Guan, Hui Hua

In [156]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy.signal import convolve2d, freqz
from skimage.exposure import rescale_intensity
```

2D convolution

Below is my 2D convolution full mode function. We assume an odd length filter. We also note that the padding size is very important to yield a properly exceuted convolution. We can note the accuracy of my convolution function compared to scipy.signal.convolve2d function with mode=full in the subsequent cell.

In [157]:

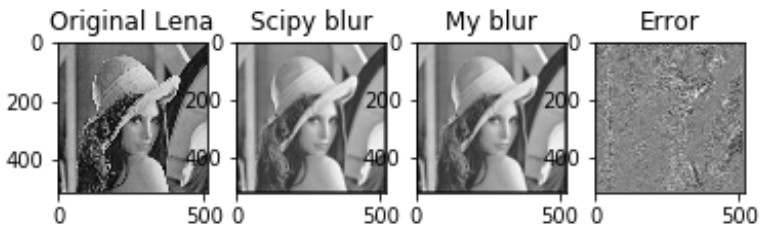
```
def conv2D(image,kernel):
    '''
    Inputs:
    image - takes an image array
    kernel - takes an odd length kernel array, assume kernel is square
    '''

    # We will first pad the image around with an appropriate amount of zero
s
    # We note that for a 3x3 kernel, in order to work properly, we need to
pad the image
    # each side by 2 zeros. This way, when sliding the kernel through the i
mage, the ninth
    # cell of the kernel will convolve with the first pixel of the image.
    # From there, we can deduce that the appropriate amount of padding is o
ne less than
    # the kernel length.
    padding = kernel.shape[0]-1
    img_pad= np.pad(img,padding,mode='constant',constant_values=0)

    # Next, we will create a array to hold the convolved image.
    # Note that the size is M+N-1, where M is the image size length, N is t
he kernel size length
    filtered_img = np.zeros((img.shape[0]+kernel.shape[0]-1,img.shape[1]+ker
nel.shape[1]-1))

    # To perform the 'convolve' operation, we would 'flip' and 'switch' the
kernel
    flipped_kernel = np.flipud(np.fliplr(kernel))


    scl= int((kernel.shape[0]-1)/2) # 3->1, 5->2, 7->3

    for row in range(scl,img_pad.shape[0]-scl):
        for col in range(scl,img_pad.shape[1]-scl):
```

```
            # As we slide through the image, we will do elementwise
multiplication and then add
            # each region up
            region = img_pad[row-scl:row+scl+1,col-scl:col+scl+1]
            filtered_img[row-scl,col-scl] =  np.sum(np.multiply(flipped_kerr
el,region))
    return filtered_img
```

Let us do a quick check on our convolution 2D function against scipy's convolve2d. We can see from the results that the error is neglible. This means my conv2D is comparable with the scipy's convolve2d function.

In [158]:

```
# We can use an averaging filter
n=9
sev=np.ones((n,n))*(1/(n*n))

# Read in image of Lena
img = cv2.imread('lena512gray.png',0)
img=img.astype('uint8')

# scipy's convolve
s=convolve2d(img, sev, mode='full')
# my convolve
ss=conv2D(img, sev)

# Show the results
plt.subplot(141)
plt.title('Original Lena')
plt.imshow(img,cmap=plt.cm.gray )

plt.subplot(142)
plt.title('Scipy blur')
plt.imshow(s, cmap=plt.cm.gray)

plt.subplot(143)
plt.imshow(ss, cmap=plt.cm.gray)
plt.title('My blur')

plt.subplot(144)
plt.imshow(s-ss, cmap=plt.cm.gray)
plt.title('Error')
plt.show()

print('The error between scipy.signal.convolve with full mode \
and my conv2D with full mode is '+ str(np.max(np.abs(s-ss))))
```




```
The error between scipy.signal.convolve with full mode and my conv2D with f
ull mode is 2.84217094304e-13
```

Below is a function we can use to normalize the pixel values from 0 to 255

In [304]:

```
def histo_norm(img):

    # Calculate the histogram and corresponding bins

    hist,_ = np.histogram(img.flatten(),256,[0,256])
    # Calculate the cdf and normalize the values to 0-255
    cdf = hist.cumsum()
    cdf_normalized = cdf * 255/ cdf[-1]

    # Replace the vales with normalized cdf values
    img = cdf_normalized[img.astype('uint8')]
    return img
```

We will read in an image into a greyscale image and then cast into uint8 type before doing any image processing

In [268]:

```
img = cv2.imread('lena512gray.png',0)
```

Below is a function we can use to display the images nicely in row and column format

In [161]:

```
def grid_display(list_of_images, list_of_titles=[], no_of_columns=3,
figsize=(20,20)):

    fig = plt.figure(figsize=figsize)
    column = 0
    for i in range(len(list_of_images)):
        column += 1
        #  check for end of column and create a new figure
        if column == no_of_columns+1:
            fig = plt.figure(figsize=figsize)
            column = 1
        fig.add_subplot(1, no_of_columns, column)
        if 'Mask' in list_of_titles[i]:
            plt.imshow(list_of_images[i], cmap='gray')

        else:
            plt.imshow(list_of_images[i], cmap='gray')

        plt.axis('off')
        if len(list_of_titles) >= len(list_of_images):
            plt.title(list_of_titles[i])
    plt.show()
```

This is the main function, which will read in an image and allow a user to specify the filter size and filter coefficients. It then call the convolution function to filter the image. The program will show the original image and the filtered image and save the filtered image into a file after properly normalizing the pixel value range to 0 to 255. The program will also calculate and display the magnitude of Fourier transform of the original and filtered image, as well as frequency response of the filter.

In [305]:

```
def main(img_path, filt_size, filt_coeff):

    # Read in image
    img = cv2.imread(img_path,0)
    #img=img.astype('uint8')

    # Make kernel from the filter coefficients and filter size
    filt_coeff=np.array(filt_coeff)
    kernel=np.zeros((1,filt_size*filt_size))
    kernel[:len(filt_coeff)]=filt_coeff
    kernel = kernel.reshape((filt_size, filt_size))
    print(kernel)

    # Filter the image with conv2D
    filt_img = conv2D(img, kernel)


    # Normalised pixel value from 0 to 255
    filt_img=histo_norm(filt_img)

    # Save filtered image into file
    cv2.imwrite('filtered_image.jpg', filt_img)

    # Calculate and display magnitude of FT of original image
    FT = np.abs(np.fft.fft2(img))
    SFT=np.log(np.fft.fftshift(FT)+1)


    # Calculate and display magnitude of FT of filtered image
    filtimage_FT = np.abs(np.fft.fft2(filt_img))
    filtimage_SFT=np.log(np.fft.fftshift(filtimage_FT)+1)

    # Frequency response of filter
    filt_FT = np.abs(np.fft.fft2(kernel))
    filt_SFT=np.log(np.fft.fftshift(filt_FT)+1)


    grid_display([img,filt_img,SFT,filtimage_SFT , filt_SFT], ['Original',
'Filtered', 'Original FT', 'Filtered FT', 'Filter Frequency Response'],\
                 5, (20,20))
    return FT, filtimage_FT, filt_FT
```

Below are the three kernels we will use: a 3x3 Gaussian filter, Laplacian operator(an edge detection) filter, and a sharpening filter. We list the coefficients here.

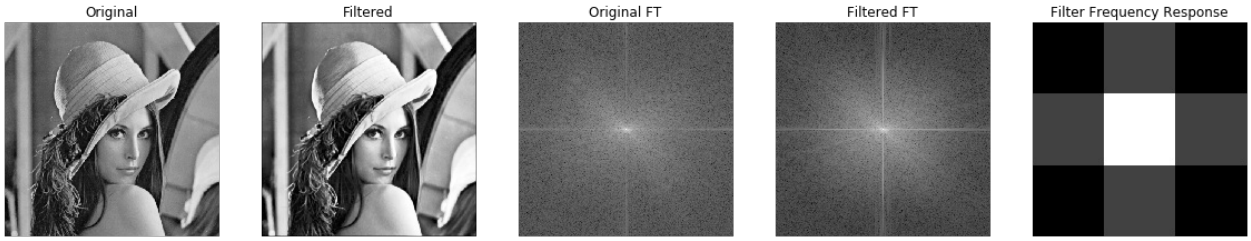
In [163]:

```
filter_H1 =[1/16, 2/16, 1/16,2/16,4/16,2/16,1/16,2/16,1/16]
filter_H2 = [-1, -1, -1,-1,8,-1,-1,-1,-1]
filter_H3 = [0, -1, 0,-1,5,-1,0,-1,0]
```

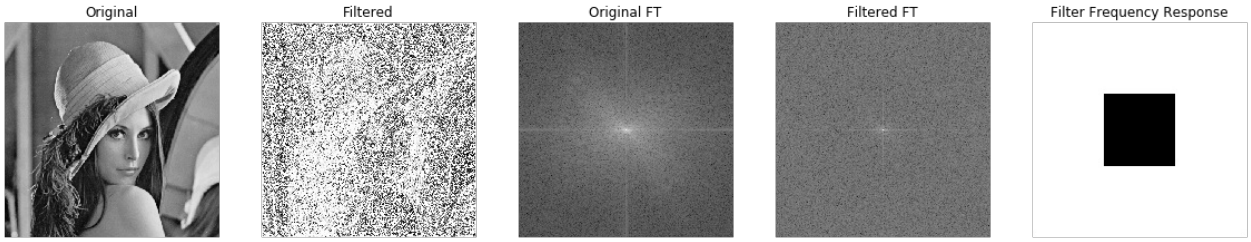
In [306]:

```
ft1,filtimgft1,filtft1=main('lena512gray.png', 3,filter_H1)
ft2,filtimgft2,filtft2=main('lena512gray.png', 3,filter_H2)
ft3,filtimgft3,filtft3 = main('lena512gray.png', 3,filter_H3)
```

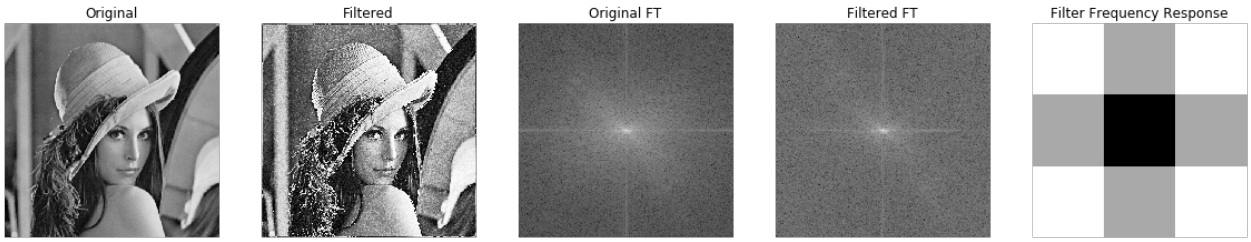
```
[[ 0.0625  0.125   0.0625]
 [ 0.125   0.25    0.125 ]
 [ 0.0625  0.125   0.0625]]
```




```
[[-1. -1. -1.]
 [-1.  8. -1.]
 [-1. -1. -1.]]
```




```
[[ 0. -1.  0.]
 [-1.  5. -1.]
 [ 0. -1.  0.]]
```



Filter discussion comments

For the 3x3 Gaussian filter, we see in the frequency response, more emphasis or 'weight' is placed on the central pixels. This would constrast the mean filter's uniformly weighted average. Because of the special weighted property seen in the frequency response, a Gaussian provides gentler smoothing and preserves edges better than a similarly sized mean filter. As a smoothing operator, it can be observed in the spatial domain that the image is more smoothed from the center and out. It is also noted that it removes high frequency components from the image.

For the 3x3 edge detection filter, we can tell from its frequency response, it detects a rapid change, which is a characteristic of an edge - a rapid change in the image intensity function. You can see a faint outline enhanced around Lena in the spatial domain. As a result of the LoG filter, any edges in the original image appear much sharper and more constrasted.

For the 3x3 sharpening filter, we can see that this high-pass filter can be used to make an image appear sharper. These filters emphasize fine details in the image. This is exactly the opposite of the low-pass Gaussian filter seen above. Notice in the frequency domain, the filter shows an exactly opposite of weight emphasis as the Gaussian filter above. Note the minus signs for the adjacent pixels and as seen in the frequency response. Thus if there is one pixel is brighter than its immediate neighbors, it gets boosted. As a result, in the spatial domain, we will see a sharpened image.

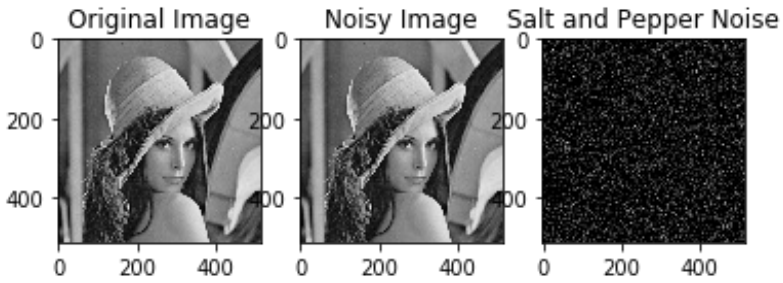
neighbors, it gets boosted. As a result, in the spatial domain, we will see a sharpened image.

Image noise removal using averaging and Gaussian filters

We create a noisy image, by adding zero mean Gaussian random noise to your image using numpy. Often times, the noise is not easily seen with our raw eyes. We can note this noise by displaying the difference between the original image and the noisy image. We should expect to see a 'salt and pepper' image, which represents the noise.

In [165]:

```
mean = 0
sigma = 0.1
noise = np.random.normal(mean, sigma, img.shape)

noisy_img = img + np.clip(noise,0,255)
noisy_img = np.clip(noisy_img, 0, 255)


plt.figure()
plt.subplot(131)
plt.imshow(img,cmap=plt.cm.gray)
plt.title('Original Image')
plt.subplot(132)
plt.title('Noisy Image')
plt.imshow(noisy_img,cmap=plt.cm.gray)
plt.subplot(133)
plt.title('Salt and Pepper Noise')
plt.imshow(noisy_img-img,cmap=plt.cm.gray)
plt.show()
```

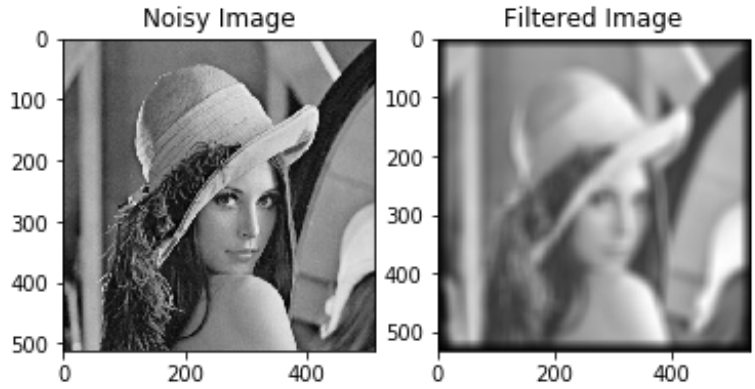


We apply the previous program to filter the noise-added image using an average filter of size n x n, where n=odd. In this case, n=21. We choose a rather large size so it is more easy to see the blurring effect.

In [166]:

```
n = 21
avg_filt = np.ones((n,n))/(n*n)

plt.figure()
plt.subplot(121)
plt.title('Noisy Image')
plt.imshow(noisy_img, cmap=plt.cm.gray)

output=conv2D(noisy_img, avg_filt)
```

```
plt.subplot(122)
plt.title('Filtered Image')
plt.imshow(output, cmap=plt.cm.gray)

plt.show()
```



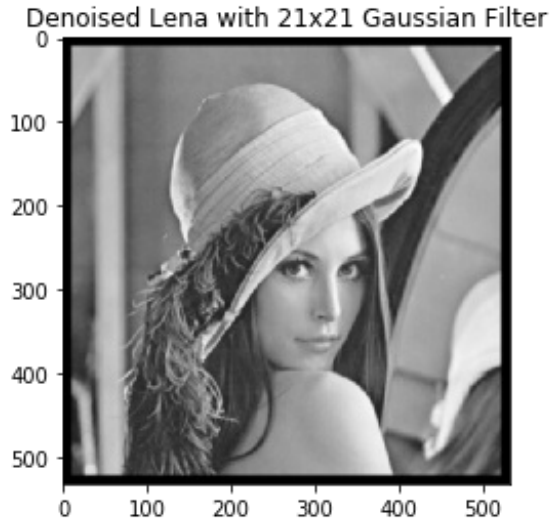
Now, we apply your previous program to filter the noise-added image using a Gaussian filter of size n x n. First, let us define a helper function to find the Gaussian kernel of a nxn filter.

In [167]:

```
def gkern(n, sig=1):
    ''' Adapted from last semester machine learning class '''
    ax = np.arange(-n // 2 + 1., n // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)

    kernel = np.exp(-(xx**2 + yy**2) / (2. * sig**2))

    return kernel / np.sum(kernel)
```

In [168]:

```
n = 21
gau_filt_21 = gkern(21)
output=conv2D(noisy_img, gau_filt_21)
plt.imshow(output, cmap=plt.cm.gray)
plt.title('Denoised Lena with 21x21 Gaussian Filter')
plt.show()
```



Try two different noise levels (0.01 and 0.1) and for each noise level different filter sizes (ex: 5x5 to

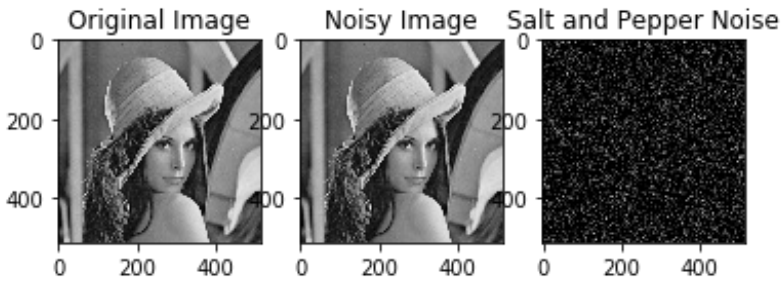
9x9 in step size of 2).

In [209]:

```
img = cv2.imread('lena512gray.png',0)
img=img.astype('uint8')
```

In [210]:

```
# Make a noisy image with noise level 0.01
mean = 0
sigma001 = 0.01
noise001 = np.random.normal(mean, sigma001, img.shape)
noisy_img001 = img + np.clip(noise001,0,255)
noisy_img_001 = np.clip(noisy_img001, 0, 255)

plt.figure()
plt.subplot(131)
plt.imshow(img,cmap=plt.cm.gray)
plt.title('Original Image')
plt.subplot(132)
plt.title('Noisy Image')
plt.imshow(noisy_img,cmap=plt.cm.gray)
plt.subplot(133)
plt.title('Salt and Pepper Noise')
plt.imshow(noisy_img_001-img,cmap=plt.cm.gray)


plt.show()
```



In [307]:

```
n5 = 5

# Denoise the noisy image
gaus_filter_5=float(1)/256*np.array([[1,4,6,4,1],[4,16,24,16,4],[6,24,36,24,
6],[4,16,26,16,4],[1,4,6,4,1]])
out5_001=conv2D(noisy_img_001, gaus_filter_5)
```

In [308]:

```
n7 = 7

# Denoise the noisy image
gaus_filter_7=gkern(n7,1)
out7_001=conv2D(noisy_img_001, gaus_filter_7)
```

In [309]:

```
n9 = 9
```

```
# Denoise the noisy image
gaus_filter_9=gkern(n9,1)
out9_001=conv2D(noisy_img_001, gaus_filter_9)
```

Now, let us change the noise level to 0.1.

In [310]:

```
# Make a noisy image with noise level 0.1
mean = 0
sigma01 = 0.1
noise01 = np.random.normal(mean, sigma01, img.shape)
noisy_img01 = img + np.clip(noise01,0,255)
noisy_img_01 = np.clip(noisy_img01, 0, 255)
```

In [311]:

```
n5 = 5

# Denoise the noisy image
gaus_filter_5=float(1)/256*np.array([[1,4,6,4,1],[4,16,24,16,4],[6,24,36,24,
6],[4,16,26,16,4],[1,4,6,4,1]])
out5_01=conv2D(noisy_img_01, gaus_filter_5)
```

In [312]:

```
n7 = 7

# Denoise the noisy image
gaus_filter_7=gkern(n7,1)
out7_01=conv2D(noisy_img_01, gaus_filter_7)
```

In [313]:

```
n9 = 9

# Denoise the noisy image
gaus_filter_9=gkern(n9,1)
out9_01=conv2D(noisy_img_01, gaus_filter_9)
```

Below is the original, noise-added, and filtered images for each combination of noise level and filter size. We then comment on for each noise level, which filter size is best for each filter and how does the two filters compare in their noise removal capability.

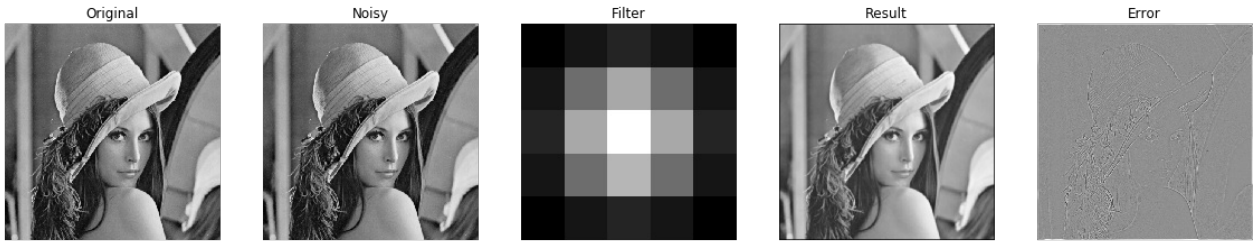
For the noisy level of 0.01, below is the results for a 5x5, 7x7, and 9x9 Gaussian filter.

In [314]:

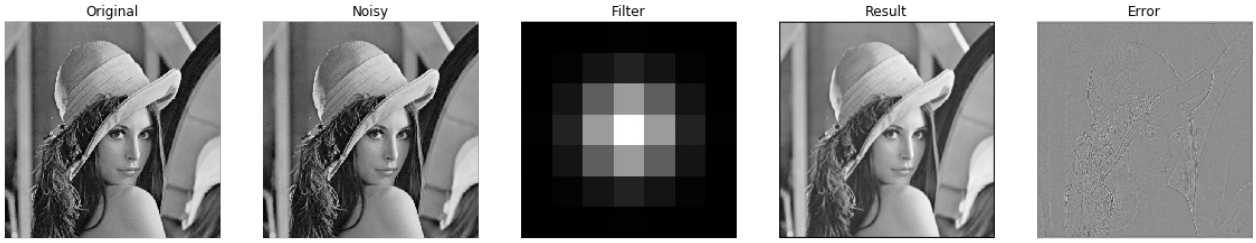
```
print('5x5')
padded5=np.pad(img,2,mode='constant',constant_values=0 )
print(np.average(np.abs(out5_001-padded5)))
grid_display([img,noisy_img_001,gaus_filter_5,out5_001,out5_001-padded5 ],
['Original', 'Noisy','Filter', 'Result', 'Error'],5)

print('7x7')
padded7=np.pad(img,3,mode='constant',constant_values=0 )
print(np.average(np.abs(out7_001-padded7)))
```

```
grid_display([img,noisy_img_001,gaus_filter_7,out7_001,out7_001-padded7], [
'Original', 'Noisy','Filter', 'Result','Error'],5)

print('9x9')
padded9=np.pad(img,4,mode='constant',constant_values=0 )
print(np.average(np.abs(out9_001-padded9)))
grid_display([img,noisy_img_001,gaus_filter_9,out9_001,out9_001-padded9], [
'Original', 'Noisy', 'Filter', 'Result','Error'],5)
```

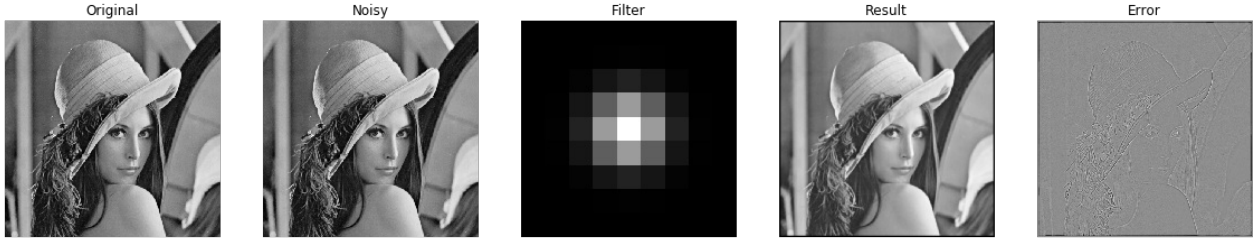
```
5x5
4.19652347623
```




```
7x7
3.93442486203
```




```
9x9
3.90670160212
```



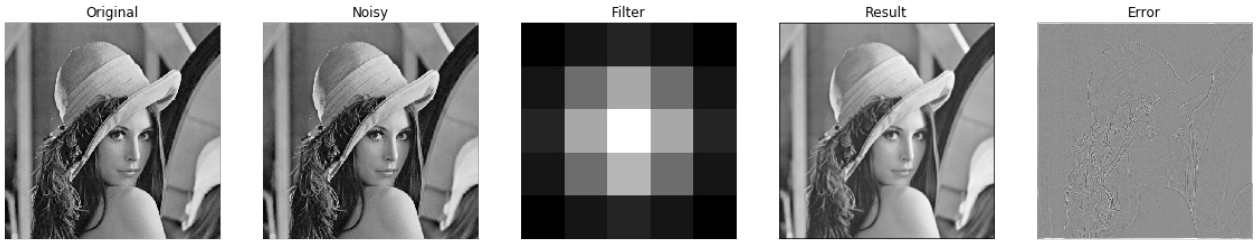
For the noisy level of 0.1, below is the results for a 5x5, 7x7, and 9x9 Gaussian filter.

In [315]:

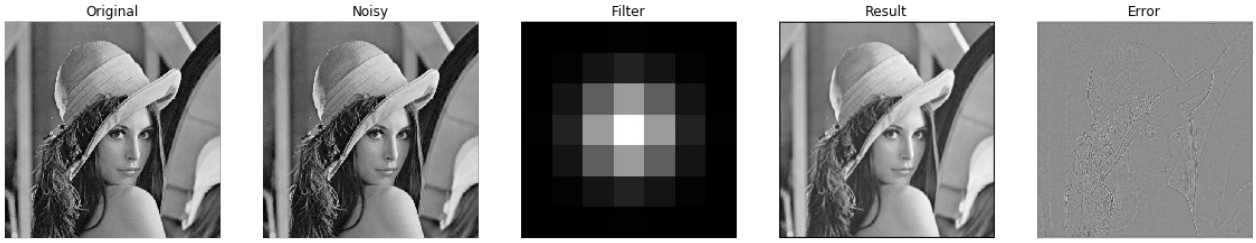
```
print('5x5')
print(np.average(np.abs(out5_01-padded5)))
grid_display([img,noisy_img_01,gaus_filter_5,out5_01,out5_01-padded5], ['Or
iginal', 'Noisy', 'Filter', 'Result', 'Error'],\
                5)

print('7x7')
print(np.average(np.abs(out7_01-padded7)))
grid_display([img,noisy_img_01,gaus_filter_7,out7_01, out7_01-padded7], ['O
riginal', 'Noisy','Filter', 'Result','Error'],\
                5)

print('9x9')
print(np.average(np.abs(out9_01-padded9)))
```

```
grid_display([img,noisy_img_01,gaus_filter_9,out9_01,out9_01-padded9], ['Or
iginal', 'Noisy','Filter', 'Result','Error'],\
                5)
```

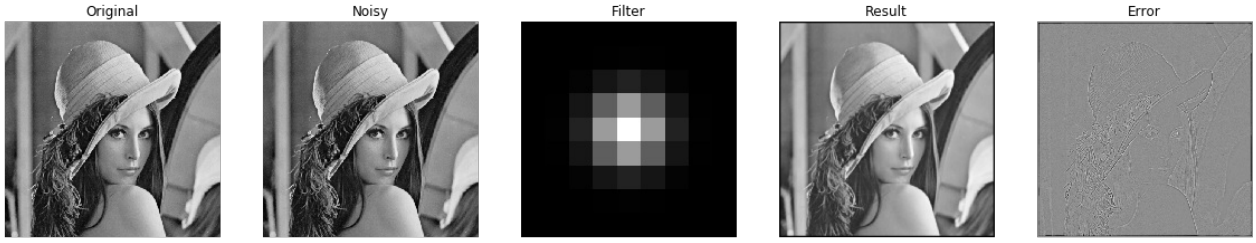
```
5x5
4.19652347623
```




```
7x7
3.93442486203
```




```
9x9
3.90670160212
```



Noise removal capability comments

For the 0.01 noise level, the 5x5 is the best because with a lower noise level, using an overly large kernel may rid fine detail. However, in here, the error is lowest and the image appears more smooth with the 9x9 kernel.

For the 0.1 noise level, 9x9 is the best because a large enough kernel must be used to overcome noise. It is suggested that the kernel size is at least 5 times the sigma of the filter. This can be confirmed because the error (e=3.90670160212) is lowest with the 9x9 kernel.

It is noted that the kernel size must increase with increasing sigma to maintain the Gaussian nature of the filter.