assignment

mostafa.yacout

April 2022

1 Introduction

In this assignment I will be showing the results of doing histogram equalization and image filtering using open-CV library using python code.

2 Histogram equalization

Histogram equalization was applied on a jet plane photo that has a very low contrast as the histogram showed us that the majority of the colour levels are located in right side of the histogram. by creating a new function, we can distribute the bins on the whole histogram or even to a specified number of bins. Results are shown below.

it is clearly shown that when the number of gray levels are 256, the image has a good contrast and by lowering the number of gray levels it the contrast become a darker.

3 Noise and filtering

In this part I added two different types of noises on a picture then I applied two different filters. firstly, I added salt and pepper noise with density 0.05 on the photo to compare between the median filter and the box filter using 7x7 and 3x3 kernel. As we can see from the results that using the median filter visually and by measuring the MSE and PSNR is way better than the box filter. the box filter didn't clear the salt and pepper noise and made the picture blurry however the median filter did. Moreover, by adding white Gaussian noise with variance 0.005, i think that the two filters are not the best filter to apply on this type of noise. However, visually and theoretically the box filter is better than the median filter, the box filter made the picture a little bit blurry but it removed the noise in contrary the median filter made the picture looks worse.there is no best filter for all the pictures, however every type of noise has it own best filter.Now let's test our filters on real noise image.visually i can consider that the median filter is better as it doesn't make the picture looks blurry. the results are shown below.

ISO

Figure 1: box filter

Figure 2: median filter

3.1 salt and pepper

picture	MSE	PSNR
median	96.93	35.96
box	301.64	30.90

3.2 Gaussian noise

picture	MSE	PSNR
median	664.05	28.43
box	422.14	29.53

Figure 3: original

Figure 4: 256 bins

Figure 5: 128 bins

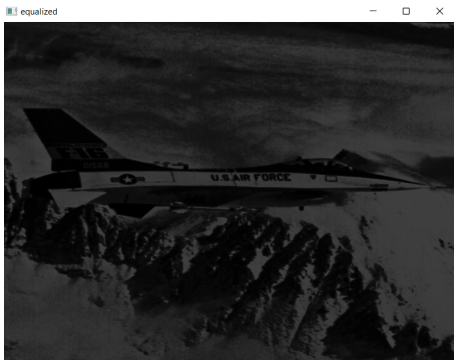

Figure 6: 64 bins

histogram equalization

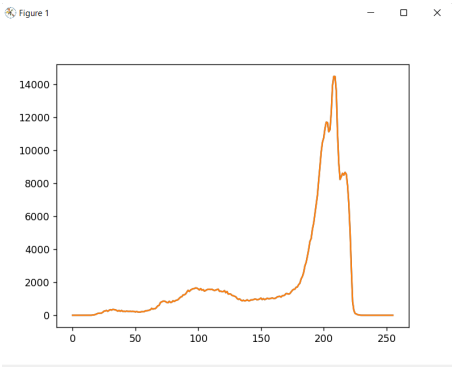

Figure 7: original

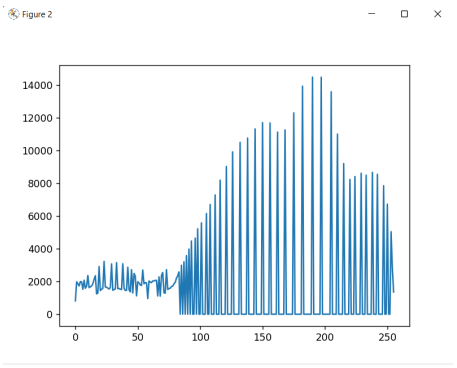

Figure 8: 256 bins

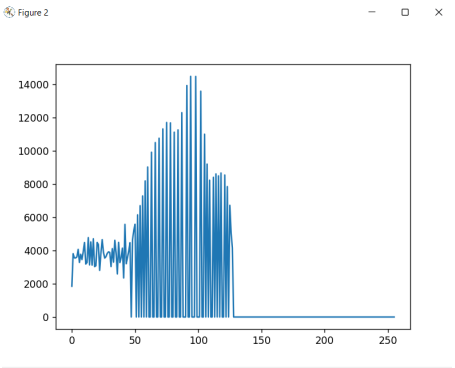

Figure 9: 128 bins

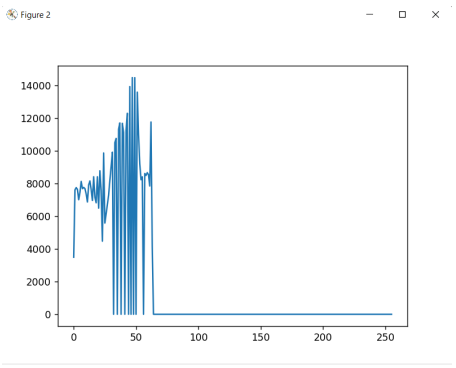

Figure 10: 64 bins

histogram

Figure 11: noisy

Figure 12: box filter

Figure 13: median filter

salt and pepper

Figure 14: noisy

Figure 15: box filter

Figure 16: median filter

white gaussian

4 Code

4.1 adding white Gaussian noise:

```
    #refrence https://scikit-image.org/docs/stable/api/skimage.util.html#random-noise
import cv2
import numpy as np
#from .dtype import img_as_float
from skimage.util import img_as_float
__all__ = ['random_noise']
img = cv2.imread('DSCN0479-001.jpg',
                 cv2.IMREAD_GRAYSCALE)

def _bernoulli(p, shape, *, random_state):
    """
    Bernoulli trials at a given probability of a given size.
    This function is meant as a lower-memory alternative to calls such as
    `np.random.choice([True, False], size=image.shape, p=[p, 1-p])`.
```

```
    While ‘np.random.choice‘ can handle many classes, for the 2-class case
    (Bernoulli trials), this function is much more efficient.
    Parameters
    ----------
    p : float
        The probability that any given trial returns ‘True‘.
    shape : int or tuple of ints
        The shape of the ndarray to return.
    seed : ‘numpy.random.Generator‘
        ‘‘Generator‘‘ instance.
    Returns
    -------
    out : ndarray[bool]
        The results of Bernoulli trials in the given ‘size‘ where success
        occurs with probability ‘p‘.
    """
    if p == 0:
        return np.zeros(shape, dtype=bool)
    if p == 1:
        return np.ones(shape, dtype=bool)
    return random_state.random(shape) <= p


def random_noise(image, mode='gaussian', seed=None, clip=True, **kwargs):
    """
    Function to add random noise of various types to a floating-point image.
    Parameters
    ----------
    image : ndarray
        Input image data. Will be converted to float.
    mode : str, optional
        One of the following strings, selecting the type of noise to add:
        - 'gaussian'  Gaussian-distributed additive noise.
        - 'localvar'  Gaussian-distributed additive noise, with specified
                      local variance at each point of ‘image‘.
        - 'poisson'   Poisson-distributed noise generated from the data.
        - 'salt'      Replaces random pixels with 1.
        - 'pepper'    Replaces random pixels with 0 (for unsigned images) or
                      -1 (for signed images).
        - 's&p'       Replaces random pixels with either 1 or ‘low_val‘, where
                      ‘low_val‘ is 0 for unsigned images or -1 for signed
                      images.
        - 'speckle'   Multiplicative noise using out = image + n*image, where
                      n is Gaussian noise with specified mean & variance.
    seed : {None, int, ‘numpy.random.Generator‘}, optional
        If ‘seed‘ is None the ‘numpy.random.Generator‘ singleton is
```

```
    used.
    If ‘seed‘ is an int, a new ‘‘Generator‘‘ instance is used,
    seeded with ‘seed‘.
    If ‘seed‘ is already a ‘‘Generator‘‘ instance then that
    instance is used.
    This will set the random seed before generating noise,
    for valid pseudo-random comparisons.
clip : bool, optional
    If True (default), the output will be clipped after noise applied
    for modes ‘’speckle’‘, ‘’poisson’‘, and ‘’gaussian’‘. This is
    needed to maintain the proper image data range. If False, clipping
    is not applied, and the output may extend beyond the range [-1, 1].
mean : float, optional
    Mean of random distribution. Used in ’gaussian’ and ’speckle’.
    Default : 0.
var : float, optional
    Variance of random distribution. Used in ’gaussian’ and ’speckle’.
    Note: variance = (standard deviation) ** 2. Default : 0.01
local_vars : ndarray, optional
    Array of positive floats, same shape as ‘image‘, defining the local
    variance at every image point. Used in ’localvar’.
amount : float, optional
    Proportion of image pixels to replace with noise on range [0, 1].
    Used in ’salt’, ’pepper’, and ’salt & pepper’. Default : 0.05
salt_vs_pepper : float, optional
    Proportion of salt vs. pepper noise for ’s&p’ on range [0, 1].
    Higher values represent more salt. Default : 0.5 (equal amounts)
Returns
-------
out : ndarray
    Output floating-point image data on range [0, 1] or [-1, 1] if the
    input ‘image‘ was unsigned or signed, respectively.
Notes
-----
Speckle, Poisson, Localvar, and Gaussian noise may generate noise outside
the valid image range. The default is to clip (not alias) these values,
but they may be preserved by setting ‘clip=False‘. Note that in this case
the output may contain values outside the ranges [0, 1] or [-1, 1].
Use this option with care.
Because of the prevalence of exclusively positive floating-point images in
intermediate calculations, it is not possible to intuit if an input is
signed based on dtype alone. Instead, negative values are explicitly
searched for. Only if found does this function assume signed input.
Unexpected results only occur in rare, poorly exposes cases (e.g. if all
values are above 50 percent gray in a signed ‘image‘). In this event,
manually scaling the input to the positive domain will solve the problem.
```

```
The Poisson distribution is only defined for positive integers. To apply
this noise type, the number of unique values in the image is found and
the next round power of two is used to scale up the floating-point result,
after which it is scaled back down to the floating-point image range.
To generate Poisson noise against a signed image, the signed image is
temporarily converted to an unsigned image in the floating point domain,
Poisson noise is generated, then it is returned to the original range.
"""
mode = mode.lower()

# Detect if a signed image was input
if image.min() < 0:
    low_clip = -1.
else:
    low_clip = 0.

image = img_as_float(image)

rng = np.random.default_rng(seed)

allowedtypes = {
    'gaussian': 'gaussian_values',
    'localvar': 'localvar_values',
    'poisson': 'poisson_values',
    'salt': 'sp_values',
    'pepper': 'sp_values',
    's&p': 's&p_values',
    'speckle': 'gaussian_values'}

kwdefaults = {
    'mean': 0.,
    'var': 0.05,
    'amount': 0.05,
    'salt_vs_pepper': 0.5,
    'local_vars': np.zeros_like(image) + 0.01}

allowedkwargs = {
    'gaussian_values': ['mean', 'var'],
    'localvar_values': ['local_vars'],
    'sp_values': ['amount'],
    's&p_values': ['amount', 'salt_vs_pepper'],
    'poisson_values': []}

for key in kwargs:
    if key not in allowedkwargs[allowedtypes[mode]]:
        raise ValueError('%s keyword not in allowed keywords %s' %
```

```
                        (key, allowedkwargs[allowedtypes[mode]]))

# Set kwarg defaults
for kw in allowedkwargs[allowedtypes[mode]]:
    kwargs.setdefault(kw, kwdefaults[kw])

if mode == 'gaussian':
    noise = rng.normal(kwargs['mean'], kwargs['var'] ** 0.5, image.shape)
    out = image + noise

elif mode == 'localvar':
    # Ensure local variance input is correct
    if (kwargs['local_vars'] <= 0).any():
        raise ValueError('All values of `local_vars` must be > 0.')

    # Safe shortcut usage broadcasts kwargs['local_vars'] as a ufunc
    out = image + rng.normal(0, kwargs['local_vars'] ** 0.5)

elif mode == 'poisson':
    # Determine unique values in image & calculate the next power of two
    vals = len(np.unique(image))
    vals = 2 ** np.ceil(np.log2(vals))

    # Ensure image is exclusively positive
    if low_clip == -1.:
        old_max = image.max()
        image = (image + 1.) / (old_max + 1.)

    # Generating noise for each unique value in image.
    out = rng.poisson(image * vals) / float(vals)

    # Return image to original range if input was signed
    if low_clip == -1.:
        out = out * (old_max + 1.) - 1.

elif mode == 'salt':
    # Re-call function with mode='s&p' and p=1 (all salt noise)
    out = random_noise(image, mode='s&p', seed=rng,
                       amount=kwargs['amount'], salt_vs_pepper=1.)

elif mode == 'pepper':
    # Re-call function with mode='s&p' and p=1 (all pepper noise)
    out = random_noise(image, mode='s&p', seed=rng,
                       amount=kwargs['amount'], salt_vs_pepper=0.)

elif mode == 's&p':
```

```
        out = image.copy()
        p = kwargs['amount']
        q = kwargs['salt_vs_pepper']
        flipped = _bernoulli(p, image.shape, random_state=rng)
        salted = _bernoulli(q, image.shape, random_state=rng)
        peppered = ~salted
        out[flipped & salted] = 1
        out[flipped & peppered] = low_clip

    elif mode == 'speckle':
        noise = rng.normal(kwargs['mean'], kwargs['var'] ** 0.5, image.shape)
        out = image + image * noise

    # Clip back to original range, if necessary
    if clip:
        out = np.clip(out, low_clip, 1.0)

    return out
a=random_noise(img, mode='gaussian', seed=None, clip=True)
cv2.imshow('a',a)
cv2.imwrite('added_noises.jpg',255*a)
cv2.waitKey(0)
```

4.2 Applying box filter

```
    ## box filter
#reference https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgpro
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread("added_noises.jpg")
img2=cv2.imread("DSCN0482-001.jpg")
def box(image):
    kernel = np.ones((7,7),np.float32)/49
    dst = cv2.filter2D(image,-1,kernel)
    plt.subplot(121), plt.imshow(image), plt.title('noise')
    plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(dst), plt.title('Averaging')
    plt.xticks([]), plt.yticks([])
    return dst
box(img)
plt.show()
cv2.imwrite('boxfilter.jpg',
            box(img))
```

4.3 Applying median filter

```
    #Video Playlist: https://www.youtube.com/playlist?list=PLHae9ggVvqPgyRQQOtENr6hKOm1UquGa
### median filter
"""
Spyder Editor

cv2.medianBlur - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_img
skimage.filters.median - https://scikit-image.org/docs/dev/api/skimage.filters.html#skimage.

See how median is much better at cleaning salt and pepper noise compared to Gaussian
"""
import cv2
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import median


#Needs 8 bit, not float.
img_gaussian_noise = cv2.imread("added_noises.jpg", 0)
original=cv2.imread("DSCN0479-001.jpg",0)
img = img_gaussian_noise
def median(image):
    median_using_cv2 = cv2.medianBlur(image, 3)
    return median_using_cv2



cv2.imshow("original",original)
cv2.imshow("before filter", img)
cv2.imshow('cv2median.jpg',median(img))
cv2.imwrite('cv2median.jpg',
           median(img))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

4.4 MSE PSNR

```
    # import the necessary packages
#reference: https://www.geeksforgeeks.org/python-peak-signal-to-noise-ratio-psnr/
#reference: https://pyimagesearch.com/2014/09/15/python-compare-two-images/
import math
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
```

```
import numpy as np
import cv2


def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

def psnr(img1, img2):
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return 100
    PIXEL_MAX = 255.0
    return print("PSNR=",20 * math.log10(PIXEL_MAX / math.sqrt(mse)))



def compare_images(imageA, imageB, title):
    # compute the mean squared error and structural similarity
    # index for the images
    m = mse(imageA, imageB)
    s = ssim(imageA, imageB)
    # setup the figure
    fig = plt.figure(title)
    plt.suptitle("MSE: %.2f, SSIM: %.2f" % (m, s))
    # show first image
    ax = fig.add_subplot(1, 2, 1)
    plt.imshow(imageA, cmap=plt.cm.gray)
    plt.axis("off")
    # show the second image
    ax = fig.add_subplot(1, 2, 2)
    plt.imshow(imageB, cmap=plt.cm.gray)
    plt.axis("off")
    # show the images
    plt.show()

    # load the images -- the original, the original + contrast,
    # and the original + photoshop
original = cv2.imread("DSCN0479-001.jpg")
```

```
filterd = cv2.imread("cv2median.jpg")
boxfilter=cv2.imread("boxfilter.jpg")
    # convert the images to grayscale
original = cv2.cvtColor(original, cv2.COLOR_BGR2GRAY)
filterd = cv2.cvtColor(filterd, cv2.COLOR_BGR2GRAY)
boxfilter=cv2.cvtColor(boxfilter,cv2.COLOR_BGR2GRAY)

    # initialize the figure
fig = plt.figure("Images")
images = ("Original", original), ("filtered", filterd),("boxfilter", boxfilter)
    # loop over the images
for (i, (name, image)) in enumerate(images):
        # show the image
    ax = fig.add_subplot(1, 3, i + 1)
    ax.set_title(name)
    plt.imshow(image, cmap=plt.cm.gray)
    plt.axis("off")
    # show the figure
plt.show()
    # compare the images
compare_images(original, filterd, "Original vs. median_filtered")
compare_images(original,boxfilter , "Original vs. box_filter")
psnr(original,filterd)
psnr(original,boxfilter)
```

4.5 salt and pepper noise

```
#refrence https://scikit-image.org/docs/stable/api/skimage.util.html#random-noise
import cv2
import numpy as np
#from .dtype import img_as_float
from skimage.util import img_as_float
__all__ = ['random_noise']
img = cv2.imread('DSCN0479-001.jpg',
                 cv2.IMREAD_GRAYSCALE)

def _bernoulli(p, shape, *, random_state):
    """
    Bernoulli trials at a given probability of a given size.
    This function is meant as a lower-memory alternative to calls such as
    `np.random.choice([True, False], size=image.shape, p=[p, 1-p])`.
    While `np.random.choice` can handle many classes, for the 2-class case
    (Bernoulli trials), this function is much more efficient.
    Parameters
    ----------
```

```
    p : float
        The probability that any given trial returns `True`.
    shape : int or tuple of ints
        The shape of the ndarray to return.
    seed : `numpy.random.Generator`
        ``Generator`` instance.
    Returns
    -------
    out : ndarray[bool]
        The results of Bernoulli trials in the given `size` where success
        occurs with probability `p`.
    """
    if p == 0:
        return np.zeros(shape, dtype=bool)
    if p == 1:
        return np.ones(shape, dtype=bool)
    return random_state.random(shape) <= p


def random_noise(image, mode='gaussian', seed=None, clip=True, **kwargs):
    """
    Function to add random noise of various types to a floating-point image.
    Parameters
    ----------
    image : ndarray
        Input image data. Will be converted to float.
    mode : str, optional
        One of the following strings, selecting the type of noise to add:
        - 'gaussian'  Gaussian-distributed additive noise.
        - 'localvar'  Gaussian-distributed additive noise, with specified
                      local variance at each point of `image`.
        - 'poisson'   Poisson-distributed noise generated from the data.
        - 'salt'      Replaces random pixels with 1.
        - 'pepper'    Replaces random pixels with 0 (for unsigned images) or
                      -1 (for signed images).
        - 's&p'       Replaces random pixels with either 1 or `low_val`, where
                      `low_val` is 0 for unsigned images or -1 for signed
                      images.
        - 'speckle'   Multiplicative noise using out = image + n*image, where
                      n is Gaussian noise with specified mean & variance.
    seed : {None, int, `numpy.random.Generator`}, optional
        If `seed` is None the `numpy.random.Generator` singleton is
        used.
        If `seed` is an int, a new ``Generator`` instance is used,
        seeded with `seed`.
        If `seed` is already a ``Generator`` instance then that
```

```
    instance is used.
    This will set the random seed before generating noise,
    for valid pseudo-random comparisons.
clip : bool, optional
    If True (default), the output will be clipped after noise applied
    for modes `'speckle'`, `'poisson'`, and `'gaussian'`. This is
    needed to maintain the proper image data range. If False, clipping
    is not applied, and the output may extend beyond the range [-1, 1].
mean : float, optional
    Mean of random distribution. Used in 'gaussian' and 'speckle'.
    Default : 0.
var : float, optional
    Variance of random distribution. Used in 'gaussian' and 'speckle'.
    Note: variance = (standard deviation) ** 2. Default : 0.01
local_vars : ndarray, optional
    Array of positive floats, same shape as `image`, defining the local
    variance at every image point. Used in 'localvar'.
amount : float, optional
    Proportion of image pixels to replace with noise on range [0, 1].
    Used in 'salt', 'pepper', and 'salt & pepper'. Default : 0.05
salt_vs_pepper : float, optional
    Proportion of salt vs. pepper noise for 's&p' on range [0, 1].
    Higher values represent more salt. Default : 0.5 (equal amounts)
Returns
-------
out : ndarray
    Output floating-point image data on range [0, 1] or [-1, 1] if the
    input `image` was unsigned or signed, respectively.
Notes
-----
Speckle, Poisson, Localvar, and Gaussian noise may generate noise outside
the valid image range. The default is to clip (not alias) these values,
but they may be preserved by setting `clip=False`. Note that in this case
the output may contain values outside the ranges [0, 1] or [-1, 1].
Use this option with care.
Because of the prevalence of exclusively positive floating-point images in
intermediate calculations, it is not possible to intuit if an input is
signed based on dtype alone. Instead, negative values are explicitly
searched for. Only if found does this function assume signed input.
Unexpected results only occur in rare, poorly exposes cases (e.g. if all
values are above 50 percent gray in a signed `image`). In this event,
manually scaling the input to the positive domain will solve the problem.
The Poisson distribution is only defined for positive integers. To apply
this noise type, the number of unique values in the image is found and
the next round power of two is used to scale up the floating-point result,
after which it is scaled back down to the floating-point image range.
```

```
To generate Poisson noise against a signed image, the signed image is
temporarily converted to an unsigned image in the floating point domain,
Poisson noise is generated, then it is returned to the original range.
"""
mode = mode.lower()

# Detect if a signed image was input
if image.min() < 0:
    low_clip = -1.
else:
    low_clip = 0.

image = img_as_float(image)

rng = np.random.default_rng(seed)

allowedtypes = {
    'gaussian': 'gaussian_values',
    'localvar': 'localvar_values',
    'poisson': 'poisson_values',
    'salt': 'sp_values',
    'pepper': 'sp_values',
    's&p': 's&p_values',
    'speckle': 'gaussian_values'}

kwdefaults = {
    'mean': 0.,
    'var': 0.05,
    'amount': 0.05,
    'salt_vs_pepper': 0.5,
    'local_vars': np.zeros_like(image) + 0.01}

allowedkwargs = {
    'gaussian_values': ['mean', 'var'],
    'localvar_values': ['local_vars'],
    'sp_values': ['amount'],
    's&p_values': ['amount', 'salt_vs_pepper'],
    'poisson_values': []}

for key in kwargs:
    if key not in allowedkwargs[allowedtypes[mode]]:
        raise ValueError('%s keyword not in allowed keywords %s' %
                         (key, allowedkwargs[allowedtypes[mode]]))

# Set kwarg defaults
for kw in allowedkwargs[allowedtypes[mode]]:
```

```
    kwargs.setdefault(kw, kwdefaults[kw])

if mode == 'gaussian':
    noise = rng.normal(kwargs['mean'], kwargs['var'] ** 0.5, image.shape)
    out = image + noise

elif mode == 'localvar':
    # Ensure local variance input is correct
    if (kwargs['local_vars'] <= 0).any():
        raise ValueError('All values of `local_vars` must be > 0.')

    # Safe shortcut usage broadcasts kwargs['local_vars'] as a ufunc
    out = image + rng.normal(0, kwargs['local_vars'] ** 0.5)

elif mode == 'poisson':
    # Determine unique values in image & calculate the next power of two
    vals = len(np.unique(image))
    vals = 2 ** np.ceil(np.log2(vals))

    # Ensure image is exclusively positive
    if low_clip == -1.:
        old_max = image.max()
        image = (image + 1.) / (old_max + 1.)

    # Generating noise for each unique value in image.
    out = rng.poisson(image * vals) / float(vals)

    # Return image to original range if input was signed
    if low_clip == -1.:
        out = out * (old_max + 1.) - 1.

elif mode == 'salt':
    # Re-call function with mode='s&p' and p=1 (all salt noise)
    out = random_noise(image, mode='s&p', seed=rng,
                       amount=kwargs['amount'], salt_vs_pepper=1.)

elif mode == 'pepper':
    # Re-call function with mode='s&p' and p=1 (all pepper noise)
    out = random_noise(image, mode='s&p', seed=rng,
                       amount=kwargs['amount'], salt_vs_pepper=0.)

elif mode == 's&p':
    out = image.copy()
    p = kwargs['amount']
    q = kwargs['salt_vs_pepper']
    flipped = _bernoulli(p, image.shape, random_state=rng)
```

```
        salted = _bernoulli(q, image.shape, random_state=rng)
        peppered = ~salted
        out[flipped & salted] = 1
        out[flipped & peppered] = low_clip

    elif mode == 'speckle':
        noise = rng.normal(kwargs['mean'], kwargs['var'] ** 0.5, image.shape)
        out = image + image * noise

    # Clip back to original range, if necessary
    if clip:
        out = np.clip(out, low_clip, 1.0)

    return out
noise_img = random_noise(img, mode='s&p',amount=0.05)
cv2.imshow('a',noise_img)
cv2.imwrite('added_noises.jpg',255*noise_img)
cv2.waitKey(0)
```

4.6 median filter

```
    #Video Playlist: https://www.youtube.com/playlist?list=PLHae9ggVvqPgyRQQOtENr6hK0m1UquGa
### median filter
"""
Spyder Editor

cv2.medianBlur - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_img
skimage.filters.median - https://scikit-image.org/docs/dev/api/skimage.filters.html#skimage.

See how median is much better at cleaning salt and pepper noise compared to Gaussian
"""
import cv2
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import numpy as np
from skimage.filters import median


#Needs 8 bit, not float.
img_salt_pepper_noise = cv2.imread("added_noises.jpg", 0)
ISO=cv2.imread("DSCN0482-001.jpg",0)
original=cv2.imread("DSCN0479-001.jpg",0)
img = img_salt_pepper_noise
img_ISO= ISO
```

```
def median(image):
    median_using_cv2 = cv2.medianBlur(image, 3)
    return median_using_cv2



cv2.imshow("original",original)
cv2.imshow("before filter", img)
cv2.imshow('cv2median.jpg',median(img))
cv2.imshow("ISO",median(img_ISO))
cv2.imwrite('cv2median.jpg',
           median(img))

cv2.waitKey(0)
cv2.destroyAllWindows()
```

4.7 box filter

```
    ## box filter
#reference https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgpro
import cv2
import numpy as np
from matplotlib import pyplot as plt

img = cv2.imread("added_noises.jpg")
img2=cv2.imread("DSCN0482-001.jpg",cv2.COLOR_BGR2GRAY)
def box(image):
    kernel = np.ones((3,3),np.float32)/9
    dst = cv2.filter2D(image,-1,kernel)
    plt.subplot(121), plt.imshow(image), plt.title('Original')
    plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(dst), plt.title('Averaging')
    plt.xticks([]), plt.yticks([])
    return dst
box(img)
plt.show()
box(img2)
plt.show()
cv2.imwrite('boxfilter_ISO.jpg',
           box(img2))

cv2.imwrite('boxfilter_addedNoise.jpg',
           box(img))
```

4.8 MSE PSNR

```
    # import the necessary packages
#reference: https://www.geeksforgeeks.org/python-peak-signal-to-noise-ratio-psnr/
#reference: https://pyimagesearch.com/2014/09/15/python-compare-two-images/
import math

from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import numpy as np
import cv2


def mse(imageA, imageB):
    # the 'Mean Squared Error' between the two images is the
    # sum of the squared difference between the two images;
    # NOTE: the two images must have the same dimension
    err = np.sum((imageA.astype("float") - imageB.astype("float")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])

    # return the MSE, the lower the error, the more "similar"
    # the two images are
    return err

def psnr(img1, img2):
    mse = np.mean((img1 - img2) ** 2)
    if mse == 0:
        return 100
    PIXEL_MAX = 255.0
    return print("PSNR=",20 * math.log10(PIXEL_MAX / math.sqrt(mse)))



def compare_images(imageA, imageB, title):
    # compute the mean squared error and structural similarity
    # index for the images
    m = mse(imageA, imageB)
    s = ssim(imageA, imageB)
    # setup the figure
    fig = plt.figure(title)
    plt.suptitle("MSE: %.2f, SSIM: %.2f" % (m, s))
    # show first image
    ax = fig.add_subplot(1, 2, 1)
    plt.imshow(imageA, cmap=plt.cm.gray)
    plt.axis("off")
    # show the second image
```

```
    ax = fig.add_subplot(1, 2, 2)
    plt.imshow(imageB, cmap=plt.cm.gray)
    plt.axis("off")
    # show the images
    plt.show()

    # load the images -- the original, the original + contrast,
    # and the original + photoshop
original = cv2.imread("DSCN0479-001.jpg")
filterd = cv2.imread("cv2median.jpg")
boxfilter=cv2.imread("boxfilter_addedNoise.jpg")
    # convert the images to grayscale
original = cv2.cvtColor(original, cv2.COLOR_BGR2GRAY)
filterd = cv2.cvtColor(filterd, cv2.COLOR_BGR2GRAY)
boxfilter=cv2.cvtColor(boxfilter,cv2.COLOR_BGR2GRAY)

    # initialize the figure
fig = plt.figure("Images")
images = ("Original", original), ("filtered", filterd),("boxfilter", boxfilter)
    # loop over the images
for (i, (name, image)) in enumerate(images):
        # show the image
    ax = fig.add_subplot(1, 3, i + 1)
    ax.set_title(name)
    plt.imshow(image, cmap=plt.cm.gray)
    plt.axis("off")
    # show the figure
plt.show()
    # compare the images
compare_images(original, filterd, "Original vs. median_filtered")
compare_images(original,boxfilter , "Original vs. box_filter")
psnr(original,filterd)
psnr(original,boxfilter)
```