ex1

May 15, 2019

1 TP1 Dimitris Proios

1.1 Requirements install

1.2 Exercise 1 -- Noise and Metrics

1.2.1 1.a) Write a function that determines the Mean Squared Error (MSE) between two images x and y.

In MSE, the lower the error, the more "similar" the two images are The two images must have the same dimension.

- MSE definition

$$MSE = \frac{1}{NM} \sum_{i=1}^{N} \sum_{i=1}^{M} (x[i,j] - y[i,j])^2$$

1.b As we see below we converted successfully the image to grayscale.

1.c The mse is 17842.766630867263, which I believe is correct since the mse is not immune to different range.

[13]:
```
import cv2
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
# 1.a
def mse(imageA, imageB) -> float:
        err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
        err /= float(imageA.shape[0] * imageA.shape[1])
        return err

def readTifGrayScale(path):
    img = cv2.imread(path, -1)
    return img

def showTifGrayScale(img, title = ""):
    plt.imshow(img, cmap = 'gray', interpolation = 'bicubic')
    plt.xticks([]), plt.yticks([])  # to hide tick values on X and Y axis
    plt.title(title)
    plt.show()
```

```
# imA = Image.open('./data/hdr_images/img01.tif')
# img1 = readTifGrayScale('./data/hdr_images/img01.tif')
# img2 = readTifGrayScale('./data/hdr_images/img02.tif')
# showTifGrayScale(img1)
# showTifGrayScale(img2)

# 1.b)  Read in a new copy of the image cameraman.tif, keep it in its original␣
 ↪datatype and range, i.e. uint8 and {0..255}.
imgCam = readTifGrayScale('./data/cameraman.tif')
showTifGrayScale(imgCam )


# 1.c)  Now read in a second copy of the image cameraman.tif but map it to␣
 ↪double and {0..1}. See Matlab im2double. Compare the two images using the MSE.␣
 ↪Can you explain the result?

imgCam_double = np.array(imgCam).astype(np.float32)
imgCam_double = np.interp(
    imgCam_double,
    (imgCam_double.min(), imgCam_double.max()),
    (0, 1)
)
print(mse(imgCam_double, imgCam ))
```

```
17842.766630867263
```

ex2

May 15, 2019

0.1 Exercise 2

0.1.1 2.a) Refractor the PNSR definition such that the PSNR is expressed as a function of the noise variance $_z ^2 $. You may assume that $\frac{2}{z} = MSE(x, y)$

- PSNR original definition

  (1) $PSNR = 10 \log_{10} \frac{a^2}{MSE(x,y)}$

  (2) $MSE(x, y) = \sigma_z^2$

  (3) (2) $PSNR = 10 \log_{10} \frac{a^2}{\sigma_z^2}$

  $PSNR = 20 \log_{10} a - 10 \log_{10} \sigma_z^2$

  $PSNR - 20 \log_{10} a = 10 \log_{10} \sigma_z^2$

  $\frac{PSNR - 20 \log_{10} a}{10} = \log_{10} \sigma_z^2$

  (3) $10^{\frac{PSNR - 20 \log_{10} a}{10}} = \sigma_z^2$

  This relationship above is used in 2.b.

0.1.2 2.b) Add Gaussian noise to an image such that the PSNR ratio with the original image is 10dB, 20dB, 30dB and 40dB. Use randn, not imnoise.

- Noise function: $Z_i = N(\mu, \sigma^2)$

[23]:
```
#2b
import cv2
import numpy as np
import math
import matplotlib.pyplot as plt

def showTifGrayScale(img, title = ""):
    plt.imshow(img, cmap = 'gray')
    plt.xticks([]), plt.yticks([])  # to hide tick values on X and Y axis
    plt.title(title)
    plt.show()

PSNRs = [10, 20, 30, 40]
```

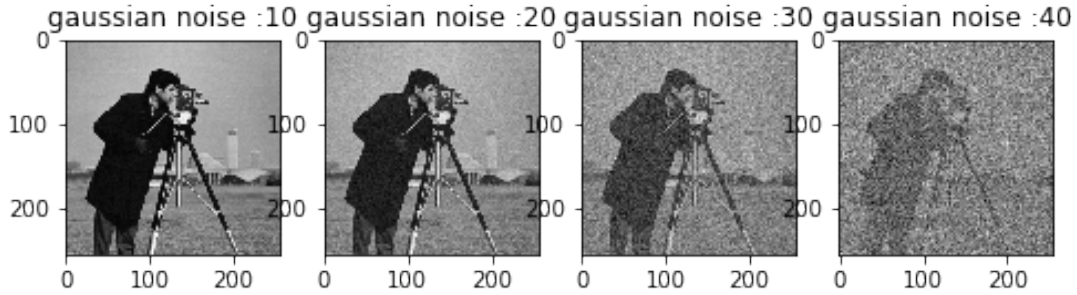
```
getVarianceForPSNR = lambda db: 10**((db-20 * math.log(255, 10))/10)
varianceList = [(db, getVarianceForPSNR(db)) for db in PSNRs ]

imgCam = cv2.imread('./data/cameraman.tif', -1)
imgCam_double = np.array(imgCam).astype("float32")
imgCam_double = np.interp(
    imgCam_double,
    (imgCam_double.min(), imgCam_double.max()),
    (0, 1)
)

showTifGrayScale(imgCam, "original")
```

0.1.3 2.c) Show the noisy images on the screen. How do they look?

They look bad the biggest the variance of the gaussian noise distribution.

[20]:
```
def experiment(db, variance, img):
    noisedImgArray, gaussNoiseMatrix = gaussianNoise(img, variance)
    return noisedImgArray

noised_images = [
    ("gaussian noise :" + str(db), experiment(db,var, imgCam_double ))
    for db,var in iter(varianceList)
]
```

```
def showImages(images, titles):
    fig=plt.figure(figsize=(8, 8))
    columns = 4
    rows = round(len(images)/columns)
    k=1
    for i in range(1, columns*rows +1):
        fig.add_subplot(rows, columns, i)
        img = images[k-1]
        plt.title(titles[k-1])
        plt.imshow(img, cmap = 'gray')
        k+= 1
    plt.show()

showImages(
    [im[1] for im in noised_images],
    [im[0] for im in noised_images],
)
```

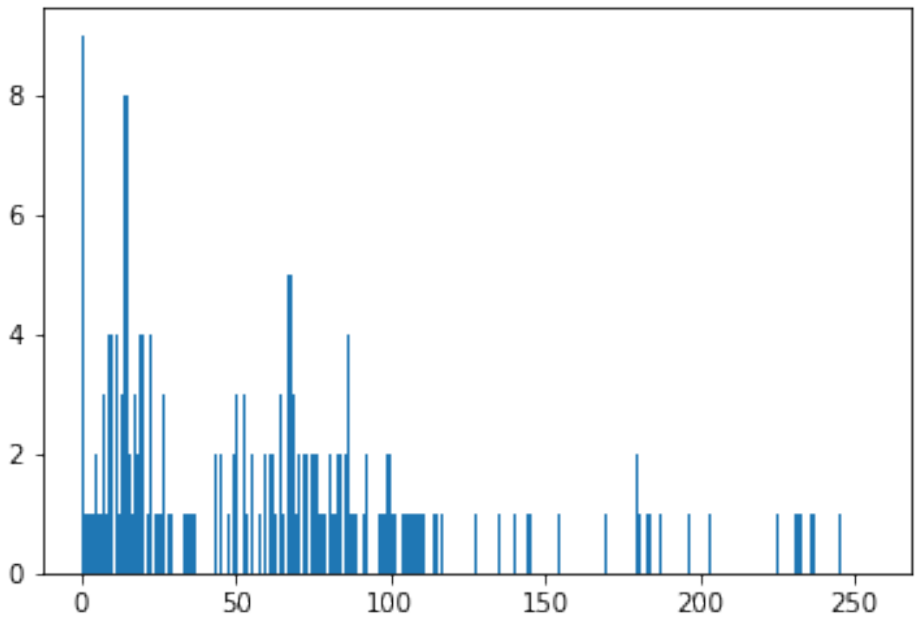


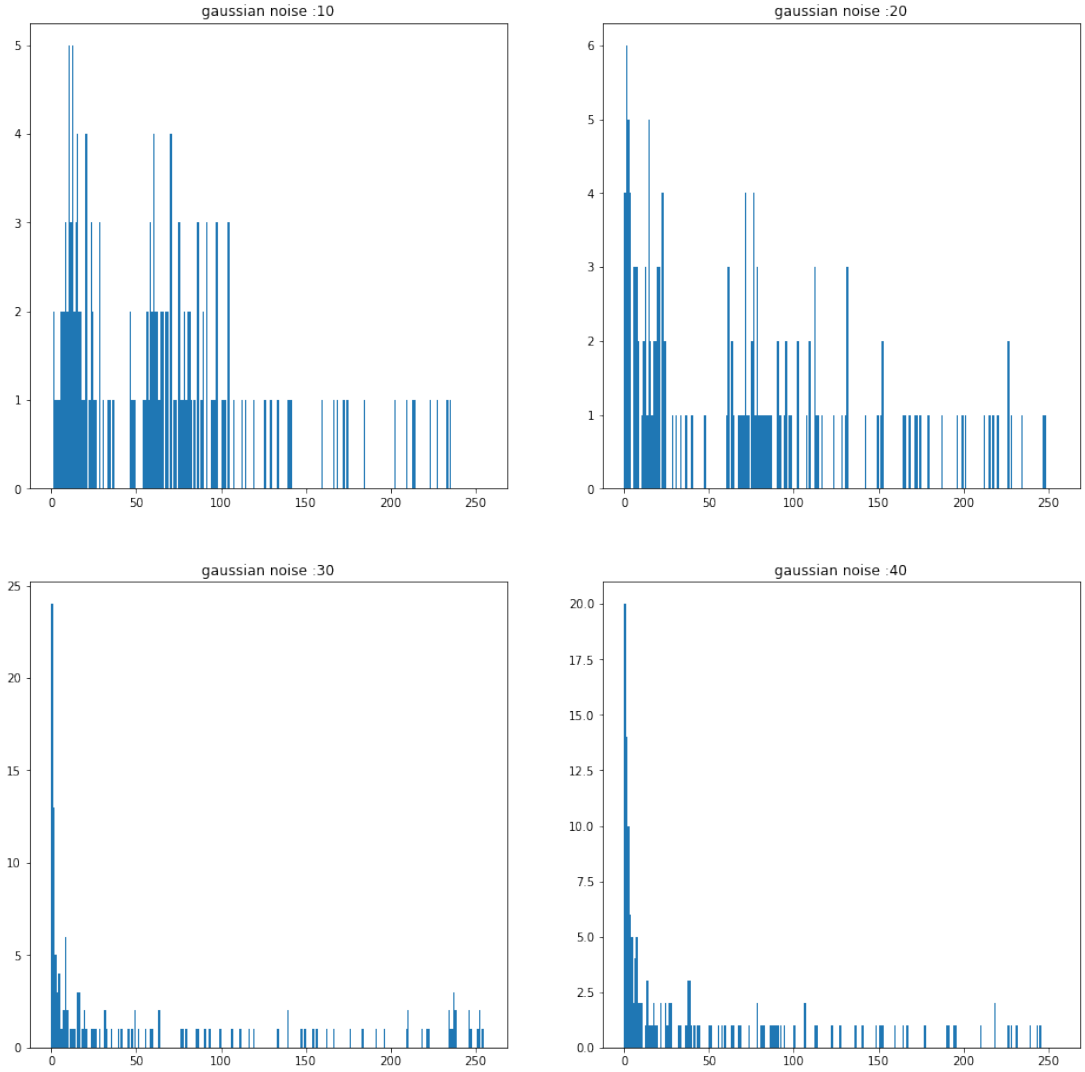
0.1.4 2.d) Show the histograms for these noisy images, can you explain what you see?

The more noisy the image the higher the difference with the original graph. We see the count for every pixel value of the image. :

[24]:
```
### 2.d) Show the histograms for these noisy images, can you explain what you␣
 ↪see?
def convertBack(im):
    imconv= np.interp(
        im,
        (im.min(), im.max()),
        (0, 255)
    )
    imconv = imconv.astype("uint8")
    return imconv

def hist(img, title=""):
```

```
    imconv = convertBack(img)
    hist , bins = np.histogram(imconv.ravel(),256,[0,256])
    plt.hist(hist, bins= bins)
    plt.title(title )
    plt.show()

hist(imgCam_double, "Original image")

titles = [im[0]  for im in noised_images]
images  = [im[1]  for im in noised_images]
fig=plt.figure(figsize=(16, 16))
columns = 2
rows = 2
k=1
for i in range(1, columns*rows +1):
        fig.add_subplot(rows, columns, i)
        img = images[k-1]
        plt.title(titles[k-1])
        imconv = convertBack(img)
        hist , bins = np.histogram(imconv.ravel(),256,[0,256])
        plt.hist(hist, bins= bins)# plt.imshow(img, cmap = 'gray')
        k+= 1
plt.show()
```





0.1.5 2.e1) Add salt & pepper Noise to an image until the PSNR ratio between the original and the noisy image is 40 dB.

The noise and pepper function is imlemented below

0.1.6 2.e.2) Visually compare it to the 40dB noisy image to which Gaussian noise was added. What can you conclude?

With salt and pepper noise function the picture is barely modified at 40db. While with gaussia noise it is geavily modified.

To validate that the salt and pper nois is coorectly implemented I trie to augmenta the noise heavily.

[26]:
```
### 2.e.1) Add salt & pepper Noise to an image until the PSNR ratio between the␣
 ↪original and the noisy image is 40 dB.
import copy

def mse(imageA, imageB) -> float:
        err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
        err /= float(imageA.shape[0] * imageA.shape[1])
        return err


def gaussianNoise(image, var, mean = 0):
      row,col = image.shape
      sigma = var**0.5
      randomGaus = np.random.normal(mean,sigma,(row,col))
      gaussNoiseMatrix = randomGaus.reshape(row,col)
      noisy = image + gaussNoiseMatrix
      return noisy, gaussNoiseMatrix

def psnr(mseres):
    PIXEL_MAX_SQUARE = 1
    mse_square = math.sqrt(mseres )
    part1 =  math.log10(PIXEL_MAX_SQUARE / mse_square)
    return 20 * part1

imgCam = cv2.imread('./data/cameraman.tif', -1)
imgCam_double = np.array(imgCam).astype("float32")
imgCam_double = np.interp(
    imgCam_double,
    (imgCam_double.min(), imgCam_double.max()),
    (0, 1)
)

def get_img_with_pepper_salt_noise(img, p, q):
    im = copy.deepcopy(img)
    randnums = np.random.rand(256,256)
    im[np.logical_and(randnums > p, randnums < q)] = 1
    im[ randnums <= p] = 0
    return im

pepperedImg = get_img_with_pepper_salt_noise(imgCam_double, p=0.00015, q=0.0003)
mseNum  = mse(imgCam_double,pepperedImg)
print("mse: ", mseNum)
print("psnr: ", psnr(mseNum))
showTifGrayScale(noised_images[3][1], "gaussian image")
showTifGrayScale(pepperedImg, "Peppered image 40 DB psnr")
showTifGrayScale(imgCam_double, "Original image")
```

```
### 2.e.2) Visually compare it to the 40dB noisy image to which Gaussian noise␣
 ↪was added. What can you conclude?
pepperedImg = get_img_with_pepper_salt_noise(imgCam_double, p=0.15, q=0.3)
mseNum  = mse(imgCam_double,pepperedImg)
print("high noise image mse: ", mseNum)
print("high noise image psnr: ", psnr(mseNum))
showTifGrayScale(pepperedImg, "Peppered image")
```

```
mse:  0.0001810497329573739
psnr:  37.42202111417113
high noise image mse:  0.0946661067722857
high noise image psnr:  10.238054832616301
```

Peppered image 40 DB psnr

Original image


Peppered image

ex3

May 15, 2019

[ ]:
```
## Low rank approximation via SVD
```

0.0.1 Exercise 3. Read the image peppers.png and convert it to grayscale. Perform its low-rank approximation for k = 1, ..., n.

0.0.2 Plot the dependence between the k and MSE of k-rank approximation version of original image. Make a conclusion.

0.0.3 Conclusion:

As we see in the experiment below each K component adds more of the "information" on the image. This is consistent with the theory since the compoents include variance of the information. This relationship is not linear since the first components are sorted to hold more of the variance observed in our data.

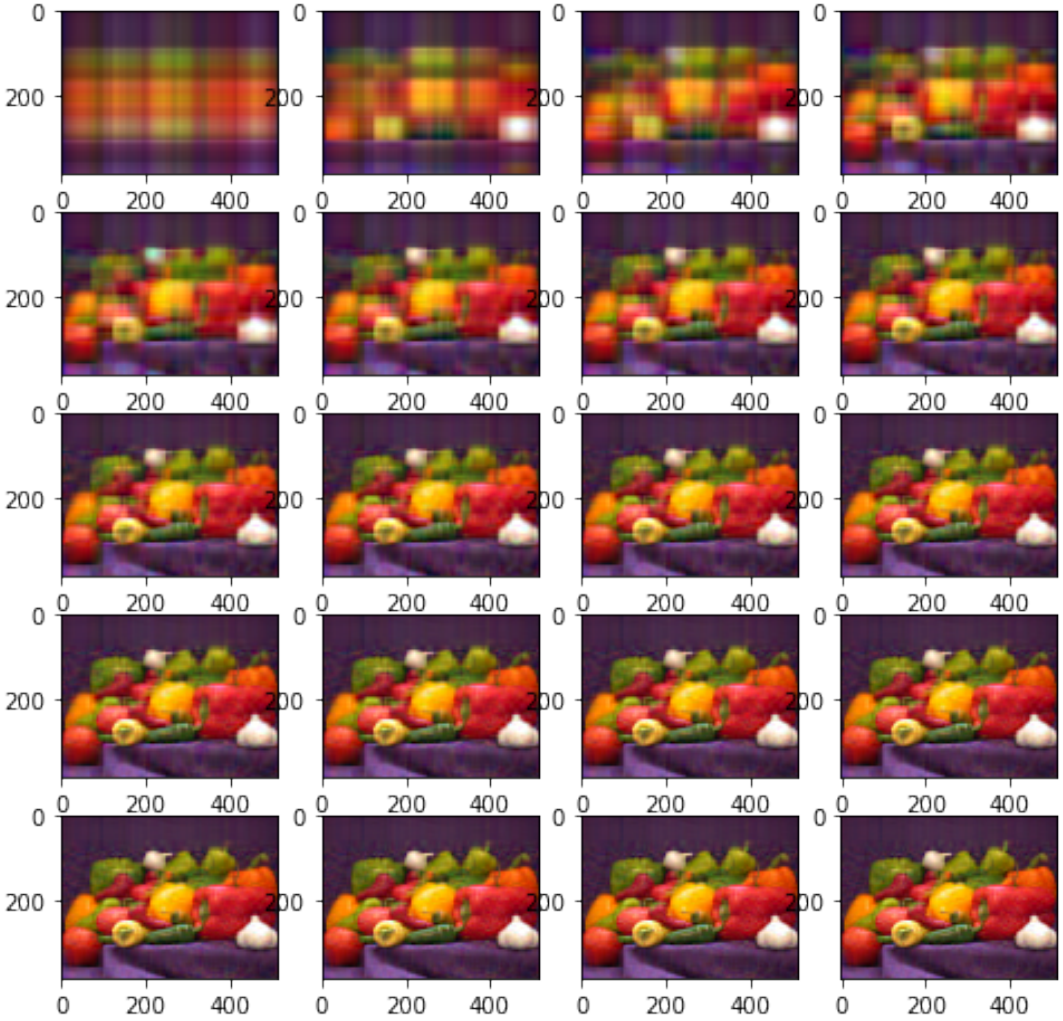
[2]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
imagePepperMatrix = mpimg.imread("./data/peppers.png")
plt.imshow(imagePepperMatrix)
U, S, VT = np.linalg.svd(imagePepperMatrix)

def mse(imageA, imageB) -> float:
        err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
        err /= float(imageA.shape[0] * imageA.shape[1])
        return err


def compress_svd(image,k):
    """
    Perform svd decomposition and truncated (using k singular values/vectors)␣
 ↪reconstruction
    returns
    --------
      reconstructed matrix reconst_matrix, array of singular values s
    """
    U,s,V = np.linalg.svd(image,full_matrices=False)
    reconst_matrix = np.dot(U[:,:k],np.dot(np.diag(s[:k]),V[:k,:]))
```

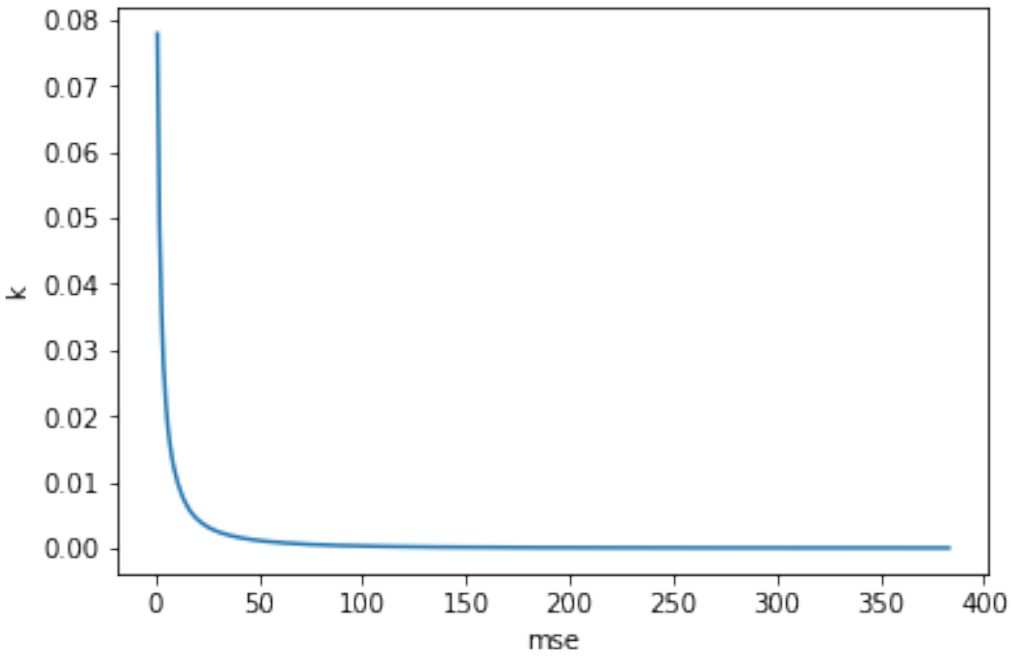
```
    return reconst_matrix,s


def compress_show_color_images_layer(image, k):
    """
     compress and display the reconstructed color image using the layer method
    """
    original_shape = image.shape
    image_reconst_layers = [compress_svd(image[:,:,i],k)[0] for i in range(3)]
    image_reconst = np.zeros(image.shape)
    for i in range(3):
        image_reconst[:,:,i] = image_reconst_layers[i]
    return image_reconst

fig=plt.figure(figsize=(8, 8))
columns = 4
rows = 5
k=1
for i in range(1, columns*rows +1):
    fig.add_subplot(rows, columns, i)
    img = compress_show_color_images_layer(imagePepperMatrix,k)
    plt.imshow(img)
    k+= 1

plt.show()

mses = [
    (
        k,
        mse(
            imagePepperMatrix,
            compress_show_color_images_layer(imagePepperMatrix,k)
        )
    )
    for k in range(1,384)
]
plt.plot(
    [i[0] for i in mses],
    [i[1] for i in mses]
)
plt.ylabel('k')
plt.xlabel('mse')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```


0
50
100
150
200
250
300
350
0
100
200
300
400
500


0
200
0
200
400


0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
k
0
50
100
150
200
250
300
350
400
mse

ex4

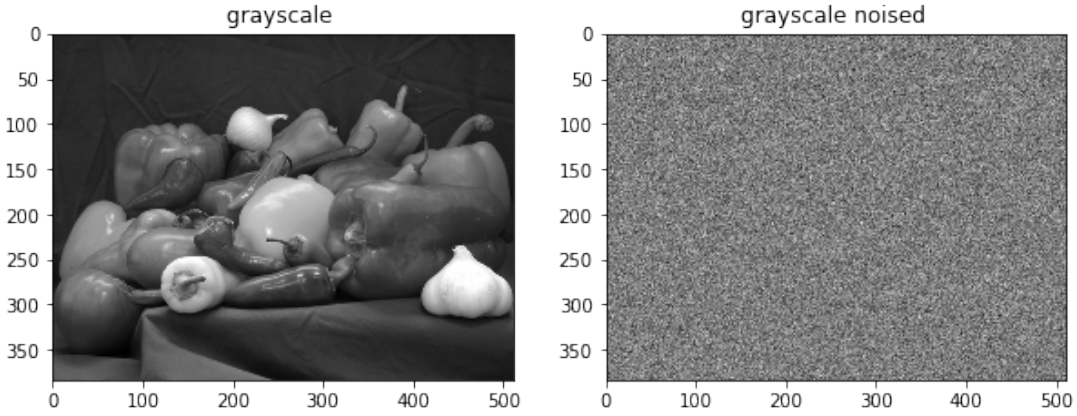
May 15, 2019

0.0.1 Exercise 4.

4.1)Read the image peppers.png and convert it to grayscale and add Gaussian noise N (0, 625). Perform its low-rank approximation for k = 1, ..., n.

[10]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import numpy as np

def mse(imageA, imageB) -> float:
        err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
        err /= float(imageA.shape[0] * imageA.shape[1])
        return err

def gaussianNoise(image, var, mean = 0):
      row,col = image.shape
      sigma = var**0.5
      randomGaus = np.random.normal(mean,sigma,(row,col))
      gaussNoiseMatrix = randomGaus.reshape(row,col)
      noisy = image + gaussNoiseMatrix
      return noisy, gaussNoiseMatrix

imagePepperMatrix = mpimg.imread("./data/peppers.png")
gray__imagePepperMatrix = cv2.cvtColor(
    imagePepperMatrix,
    # cv2.COLOR_BGR2GRAY
    cv2.COLOR_RGB2GRAY
)
fig=plt.figure(figsize=(10, 10))
fig.add_subplot(1, 2, 1)
plt.title( "grayscale ")
plt.imshow(gray__imagePepperMatrix,  cmap='gray' )
noised__gray__imagePepperMatrix, noiseMatrix= ␣
 ↪gaussianNoise(gray__imagePepperMatrix, 625, 0)
fig.add_subplot(1, 2, 2)
plt.title( "grayscale noised ")
plt.imshow(noised__gray__imagePepperMatrix,  cmap='gray' )
```

```
plt.show()
```

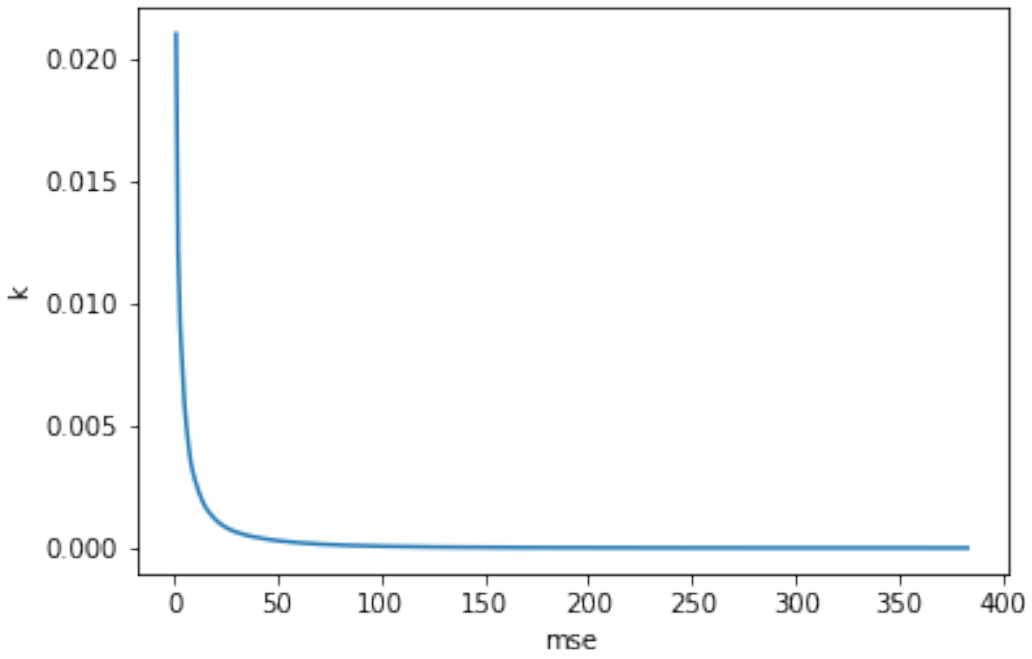


4.2 Plot the dependence between the k and MSE of k-rank approximation version of original image. Make a conclusion. The Plot the dependence between the k and MSE of k-rank approximation version of original image. (mistake?) I am plotting the approximation with grayscale noised image.

Conclusion The svd reconstruction ratio is maintained even with the noised image.

[11]:
```
#### 4.2 Plot the dependence between the k and MSE of k-rank approximation␣
 ↪version of original image. Make a conclusion.

def grayscale_compress_svd(image,k):
    """
    Perform svd decomposition and truncated (using k singular values/vectors)␣
 ↪reconstruction
    returns
    --------
      reconstructed matrix reconst_matrix, array of singular values s
    """
    U,s,V = np.linalg.svd(image,full_matrices=False)
    reconst_matrix = np.dot(U[:,:k],np.dot(np.diag(s[:k]),V[:k,:]))
    return reconst_matrix,s


def grayscale_compress_show_color_images_layer(image, k):
    """
     compress and display the reconstructed color image using the layer method
    """
    original_shape = image.shape
    image_reconst_layers = grayscale_compress_svd(image ,k)
```

```
    image_reconst = image_reconst_layers
    return image_reconst[0]

mses = [
    (
        k,
        mse(
            gray__imagePepperMatrix,
            grayscale_compress_show_color_images_layer(gray__imagePepperMatrix,k)
        )
    )
    for k in range(1,384)
]
plt.plot(
    [i[0] for i in mses],
    [i[1] for i in mses]
)
plt.ylabel('k')
plt.xlabel('mse')
plt.show()
```



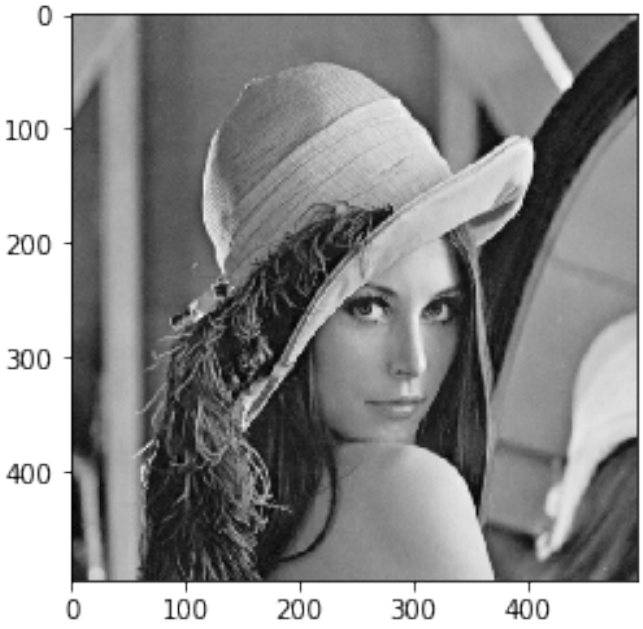
ex5

May 15, 2019

0.0.1 TODO Exercise 5. ex

(a) Read the image lena.png and convert it to grayscale.

[67]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import cv2
import numpy as np
imageLena = mpimg.imread("./data/lena.png")
imageLena  = imageLena *256
imageLena
output = cv2.cvtColor(
    imageLena,
    cv2.COLOR_RGB2GRAY
)
gray__imageLena= output .copy()

plt.imshow(gray__imageLena, cmap = 'gray' )
plt.show()
(h, w) = gray__imageLena.shape[:2]
```



[67]:
```
array([[160.80013 , 162.80797 , 162.80797 , ..., 171.45676 , 172.23178 ,
        152.41035 ],
       [161.80405 , 163.58301 , 162.80797 , ..., 171.22787 , 175.71841 ,
        155.42213 ],
       [163.29387 , 163.65428 , 162.28995 , ..., 169.69487 , 171.70271 ,
        155.18219 ],
       ...,
       [ 50.322575,  44.299046,  50.208126, ..., 101.09691 , 102.66905 ,
         97.44264 ],
       [ 45.302967,  42.291203,  51.212048, ..., 100.48653 , 103.755295,
         99.30392 ],
       [ 43.295124,  43.295124,  55.227734, ..., 101.49045 , 105.24512 ,
        104.39479 ]], dtype=float32)
```

0.0.2 2.b.1) Add a watermark to the image with and without applying NVF function the different values of $\frac{z}{2}$ (10, 25, 50, 75) and D. Choose the window size appropriate to used image.

The Noise Visibility Function (NVF) describes noise visibility in an image. The most known form of NVF is given as:

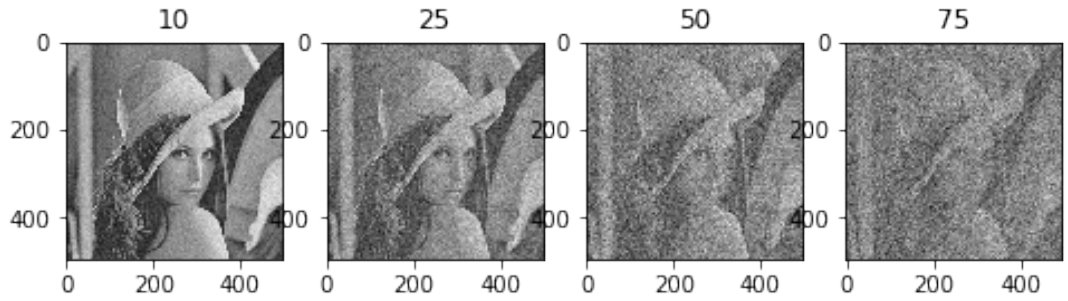
$NVF = \frac{1}{1+\theta\sigma^2_{\bar{x}}(i,j)}$ $\theta = \frac{D}{\sigma^2_{\bar{x}max}}$

- $\sigma_{x_{max}}$ (i, j) denotes the local variance of the image in a window centred on the pixel with coordinates (i, j),
- plays the role of contrast adjustment for every particular image, $\sigma^2_{\bar{x}_{max}}$ is the maximum local variance for a given image

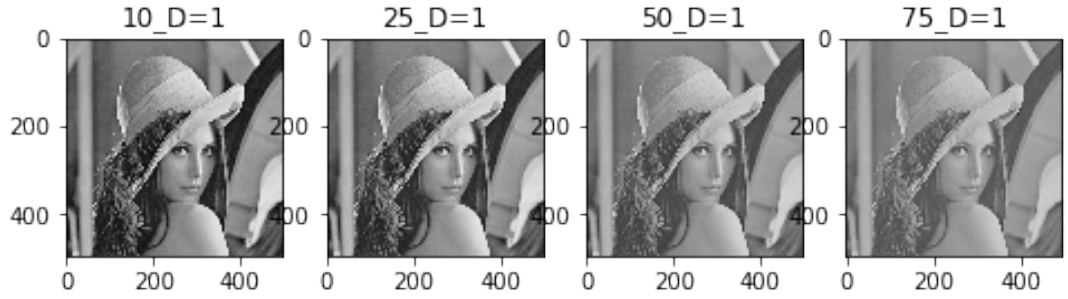
- D is an experimentally determined parameter.
- The final embedding equation is: $y_{i,j} = x_{i,j} + (1NVF)z_{i,j}$

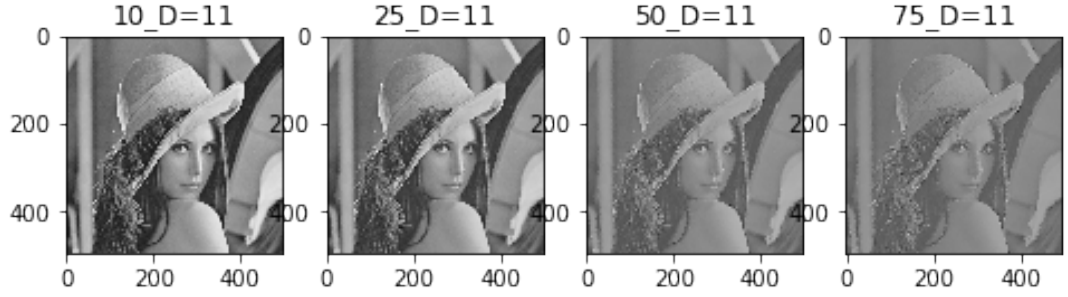
[87]:
```
# Here we generate the different watermarks for variances [10, 25, 50, 75]

def generateWatermark(window_size, sigma, mu=0, ):
    (row, col) = window_size
    return np.random.normal(mu,sigma,(row,col))

watermark_size = gray__imageLena.shape[:2]
watermark_list = [
    (generateWatermark(watermark_size, sigma), sigma)
    for sigma in [10, 25, 50, 75]
]

# Add watermark to the image without the NVF function
#with equation $y = x + z$ which is equivalent to adding gaussian noise

def addWatermark(img, z):
    return np.add(img, z)

def showImages(images, titles):
    fig=plt.figure(figsize=(8, 8))
    columns = 4
    rows = round(len(images)/columns)
    k=1
    for i in range(1, columns*rows +1):
        fig.add_subplot(rows, columns, i)
        img = images[k-1]
        plt.imshow(img, cmap = 'gray')
        plt.title(titles[k-1])
        k+= 1
    plt.show()

noNVFwatermarked_Img_list = [
    (addWatermark(gray__imageLena, watermark[0]), str(watermark[1]))
    for watermark in watermark_list
]

showImages(
    [im[0] for im in noNVFwatermarked_Img_list],
    [im[1] for im in noNVFwatermarked_Img_list]
)

# Here we implement the description functions
from scipy import ndimage
lena_variance_matrix = ndimage.generic_filter(gray__imageLena, np.var, size=4)
```

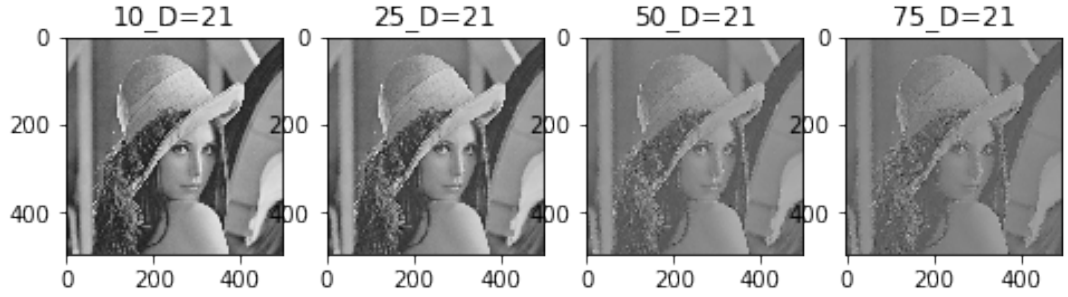
```
maxLocalVar = lena_variance_matrix.max()
Ds = range(1,51, 10)

for D in Ds:
    theta = D / maxLocalVar
    print(theta)
    NVF_matrix =  1/(1+ theta*lena_variance_matrix)

    def embeddingEquation(NVF_matrix, x, z):
        return np.add(x, np.multiply((1-NVF_matrix ), z))

    NVFwatermarked_Img_list = [
        (
            embeddingEquation(NVF_matrix, gray__imageLena, watermark[0]),
            str(watermark[1]) + "_D=" + str(D)
        )
        for watermark in watermark_list
    ]
    showImages(
        [im[0] for im in NVFwatermarked_Img_list],
        [im[1] for im in NVFwatermarked_Img_list]
    )
```

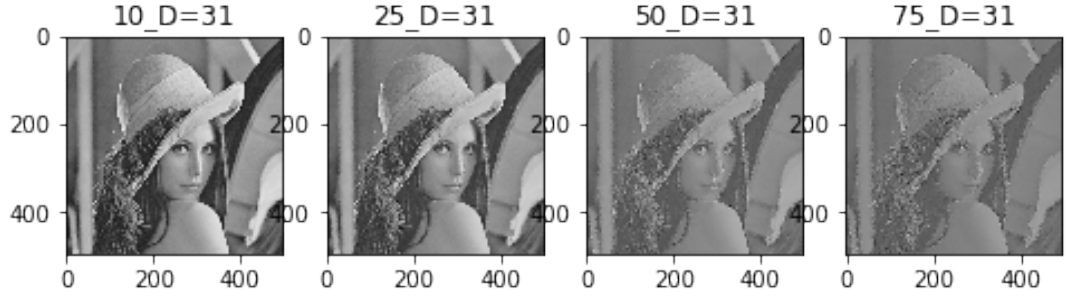


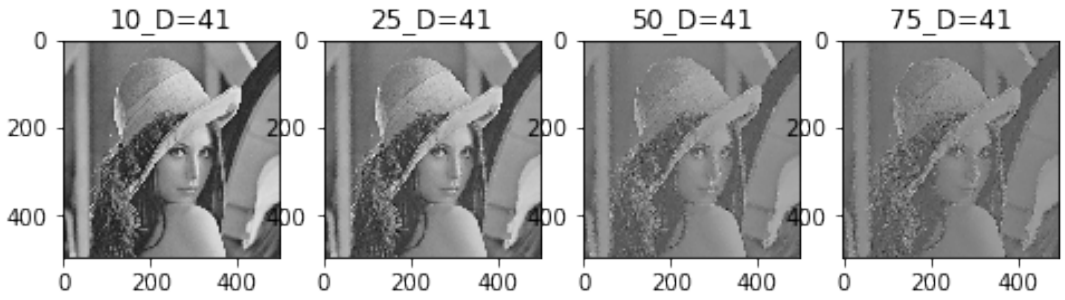

```
0.0001630953068290187
0.0017940483751192058
0.0034250014434093925
0.00505595451169958
0.006686907579989767
```


10_D=1
25_D=1
50_D=1
75_D=1


10_D=11
25_D=11
50_D=11
75_D=11


10_D=21
25_D=21
50_D=21
75_D=21


10_D=31
25_D=31
50_D=31
75_D=31



0.0.3 2.c) Report the dependency between the parameters $\frac{z}{2}$, D and original image.

From the small experiment we ran above we determine that greater values of D make the influece of the window bigger. The $\frac{z}{2}$ as expected increases the noise in the image.

ex6

May 15, 2019

1 Exercise 6. ex

(a) You are given a set of images hdr images. Combine the images (not necessary all) to one image in such a way that the result image has higher quality then all given images in the set. You can sum, subtract the images, divide by some constant, multiply by some mask, etc.

[21]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt

def showImages(images, titles):
    fig=plt.figure(figsize=(20, 20))
    columns = 2
    rows = 2
    k=1
    for i in range(1, columns*rows +1):
        fig.add_subplot(rows, columns, i)
        im = images[k-1]
        rgb = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
        plt.imshow(rgb, cmap=plt.cm.Spectral)
        plt.title(titles[k-1])
        k+= 1
    plt.show()

folder_path = "./data/hdr_images/"
img_titles = ["img01.tif", "img02.tif", "img03.tif", "img04.tif", "img05.tif"]
img_fn= [ folder_path +f for f in img_titles ]
img_list = [cv2.imread(fn, -1) for fn in img_fn]

# Exposure fusion using Mertens
merge_mertens = cv2.createMergeMertens()
res_mertens = merge_mertens.process(img_list)
```

(b) Visualise the results and explain how did you obtain them.

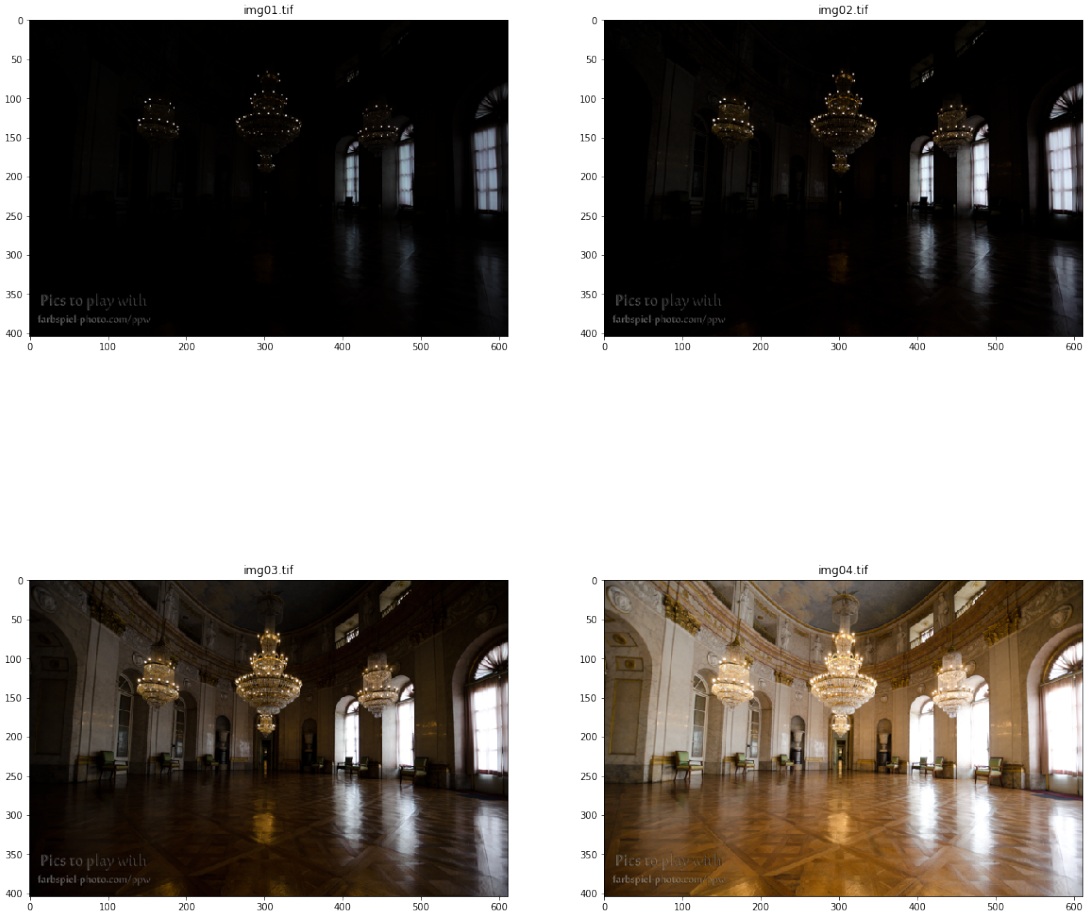
Pixels are weighted using contrast, saturation and well-exposedness measures, then images are combined using laplacian pyramids.

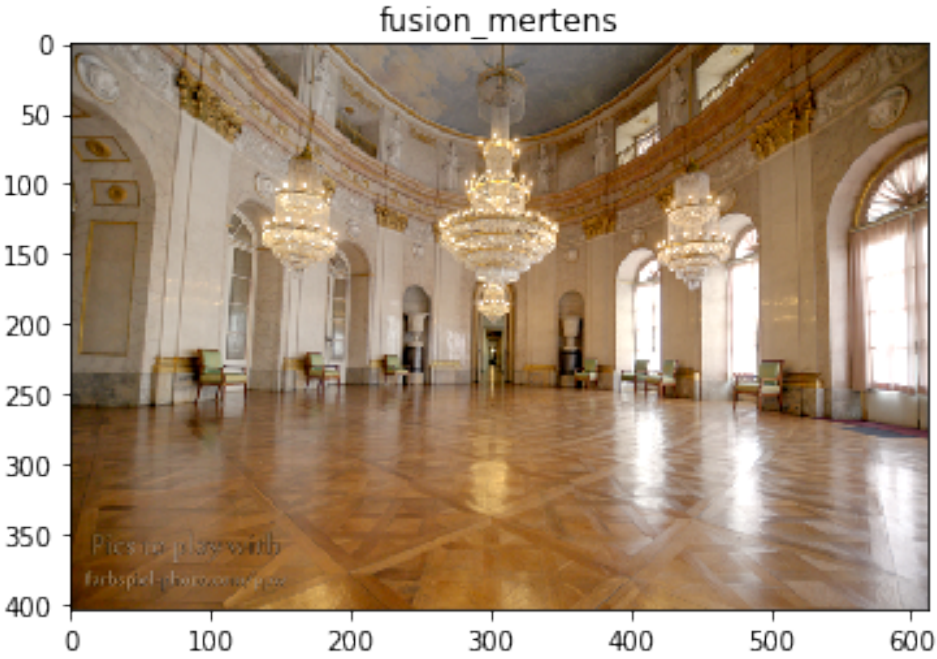
The resulting image weight is constructed as weighted average of contrast, saturation and well-exposedness measures. The library of openCV relies on pyramidal image decomposition. It assumes that the images are perfectly aligned. Exposure fusion computes the desired image by keeping only the “best” parts in the multi-exposure image sequence. This process is guided by a set of quality measures, which is aaggregated in a scalar-valued weight map. It is useful to think of the input sequence as a stack of images. The final image is then obtained by collapsing the stack using weighted blending

source: https://mericam.github.io/papers/exposure_fusion_reduced.pdf

Solution for a, b Base on this example: TODO https://docs.opencv.org/4.1.0/d2/df0/tutorial_py_hdr.html

[24]:
```
showImages(img_list, img_titles)

res_mertens_8bit = np.clip(res_mertens*255, 0, 255).astype('uint8')
rgb = cv2.cvtColor(res_mertens_8bit, cv2.COLOR_BGR2RGB)
plt.imshow(rgb, cmap = plt.cm.Spectral)
plt.title("fusion_mertens")
plt.show()
```




fusion_mertens
0
50
100
150
200
250
300
350
400
0
100
200
300
400
500
600

ex7

May 15, 2019

0.0.1 You are given a "renoir" set of two images (reference and noisy) from the RENOIR dataset 2 .

(a) Visualise all color channels of both images. Are the all channels equally affected by the noise?

No green not affected as much as red and blue.

[1]:
```
import scipy.ndimage as ndimage
import matplotlib.pyplot as plt
import cv2
import numpy as np

im1 = ndimage.imread("./data/renoir/Reference.bmp")
im1 = np.asarray(im1)
im2 = ndimage.imread("./data/renoir/Noisy.bmp")
im2 = np.asarray(im2)
plt.imshow(im1, cmap=plt.cm.Spectral)
plt.show()
plt.imshow(im2, cmap=plt.cm.Spectral)
plt.show()

#plt.colormaps()

red_images = im2[:,:,0]
green_images = im2[:,:,1]
blue_images = im2[:,:,2]
plt.imshow(red_images, cmap=plt.cm.Reds)
plt.show()
plt.imshow(green_images, cmap=plt.cm.Greens)
plt.show()
plt.imshow(blue_images, cmap=plt.cm.Blues)
plt.show()
```

```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:6:
DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0.
Use ``matplotlib.pyplot.imread`` instead.
```

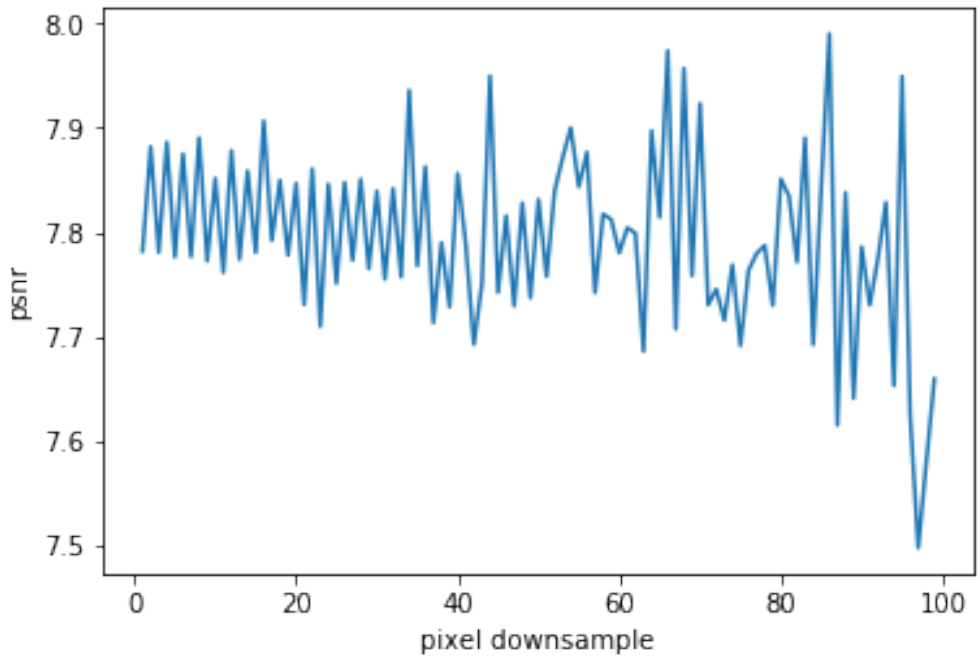
```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:8:
DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0.
Use ``matplotlib.pyplot.imread`` instead.


<Figure size 640x480 with 1 Axes>


<Figure size 640x480 with 1 Axes>


<Figure size 640x480 with 1 Axes>


<Figure size 640x480 with 1 Axes>


<Figure size 640x480 with 1 Axes>
```

0.0.2 (b) Try to decrease the noise via image down/up sampling.

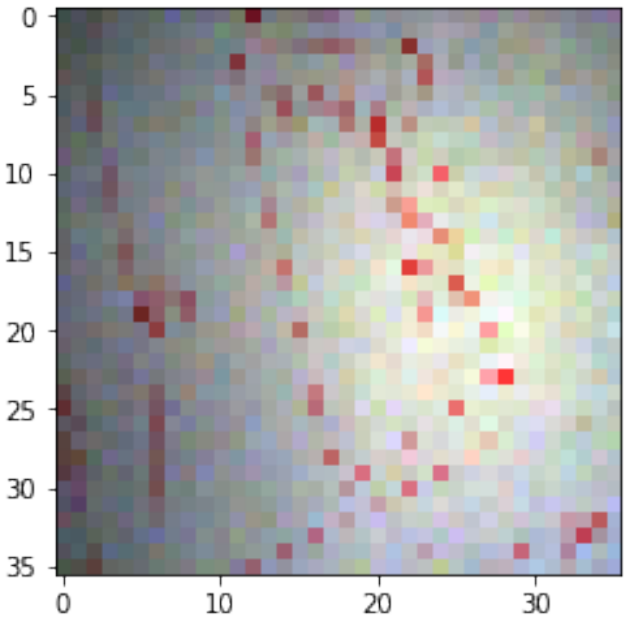
b.a) Do it for the RGB image. Measure the PSNR between the reference and de-noised images.

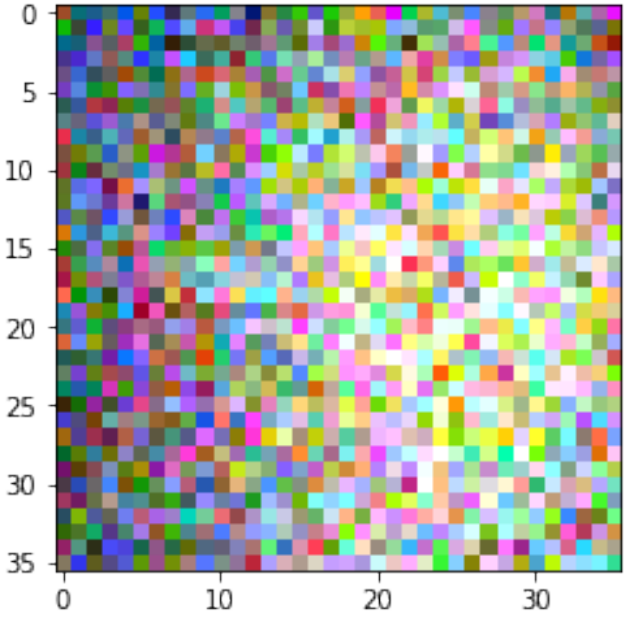
[33]:
```
from helpers import mse
import math
# Downsample an image by skipping indicies

def downsample_image(img, skip):
     return img[::skip,::skip]

def psnr(mseres, PIXEL_MAX_SQUARE = 255):
    mse_square = math.sqrt(mseres )
    part1 =  math.log10(PIXEL_MAX_SQUARE / mse_square)
    return 20 * part1



# downsampling
psnr_list = [psnr(mse( downsample_image(im1, skip), downsample_image(im2,␣
 ↪skip))) for skip in range(1,100)]
plt.plot([skip for skip in range(1,100)], psnr_list)
plt.xlabel("pixel downsample")
plt.ylabel("psnr")
plt.show()

print("no downsampled pair value psnr: " )
print(psnr(mse(im1,im2)))
```

```
print("max psnr downsampled value: ")
print(max(psnr_list))
print("best downsampling skip value: ")

print(psnr_list.index(max(psnr_list)))
best_ratio = psnr_list.index(max(psnr_list))

plt.imshow(downsample_image(im1, best_ratio))
plt.show()
plt.imshow(downsample_image(im2, best_ratio))
plt.show()


# best_ratio, im1.shape, downsample_image(im1, best_ratio).shape
```



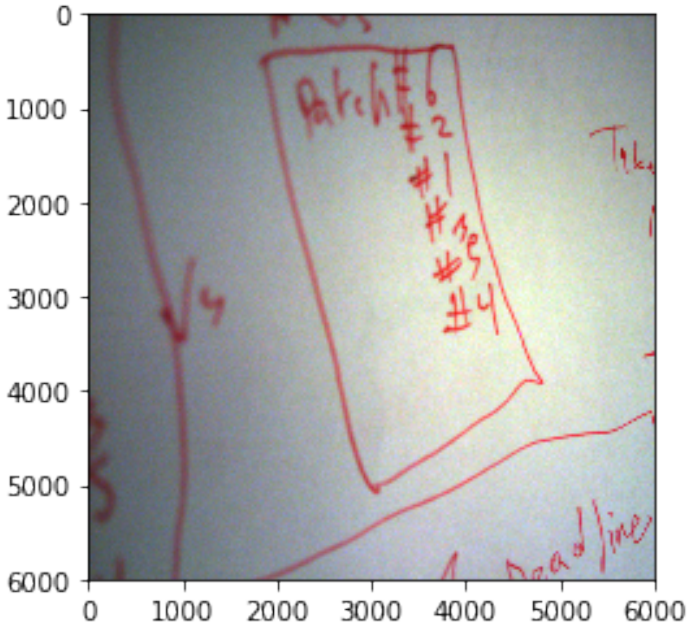

```
no downsampled pair value psnr:
7.781946412448223
max psnr downsampled value:
7.989865945271479
best downsampling skip value:
85
```



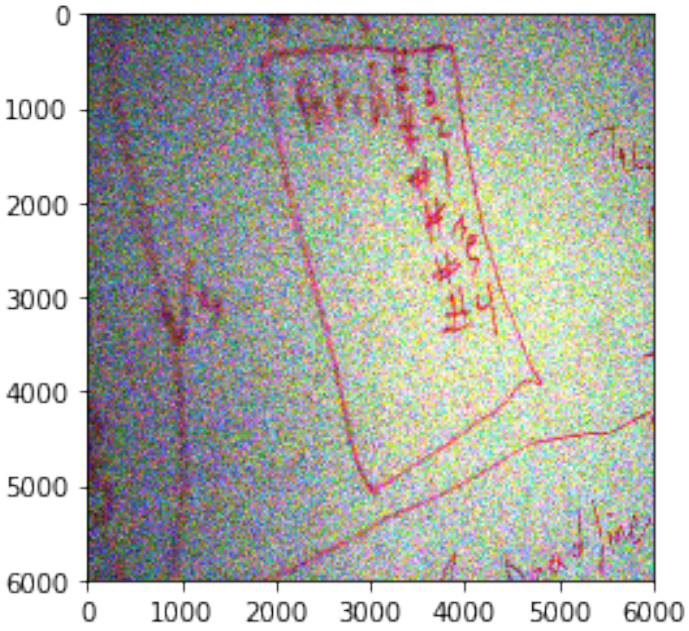


[5]:
```
#upsampling using scipy.ndimage
import skimage.transform
def upsample_skimage(factor, input_img):
        # Order=1 is bilinear upsampling
```

```
    return skimage.transform.rescale(input_img,
                                     factor,
                                     mode='constant',
                                     cval=0,
                                     order=1)

upsampled_im1 = upsample_skimage(factor=2, input_img=im1)
upsampled_im2 = upsample_skimage(factor=2, input_img=im2)
plt.imshow(upsampled_im1)
plt.show()
plt.imshow(upsampled_im2)
psnr(mse(upsampled_im1,upsampled_im2))
```



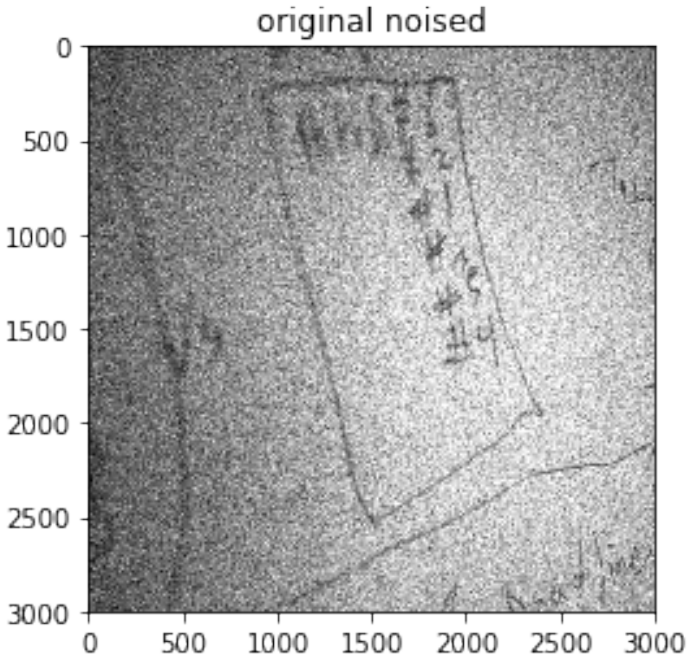
[5]: `57.29515427642124`

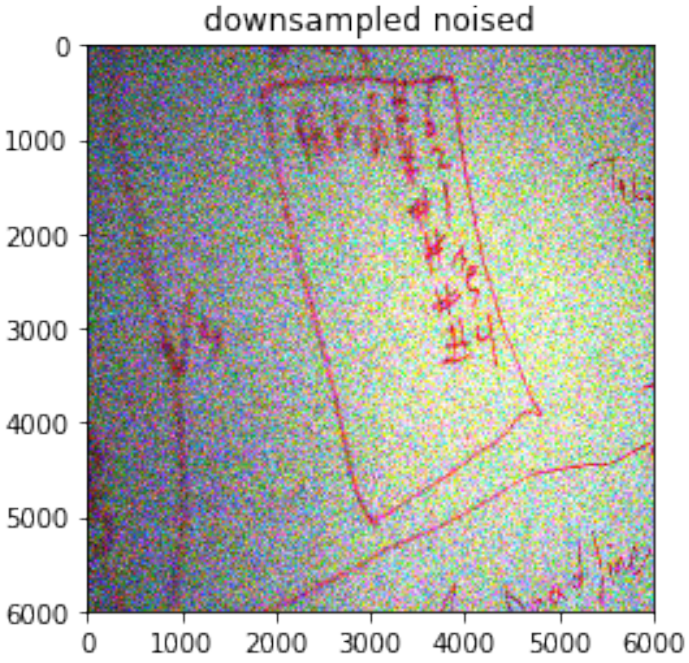


b.b) Do it for the grayscale image. Measure the PSNR between the grayscale reference and de-noised images.

[38]:
```
im1_gr = cv2.cvtColor(
    im1,
    cv2.COLOR_RGB2GRAY
)
im2_gr = cv2.cvtColor(
    im2,
    cv2.COLOR_RGB2GRAY
)
plt.title("original noised")
plt.imshow(im2_gr, cmap = 'gray' )
plt.show()

d1=downsample_image(im1_gr, 3)
d2=downsample_image(im2_gr, 3)
plt.imshow(upsampled_im2, cmap = 'gray')
plt.title("downsampled noised")
plt.show()
print("downsampled psnr")
print(psnr(mse( d1, d2)))

#upscale
upsampled_im1 = upsample_skimage(factor=3, input_img=im1_gr)
```

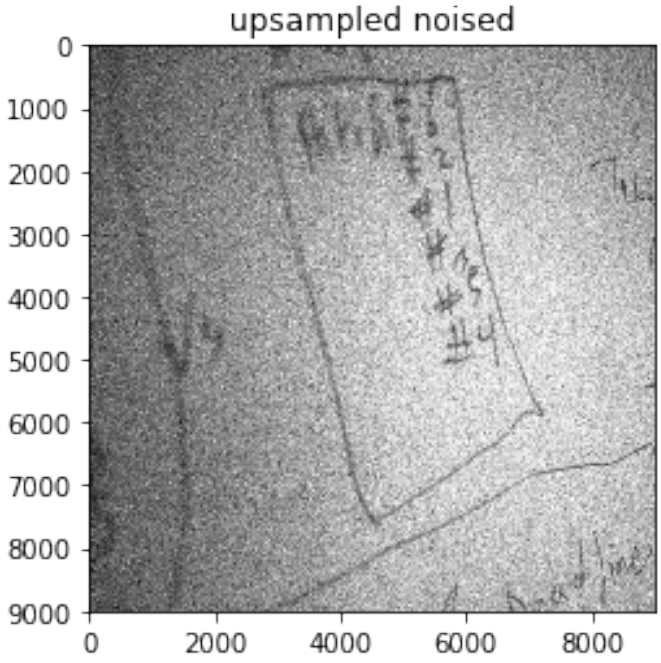
```
upsampled_im2 = upsample_skimage(factor=3, input_img=im2_gr)
plt.title("upsampled noised")
plt.imshow(upsampled_im2, cmap = 'gray')
print("upsampled psnr")
psnr(mse(upsampled_im1,upsampled_im2))
```





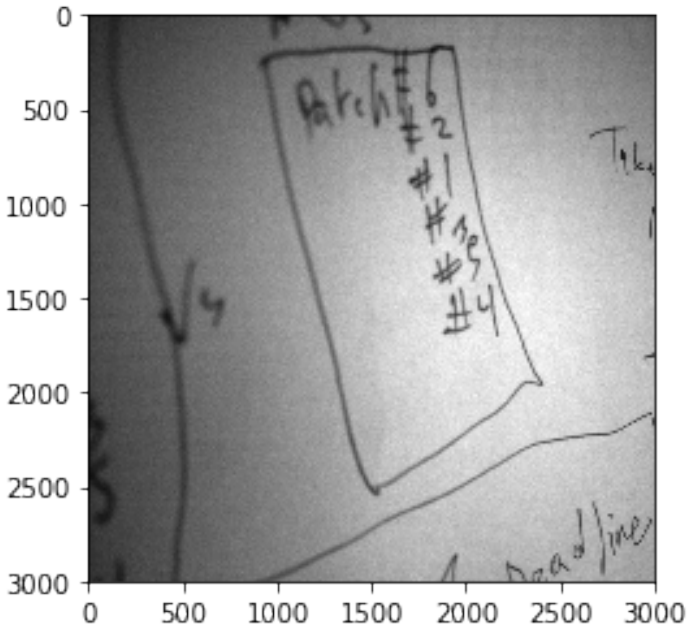

```
downsampled psnr
19.06967847453363
upsampled psnr
```

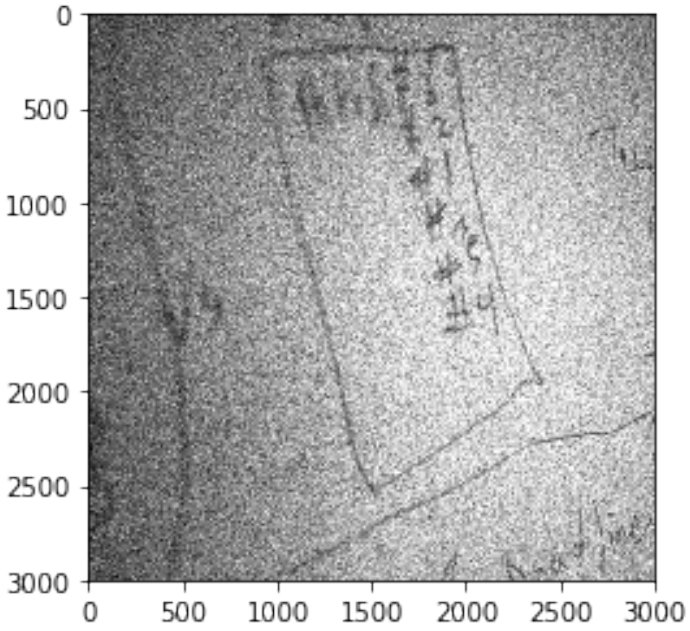
[38]: `69.10304970013075`



b.c) Do the denoising for the RGB image and after convert it to the grayscale. Measure the PSNR. Does the obtained result is different from the (b)? Explain the result. I used the best from the two techniques the psnr dropped which implies quality drop. Upsampling is sensitive to color mapping since the transform from rgb to grayscale is not linear.

[39]:
```
upsampled_im1 = upsample_skimage(factor=3, input_img=im1)
upsampled_im2 = upsample_skimage(factor=3, input_img=im2)

upsampled_im1_gr = cv2.cvtColor(
    im1,
    cv2.COLOR_RGB2GRAY
)
upsampled_im2_gr = cv2.cvtColor(
    im2,
    cv2.COLOR_RGB2GRAY
)
plt.imshow(upsampled_im1_gr,cmap='gray')
plt.show()
plt.imshow(upsampled_im2_gr, cmap='gray')
plt.show()

print(psnr(mse(upsampled_im1_gr,upsampled_im2_gr)))
```






```
19.076533001482233
```

0.0.3 (c) What other methods can you suggest to improve the noisy image quality?

Deep neural network autoencoder eg. https://papers.nips.cc/paper/4686-image-denoising-and-inpainting-with-deep-neural-networks.pdf

ex8

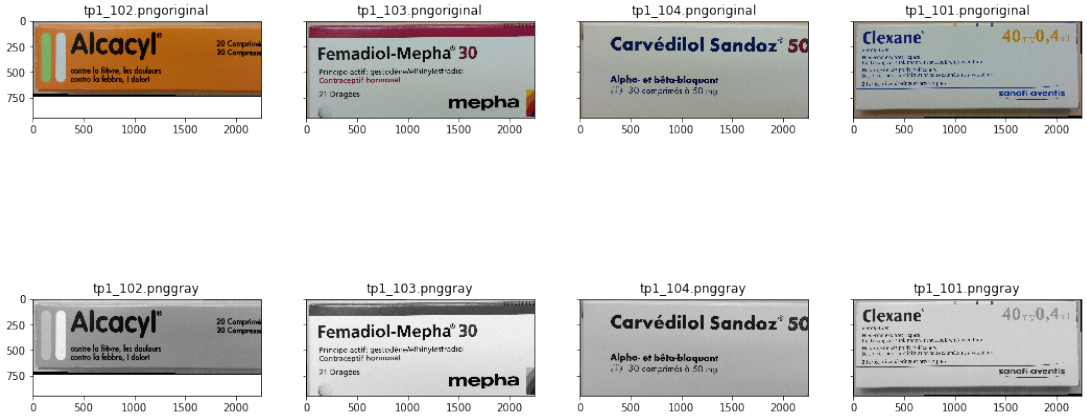
May 15, 2019

1 Exercise 8. You are given a set of 4 images: tp1 101.png - tp1 104.png. For one of these images perform the segmentation of the text information. See the example in Figure 3. Some graphical elements can be segmented as well.

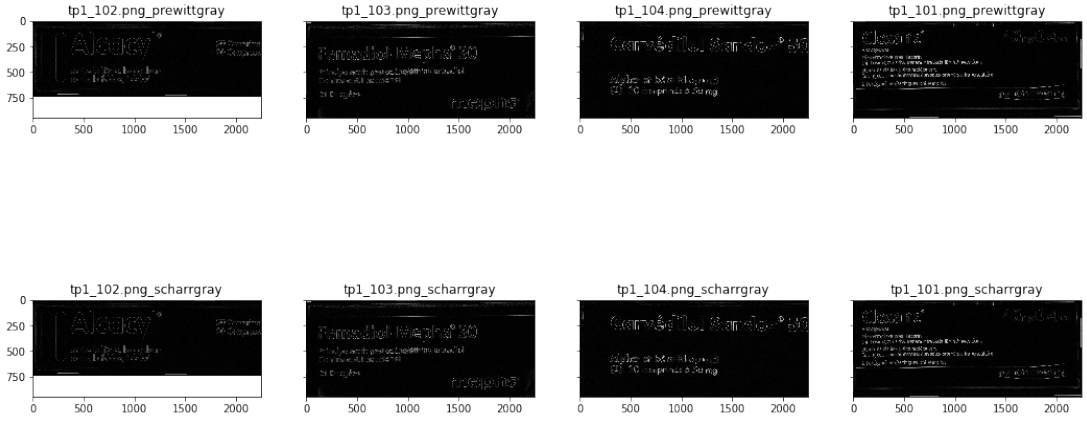
Hint: use edge detection and image filtering techniques. The next Matlab function can be useful imdilate, imfill, bwconncomp, regionprops

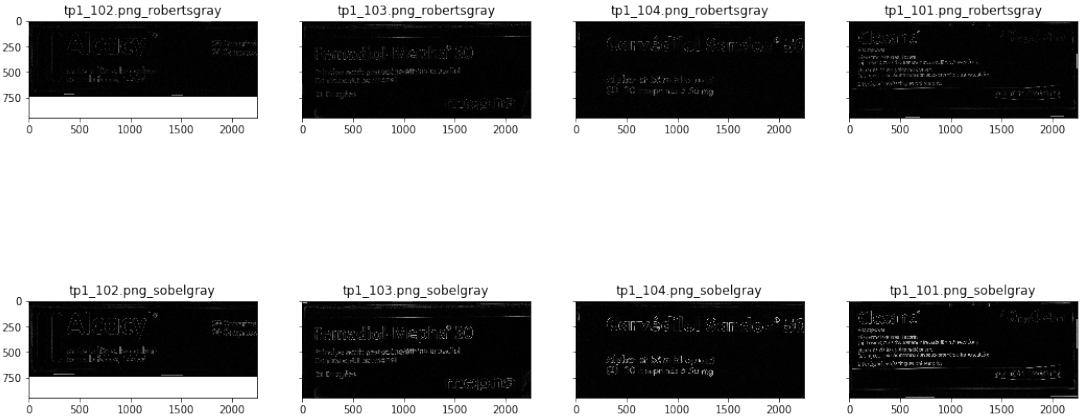
[90]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.filters import roberts, sobel, scharr, prewitt, gaussian
import os
import matplotlib.image as mpimg

class ImageType():
    def __init__(self, **kwargs):
        self.name = kwargs["name"]
        self.data = kwargs["data"]

def toGrayScale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

def showImgs(images, title, colormap):
    fig, ax = plt.subplots(
        ncols=4,
        sharex=True,
        sharey=True,
        figsize=(18, 14)
    )

    for index, im in enumerate(images):
        ax[index].imshow(im.data, colormap)
        ax[index].set_title(im.name + title)


images = [
```

```
    ImageType(name=file, data=mpimg.imread("./data/" + file))
    for file in os.listdir("./data/") if file.startswith("tp1_")
]

images_gray = [
    ImageType(name=file, data=toGrayScale(mpimg.imread("./data/" + file)))
    for file in os.listdir("./data/") if file.startswith("tp1_")
]
showImgs(images, "original", plt.cm.Spectral)
showImgs(images_gray, "gray", "gray")
```



[99]:
```
filtered = {}
for filter in [ prewitt,scharr, roberts, sobel]:
    temp = [
        ImageType(name=im.name + "_" + filter.__name__, data=filter(im.data))
        for im in images_gray
    ]
    filtered[filter.__name__] = temp
    showImgs(temp, "gray", "gray")
```




tp1_102.png_robertsgray
tp1_103.png_robertsgray
tp1_104.png_robertsgray
tp1_101.png_robertsgray
tp1_102.png_sobelgray
tp1_103.png_sobelgray
tp1_104.png_sobelgray
tp1_101.png_sobelgray

ex9

May 15, 2019

0.1 Exercise 9 Nuts and bolts

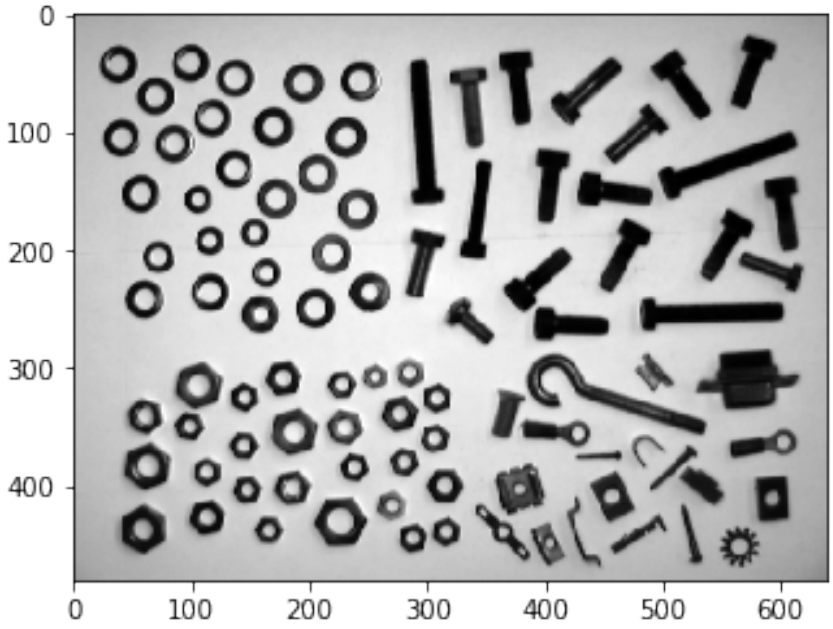
In this little project you will design and test a program that can recognize various nuts and bolts in an image using Matlab's morphological functions and a bit of statistics.

The image can be seen in Figure 4. Matlab contains an excellent tutorial segmenting and counting rice in an image, which you can work through as preparation. The principle steps that need to be done are the following: - Segment the foreground which contains all parts, from the background. You can use 4 http://www.mathworks.ch/ch/help/images/image-enhancement-and-analysis.html morphological opening, e.g. imopen to ascertain background statistics or use the so called Otsi's method implemented by Matlab graythresh. - Use morphology to remove any noise from the image - Select all individual items using Matlab's bwlabel and bwconncomp. - To gather statistics deploy Matlab's regionprops function. It is capable of collecting a vast amount of information on binary objects which in term can be used to distinguish the various parts from each other. - Find a combination of metrics to separate the different parts as best as possible.

0.1.1 Exercise 9. ex

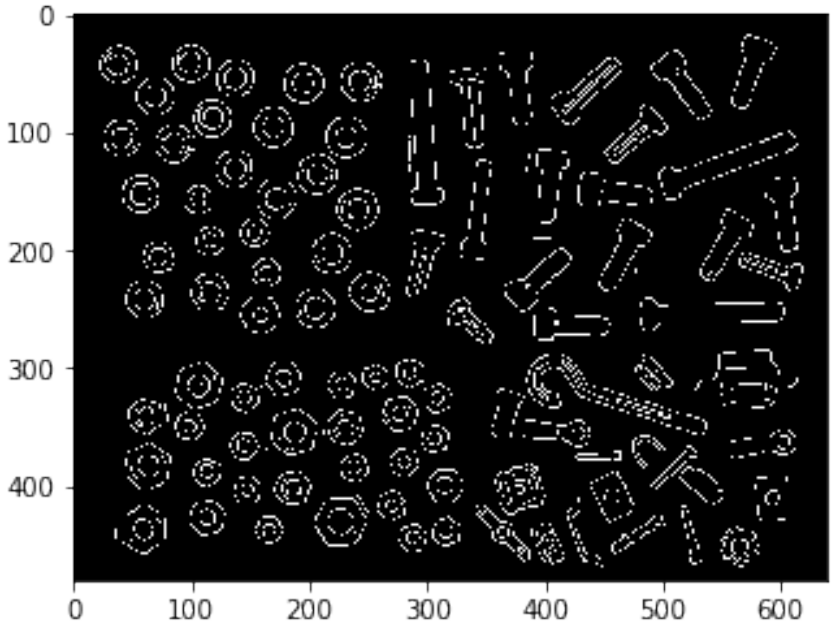
1. Implement the image segmentation and statistics gathering functions

[40]:
```
import skimage
import matplotlib.image as mping
import matplotlib.pyplot as plt
import numpy as np
import cv2
from scipy import ndimage as ndi
from  skimage.measure import label
def toGrayScale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

parts = toGrayScale(np.asarray(mping.imread("./data/parts.png")))
plt.imshow(parts, cmap="gray")
plt.show()
```



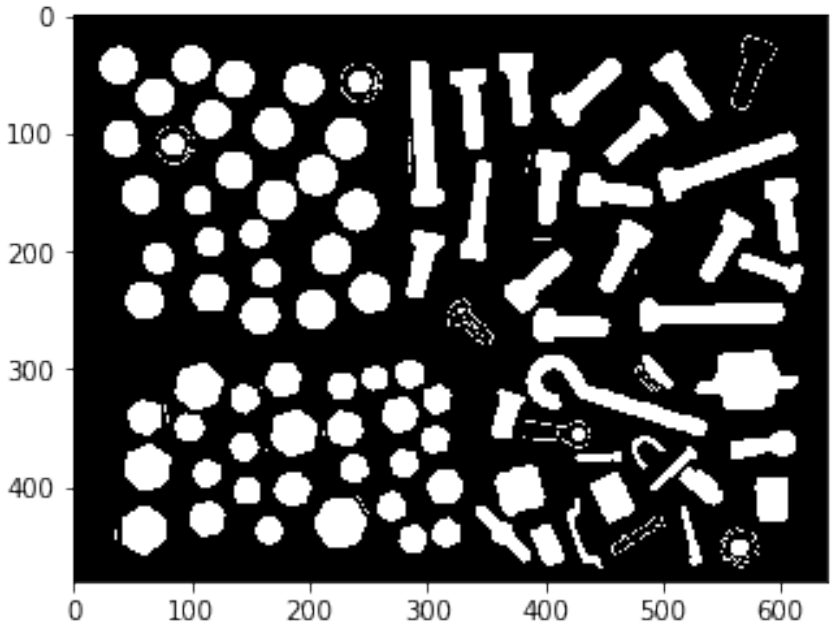
[34]:
```
edges = skimage.feature.canny(parts)
plt.imshow(edges , cmap="gray")
edges
```

[34]:
```
array([[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```



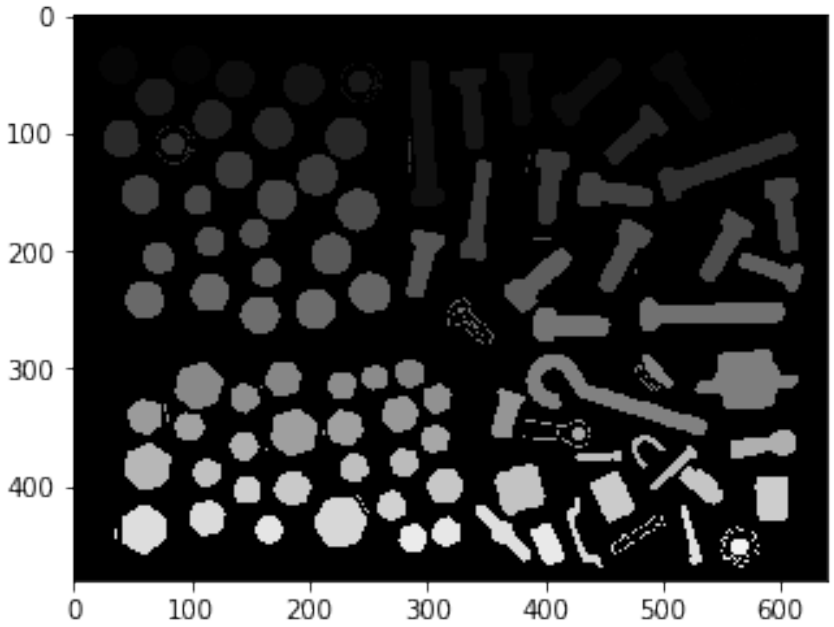
[29]:
```
fill_parts= ndi.binary_fill_holes(edges)
plt.imshow(fill_parts, cmap="gray")
```

[29]: <matplotlib.image.AxesImage at 0x7f0e225a66d8>



[50]:
```
labels =label(fill_parts)
labels.shape
plt.imshow(labels, cmap="gray")
```

[50]:
```
<matplotlib.image.AxesImage at 0x7f0e22338ef0>
```



[ ]:
```
areas = [r.area for r in skimage.measure.regionprops(labels)]
# plt.imshow(areas, cmap="gray")
```

[1]:
```
# otsu method from skimage

plt.show()
```

```
<Figure size 800x250 with 3 Axes>
```

2. Report on what statistics work and why (not).