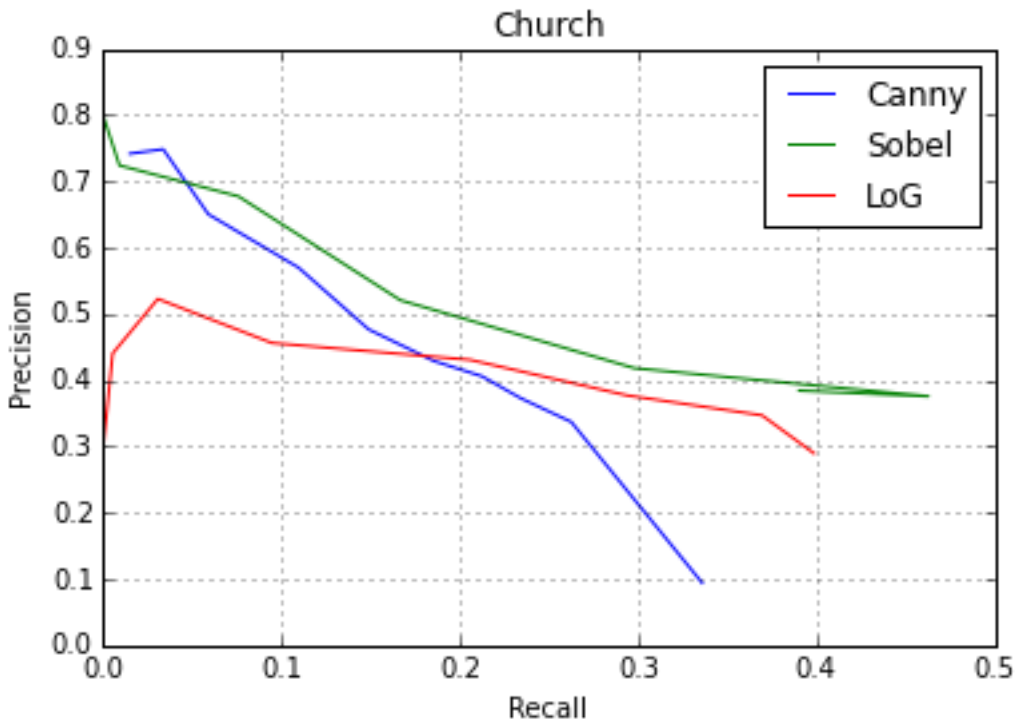
Then we calculated each three method on each three images, so its 3x3 graph ploted. We plot the three method of one image in one graph, so the result is shown below.

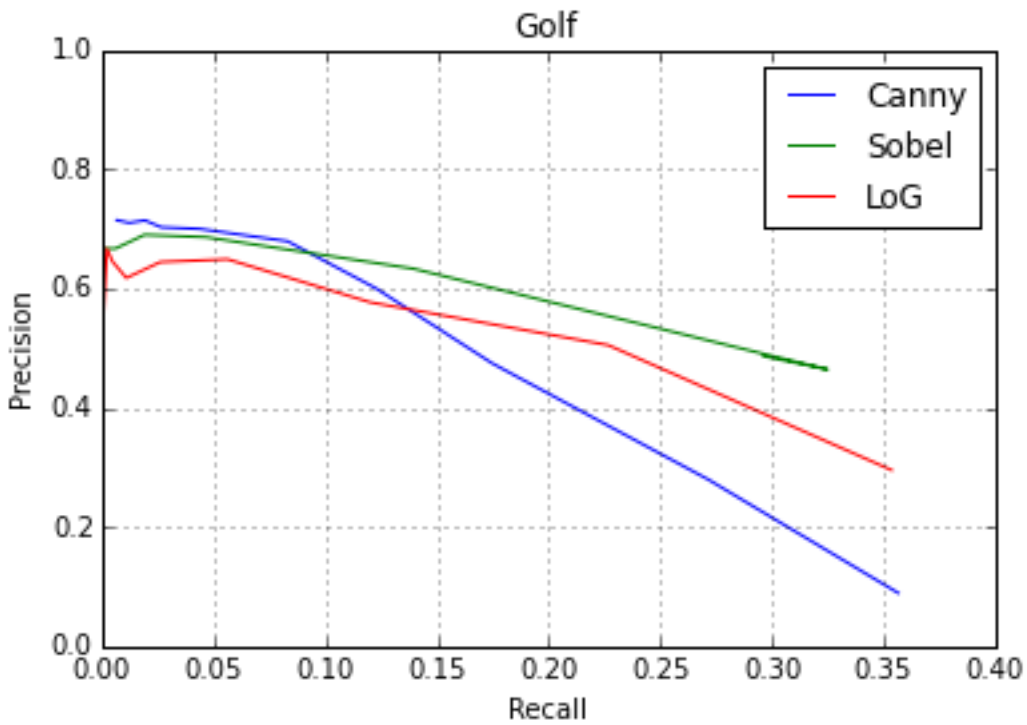
In [161]:

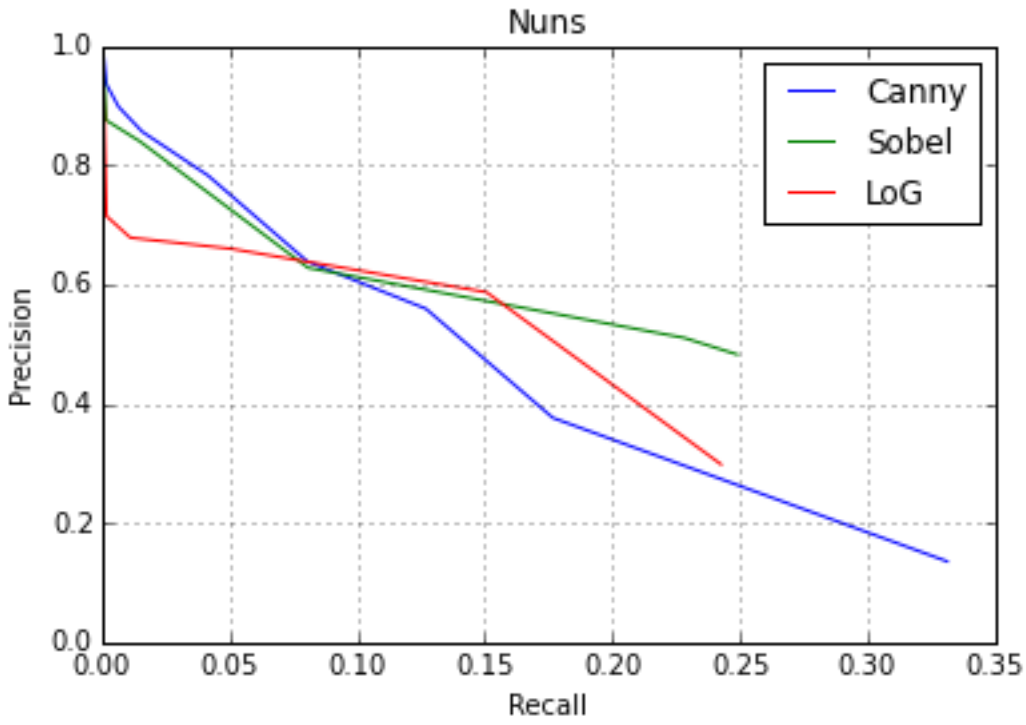
```
def pecision_recall(pe,re,fs,gt,ed):
    y_true = np.ravel(gt)
    y_pred = np.ravel(ed)
    a,b,c,d = precision_recall_fscore_support(y_true, y_pred,average='binary')
    pe.append(a)
    re.append(b)
    fs.append(c)

def plot(st):
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.legend(loc=1)
    plt.title(st)
    plt.grid(True)
    plt.show()


#Canny
thres_can = np.arange(0,1000,100)
peci_can_1 = []
recall_can_1 = []
fscore_can_1 = []
peci_can_2 = []
```

```
recall_can_2 = []

fscore_can_2 = []
peci_can_3 = []
recall_can_3 = []
fscore_can_3 = []
for i in range(len(thres_can)):
    can_edge1,can_edge2,can_edge3 = cannythres(thres_can[i],thres_can[i])
    pecision_recall(peci_can_1,recall_can_1,fscore_can_1,im1gt,can_edge1)
    pecision_recall(peci_can_2,recall_can_2,fscore_can_2,im2gt,can_edge2)
    pecision_recall(peci_can_3,recall_can_3,fscore_can_3,im3gt,can_edge3)

#Sobel
thres_sob = np.arange(0,1,0.1)
peci_sob_1 = []
recall_sob_1 = []
fscore_sob_1 = []
peci_sob_2 = []
recall_sob_2 = []
fscore_sob_2 = []
peci_sob_3 = []
recall_sob_3 = []
fscore_sob_3 = []
for j in range(len(thres_sob)):
    sob_edge1,sob_edge2,sob_edge3 = sobthres(thres_sob[j])
    pecision_recall(peci_sob_1,recall_sob_1,fscore_sob_1,im1gt,sob_edge1)
    pecision_recall(peci_sob_2,recall_sob_2,fscore_sob_2,im2gt,sob_edge2)
    pecision_recall(peci_sob_3,recall_sob_3,fscore_sob_3,im3gt,sob_edge3)

#LoG
thres_log = np.arange(0,30,3)
peci_log_1 = []
recall_log_1 = []
fscore_log_1 = []
peci_log_2 = []
recall_log_2 = []
fscore_log_2 = []
peci_log_3 = []
recall_log_3 = []
fscore_log_3 = []
for k in range(len(thres_log)):
    log_edge1,log_edge2,log_edge3 = LogPlot(2,thres_log[k])
    pecision_recall(peci_log_1,recall_log_1,fscore_log_1,im1gt,log_edge1)
    pecision_recall(peci_log_2,recall_log_2,fscore_log_2,im2gt,log_edge2)
    pecision_recall(peci_log_3,recall_log_3,fscore_log_3,im3gt,log_edge3)

plt.plot(recall_can_1, peci_can_1,label='Canny')
plt.plot(recall_sob_1, peci_sob_1,label='Sobel')
plt.plot(recall_log_1, peci_log_1,label='LoG')
plot('Church')
plt.plot(recall_can_2, peci_can_2,label='Canny')
plt.plot(recall_sob_2, peci_sob_2,label='Sobel')
plt.plot(recall_log_2, peci_log_2,label='LoG')
plot('Golf')
```

```
plt.plot(recall_can_3, peci_can_3,label='Canny')

plt.plot(recall_sob_3, peci_sob_3,label='Sobel')
plt.plot(recall_log_3, peci_log_3,label='LoG')
plot('Nuns')
```







Question 3

In order to widen the edge without blurring it, we convolved the GT image with a 3x3 filter, which is 0 at corner and 1 at four directions and 1 at the center, so that for each pixel, as long as there is a 1 that is 1 pixel away from it, it can be considered as 1. As a result, each edge pixel is expanded to 1 pixel on all four directions.

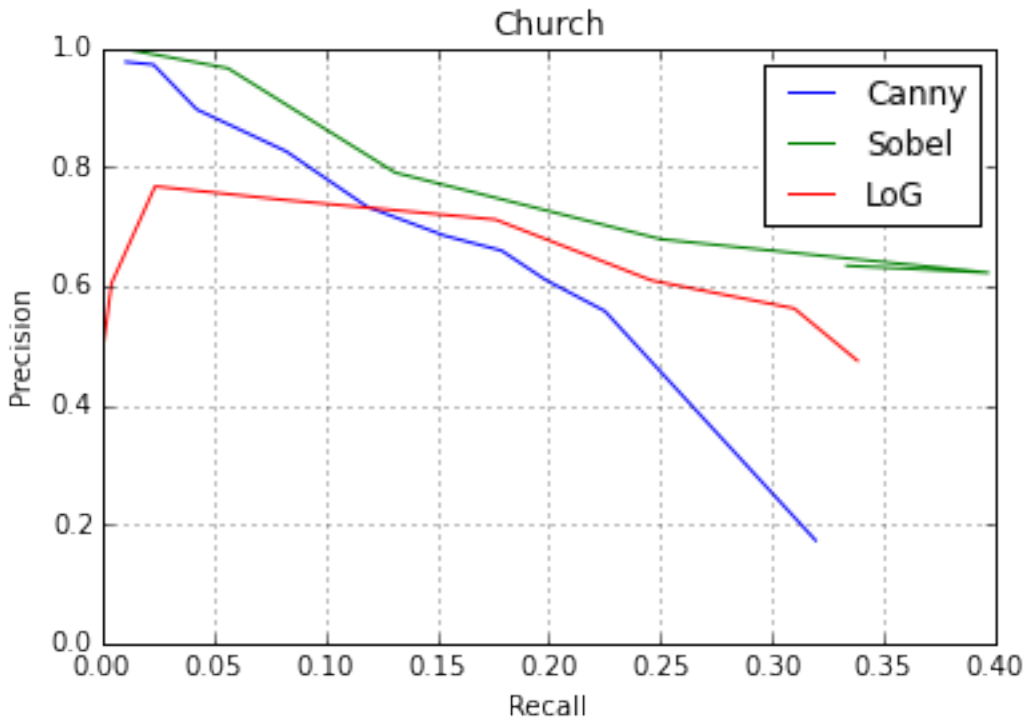
As shown in the graphs below, all images' PR curve are significantly closer to the upper right corner, which means that all three detectors are more accurate on each images.

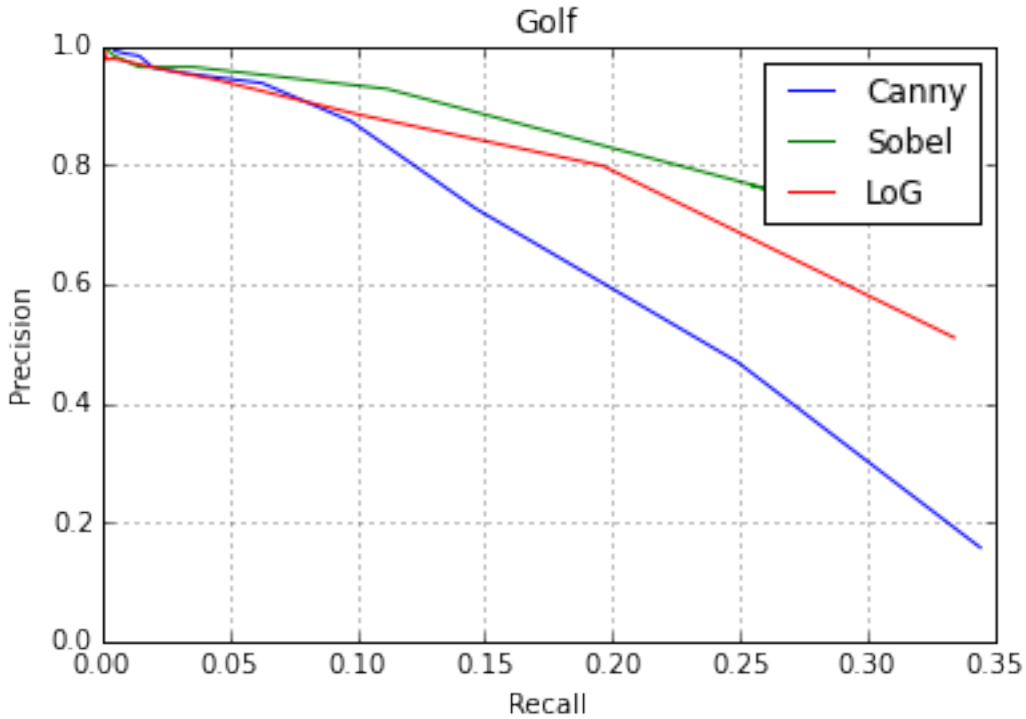
In [162]:

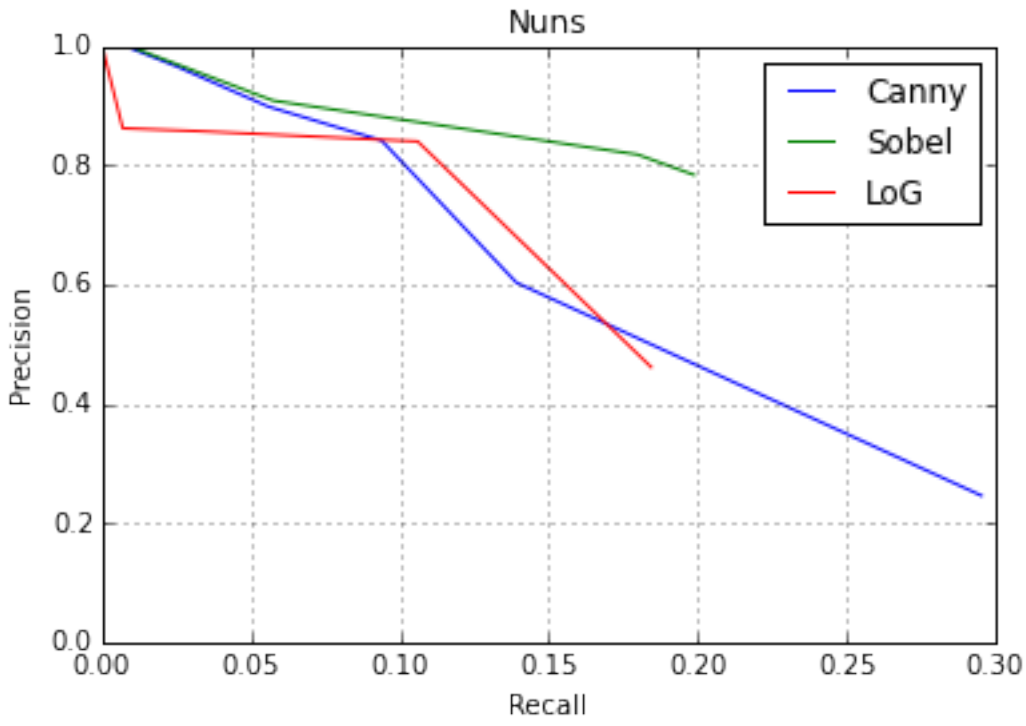
```
def wider(im):
    k = np.array([[0, 1, 0],
                  [1, 1, 1],
                  [0, 1, 0]])
    x = signal.convolve2d(im, k, boundary='symm', mode='same')
    w, h = x.shape[0], x.shape[1]
    for i in range(w):
        for j in range(h):
            if (x[i][j] > 0):
                x[i][j]=1
    return x

x = wider(im1gt)
y = wider(im2gt)
z = wider(im3gt)


#Canny
thres_can = np.arange(0,1000,100)
peci_can_1 = []
recall_can_1 = []
fscore_can_1 = []
```

```
peci_can_2 = []
recall_can_2 = []
fscore_can_2 = []
peci_can_3 = []
recall_can_3 = []
fscore_can_3 = []
for i in range(len(thres_can)):
    can_edge1,can_edge2,can_edge3 = cannythres(thres_can[i],thres_can[i])
    pecision_recall(peci_can_1,recall_can_1,fscore_can_1,x,can_edge1)
    pecision_recall(peci_can_2,recall_can_2,fscore_can_2,y,can_edge2)
    pecision_recall(peci_can_3,recall_can_3,fscore_can_3,z,can_edge3)

#Sobel
thres_sob = np.arange(0,1,0.1)
peci_sob_1 = []
recall_sob_1 = []
fscore_sob_1 = []
peci_sob_2 = []
recall_sob_2 = []
fscore_sob_2 = []
peci_sob_3 = []
recall_sob_3 = []
fscore_sob_3 = []
for j in range(len(thres_sob)):
    sob_edge1,sob_edge2,sob_edge3 = sobthres(thres_sob[j])
    pecision_recall(peci_sob_1,recall_sob_1,fscore_sob_1,x,sob_edge1)
    pecision_recall(peci_sob_2,recall_sob_2,fscore_sob_2,y,sob_edge2)
    pecision_recall(peci_sob_3,recall_sob_3,fscore_sob_3,z,sob_edge3)

#LoG
thres_log = np.arange(0,30,3)
peci_log_1 = []
recall_log_1 = []
fscore_log_1 = []
peci_log_2 = []
recall_log_2 = []
fscore_log_2 = []
peci_log_3 = []
recall_log_3 = []
fscore_log_3 = []
for k in range(len(thres_log)):
    log_edge1,log_edge2,log_edge3 = LogPlot(2,thres_log[k])
    pecision_recall(peci_log_1,recall_log_1,fscore_log_1,x,log_edge1)
    pecision_recall(peci_log_2,recall_log_2,fscore_log_2,y,log_edge2)
    pecision_recall(peci_log_3,recall_log_3,fscore_log_3,z,log_edge3)

plt.plot(recall_can_1, peci_can_1,label='Canny')
plt.plot(recall_sob_1, peci_sob_1,label='Sobel')
plt.plot(recall_log_1, peci_log_1,label='LoG')
plot('Church')
plt.plot(recall_can_2, peci_can_2,label='Canny')
plt.plot(recall_sob_2, peci_sob_2,label='Sobel')
plt.plot(recall_log_2, peci_log_2,label='LoG')
```

```
plot('Golf')

plt.plot(recall_can_3, peci_can_3,label='Canny')
plt.plot(recall_sob_3, peci_sob_3,label='Sobel')
plt.plot(recall_log_3, peci_log_3,label='LoG')
plot('Nuns')
```





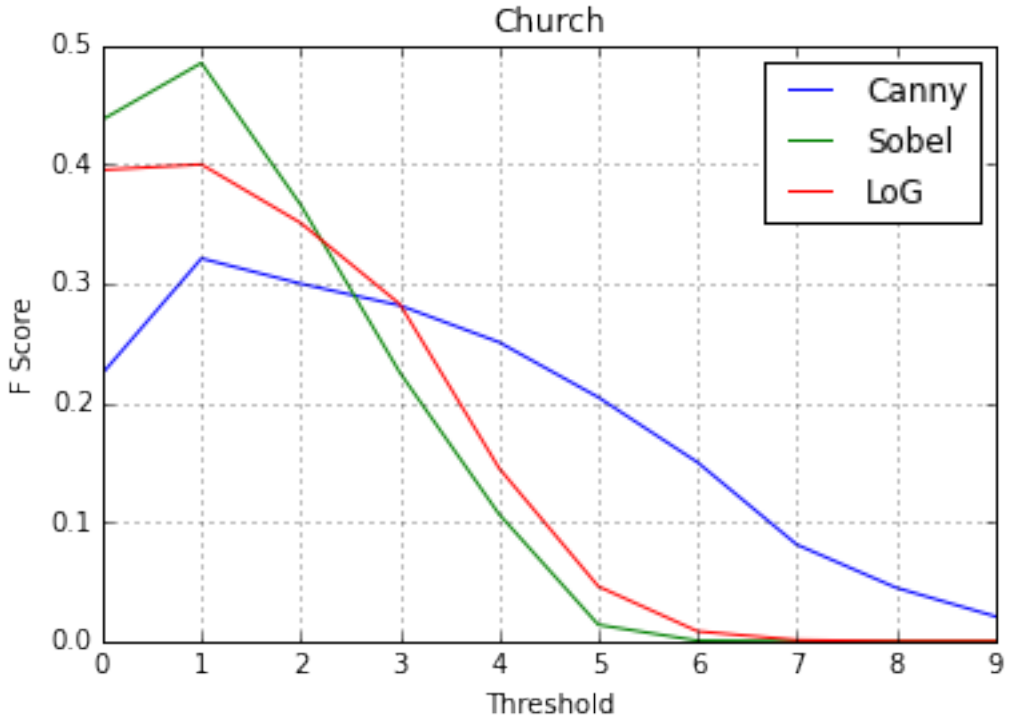


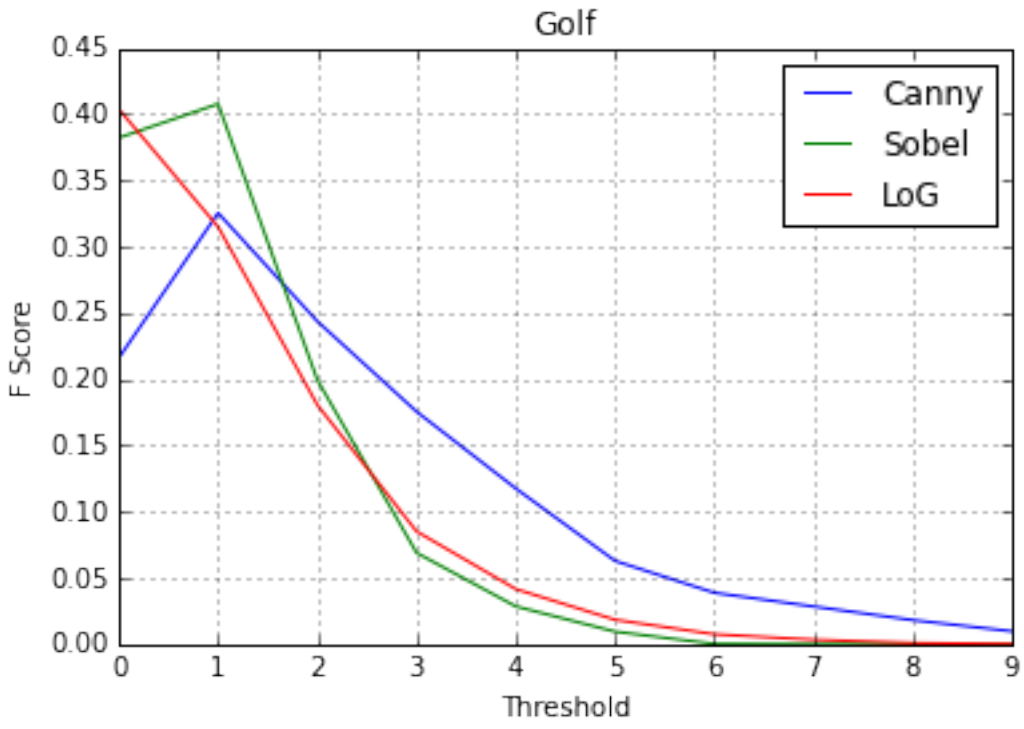
Question 4

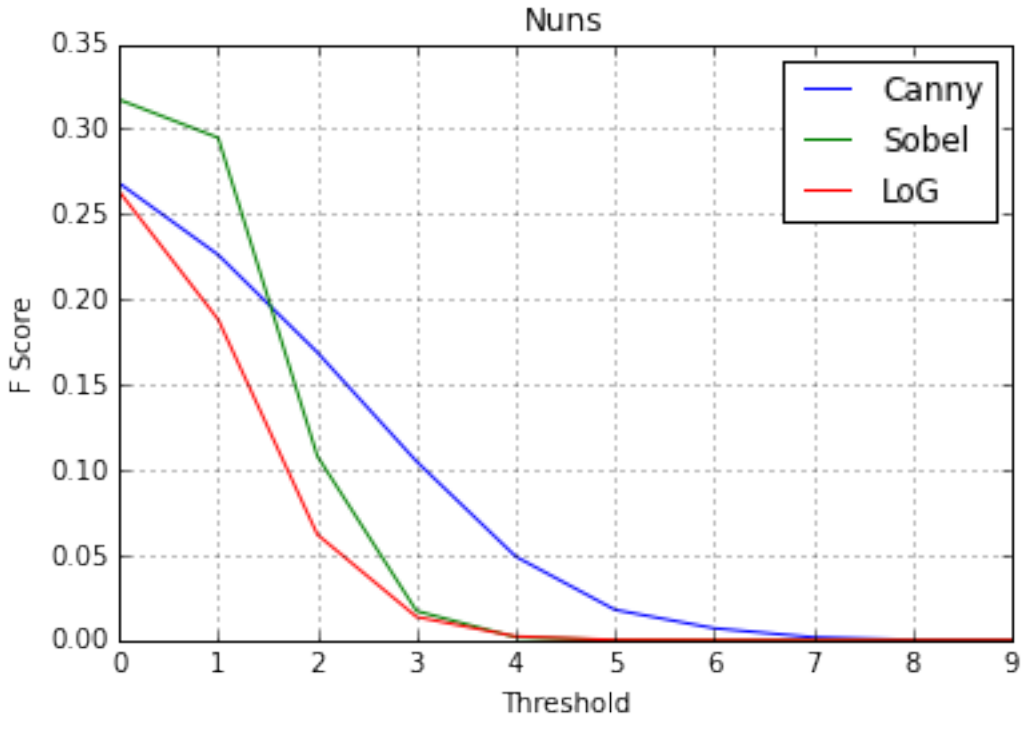
We plot the F-measure as the same way in question 2. The result shows that Sobel has the best result overall, because the highest point in each image is the Sobel filter. Even though Sobel doesn't have good performance as threshold goes up, it doesn't matteres to the result becasue the threshold can always be set as the highest point in each line.

In [163]:

```
def plotf(st):
    plt.xlabel('Threshold')
    plt.ylabel('F Score')
    plt.legend(loc=1)
    plt.title(st)
    plt.grid(True)
    plt.show()

x = np.arange(0,10)

plt.plot(x,fscore_can_1,label='Canny')
plt.plot(x,fscore_sob_1,label='Sobel')
plt.plot(x,fscore_log_1,label='LoG')
plotf('Church')
plt.plot(x,fscore_can_2, label='Canny')
plt.plot(x,fscore_sob_2, label='Sobel')
plt.plot(x,fscore_log_2, label='LoG')
plotf('Golf')
plt.plot(x,fscore_can_3, label='Canny')
plt.plot(x,fscore_sob_3, label='Sobel')
plt.plot(x,fscore_log_3, label='LoG')
plotf('Nuns')
```


Church
F Score
Threshold
Canny
Sobel
LoG
0.0
0.1
0.2
0.3
0.4
0.5
0
1
2
3
4
5
6
7
8
9


Golf
F Score
Threshold
Canny
Sobel
LoG
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0
1
2
3
4
5
6
7
8
9


Nuns
F Score
Threshold
Canny
Sobel
LoG
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
1
2
3
4
5
6
7
8
9