Directed Study Report

Saori Shigehisa

Dr. Sachin Meena

May 1, 2022

Introduction

Since I have been working in the electrical/mechanical engineering field, I wanted to explore various other fields such as data science to not only improve the breadth of my computer science knowledge but the depth, so I asked Dr. Meena to lead my directed study. During the semester, 3 different phases occurred: learning the basic ideas of machine learning, learning the basic ideas of image processing, and practical work using Python, Jupyter, and Anaconda.

Procedures

1st Phase Report:

In the 1st phase, to start understanding AI (deep learning), Dr. Meena suggested that I start by understanding machine learning. To become familiar with these fundamentals, I started reading a textbook called, “An Introduction to Statistical Learning - with Applications in R” by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. In this phase, I will report the technical terms that I learned from the textbook.

Prediction Equation:

To start the field of machine learning, I learned the most important equation called the “prediction equation”:

$$Y = f(X) + \epsilon$$

$\hat{f}$: estimation of f

$\hat{Y}$: result of prediction of Y.
(prediction)

f: mapping (fixed)

X: feature (age, blood sample - etc)

Y: category variable, output.

According to the textbook, "the input variables are typically denoted using the symbol X, with a subscript to distinguish them. So X1 might be the TV budget, X2 the radio budget, and X3 the newspaper budget. The inputs go by different names, such as predictors, independent variables, features, or sometimes just variables. The output variable—in this case, sales—is often called the response or dependent variable, and is typically denoted using the symbol Y. Throughout this book, we will use all of these terms interchangeably."

This figure from the textbook let me easily understand how the equation works:

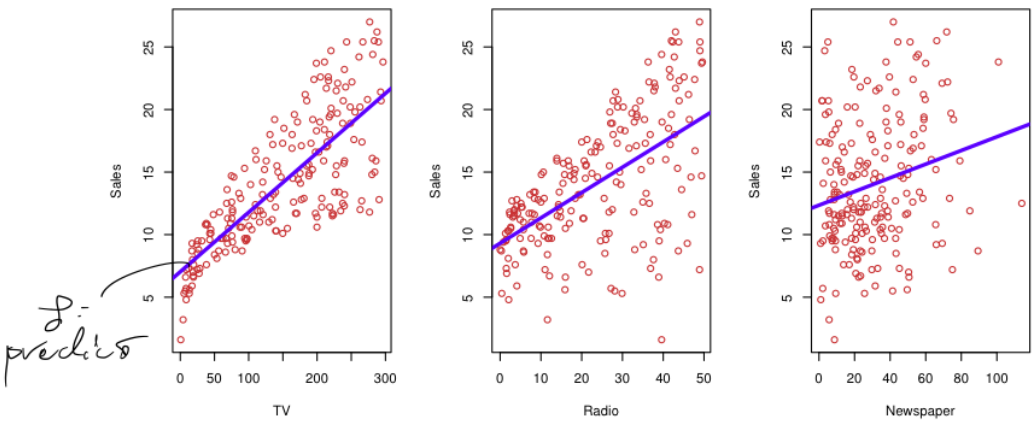


FIGURE 2.1. *The* `Advertising` *data set. The plot displays* `sales`*, in thousands of units, as a function of* `TV`*,* `radio`*, and* `newspaper` *budgets, in thousands of dollars, for* 200 *different markets. In each plot we show the simple least squares fit of* `sales` *to that variable, as described in Chapter 3. In other words, each blue line represents a simple model that can be used to predict* `sales` *using* `TV`*,* `radio`*, and* `newspaper`*, respectively.*

feacher is X. (information)

curve fit

features

More generally, suppose that we observe a quantitative response Y and p different predictors, $X_1, X_2, \ldots, X_p$. We assume that there is some relationship between Y and $X = (X_1, X_2, \ldots, X_p)$, which can be written in the very general form

ages

prediction

categoly variable — output

$$Y = f(X) + \epsilon. \tag{2.1}$$

fixed — mapping for X&Y.

Here f is some fixed but unknown function of $X_1, \ldots, X_p$, and ϵ is a random *error term*, which is independent of X and has mean zero. In this formulation, f represents the *systematic* information that X provides about Y. error term systematic

As another example, consider the left-hand panel of Figure 2.2, a plot of

Irreducible Errors and Reducible Errors:

$$\begin{aligned} \mathrm{E}(Y-\hat{Y})^2 &= \mathrm{E}[f(X)+\epsilon-\hat{f}(X)]^2 \\ &= \underbrace{[f(X)-\hat{f}(X)]^2}_{\text{Reducible}} + \underbrace{\mathrm{Var}(\epsilon)}_{\text{Irreducible}} , \end{aligned}$$

average / expected value

- The accuracy of $\hat{Y}$ as a prediction of Y (category variable) depends on errors. These are called "irreducible errors" and "reducible errors."
- The reason why we cannot reduce "irreducible errors" is because no matter how well we estimate f, we cannot reduce the error introduced by ε.

Training and Inference:

- Difference between "Training" and "Inference"

In training, we use neural network
ex) Image: is the picture a cat? No, Yes — learning

inference, we use prediction.

Parametric and Non-Parametric:

Parametric V.S. Non-parametric
machine learning algorithms

Parametric machine learning algorithms
are any machine learning algorithms
if it learns using a predefined
mapped function.

No matter how much amount of data
you're gonna throw at this algorithms,
it will not change how many parameters
it needs to learn to predict or classify.

$$\boxed{Y = B_0 + B_1(X)}$$

this is called linear regression

all you have to do is estimate the
value for the coefficients betas.

one of the methods of parametric ML algorithms

Linear Regression : $\boxed{Y = B_0 + B_1(X)}$

If the equation does not follow a linear distribution, then

you can use 2 methods:

① use transformation like logarithmic / square

In order to estimate close enough or no matter what we do, the data will not following a linear distribution, in that case, the assumption is wrong.

(your prediction is out)

Parametric:

Parametric ML algorithms

methods
- regression
- LDA
- Perceptron

this is where deep learning started.

advantages:
- simple
- fast
- less data

disadvantages:
- constrainted (stick to the assumptions & its parameters)
- limited complexity
- poor fit

Non-Parametric:

Non-Parametric ML algorithms

methods
- K-Nearest Neighbours (KNN)
- ⋮

Ex. of Non-Parametric ML algorithms
- Decision Trees
- Naive Bayes
- SVM
- Neural Networks

advantages:
- flexibility
- Power
- Performance

disadvantages:
- More data
- slower
- overfitting

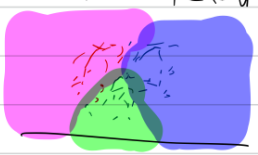
Supervised Learning and Unsupervised Learning:

The following section explains the difference between Supervised Learning and Unsupervised Learning.

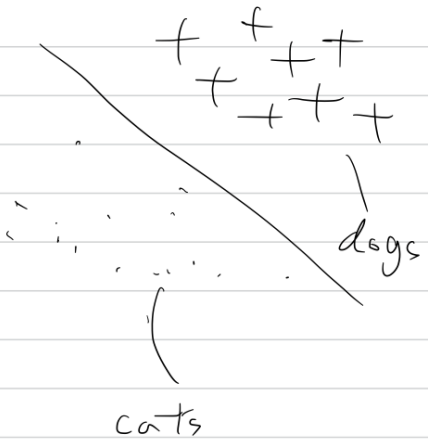
Supervised Learning:

Supervised learning has 2 types

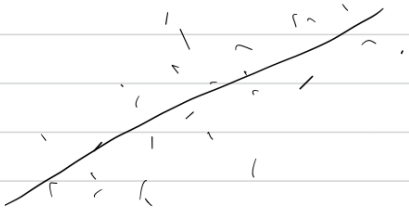
supervised machine learning model produce classification regions.



- classification



- regression



Unsupervised Learning:

Unsupervised learning

this model work on its own to discover information that may not be visible to the human eye.

Ex) groups / clusters, perform density estimation,

dimensionality reduction

Un-supervised learning

- less tests
- less models
- less control

difference b/w supervised learning / unsupervised learning

2nd Phase Report:

Because I have never touched anything practical in the field of AI, Dr. Meena told me to start by reading the textbook. In the first phase, I learned basic knowledge about machine learning. To begin, Dr. Meena told me to start by understanding digital image processing. In this phase, I was tasked to understand the basics of image processing. By reading the textbook, called “Digital Image Processing" by Rafael C. Gonzalez and Richard E. Woods, I became familiar with technical words, their definition, and their innerworkings. I will report the technical words and how they work here:

Pixel:

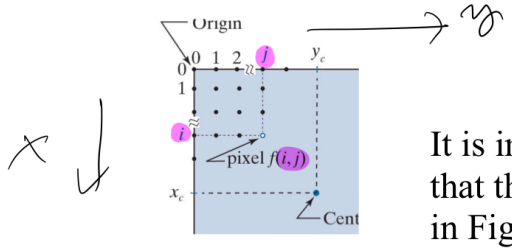
During the session, I learned that a Pixel is a digital image and its element. It can be often denoted as F(i,j) to indicate the specific position.

the third representation is an array (matrix) composed of the numerical values of f(x, y). This is the representation used for computer processing. In equation form, we write the representation of an

Coordinate convention used to represent digital images.

" pixel throughout the book to denote a digital image and its elements.

As mentioned earlier, we generally use f(i, j) when referring to a pixel with coordinates (i, j).



It is important to note that the representation in Fig. 2.19 , in which the positive x-axis extends downward and the positive y-axis extends to the right

RGB channel:

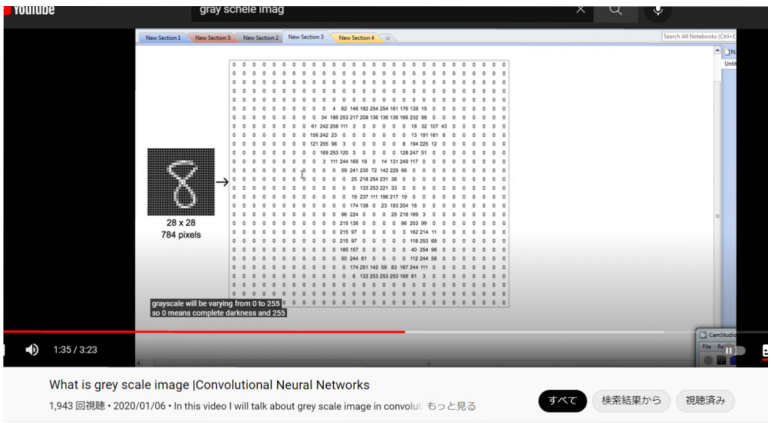
when you get close to the screen, each pixel emits 3 colors. Red, Green, and Blue.

(Even it's black & white colored screen, each pixel emits Red, Green, and Blue.)

Gray Scale Image:

grayscale will be made with numbers from 0 to 255.

0 is the darkest and as the number increases, it will get close to white.



the gray schale image is made with pixels.

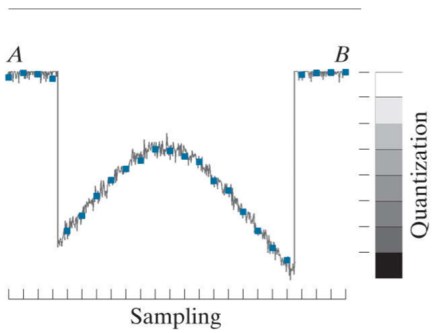
Sampling and Quantization:

there is a continuous image I that we want to convert to digital form. this is continuous respect to the x- & y- coordinates & also in amplitude.

To digitize it, we need to sample the function in both coordinates & in amplitude.

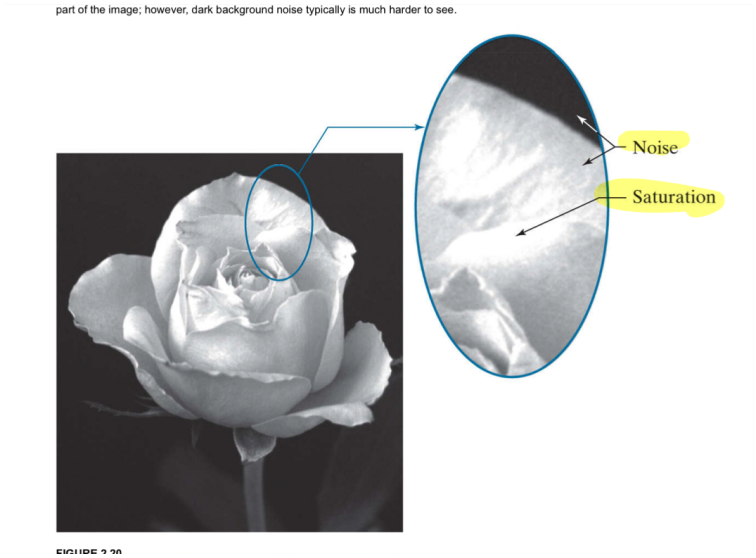
Digitizing the coordinate values is called <u>sampling</u>.

digitizing the amplitude values is called "<u>quantization</u>."



Noise and Saturation:

The dark background is noisier, but the noise is difficult to see.



Dynamic range establishes:

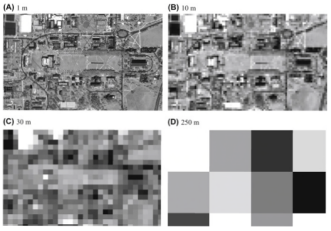
The dynamic range establishes the lowest and highest intensity levels that a system can represent and, consequently, that an image can have.

Linear Indexing:

Another type of indexing used extensively in programming image processing algorithms is linear indexing, which consists of a 1-D string of nonnegative integers based on computing offsets from coordinates (0, 0).

Spatial Resolution:

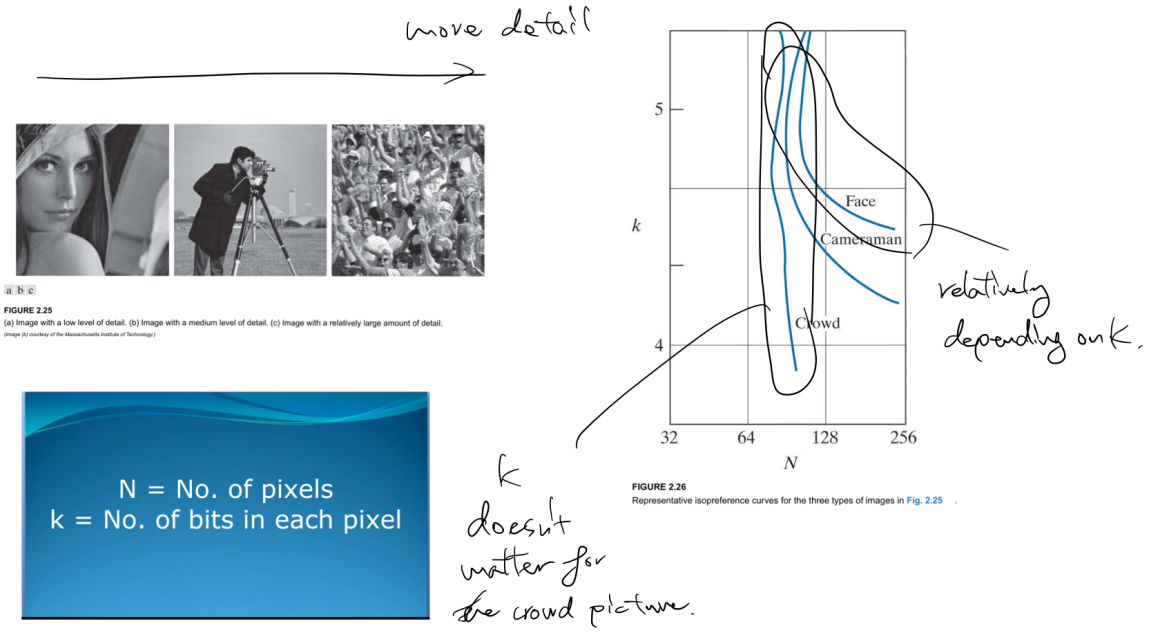
Spatial resolution is a measure of the smallest object that can be resolved by the sensor.



Intensity Resolution:

Intensity level resolution is given in terms of the number of bits used to store each intensity level. Higher the intensity level, higher the image quality & higher the size of image.

Isopreference curves:



To deeply understand "Isopreference curves," this youtube video helped me a lot: [Isopreference curves](#).

Digital Image Interpolation:

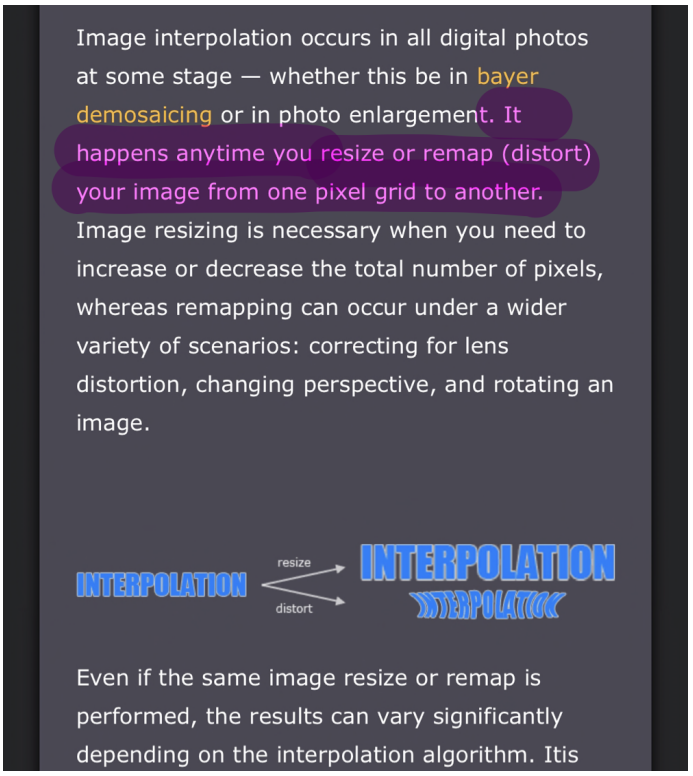


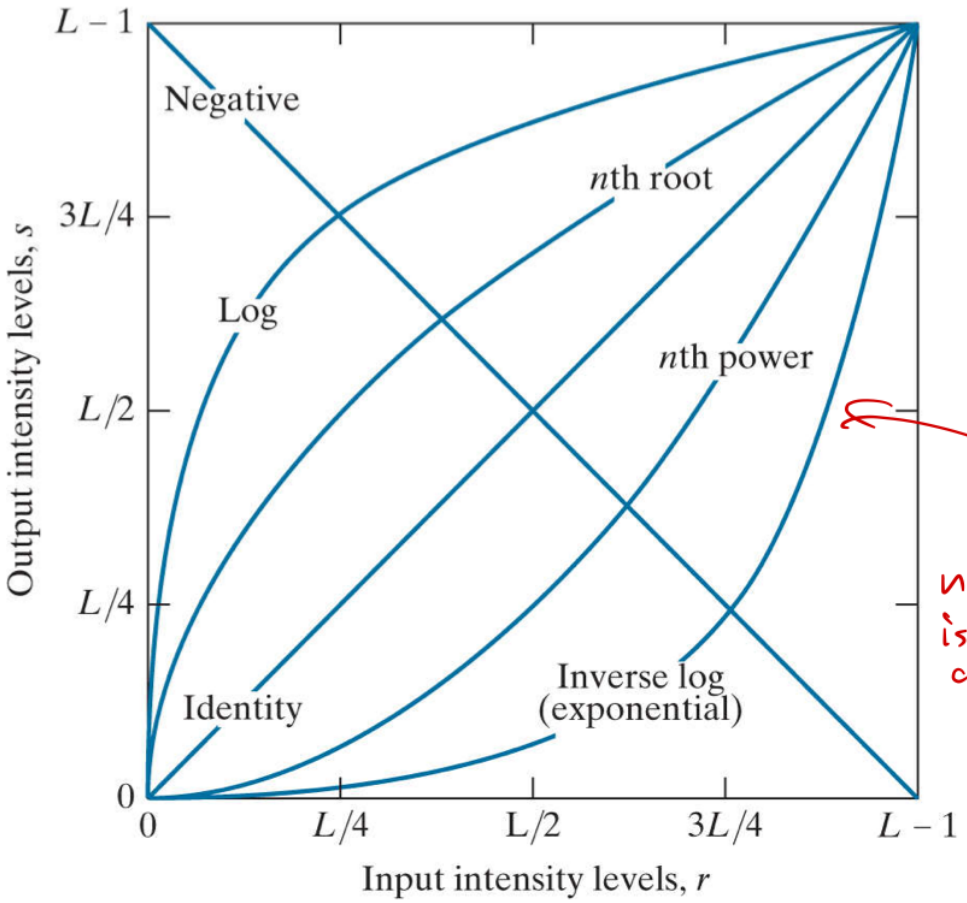
Image Negatives:

Image Negatives

This is used when the background is darker (black)

The negative of an image with intensity levels in the range $[0, L-1]$ is obtained by using the negative transformation function shown in Fig. 3.3 , which has the form:

$$s = L - 1 - r \tag{3-3}$$

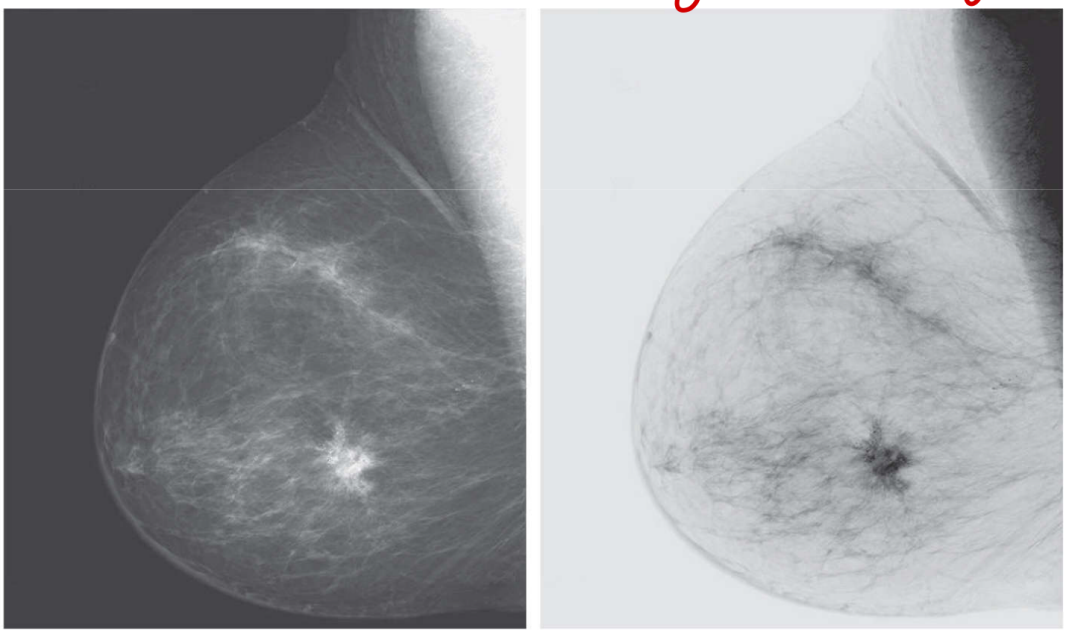


point of "Image negatives" is this curve.

FIGURE 3.3
Some basic intensity transformation functions. Each curve was scaled *independently* so that all curves would fit in the same graph. Our interest here is on the *shapes* of the curves, not on their relative values.

Reversing the intensity levels of a digital image in this manner produces the equivalent of a photographic negative. This type of processing is used, for example, in enhancing white or gray detail embedded in dark regions of an image, especially when the black areas are dominant in size. Figure 3.4 shows an example. The original image is a digital mammogram showing a small lesion. Despite the fact that the visual content is the same in both images, some viewers find it easier to analyze the fine details of the breast tissue using the negative image.

Negative Image



Gamma Transform:

$$Gamma\ Transform \neq Gamma\ Correction$$

(Gamma transform)

Example 3.1: Contrast enhancement using power-law intensity transformations.

In addition to gamma correction, power-law transformations are useful for general-purpose contrast manipulation. **Figure 3.8(a)** shows a magnetic resonance image (MRI) of a human upper thoracic spine with a fracture dislocation. The fracture is visible in the region highlighted by the circle. Because the image is predominantly dark, an expansion of intensity levels is desirable. This can be accomplished using a power-law transformation with a fractional exponent. The other images shown in the figure were obtained by processing **Fig. 3.8(a)** with the power-law transformation function of **Eq. (3-5)**. The values of gamma corresponding to images (b) through (d) are 0.6, 0.4, and 0.3, respectively ($c = 1$ in all cases). Observe that as gamma decreased from 0.6 to 0.4, more detail became visible. A further decrease of gamma to 0.3 enhanced a little more detail in the background, but began to reduce contrast to the point where the image started to have a very slight "washed-out" appearance, especially in the background. The best enhancement in terms of contrast and discernible detail was obtained with $\gamma = 0.4$. A value of $\gamma = 0.3$ is an approximate limit below which contrast in this particular image would be reduced to an unacceptable level.

a b
c d

gamma correction
&
gamma transform

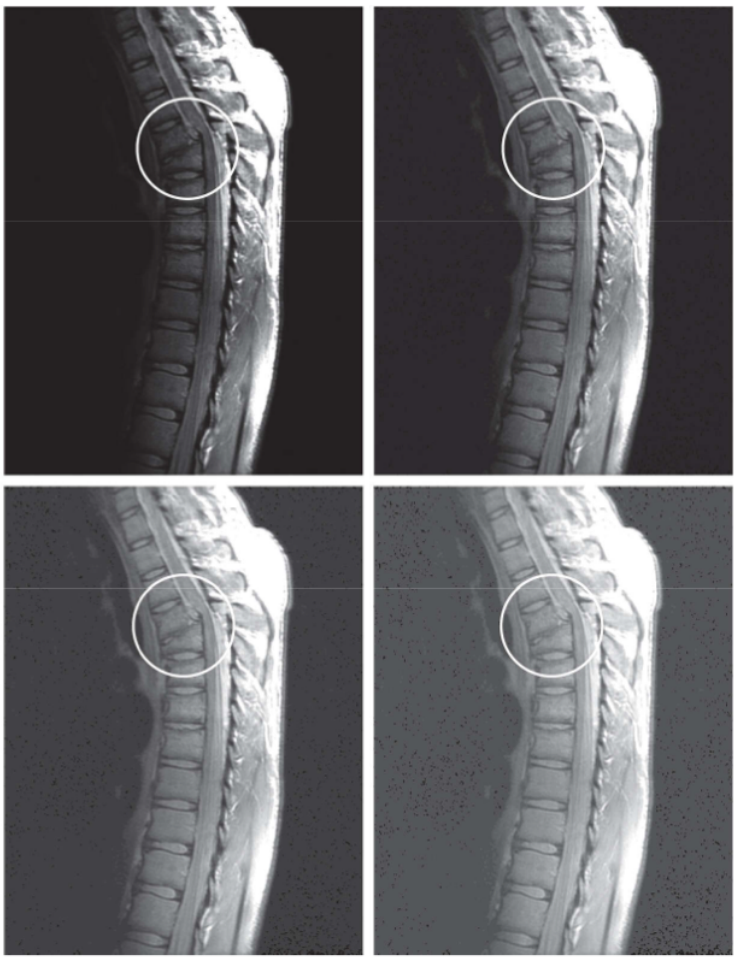


FIGURE 3.8

(a) Magnetic resonance image (MRI) of a fractured human spine (the region of the fracture is enclosed by the circle). (b)–(d) Results of applying the transformation in **Eq. (3-5)** with $c = 1$ and $\gamma = 0.6,\ 0.4,$ and 0.3, respectively. (Original image courtesy of Dr. David R. Pickens, Department of Radiology and Radiological Sciences, Vanderbilt University Medical Center.)

Histogram:

Ex of histogram equalization

$$s_0 = T(r_0) = 7\sum_{j=0}^{0} p_r(r_j) = 7p_r(r_0) = 1.33$$

TABLE 3.1

Intensity distribution and histogram values for a 3-bit, 64×64 digital image.

r_k	n_k	$p_r(r_k) = n_k/MN$
$r_0 = 0$	790	0.19
$r_1 = 1$	1023	0.25
$r_2 = 2$	850	0.21
$r_3 = 3$	656	0.16
$r_4 = 4$	329	0.08
$r_5 = 5$	245	0.06
$r_6 = 6$	122	0.03
$r_7 = 7$	81	0.02

Similarly, $s_1 = T(r_1) = 3.08$ $s_2 = 4.55$ $s_3 = 5.67$ $s_4 = 6.23$ $s_5 = 6.65$ $s_6 = 6.86$ and $s_7 = 7.00$ This transformation function has the staircase shape shown in Fig. 3.19(b) .

At this point, the s values are fractional because they were generated by summing probability values, so we round them to their nearest integer values in the range [0, 7]:

1, 3, 5, 6, 7 5 distinct intensity levels

$$s_0 = 1.33 \rightarrow 1 \quad s_2 = 4.55 \rightarrow 5 \quad s_4 = 6.23 \rightarrow 6 \quad s_6 = 6.86 \rightarrow 7$$
$$s_1 = 3.08 \rightarrow 3 \quad s_3 = 5.67 \rightarrow 6 \quad s_5 = 6.65 \rightarrow 7 \quad s_7 = 7.00 \rightarrow 7$$

These are the values of the equalized histogram. Observe that the transformation yielded only five distinct intensity levels. Because $r_0 = 0$ was mapped to $s_0 = 1$, there are 790 pixels in the histogram equalized image with this value (see Table 3.1). Also, there are 1023 pixels with a value of $s_1 = 3$ and 850 pixels with a value of $s_2 = 5$. However, both r_3 and r_4 were mapped to the same value, 6, so there are $(656 + 329) = 985$ pixels in the equalized image with this value. Similarly, there are $(245 + 122 + 81) = 448$ pixels with a value of 7 in the histogram equalized image. Dividing these numbers by $MN = 4096$ yielded the equalized histogram in Fig. 3.19(c) .

3rd Phase Report:

I still struggled to understand these concepts fully, so Dr. Meena suggested I practice python coding. In this last phase, I saw how to read/display images in python using Jupytor and Anaconda. In this phase, I also learned that there is a library in python for image processing. At the end of this phase, I better understood how the previous ideas could be applied.

Basics.py:

To practically start working on the image processor using Python, in the first week of the 3rd Phase, I tried to become familiar with basic python libraries for image processing. I went through another student's work first. Based on his workshop and internet sources, I became more comfortable with basic image processing. For this session, I created a program

that does greyscale, rotates an image, and zooms in/out an image. The followings are the questions and the answers that I came up with during the session:

 # Questions:

1 How can I know the address of the folder that I made on Jupyter?

2 How do you change the order of the bracket -> In[]?

3 What is the point of using both Anaconda and Jupyter?

4 Is Jupyter an environment such as VS?

5 When OpenCV is working and how?

#

 # Answers:

1 C:\Users\Saori Shigehisa\Documents\AIPoject-OpenCV

2 They show how many cells I ran

3 Anaconda is just a bundle that contains an implementation of python with a bunch of useful packages, and Jupyter

notebook is just a convenient platform on anaconda that allows to code

4 Yes, VS is just an IDE, platform, or "an environment."

5 OpenCV is a library and it is imported

The codes/outputs are below:

In [25]:

```
# Importing modules
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
import cv2 # OpenCV
import glob # loading paths

# load all images (pngs)
images = glob.glob('*.jpg') #** means all the pictures of the array, and * means only 1 picture

print(images)

print(type(images))
print(np.shape(images))
print(images[1])
```

```
['gamma_adjusted.jpg', 'NightSightColor.jpg', 'rotated_image.jpg']
<class 'list'>
(3,)
NightSightColor.jpg
```

In [26]:

```
# load as color image
img = cv2.imread(images[1], cv2.IMREAD_COLOR)

# load as grayscale
img_gray = cv2.imread(images[1], cv2.IMREAD_GRAYSCALE)
print()

#print(np.shape(img))
#print(img)

#print(np.shape(img_gray))
#print(img_gray)
#z = np.ones([2,2,5], dtype=bool)
#print(z)
```

In [27]:

```
# load color image with integers
img = cv2.imread(images[1], 1)

# grayscale
img_gray = cv2.imread(images[1], 0)
```

In [29]:

```
# display color and grayscale images
fig, axes = plt.subplots(1, 2, figsize=(4, 4), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original color image")
ax[0].imshow(img[...,::-1], interpolation = "nearest")

ax[1].set_title("Image as grayscale")
ax[1].imshow(img_gray, cmap=plt.cm.Greys_r, interpolation = "nearest")

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()
```

In [30]:

```
# Extra just to show colored actually changes
imgNightSightColor = cv2.imread('NightSightColor.jpg',1)
imgNightSightGrey = cv2.imread('NightSightColor.jpg',0)

print(imgNightSightColor)
```

```
[[[248 181 186]
  [250 183 188]
  [252 185 190]
  ...
  [253 236 239]
  [252 235 238]
  [249 232 235]]

 [[248 181 186]
  [248 181 186]
  [249 182 187]
  ...
  [253 236 239]
  [253 236 239]
  [250 233 236]]

 [[248 181 186]
  [247 180 185]
  [245 178 183]
  ...
  [254 237 240]
  [253 236 239]
  [250 233 236]]

 ...
```

```
 ...

 [[127  87  88]
  [124  84  85]
  [122  82  83]
  ...
  [165 136 152]
  [164 135 151]
  [165 136 152]]

 [[129  87  88]
  [128  86  87]
  [126  85  83]
  ...
  [160 128 145]
  [159 127 144]
  [160 128 145]]

 [[124  82  83]
  [123  81  82]
  [121  80  78]
  ...
  [160 126 143]
  [160 126 143]
  [162 128 145]]]
```

In [31]:

```python
# display color and grayscale images
fig, axes = plt.subplots(1, 2, figsize=(10, 10), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original color image")
ax[0].imshow(imgNightSightColor)

ax[1].set_title("Image as grayscale")
ax[1].imshow(imgNightSightGrey, cmap=plt.cm.Greys_r, interpolation = "nearest")

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()
```

Original color image

Image as grayscale

In [32]:

```python
print(np.shape(img))
print(type(img))

height, width = img.shape[:2]

# This is a slice, it's extremely common in python and data science
# [:2] from 0 index up to 2, so excluding 2
print(img.shape[:2])
# [1:] from 1 index to the end
print(img.shape[1:])
# [1:2] from 1 index up to 2
print(img.shape[1:2])

# Height and width as we expected
print(height, width)

#center =

# lists(mutable) vs. tuples(immutable)
# list here
list1 = [1, 2, 3, 4, 5]
print(list1)
```

```python

list1[0] = list1[0] + 1
print(list1)

list1.append(1)
print(list1)

list1.append('hello')
print(list1)

print(type(list1[0]))
print(type(list1[6]))

list1 = list1[:2] + list1[:5]
print(list1)

# tuple here
tuple1 = (1, 2, 3)
print(tuple1)

# This is an error because you can't change tuples
#tuple1[0] = 1 + tuple1[0]
```

```
(1200, 1600, 3)
<class 'numpy.ndarray'>
(1200, 1600)
(1600, 3)
(1600,)
1200 1600
[1, 2, 3, 4, 5]
[2, 2, 3, 4, 5]
[2, 2, 3, 4, 5, 1]
[2, 2, 3, 4, 5, 1, 'hello']
<class 'int'>
<class 'str'>
[2, 2, 2, 2, 3, 4, 5]
(1, 2, 3)
```

In [33]:

```
center = (width/2, height/2)

# using cv2.getRotationMatrix2D() to get the rotation matrix
# angle is rotating angle, scale is zooming in or out (from where the center)
rotate_matrix = cv2.getRotationMatrix2D(center=center, angle=45, scale=1)
```

In [34]:

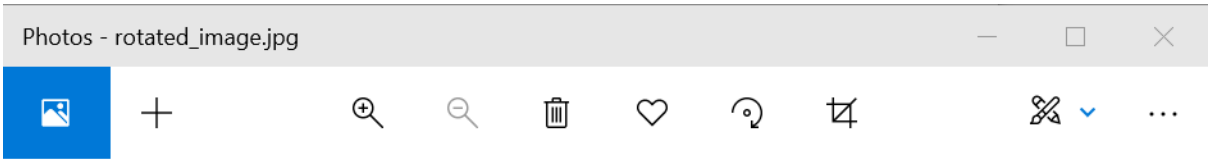
```
# rotate the image using cv2.warpAffine
rotated_image = cv2.warpAffine(src=img, M=rotate_matrix, dsize=(width, height))
```

In [35]:

```
# cv2.imshow('Original image', img)
cv2.imshow('Rotated image', rotated_image)
# wait indefinitely, press any key on keyboard to exit
cv2.waitKey(0)
# save the rotated image to disk
cv2.imwrite('rotated_image.jpg', rotated_image)
```

Out[35]:

```
True
```



GammaCorrection.py:

In this session, based on the idea from the internet and the textbook, “Digital Image Processing," I implemented a program for a gamma correction, smoothing filter of an image, Sobel edge detector, and finding the edges in an image. I learned that gamma correction is making the image brighter or darker, so you can more easily identify lines and distinguishing features. I learned how it works by seeing the result of the code one by one. The followings are the questions and the answers that I came up with during the session:

Questions:

#

1 I got this: In []: <- inside of the bracket empty! How do you run the cell individually?

2 What is gamma correction?

#

Answers:

#

1 You need to shit + enter

2 Just make the image brighter or darker, so you can more easily identify lines and distinguishing features

The codes/outputs are below:

In [26]:

```
# Importing modules
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
import cv2 # OpenCV
import glob # loading paths

# load an image (in png)
images = glob.glob('*.png') #** means all the pictures of the array, and * means only 1 image

print(images)

print(type(images))
print(np.shape(images))
print(images[1])
```

```
['aerial.png', 'fracture_spine.png']
<class 'list'>
(2,)
fracture_spine.png
```

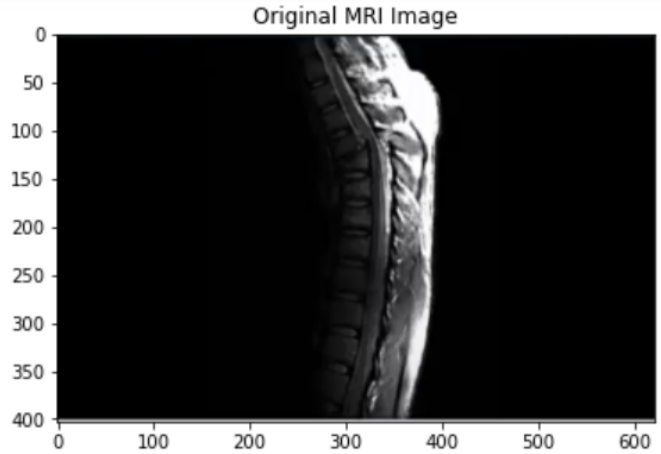
In [14]:

```
# load as color image
# [0] is the aerial image in the array and [1] is the fracture_spine (MRI) image in the array
# see what pictures are in the array above result
img = cv2.imread(images[1], cv2.IMREAD_COLOR)
print()
```

In [16]:

```
# apply gamma correction on the MRI image
# displaying the title
plt.title("Original MRI Image")

plt.imshow(img)
```

Out[16]: <matplotlib.image.AxesImage at 0x16ff6e4b280>



In [17]:

```
def adjust_gamma(image, gamma=1.0):
    # build a lookup table mapping the pixel values [0, 255] to
    # their adjusted gamma values
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255
        for i in np.arange(0, 256)]).astype("uint8")

    # apply gamma correction using the lookup table
    return cv2.LUT(image, table)
```

In [18]:

```
adjusted = adjust_gamma(img, gamma=2.35)

# print(adjusted)
# (row, column, channels like RGB)
print(np.shape(adjusted))
```

```
(403, 622, 3)
```

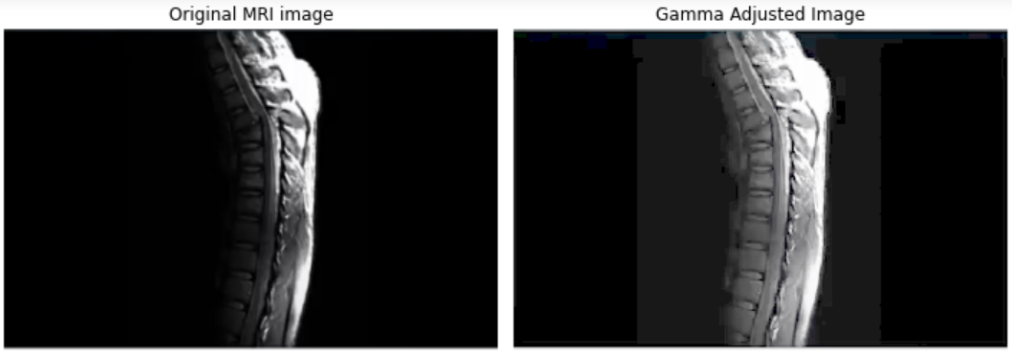
In [25]:

```
# cv2.imshow('Original image', img)
cv2.imshow('adjusted', adjusted)
# wait indefinitely, press any key on keyboard to exit
cv2.waitKey(0)
# save the rotated image to disk
cv2.imwrite('gamma_adjusted.jpg', adjusted)
```

Out[25]: True

In [24]:

```
# display original and gamma adjusted images
fig, axes = plt.subplots(1, 2, figsize=(10, 10), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original MRI image")
ax[0].imshow(img)

ax[1].set_title("Gamma Adjusted Image")
ax[1].imshow(adjusted, cmap=plt.cm.Greys_r, interpolation = "nearest")

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()
```



In [28]:

```
# now, apply gamma correction on the aerial image
print(images[0]) #check if the image in the first array is the aerial picture
```

```
aerial.png
```

In [29]:

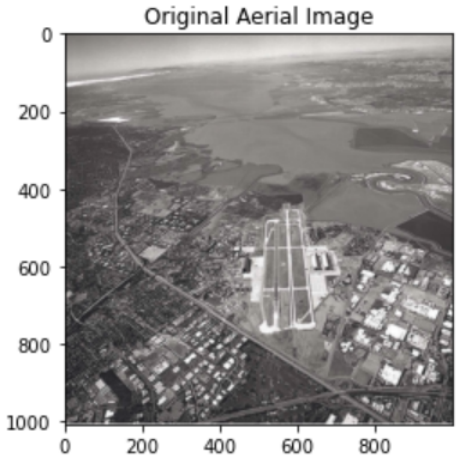
```
# load as color image
# [0] is the aerial image in the array and [1] is the fracture_spine (MRI) image in the array
# see what pictures are in the array above result
img = cv2.imread(images[0], cv2.IMREAD_COLOR)
print()
```

In [30]:

```
# apply gamma correction on the MRI image
# displaying the title
plt.title("Original Aerial Image")

plt.imshow(img)
```

Out[30]:

```
<matplotlib.image.AxesImage at 0x16ffae31e80>
```



In [31]:

```python
def adjust_gamma(image, gamma=1.0):
    # build a lookup table mapping the pixel values [0, 255] to
    # their adjusted gamma values
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255
        for i in np.arange(0, 256)]).astype("uint8")

    # apply gamma correction using the lookup table
    return cv2.LUT(image, table)
```

In [68]:

```python
adjusted = adjust_gamma(img, gamma= 0.05)

# print(adjusted)
# (row, column, channels like RGB)
print(np.shape(adjusted))
```

```
(1008, 1000, 3)
```

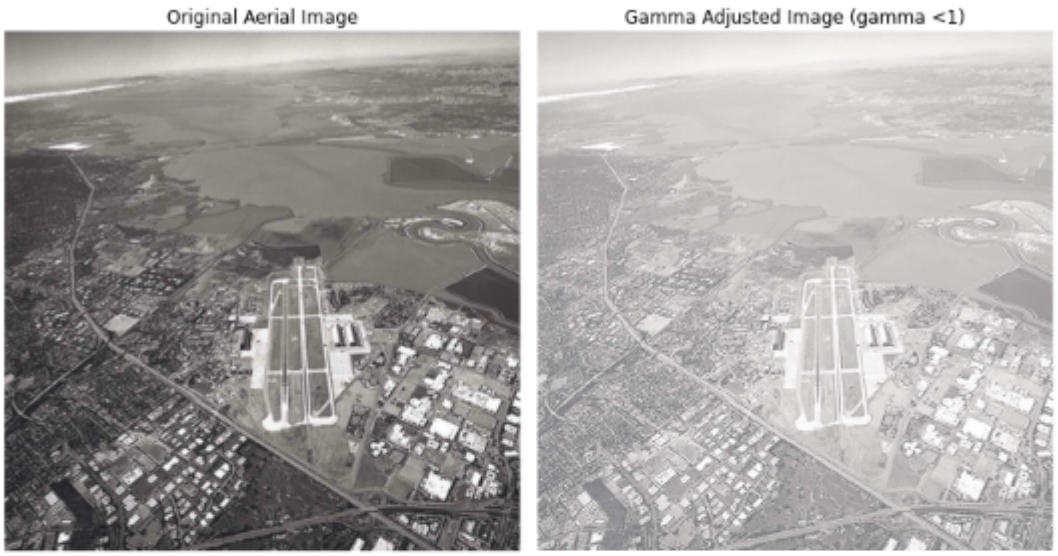
In [39]:

```python
# cv2.imshow('Original image', img)
cv2.imshow('adjusted', adjusted)
# wait indefinitely, press any key on keyboard to exit
cv2.waitKey(0)
# save the rotated image to disk
cv2.imwrite('gamma_adjusted.jpg', adjusted)
```

Out[39]: True

In [44]:

```python
# display original and gamma adjusted images
fig, axes = plt.subplots(1, 2, figsize=(10, 10), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original Aerial Image")
ax[0].imshow(img)

ax[1].set_title("Gamma Adjusted Image (gamma > 1)")
ax[1].imshow(adjusted, cmap=plt.cm.Greys_r, interpolation = "nearest")

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()
```



In [45]:
```
# Try using a different image
imgNightSight = cv2.imread("NightSightColor.jpg", cv2.IMREAD_COLOR)
adjustedNS1 = adjust_gamma(imgNightSight, gamma=2)

# if the value is 1<gamma, the image will become darker
adjustedNS2 = adjust_gamma(imgNightSight, gamma=0.05)
```

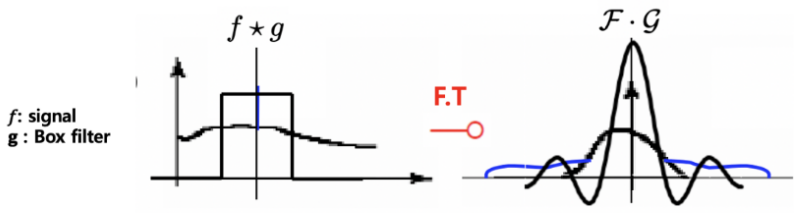
In [52]:
```
# display original and gamma adjusted images
fig, axes = plt.subplots(1, 3, figsize=(20, 20), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original Night Sight Image")
ax[0].imshow(imgNightSight)

ax[1].set_title("Gamma Adjusted Image (gamma > 1)")
ax[1].imshow(adjustedNS1, cmap=plt.cm.Greys_r, interpolation = "nearest")

ax[2].set_title("Gamma Adjusted Image(gamma < 1)")
ax[2].imshow(adjustedNS2, cmap=plt.cm.Greys_r, interpolation = "nearest")
```

Out[52]: <matplotlib.image.AxesImage at 0x23e3604e100>

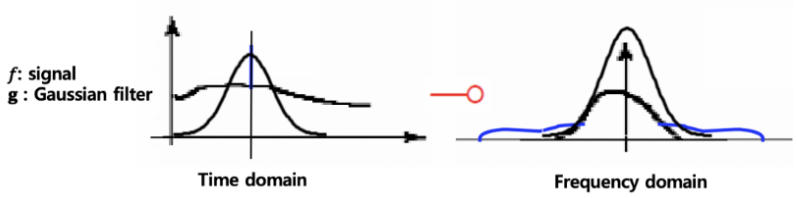
SmoothingFilters.py:

In this session, based on the idea from the internet and the textbook, “Digital Image Processing," I tried to implement a program that smooths the image by convolving a kernel filter over the image. In this session, I learned that there are various kernels used to smooth images such as Gaussian, Median, Normalized Box Filter, etc. I tested to see the differences between each kernel filter and their effects on the image. In addition, according to the article,

"[CV] 2. Image Processing Basic: Gaussian and Median Filter, Separable 2D filter," I learned that "the Gaussian filter is a better choice for **g** as its Fourier-transformed shape is the ideal low-pass filter, allowing only low frequencies to survive. On the other hand, the box filter turns into the sinc function after Fourier transforms. It not only suppresses high frequencies but also some low-frequency components, which are responsible for general image structures and are supposed not to be lost." Therefore, in this case, the Gaussian filter is more efficient to reduce the noise compared to the Normalized Box Filter.



Blue line: noise pixel (left) and its corresponding frequency components (right) introduced by the noise pixel



The codes/outputs are below:

```python
# Importing modules
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
import cv2 # OpenCV
import glob # loading paths
from matplotlib.pyplot import figure

# load an image (in jpg)
images = glob.glob('*.jpg') #** means all the pictures of the array, and * means only 1 image

print(images)

print(type(images))
print(np.shape(images))
print(images[1])
```

```
['gamma_adjusted.jpg', 'NightSightColor.jpg', 'rotated_image.jpg']
<class 'list'>
(3,)
NightSightColor.jpg
```

In [2]:

```
# load as color image
# [0] is the aerial image in the array and [1] is the fracture_spine (MRI) image in the array
# see what pictures are in the array above result
img = cv2.imread(images[1], cv2.IMREAD_COLOR)
print()
```

In [3]:

```
# displaying the title
figure(figsize=(10,10))

plt.title("Original Image")

plt.imshow(img)
```

Out[3]: <matplotlib.image.AxesImage at 0x1cbf26c6190>

In [4]:

```
# Applying Normalized Box Filter with a [3x3] matrix Kernel
img_blur = cv2.blur(img, (20, 20))

# Applying 10 convolutions to see more effect
for i in range(100):
    img_blur = cv2.blur(img, (3, 3))

figure(figsize=(10,10))

plt.title("Image Blur")

plt.imshow(img_blur)

print(img_blur.shape)
```

```
(1200, 1600, 3)
```

Image Blur
0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

In [5]:

```
# display color and grayscale images
fig, axes = plt.subplots(1, 2, figsize=(20, 20), sharex=True, sharey=True)
ax = axes.ravel()

ax[0].set_title("Original Image")
ax[0].imshow(img)

ax[1].set_title("Normalized Box Filter Image")
ax[1].imshow(img_blur, cmap=plt.cm.Greys_r, interpolation = "nearest")

for a in ax:
    a.set_axis_off()

fig.tight_layout()
plt.show()
```

In [13]:

```
#print(img.shape)

img_gaussian = cv2.GaussianBlur(img, (21,21),sigmaX=0)

figure(figsize=(10,10))

plt.title("Image Gaussian Blur")

plt.imshow(img_gaussian)

#print(img_gaussian.shape)
```

Out[13]: <matplotlib.image.AxesImage at 0x1f1af0c72e0>

In [14]:

```
img_median = cv2.medianBlur(img, 11)

figure(figsize=(10,10))

plt.title("Image Median Blur")

plt.imshow(img_median)
```

Out[14]: <matplotlib.image.AxesImage at 0x1f1af6dbac0>

In [15]:

```
# create figure
fig = plt.figure(figsize=(20, 20))

# setting values to rows and column variables
rows = 1
columns = 4

# Adds a subplot at the 1st position
fig.add_subplot(rows, columns, 1)

# showing image
plt.imshow(img)
plt.axis('off')
plt.title("Original")

# Adds a subplot at the 2nd position
fig.add_subplot(rows, columns, 2)

# showing image
plt.imshow(img_blur)
plt.axis('off')
plt.title("Normalized Box")

# Adds a subplot at the 3rd position
fig.add_subplot(rows, columns, 3)

# showing image
plt.imshow(img_gaussian)
plt.axis('off')
plt.title("Gaussian")

# Adds a subplot at the 4th position
fig.add_subplot(rows, columns, 4)

# showing image
plt.imshow(img_median)
plt.axis('off')
plt.title("Median")

# Tightness for the images
plt.tight_layout ()
plt.subplots_adjust(hspace=0, wspace=.1)
```

SobelEdgeAndCannyEdgeDetections.py:

In this session, the image on a Sobel Edge Detection and the image on a Canny Edge Detection were compared to see if there is a difference. I tried with 2 different images to see

how Sobel Filter works on the X-axis and Y-axis. The Sobel X image is filtered X-axis out and the Sobel Y image is filtered Y-axis out. Compare the image to a Canny Edge Detection and see if there is a difference. In addition, I also learned how to use this code on Google Colab which has the same extension (.ipynb).

In [97]:

```
# This program is about a Sobel Edge Detection and a Canny Edge Detection
# I tried with 2 different images to see how Sobel Filter works on X-axis and Y-axis
# # The Sobel X image is filtered X axis out and the Sobel Y image is filtered Y axis out
# Compare the image to a Canny Edge Detection and see if there is a difference
# In addition, I also learned how to use this code on Google Colab which has the same extension (.ipynb)
```

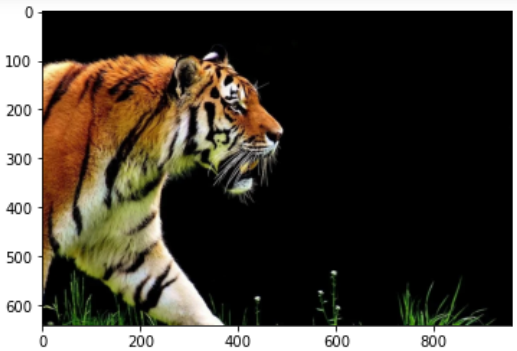
In [98]:

```
# Importing modules
import numpy as np
%matplotlib inline
from matplotlib import pyplot as plt
import cv2 # OpenCV
import glob # loading paths

# load an image (in png)
argv = glob.glob('*.png') #** means all the pictures of the array, and * means only 1 image

print(argv)

print(type(argv))
print(np.shape(argv)) # how many images are in the directory
print(argv[4]) # you want to use the 4th image 'SobelEdgeTiger.png' in the list
```

```
['aerial.png', 'circuitboard.png', 'fracture_spine.png', 'saltandpepper.png', 'SobelEdgeTiger.png', 'SobelXY.png']
<class 'list'>
(6,)
SobelEdgeTiger.png
```

In [99]:

```
src = cv2.imread('SobelEdgeTiger.png')
# OpenCV reads images in BGR, but matplotlib shows images in RGB, so
# you have to convert to RGB before using plt.imshow
plt.imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))

img = cv2.imread(argv[4], cv2.IMREAD_COLOR)
print()
```



In [113]:

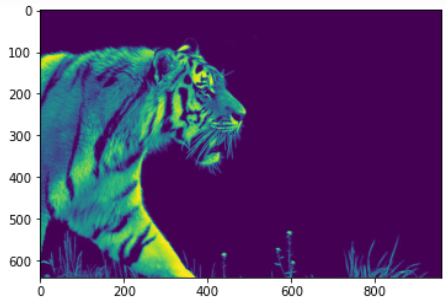
```
# If you want to open on Google Colab, upload theimage from local PC
# from google.colab import files
# uploaded = files.upload()

# Check the uploaded file. (It seems that you can execute Linux commands with the! + Command)
# If the uploaded file is called "SobelXY.png," it will be as follows.

# !ls
# SobelXY.png
```

In [114]:

```
# If you don't have the following code,
# "plt.imshow(gray, cmap='gray'),"
# matplotlib reads the grayscale image into a blue green color by default
bgdefault = cv2.imread('SobelEdgeTiger.png', cv2.IMREAD_GRAYSCALE)
plt.imshow(bgdefault)
```

Out[114]:

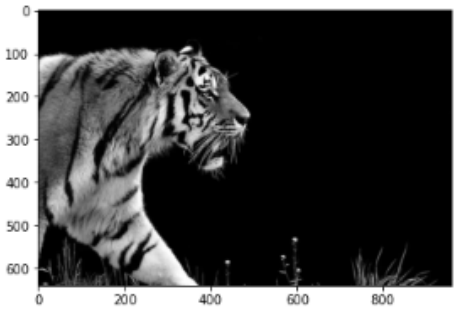
```
<matplotlib.image.AxesImage at 0x1bf1bf3dbb0>
```



In [115]:

```
# Now, channge the image into grayscale

# either codes work:

#gray = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)
#plt.imshow(gray) # show the image
# OR

gray = cv2.imread('SobelEdgeTiger.png', cv2.IMREAD_GRAYSCALE)
# cmap needs to be set as gray because matplotlib reads the grayscale image into a blue green color by default
# when you want to use plt package, you have to indicate "...., cmap='gray')"
plt.imshow(gray, cmap='gray') # show the image
```

Out[115]: <matplotlib.image.AxesImage at 0x1bf1bf9fee0>



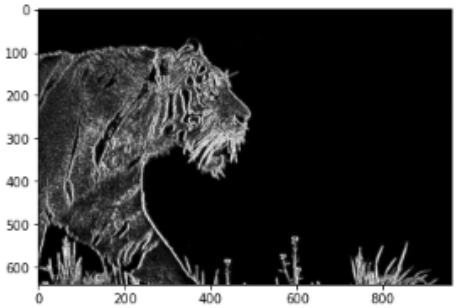
In [116]:

```
# Sobel Filter
scale = 1
delta = 0
ddepth = cv2.CV_16S

grad_x = cv2.Sobel(gray, ddepth, 1, 0, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)
grad_y = cv2.Sobel(gray, ddepth, 0, 1, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)

#Conversion Filter
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)
grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
```

In [117]:

```
# Sobel Filter only on x axis
# The Sobel X image is filtered X axis out
plt.imshow(abs_grad_x, cmap='gray')
```

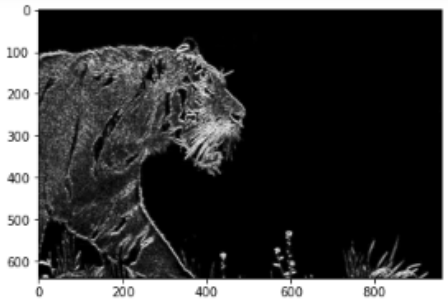
Out[117]: <matplotlib.image.AxesImage at 0x1bf1c005eb0>



In [118]:

```
# Sobel Filter only on y axis
# The Sobel Y image is filtered X axis out
plt.imshow(abs_grad_y, cmap='gray')
```

Out[118]: <matplotlib.image.AxesImage at 0x1bf1d6e9dc0>



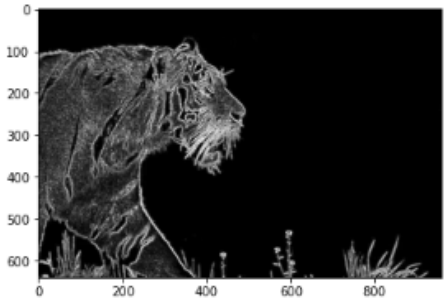
In [119]:

```
#Conversion Filter
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)

grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
```

In [120]:

```
##Display the image after conversion filter
plt.imshow(grad, cmap='gray')
```

Out[120]: <matplotlib.image.AxesImage at 0x1bf1c0e02b0>



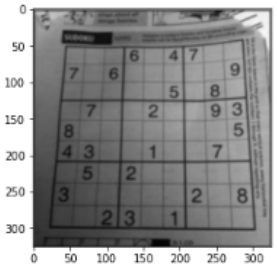
In [121]:

```
# Now, try with a different image
print(argv)

print(type(argv))
print(argv[5]) # you want to use the 5th image (SobelXY.png) in the list
```

```
['aerial.png', 'circuitboard.png', 'fracture_spine.png', 'saltandpepper.png', 'SobelEdgeTiger.png', 'SobelXY.png']
<class 'list'>
SobelXY.png
```

In [122]:

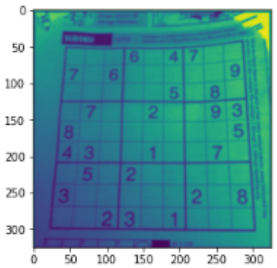
```
src = cv2.imread('SobelXY.png')
# OpenCV reads images in BGR, but matplotlib shows images in RGB, so
# you have to convert to RGB before using plt.imshow
# However, 'SobelXY.png' is already black and white image, so it shows in grayscale
plt.imshow(cv2.cvtColor(src, cv2.COLOR_BGR2RGB))

img = cv2.imread(argv[5], cv2.IMREAD_COLOR)
print()
```



In [123]:

```
# matplotlib reads the grayscale image into a blue green color by default
bgdefault = cv2.imread('SobelXY.png', cv2.IMREAD_GRAYSCALE)
plt.imshow(bgdefault)
```

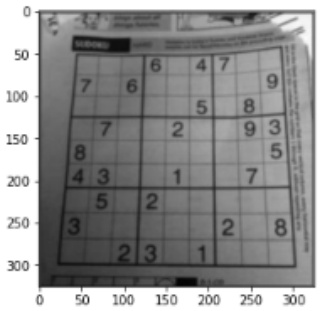
Out[123]: <matplotlib.image.AxesImage at 0x1bf1c0e92b0>



In [124]:

```
gray = cv2.imread('SobelXY.png', cv2.IMREAD_GRAYSCALE)
# cmap needs to be set as gray because matplotlib reads the grayscale image into a blue green color by default
# when you want to use plt package, you have to indicate "...., cmap='gray')"
plt.imshow(gray, cmap='gray') # show the image
```

Out[124]: <matplotlib.image.AxesImage at 0x1bf1c070070>



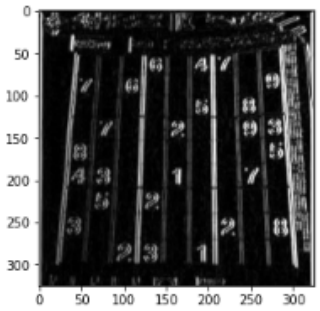
In [125]:

```
# Sobel Filter
scale = 1
delta = 0
ddepth = cv2.CV_16S

grad_x = cv2.Sobel(gray, ddepth, 1, 0, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)
grad_y = cv2.Sobel(gray, ddepth, 0, 1, ksize=3, scale=scale, delta=delta, borderType=cv2.BORDER_DEFAULT)

#Conversion Filter
abs_grad_x = cv2.convertScaleAbs(grad_x)
abs_grad_y = cv2.convertScaleAbs(grad_y)
grad = cv2.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
```

In [126]:

```
# Sobel Filter only on x axis
# The Sobel X image is filtered X axis out
plt.imshow(abs_grad_x, cmap='gray')
```

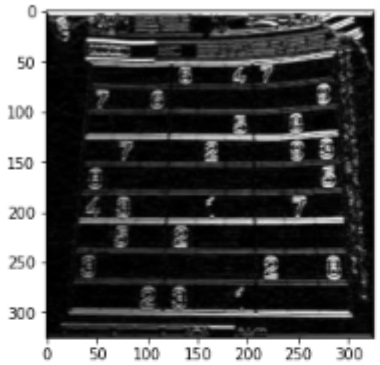
Out[126]: <matplotlib.image.AxesImage at 0x1bf1c12ecd0>



In [127]:

```
# Sobel Filter only on y axis
# The Sobel Y image is filtered X axis out
plt.imshow(abs_grad_y, cmap='gray')
```

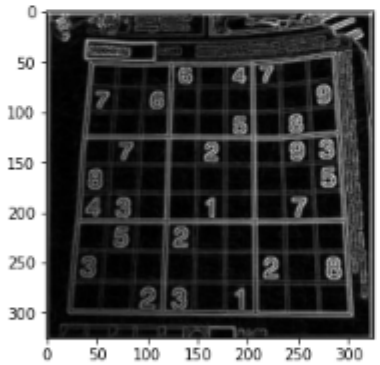
Out[127]: <matplotlib.image.AxesImage at 0x1bf1c198d60>



In [128]:

```
#Display the image after conversion filter
plt.imshow(grad, cmap='gray')
```

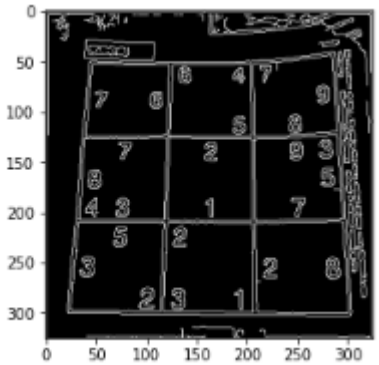
Out[128]: <matplotlib.image.AxesImage at 0x1bf1c201c40>



In [129]:

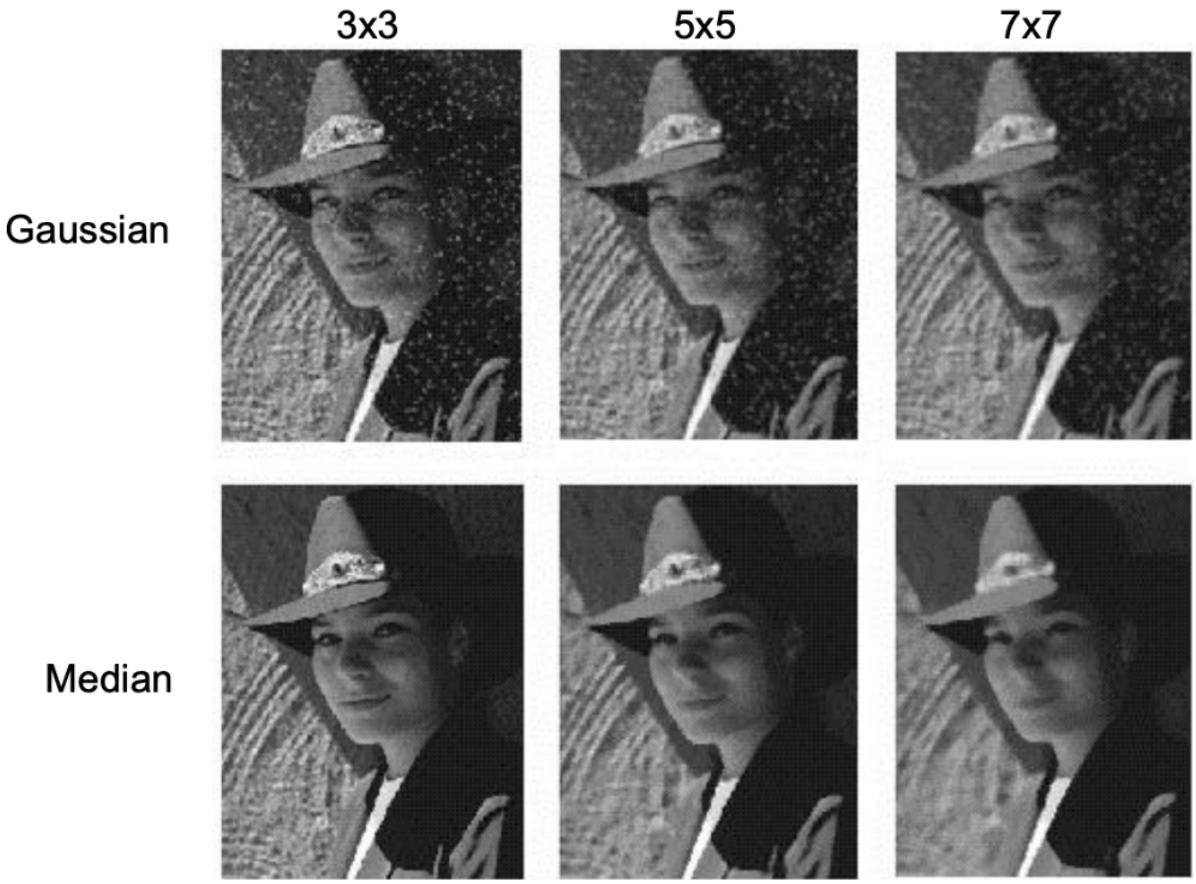
```
# Canny Edge Detection
canny = cv2.Canny(image=img, threshold1=100, threshold2=200) # Canny Edge Detection

# Display Canny Edge Detection Image as a window
# cv2.imshow('Canny Edge Detection', edges)
# cv2.waitKey(0)
# cv2.destroyAllWindows()

plt.imshow(canny, cmap='gray')
```

Out[129]: <matplotlib.image.AxesImage at 0x1bf1c26ebe0>



MedianFiltering.py:

When I implemented SmoothingFilters.py, I learned that the Gaussian filter is more efficient to reduce the noise compared to the Normalized Box Filter in that situation. In this session, a “salt and pepper” noise was added to an image and then a Median filter was used on the image. Then, the image with a Gaussian filter was compared with the Median filtered image to see if there is a difference. The Gaussian filter didn’t get rid of any of the salt and pepper noise. The median filter allowed the center pixel value to be replaced by a value that is present in the surrounding pixels. This differs from Gaussian which will use the weighted average instead, where outliers can heavily skew the average, resulting in almost no noise reduction in this case. According to the article, "[CV] 2. Image Processing Basic: Gaussian and Median Filter, Separable 2D filter, "when the noise level is too high, although the amount of noise pixel decreases with increasing Gaussian filter size, they still exist in the image. The median filter, on the other hand, already removes most of the noise pixels with a 3 x 3 filter size. By applying a larger filter size, the Median filter further excludes noise pixels but it loses a lot of image-structure information and image details."



The codes/outputs are below:

In [9]:

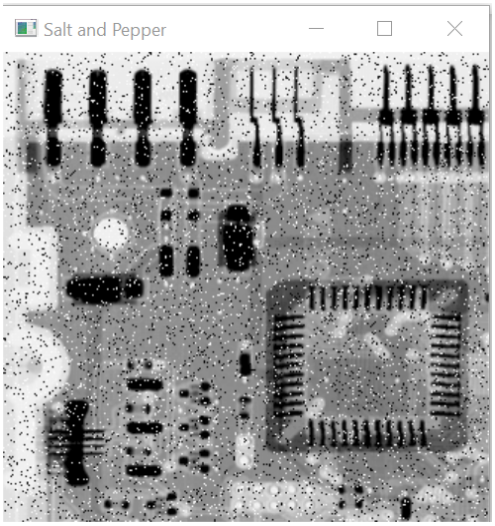
```python
import random
import cv2
# import plt for the image title
from matplotlib import pyplot as plt
# import figure
from matplotlib.pyplot import figure

def add_noise(img):

    # get the dimensions of the image
    row , col = img.shape

    # randomly pick some pixels in the
    # image for coloring them white
    # pick a random number between 300 and 10000
    number_of_pixels = random.randint(300, 10000)
    for i in range(number_of_pixels):

        # pick a random y coordinate
        y_coord=random.randint(0, row - 1)

        # pick a random x coordinate
        x_coord=random.randint(0, col - 1)

        # color that pixel to white
        img[y_coord][x_coord] = 255

    # randomly pick some pixels in
    # the image for coloring them black
    # pick a random number between 300 and 10000
    number_of_pixels = random.randint(300 , 10000)
    for i in range(number_of_pixels):

        # pick a random y coordinate
        y_coord=random.randint(0, row - 1)

        # pick a random x coordinate
        x_coord=random.randint(0, col - 1)

        # color that pixel to black
        img[y_coord][x_coord] = 0

    return img

# salt-and-pepper noise can
# be applied only to grayscale images
# read the color image in grayscale image
img = cv2.imread('circuitboard.png', cv2.IMREAD_GRAYSCALE)

#store the image
cv2.imwrite('saltandpepper.png', add_noise(img))
```

Out[9]: True

In [10]:

```python
cv2.imshow('Salt and Pepper', img) # display img with salt and pepper noise
cv2.waitKey(0)        # wait for a key press to
cv2.destroyAllWindows # close the img window
```

Out[10]: <function destroyAllWindows>



In [11]:
```
img = cv2.imread('saltandpepper.png') # load image
img_median = cv2.medianBlur(img, 5) # add median filter to image

cv2.imshow('Median Filter', img_median) # display img with a median filter
cv2.waitKey(0)        # wait for a key press to
cv2.destroyAllWindows # close the img window
```

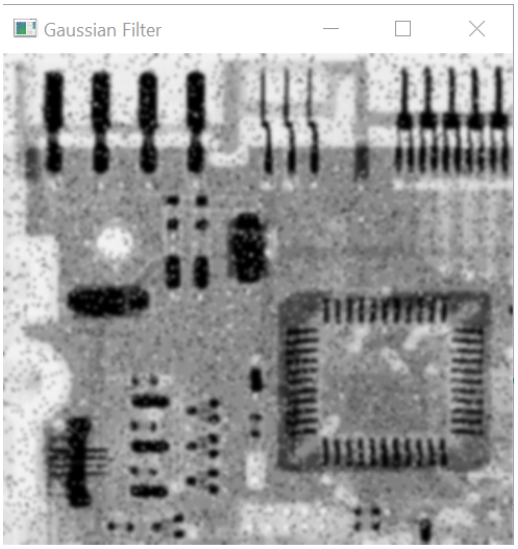
Out[11]: <function destroyAllWindows>

In [12]:

```
img_gaussian = cv2.GaussianBlur(img, (5,5),sigmaX=0) # (img, (5,5).. <- as increasing the numbers, it becomes more blurry
figure(figsize=(5,5))

cv2.imshow('Gaussian Filter', img_gaussian) # display img with a gaussian filter
cv2.waitKey(0)        # wait for a key press to
cv2.destroyAllWindows # close the img window

# plt.title("Image Gaussian Blur")
# plt.imshow(img_gaussian)
```

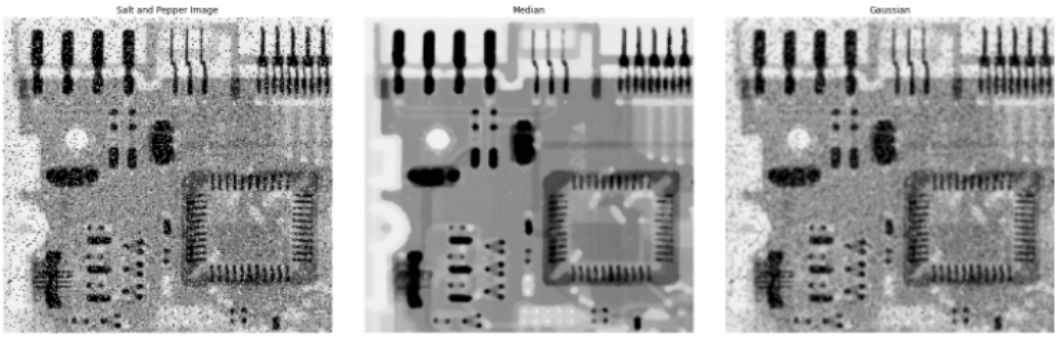
Out[12]:

```
<function destroyAllWindows>

<Figure size 360x360 with 0 Axes>
```



In [13]:

```python
# create figure
fig = plt.figure(figsize=(20, 20))

# setting values to rows and column variables
rows = 1
columns = 3

# Adds a subplot at the 1st position
fig.add_subplot(rows, columns, 1)
# showing image
plt.imshow(img)
plt.axis('off')
plt.title("Salt and Pepper Image")

# Adds a subplot at the 2nd position
fig.add_subplot(rows, columns, 2)
# showing image
plt.imshow(img_median)
plt.axis('off')
plt.title("Median")

# Adds a subplot at the 3rd position
fig.add_subplot(rows, columns, 3)
# showing image
plt.imshow(img_gaussian)
plt.axis('off')
plt.title("Gaussian")

# Tightness for the images
plt.tight_layout ()
plt.subplots_adjust(hspace=0, wspace=.1)
```



Conclusions

Throughout the directed study, I was able to learn the theoretical ideas of machine learning and image processing. In addition, I practically implemented filters and edge detectors and learned how each filter/edge detector is different. I also became able to distinguish which filter/edge detector is more suitable depending on the case. Before I started learning with Dr. Meena, I did not even know how to begin learning the field of artificial intelligence. Throughout the directed study, I realized that AI is a deep and wide field. Before starting to learn the field of AI, I also realized I needed to build foundations. To strengthen my foundations, understanding the basics of machine learning and image processing were paramount. This directed study has taught me how to study individually in an effective manner. I also believe the materials that I learned under Dr. Meena will definitely help me to further learn on my own.

References

Gonzalez, & Woods. (2018). *Digital Image Processing*. Pearson.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An introduction to statistical learning: With applications in R*. Springer.

jun94. (2020, November 12). *[CV] 2. gaussian and median filter, separable 2D filter*. Medium. Retrieved May 1, 2022, from https://medium.com/jun94-devpblog/cv-2-gaussian-and-median-filter-separable-2d-filter-2d11ee022c66