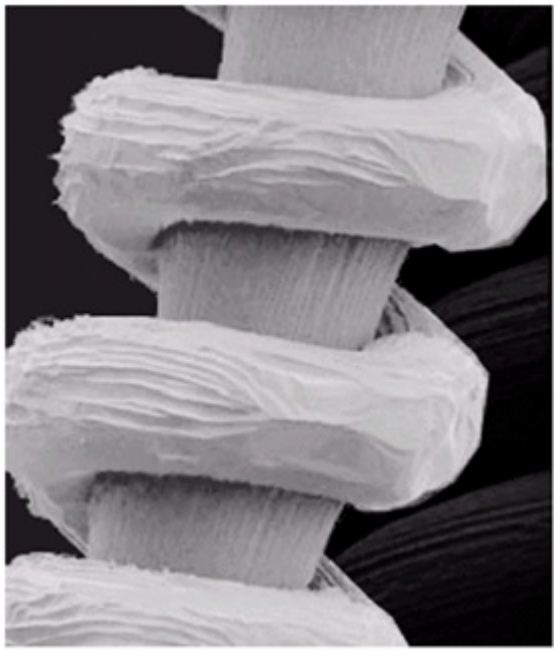
In [ ]:
```
import pandas as pd
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
import math
from IPython.display import Image, display, HTML

print(pd.__version__)
print(cv.__version__)
```

```
2.1.0
4.8.0
```

In [ ]:
```
#%%[markdown]
# ![title](assignment2_image1.jpg)
image = cv.imread("./image/assignment2_image1.jpg")
gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)

display(HTML('<h3>Original image</h3>'))
display(Image(filename='./image/assignment2_image1.jpg'))
```

Original image



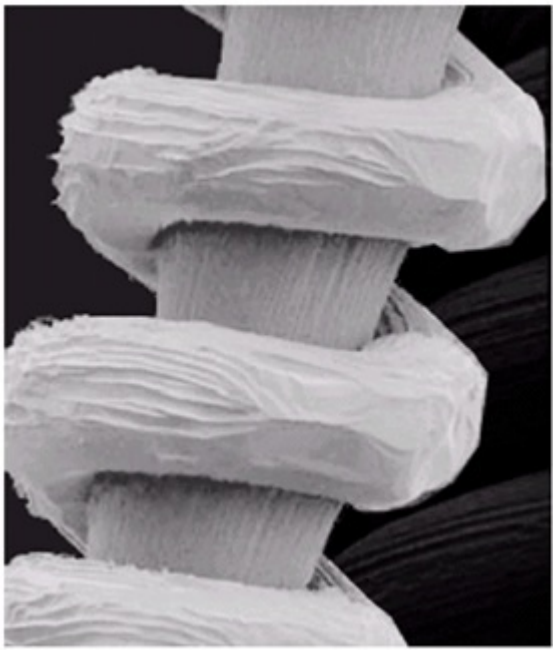
1. Gamma Correction with value from 0.5 - 2.0 (step = 0.1)

In [ ]:
```
# Define the initial gamma value
gamma = 0.5

# Perform gamma correction
while(gamma < 2.01):
    gamma_corrected = np.power(img/255.0, gamma)
    gamma_corrected = np.uint8(gamma_corrected * 255.0)
    cv.imwrite(f'./image/gamma/output_{gamma}.jpg', gamma_corrected)

    gamma = round(gamma + 0.1, 2)
```

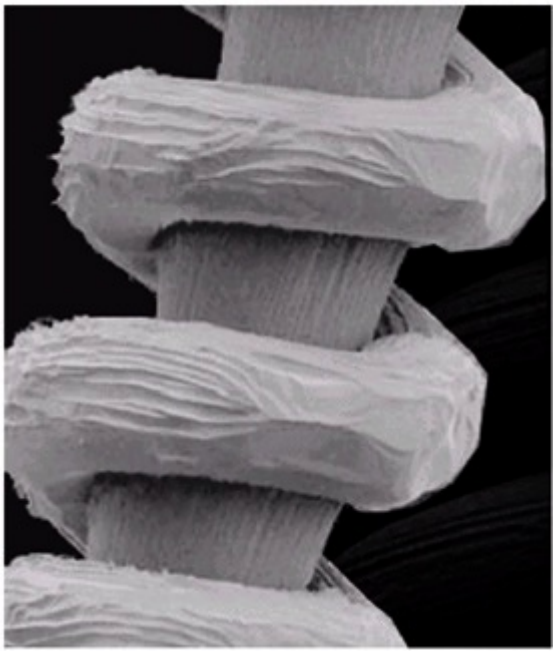
```
# display all new images
i = 0.5
while (i < 2.01):
    # display(HTML(f'<h3>gamma = {i}</h3>'))
    # display(Image(filename=f'./image/gamma/output_{i}.jpg') )
    # print(i)
    i = round(i + 0.1, 2)

display(HTML('<h3>gamma = 0.5</h3>'))
display(Image(filename=f'./image/gamma/output_0.5.jpg'))

display(HTML('<h3>gamma = 1.0</h3>'))
display(Image(filename=f'./image/gamma/output_1.0.jpg'))

display(HTML('<h3>gamma = 1.5</h3>'))
display(Image(filename=f'./image/gamma/output_1.5.jpg'))

display(HTML('<h3>gamma = 2.0</h3>'))
display(Image(filename=f'./image/gamma/output_2.0.jpg'))
```

gamma = 0.5

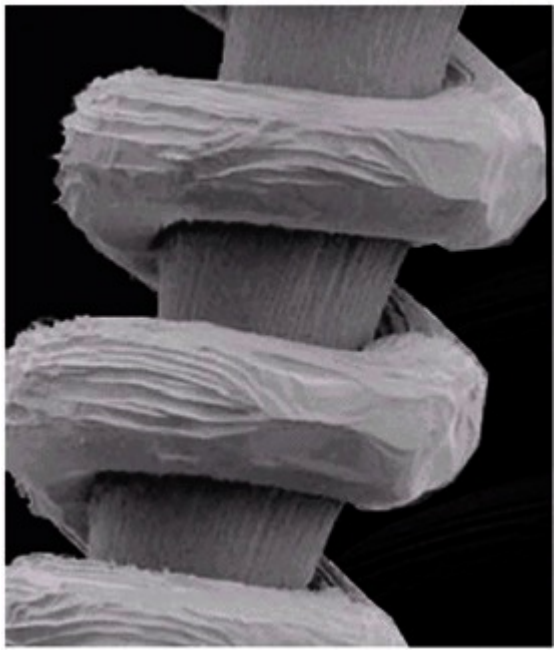
gamma = 1.0



gamma = 1.5



gamma = 2.0



> **จากผลการรัน เราจะเห็นได้ว่าช่วงที่ให้ผลลัพธ์ดีที่สุด คือ ช่วงที่ค่า gamma ประมาณ 0.5-1.0 แต่ก็ยังได้ภาพที่ปรับแต่งได้ไม่ดีที่สุด เพราะ อัลกอริทึมปรับแต่งส่วนอื่นที่เราไม่ต้องการให้สว่างขึ้นด้วย**

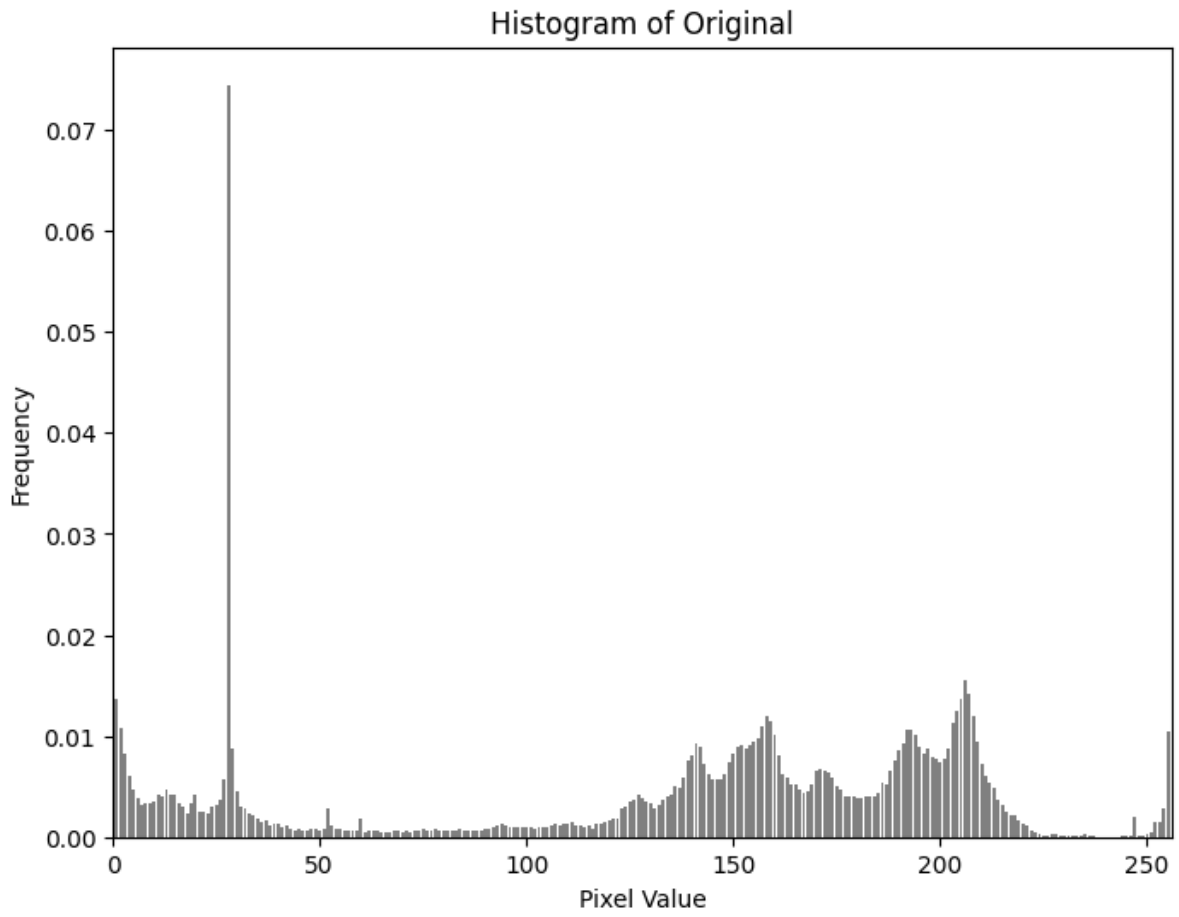
---

2. Global Histogram Equalization

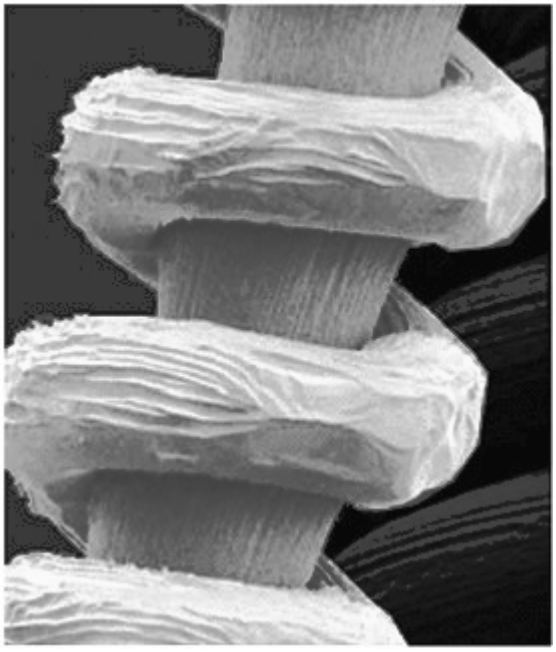
In [ ]:

```
# Load the image (8 bits pixel)
image = cv.imread("./image/assignment2_image1.jpg", cv.IMREAD_GRAYSCALE)  # Read as
width, height = image.shape
n = width * height

"""
image.flatten() returns 1 array dimension of image
bins is interval that we seperate to count group of values
ex. bins = 8 -> [frequency of 0-7], [frequency of 8-15], [frequency of 16-23],...
in this case we want to count every value that appear in image
so we set bins = 256 to count every value one by one
range = considering data, ignore if not in range
"""
# Calculate the histogram of the image
hist, bins = np.histogram(image.flatten(), bins=256, range=(0, 256))

# downscale with its size to caculate CDF
hist_downscaled = hist / n

# Display histogram
plt.figure(figsize=(8, 6))
plt.title('Histogram of Original')
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
plt.xlim([0, 256])
plt.bar(np.arange(256), hist_downscaled, color='gray')
plt.show()

# Calculate the cumulative distribution function (CDF) of the histogram
cdf = hist_downscaled.cumsum()

# upscale back to replace original image
```

```
cdf_upscaled = cdf * 256

"""
Apply histogram equalization to the image using the lookup table (its numpy functic
it will map each value pixel with value in cdf
ex. pixel of image with value = 252 will be mapped with value of index at 252 in cd
"""

'''
More example :

a = np.array([0,3,2,6,42,7])
b= np.array([[5,5,5,5,5],[1,1,1,1,1]])

print(a[b]) -> [[7 7 7 7 7], [3 3 3 3 3]]
'''

# equivalent with "equalized_image = cdf[image]"
equalized_image = np.take(cdf_upscaled, image)

cv.imwrite('./image/output_global.jpg', equalized_image)

display(HTML('<h3>Enhanced image</h3>'))
display(Image("./image/output_global.jpg") )
```



Enhanced image



> **จากผลการรัน เราจะเห็นได้ว่าผลลัพธ์ที่ได้มีความใกล้เคียงกับการใช้ gamma correction แต่มีปัญหาเหมือนกันคือ ส่วนที่ไม่ต้องการสว่างขึ้นมาด้วย**

---

3. Local Histogram Equalization with 3 neighborhood sizes: 3x3, 7x7, and 11x11.

In [ ]:

```
def caculatingMean(tile):
    n = tile.shape[0] * tile.shape[1]

    # Calculate the histogram of the image
    hist, bins = np.histogram(tile.flatten(), bins=256, range=(0, 256))

    # downscale with its size to caculate CDF
    hist_normalized = hist / np.sum(hist)

    mean = 0

    for i in range(tile.shape[1]):
        for j in range(tile.shape[0]):

            pixel_value = tile[j][i]
            mean += pixel_value * hist_normalized[pixel_value]

    return mean

def caculatingStd(tile):
    n = tile.shape[0] * tile.shape[1]

    # Calculate the histogram of the image
    hist, bins = np.histogram(tile.flatten(), bins=256, range=(0, 256))

    # downscale with its size to caculate CDF
    hist_normalized = hist / np.sum(hist)

    mean = caculatingMean(tile)
    var = 0
```

```
    for i in range(tile.shape[1]):
        for j in range(tile.shape[0]):
            pixel_value = tile[j][i]
            var += ( (pixel_value - mean)**2 ) * hist_normalized[pixel_value]

    return math.sqrt(var)

# tile = np.array( [ [255, 253, 255], [255, 255, 255], [252, 255, 255] ] , dtype=np
# # tile = tile.flatten()
# mean = caculatingMean(tile)
# print(mean)
# std = caculatingStd(tile)
# print(std)
```

In [ ]:

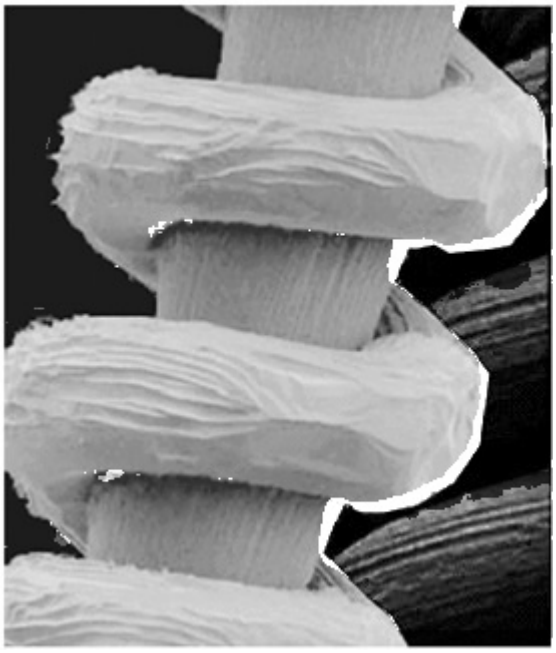
```
def localEqualization(image, size):
    enhance_image = image.copy()
    height = image.shape[1]
    width = image.shape[0]
    # global_mean = float(np.mean(gray_image))
    # global_std = float(np.std(gray_image))
    global_mean = caculatingMean(image)
    global_std = caculatingStd(image)
    # global_mean_intensity = float(cv.meanStdDev(image)[0][0])
    # global_standard_deviation = float(cv.meanStdDev(image)[1][0])


    # Define the size of the tiles
    tile_height = size[0]
    tile_width = size[1]

    # Initialize a list to store the tiles
    tiles = []

    # Split the array into 3x3-sized tiles
    for i in range(0, height):
        for j in range(0, width):
            # Define the coordinates of the current tile
            y1 = i
            y2 = i + tile_height
            x1 = j
            x2 = j + tile_width

            # Extract the current sized tile from the array
            tile = image[x1:x2, y1:y2]
            center_pixel_x = j+1
            center_pixel_y = i+1

            # out of bound (index)
            if center_pixel_y > height-1:
                continue
            if center_pixel_x > width-1:
                continue


            local_mean = caculatingMean(tile)
            local_std = caculatingStd(tile)

            """
            E: The mutiple enhancement factor.
            k0: The upper bound of the global mean
            k1: The lower bound on the global std
            k2: The upper bound on the global std
            """
```

```
            # Best performance for 3 x 3
            # k0 = 0.00055
            # k1 = 0.0000002
            # k2 = 0.000012
            # e = 3.5

            # acceptable performance for 7 x 7
            # k0 = 0.00075
            # k1 = 0.000002
            # k2 = 0.0001
            # e = 3.5

            # acceptable performance for 11 x 11
            # k0 = 0.00055
            # k1 = 0.0000002
            # k2 = 0.000012
            # e = 3.5

            k0 = 0.002
            k1 = 0.00002
            k2 = 0.001
            e = 3.5

            # print(local_mean)
            # coor = (center_pixel_x, center_pixel_y)
            # print(coor)
            # s = [(172, 260), (246, 246), (267, 241), (231, 291), (207, 292), (218
            # if coor in s:
            #     print(coor)
            #     print("local mean : " + str(local_mean))
            #     print("global_mean : " + str(global_mean))
            #     print("global mean : " + str(k0*global_mean))
            #     print()
            #     print("local std : " + str(local_std))
            #     print("Global std : " + str(global_std))
            #     print("lower bound : " + str(k1*global_std))
            #     print("upper bound : " + str(k2*global_std))
            #     print()

            mean_condition = local_mean <= k0*global_mean
            std_condition = k1*global_std <= local_std <= k2*global_std

            if mean_condition and std_condition:
                old_value = image[center_pixel_x, center_pixel_y]
                new_value = old_value * e
                enhance_image[center_pixel_x, center_pixel_y] = min(255, new_value)

    return enhance_image

grid = (11, 11)
enhanced_image = localEqualization(gray_image, grid)

cv.imwrite(f'./image/local_equalization/output_local_{grid}.jpg', enhanced_image)

display(HTML('<h3>3 x 3 </h3>'))
display(Image(f"./image/local_equalization/output_local_(3, 3).jpg") )

display(HTML('<h3>7 x 7 </h3>'))
display(Image(f"./image/local_equalization/output_local_(7, 7).jpg") )

display(HTML('<h3>11 x 11 </h3>'))
display(Image(f"./image/local_equalization/output_local_(11, 11).jpg") )
```

3 x 3

7 x 7

11 x 11



> **จากผลการรัน เราจะเห็นได้ว่าผลลัพธ์ที่ได้ค่อนข้างแตกต่างจาก gamma correction และ global equalization โดย**
>
> **local equalization จะปรับเฉพาะส่วนด้านขวามือ (เส้นสีเทาข้างหลัง) ให้สว่างขึ้นโดยไปยุ่งกับส่วนที่ไม่เกี่ยวข้องน้อยมาก จากการลองรัน neighbor ขนาด 3 x 3, 7 x 7, 11 x 11 จะเห็นได้ชัดว่าภาพมีแนวโน้มที่จะมีคุณภาพมากขึ้นตามจำนวน neighbor ที่ใหญ่ขึ้น จะสังเกตว่า 3 x 3 จะมีความไม่ต่อเนื่องของการเติมความสว่างบ้าง และค่อยๆลดลงเมื่อเปลี่ยนเป็น 7 x 7 และ 11 x 11 ทั้งนี้ ขึ้นอยู่กับการปรับค่า parameter (k0, k1, k2, e) ด้วย**

โดยจากการทดลองได้ parameter ที่ทำให้ได้ผลลัพธ์ในระดับที่รับได้คือ

```
Best performance for 3 x 3
k0 = 0.00055
k1 = 0.0000002
k2 = 0.000012
e = 3.5

acceptable performance for 7 x 7
k0 = 0.00075
k1 = 0.000002
k2 = 0.0001
e = 3.5

acceptable performance for 11 x 11
k0 = 0.00055
k1 = 0.0000002
k2 = 0.000012
e = 3.5
```

> **โดยสรุป ในวิธีการทั้ง 3 วิธี วิธีที่ได้ผลลัพธ์ดีที่สุด คือ Local equalization เนื่องจากเกิดความเสียหายกับส่วนที่ไม่ต้องการปรับแต่งน้อยที่สุด แต่ก็มีข้อเสียคือ ต้องใช้เวลาในการหา parameter ที่ดีที่สุด**