1) a. w is const. then we can

take her out from the integrl.

so, we get MSE and MAD functions

and according the lecture 3

$$\hat{f}_2(t) = \sum \gamma_i \cdot 1_{\Delta_i}(t) = \sum_{i=1}^{N} \left(\frac{1}{\Delta_i} \cdot \int_{\Delta_i} y(t)\,dt \right) \cdot 1_{\Delta_i}(t)$$

$$\hat{f}_1(t) = \text{"} = \sum_{i=1}^{N} \theta_{i\,median} \cdot 1_{\Delta_i}(t)$$

b. $$\varepsilon^2(f, \hat{f}) = \int_0^1 \left| f(x) - \hat{f}(x) \right|^2 \cdot w(x)\,dx$$

linearity of the integral $$= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} \left| f(x) - \hat{f}(x) \right|^2 \cdot w(x)\,dx$$

$$= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} \left| f(x) - \gamma_i 1_{\Delta_i} \right|^2 \cdot w(x)\,dx$$

We'll derive and equate to zero:

$$\frac{\partial \varepsilon^2(f, \hat{f})}{\partial \gamma_i} = \frac{\partial}{\partial \gamma_i} \left(\sum_{i=1}^{\tilde{}} \int_{\Delta_i} \left| f(x) - \hat{f}(x) \right|^2 \cdot w(x)\,dx \right)$$

$$\overset{1}{=} \frac{\partial}{\partial \gamma_i} \int_{\Delta_i} \left| f(x) - \hat{f}(x) \right|^2 \cdot w(x)\,dx$$

$$\overset{2}{=} \int_{\Delta_i} \frac{\partial}{\partial \gamma_i} \left| f(x) - \gamma_i 1_{\Delta_i} \right|^2 \cdot w(x)\,dx$$

e

3

$$= \int_{\Delta_i} -\left(2f(x) - 2\gamma_i^* 1_{\Delta_i}(x)\right) \cdot w(x)\,dx = 0$$

$$\Rightarrow \quad \int_{\Delta_i} 2f(x)\cdot w(x)\,dx = \int_{\Delta_i} 2\gamma_i^* 1_{\Delta_i}(x)\cdot w(x)\,dx$$

$$\Rightarrow \quad \gamma_i^* = \frac{\int_{\Delta_i} f(x)\cdot w(x)\,dx}{\int_{\Delta_i} w(x)\,dx}$$

1 all the parts quaks to zero

ecsep the part of the γ_i

2 acorrding to Leibniz's theorem.

3 rulles of derivation

c. $$\mathcal{E}(f,\hat{f}) = \int_0^1 |f(x)-\hat{f}(x)| \cdot w(x)\,dx$$

$$= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} |f(x)-\hat{f}(x)| \cdot w(x)\,dx$$

$$= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} |f(x)-\gamma_i 1_{\Delta_i}| \cdot w(x)\,dx$$

We'll derive and equate to zero:

$$\frac{\partial \mathcal{E}(f,\hat{f})}{\partial \gamma_i} = \frac{\partial}{\partial \gamma_i}\left(\sum_{i=1}^{\tilde{}} \int_{\Delta_i} |f(x)-\gamma_i 1_{\Delta_i}| \cdot w(x)\,dx \right)$$

$$= \frac{\partial}{\partial \gamma_i} \int_{\Delta_i} |f(x)-\gamma_i 1_{\Delta_i}| \cdot w(x)\,dx$$

reasons like before

$$= \int_{\Delta_i} \frac{\partial}{\partial \gamma_i} |f(x)-\gamma_i 1_{\Delta_i}| \cdot w(x)\,dx$$

$$= \int_{\Delta_i} \frac{\partial}{\partial \gamma_i} \sqrt{(f(x)-\gamma_i 1_{\Delta_i})^2} \cdot w(x)\,dx$$

$$= \int_{\Delta_i} -\frac{1}{2\sqrt{(f(x)-\gamma_i 1_{\Delta_i})^2}} \cdot 2(f(x)-\gamma_i 1_{\Delta_i}) w(x)\,dx$$

$$= -\int \operatorname{sign}(f(x)-\gamma_i^*) \cdot w(x) \cdot dx = 0$$

$\Rightarrow$ γ_i is the weighted median

$\Rightarrow$ $$\hat{f} = \sum_{i=1}^{\tilde{}} \gamma_i^* \cdot 1_{\Delta_i}$$

d. $\varepsilon^p(f, \hat{f}) = \int_0^1 |f(x) - \hat{f}(x)|^p \cdot w(x)\,dx$

linearity of the integral $= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} |f(x) - \hat{f}(x)|^p \cdot w(x)\,dx$

$= \sum_{i=1}^{\tilde{}} \int_{\Delta_i} |f(x) - \gamma_i 1_{\Delta_i}|^p \cdot w(x)\,dx$

We'll define $\varepsilon_i^p(f_i, \hat{f}_i) = \int_{\Delta_i} |f_i(x) - \hat{f}_i(x)|^p \cdot w(x)\,dx$

$= \int_{\Delta_i} |f_i(x) - \gamma_i 1_{\Delta_i}|^p \cdot w(x)\,dx$

because each f_i is defined solely by the γ_i. the term ε_i^p are independent for each i.

and we got $\varepsilon^p = \sum \varepsilon_i^p$, as requested.

e.

i define $W_{f_i,\hat{f}_i} = |f_i - \hat{f}_i|^{p-2} > 0$

and than

$$|f_i - \hat{f}_i|^p = |f_i - \hat{f}_i|^{p-2} \cdot (f_i - \hat{f}_i)^2 = W_{f_i,\hat{f}_i} \cdot |f_i - \hat{f}_i|^{p-2}$$

as requested.

ii $$\varepsilon_i^p(f_i, \hat{f}_i) = \int |f(x) - \hat{f}(x)|^p \cdot w(x)\, dx$$

$$= \int |f(x) - \hat{f}(x)|^2 \cdot w'(x)\, dx$$

where $$w'(x) = |f(x) - \hat{f}(x)|^{p-2} \cdot w(x)$$

iii because it's was like clause b.

but now we have differentiate the

term s.t. it's diffecult to derivtion.

iv the term w' isn't desined for

$p = 1$ and $f_i(x) = \hat{f}_i(x)$

v define function calculate$(f_i, \hat{f}_i)$ s.t.

return weight function $w_i' = W_{f_i,\hat{f}_i}$.

The algorithm:

input: f_i, initial $\hat{f}_i$.

inifialzyion: $w_i = calculate(f, \hat{f})$

Iterate:

1. $\hat{f}_i^{next} = \min \int (f - \hat{f})^2 w_i \, dx$

2. $\hat{f}_i = \hat{f}_i^{next}$

3. $w_i = calculate(f, \hat{f})$

f) The algorithm:

input: f_i, w_i, $\int$ (small number) .

inifialzyion: for each interval

$\hat{f}_i = 0$, $w_i = \min\left\{\frac{1}{\varepsilon}, |f - \hat{f}|^{p-2} w\right\}$

Iterate: (until the error smaller than the given $\int$)

1. $\gamma_i^* = \frac{\int_i f \cdot w \, dx}{\int_i w \, dx}$

2. $\hat{f}_i = \gamma_i^* \cdot 1_{\Delta_i}$

3. $w_i = \min\left\{\frac{1}{\varepsilon}, |f - \hat{f}|^{p-2} w\right\}$

g. IRLS

2

a) $\int_{x \in \Delta_i} (x - N_i)^K \, dx = \left. \frac{(x - N_i)^{K+1}}{K+1} \right|_{x = \frac{i}{N}}^{x = \frac{i-1}{N}}$ ‼

$= \frac{\left(\frac{i}{N} - N_i\right)^{K+1}}{(K+1)} - \frac{\left(\frac{i-1}{N} - N_i\right)^{K+1}}{(K+1)}$

N_i is the center of $\left(\frac{i-1}{N}, \frac{i}{N}\right)$

$= \frac{\left(\frac{i}{N} - \left(\frac{i}{N} - \frac{1}{2N}\right)\right)^{K+1}}{(K+1)} - \frac{\left(\frac{i-1}{N} - \left(\frac{i-1}{N} + \frac{1}{2N}\right)\right)^{K+1}}{(K+1)}$

$= \frac{\left(\frac{1}{2N}\right)^{K+1}}{(K+1)} - \frac{\left(-\frac{1}{2N}\right)^{K+1}}{(K+1)}$

$= \frac{\left(\frac{1}{N}\right)^{K+1} - \left(-\frac{1}{N}\right)^{K+1}}{2^{K+1}(K+1)}$

- if K odd $\Rightarrow K+1$ even $\Rightarrow \left(-\frac{1}{N}\right)^{K+1} = \left(\frac{1}{N}\right)^{K+1}$

$\Rightarrow \frac{\left(\frac{1}{N}\right)^{K+1} - \left(\frac{1}{N}\right)^{K+1}}{2^{K+1}(K+1)} = \boxed{0}$

if K even $\Rightarrow K+1$ odd $\Rightarrow \left(-\frac{1}{N}\right)^{K+1} = -\left(\frac{1}{N}\right)^{K+1}$

$\Rightarrow \frac{\left(\frac{1}{N}\right)^{K+1} + \left(\frac{1}{N}\right)^{K+1}}{2^{K+1}(K+1)} = \frac{2\left(\frac{1}{N}\right)^{K+1}}{2^{K+1}(K+1)} = \boxed{\frac{|\Delta_i|^{K+1}}{2^K (K+1)}}$

b) $\int_{[0,1)} (\phi(x) - \hat{\phi}(x))^2 dx = \sum_{i=1}^{N} \int_{\Delta_i} (\phi(x) - \hat{\phi}(x))^2 dx$

$$= \sum_{i=1}^{N} \int_{\Delta_i} \phi^2(x) dx - 2 \int \phi(x) \hat{\phi}(x) dx + \int \hat{\phi}^2(x) dx$$

$$MSE(\phi, \hat{\phi}) = \sum_{i=1}^{N} \int \phi^2(x) dx - 2 \int \phi(x) (a_i (x - x_i) + c_i) dx +$$
$$+ \int (a_i (x - x_i) + c_i)^2 dx$$

$$\frac{\partial MSE(\phi, \hat{\phi})}{\partial a_i} = -2 \int \phi(x) \frac{\partial}{\partial a_i} (a_i (x - x_i) + c_i) dx + \int \frac{\partial}{\partial a_i} (a_i (x - x_i) + c_i)^2 dx$$

$$= -2 \int \phi(x) (x - x_i) dx + 2 \int (a_i (x - x_i) + c_i)(x - x_i) dx$$

$$= -2 \int \phi(x)(x - x_i) dx + 2 \int a_i (x - x_i)^2 dx + \underbrace{2 c_i \int (x - x_i) dx}_{=0}$$

$$= -2 \int \phi(x)(x - x_i) dx + 2 \int a_i (x - x_i)^2 dx$$

compare to 0: $\int \phi(x)(x - x_i) dx = a_i \frac{\Delta_i^3}{2^2 (2+1)}$ (precedent question)

$$= \frac{a_i}{12 N^3}$$

$$\Rightarrow \boxed{a_i = 12 N^3 \int \phi(x)(x - x_i) dx}$$

$$\frac{\partial MSE(\phi,\hat{\phi})}{\partial c_i} = -2\int \phi(x)dx + 2\int a_i(x-x_i)+c_i\,dx$$

$$= -2\int \phi(x)dx + 2a_i \underbrace{\int (x-x_i)dx}_{=0} + 2\int c_i\,dx$$

$$= -2\int \phi(x)dx + 2\int c_i\,dx$$

compare to 0:

$$\int \phi(x)dx = \frac{c_i}{N}$$

$$\Rightarrow \boxed{c_i = N\int \phi(x)dx}$$

c) $$MSE = \sum_{i=1}^{N}\int \phi(x) - (a_i(x-x_i)+c_i)^2\,dx$$

$$\boxed{= \sum_{i=1}^{N}\int\left(\phi(x) - 12N^3\int \phi(x)(x-x_i)dx\,(x-x_i) - N\int\phi(x)dx\right)^2 dx}$$

d) The optimal MSE with constant wise approximation on each interval is

$$\sum_{i}^{N}\int\left(\phi(x) - N\int\phi(x)\right)^2 dx$$

which is equivalent to $\sum_{i=1}^{N}\int(\phi(x)-c_i)^2dx$ ($a_i = 0$ in this case)

So chosing optimal a_i which is $12N^3\int\phi(x)(x-x_i)dx$ can only decrease the value of the MSE

In [1]:

```
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Quantization

Q1

In [2]:

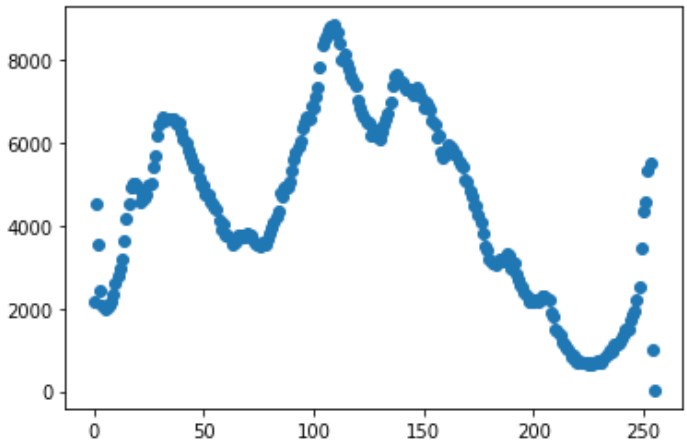
```
#Open the image as greyscale
im = Image.open(r"./Truman_receives_menorah.jpg").convert('L')
```

In [3]:

```
#We normalize the distribution of pixels of each colour in the greyscale image
#Thus an xpproximation of the pdf is achieved
num_of_pixels = im.size[0] * im.size[1]
hist = pd.Series(im.histogram())
pdf = hist.apply(lambda x: x/num_of_pixels)

# plotting the histogram for clause 1
plt.scatter(hist.index, hist)
```

Out[3]:

```
<matplotlib.collections.PathCollection at 0x7f4412968690>
```



In [4]:

```
print(num_of_pixels)
```

```
1138800
```

Q2. a)

In [5]:

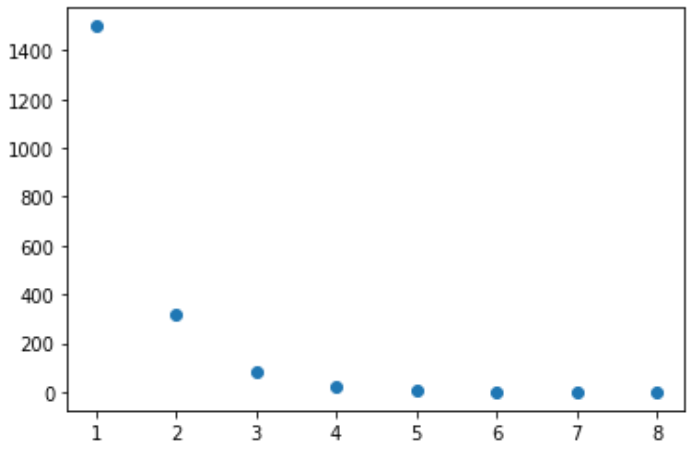
```
def uni_quantize(pixel_color, interval_size, start_of_d_l):
    """ Returns the uniform quantizer for the received pixel_color."""
    return interval_size*((pixel_color-start_of_d_l)//interval_size + 0.5) + start_of_d_
l

def mse_term(pixel_color, pixel_quantization):
    """ Returns the MSE term. """
    return np.power(pixel_color-pixel_quantization, 2)
```

```
def mse_uniform(pdf, function_range, num_of_intervals, start_of_decision_boundries):
    """ Returns the total uniform MSE by calculating the weighted sum of the MSE terms.
        This calculation is equivalent to summing all the terms of the image pixels and d
ividing it by the total
        number of pixels in the image. """
    total_mse = 0
    for color in range(len(pdf)):
        total_mse += pdf[color] * mse_term(color, uni_quantize(color, function_range / n
um_of_intervals, start_of_decision_boundries))
    return total_mse


mse_for_b_bits = [0] * 8  # the array in which we will store the MSE for every b
start_of_decision_boundries = -0.5
function_range = 256
# filling the array for every b
for b in range(1,9):
    num_of_intervals = np.power(2, b)
    mse_for_b_bits[b-1] = mse_uniform(pdf, function_range, num_of_intervals, start_of_de
cision_boundries)
    print("for b = %s: %s" % (b, mse_for_b_bits[b-1]) )

mse_for_b_bits_uni_series = pd.Series(mse_for_b_bits)
plt.scatter(mse_for_b_bits_uni_series.index + 1, mse_for_b_bits_uni_series)
```

```
for b = 1: 1499.3780874604845
for b = 2: 319.1266508605549
for b = 3: 84.34553389532843
for b = 4: 21.239724271162626
for b = 5: 5.206945907973306
for b = 6: 1.2472971548998946
for b = 7: 0.2500000000000002
for b = 8: 0.0
```

Out[5]:

```
<matplotlib.collections.PathCollection at 0x7f441564a450>
```



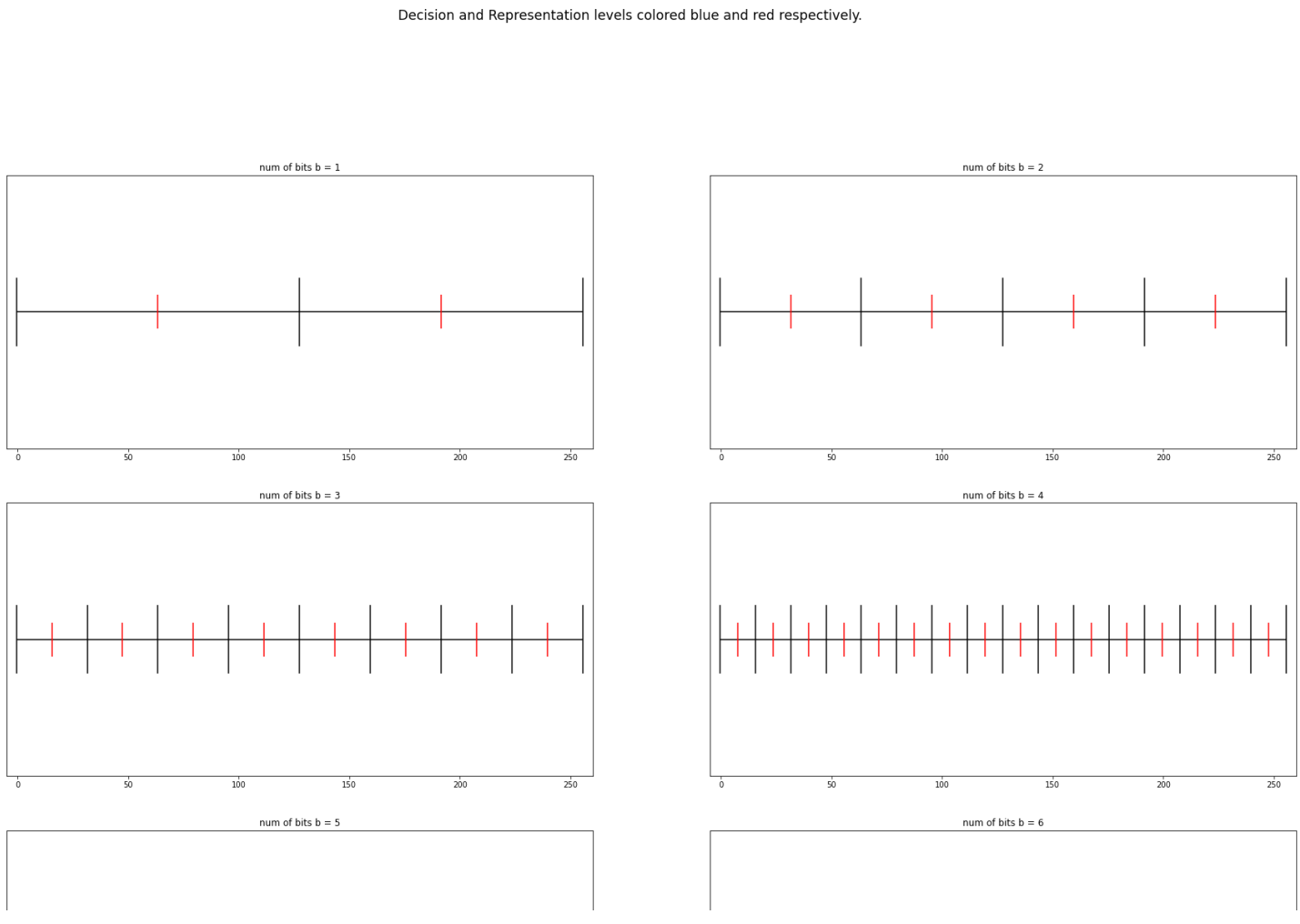
Q2. b)

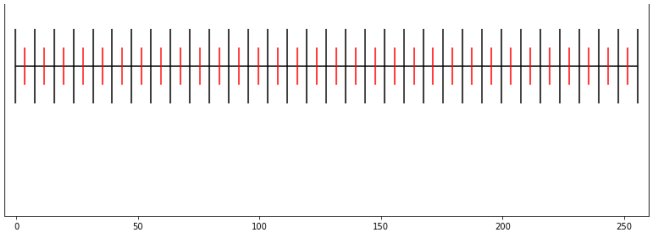
In [7]:

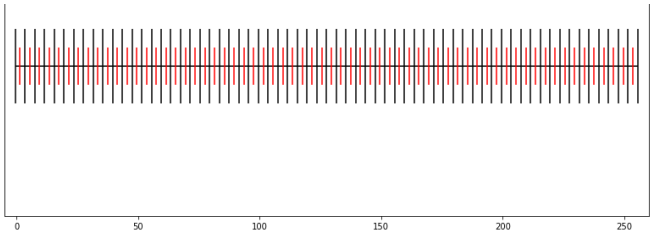
```
def get_uni_boundries(b, func_range):
    """ Returns an array with the uniform boundaried (decision levels),
        with the lowest boundary at -0.5."""
    uni_boundries = [-0.5]
    num_of_reps = np.power(2,b)
    interval_size = func_range / num_of_reps

    for i in range(num_of_reps):
        uni_boundries.append(uni_boundries[i] + interval_size)

    return uni_boundries
```

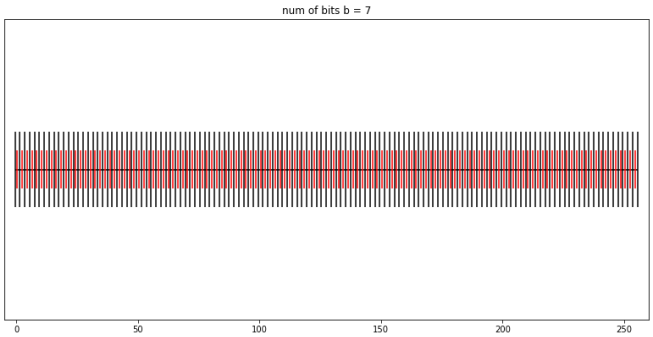
In [8]:

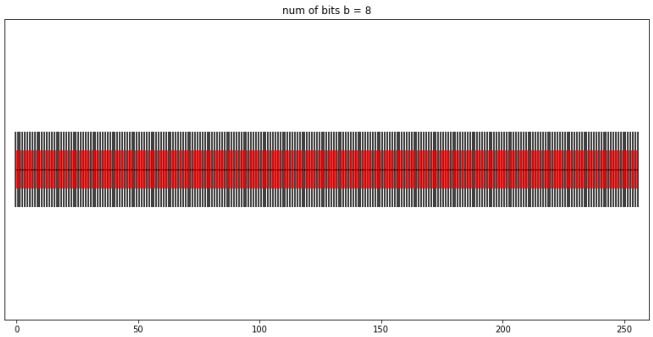
```
uni_boundries = [get_uni_boundries(b, 256) for b in range(1,9)]

uni_reps = [[], [], [], [], [], [], [], []]  # a list of lists containing the uniform rep
resantation levels for every b
for b in range(1,9):
    for i in range(len(uni_boundries[b-1])-1):
        # the representator is in the middle of the interval
        uni_reps[b-1].append((uni_boundries[b-1][i]+uni_boundries[b-1][i+1])/2)

# set up the figure
def plot_quantization(boundries_list, reps_list):
    fig = plt.figure(figsize=(30, 30))
    fig.suptitle("Decision and Representation levels colored blue and red respectively.",
size="xx-large")
    for b in range(1,9):
        ax = fig.add_subplot(4,2,b)
        ax.set_xlim(-5,260)
        ax.set_ylim(0,4)

        ax.set_title("num of bits b = %d" % (b))

        # draw lines
        xmin = -0.5
        xmax = 255.5
        y = 2
        height = 1

        plt.hlines(y, xmin, xmax)

        plt.vlines(boundries_list[b-1], y - height / 2., y + height / 2.)

        plt.vlines(reps_list[b-1], y - height / 4., y + height / 4., linestyles="solid",
colors="red")
        ax.get_yaxis().set_visible(False)
    # plt.tight_layout()
    plt.show()

plot_quantization(uni_boundries, uni_reps)
```











Q3.

In [9]:

```
def calculate_r_i(pdf, lower_boundry, higher_boundry):
    """ Returns the representator/quantizer of the interval."""
    weighted_sum = 0
    prob_sum = 0
    i = np.ceil(lower_boundry)
    while i < higher_boundry:
        weighted_sum += i*pdf[i]
        prob_sum += pdf[i]
        i+=1
    return weighted_sum / prob_sum


def ml_quantizer(color, decision_levels, rep_levels):
    """ Returns the value of Q(color)."""
    for i in range(len(decision_levels) - 1):
        if decision_levels[i] <=color and color < decision_levels[i+1]:
            # found the color's interval, return its representator
            return rep_levels[i]


def mse_ml(pdf, decision_levels, rep_levels):
    """ Returns the total MSE for the given quatization.
        This calculation is equivalent to summing all the terms of the image pixels and d
ividing it by the total
        number of pixels in the image."""
    total_mse = 0
    for color in range(len(pdf)):
        total_mse += pdf[color] * mse_term(color, ml_quantizer(color, decision_levels, r
ep_levels))
    return total_mse

#initial d contains j+1 values, where the first and last are phi low and high respectivel
y
def max_loyd(pdf, initial_d, epsilon):
    d = initial_d
    r = [calculate_r_i(pdf, d[i], d[i+1]) for i in range(len(d) - 1)]
    last_mse = mse_ml(pdf, d, r)
    curr_mse = last_mse
    last_diff = epsilon
    while(last_diff>=epsilon):  # will stop iterating when the diff improves by less than
epsilon
        for i in range(len(r)):
            if i!=0:
                d[i] = (r[i-1]+r[i])/2
        r = [calculate_r_i(pdf, d[i], d[i+1]) for i in range(len(d) - 1)]
        last_mse = curr_mse
```

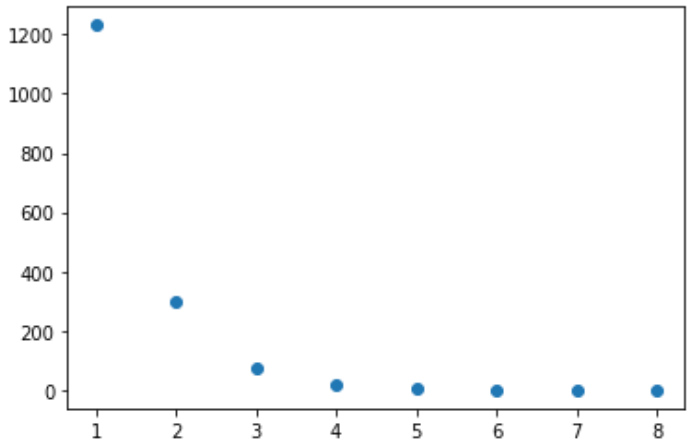
```
        curr_mse = mse_ml(pdf, d, r)
        last_diff = last_mse - curr_mse
    return r,d,curr_mse
```

Q4. a+b)

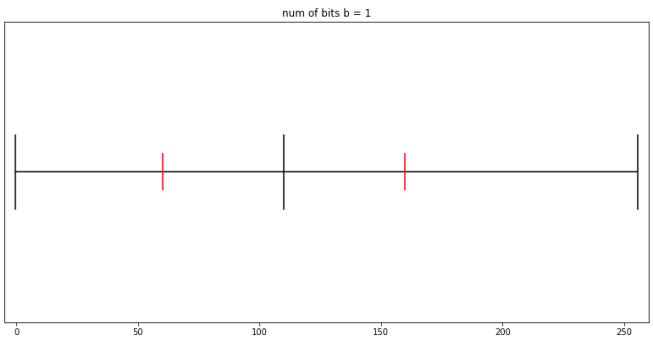
In [10]:

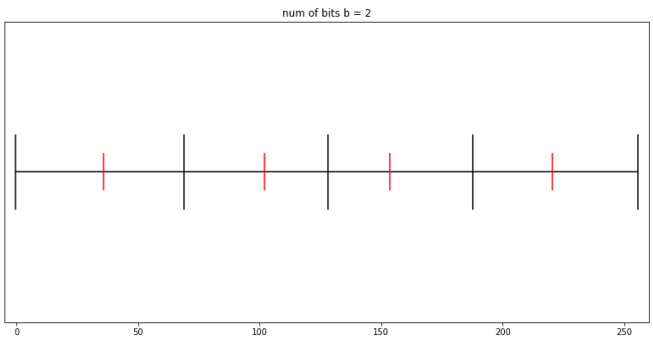
```
mse_for_b_bits = [0] * 8  # array for the MSE for every b
r_list = []  # will be a list of lists, with the representation levels for every b
d_list = []  # will be a list of lists, with the decision levels for every b
for b in range(1,9):
    r, d, mse_for_b_bits[b-1] = max_loyd(pdf, get_uni_boundries(b, 256), 0.5)
    r_list.append(r)
    d_list.append(d)
    print("for b = %s: %s" % (b, mse_for_b_bits[b-1]) )
# clause a: MSE as a function
mse_for_b_bits_ML_series = pd.Series(mse_for_b_bits)
plt.scatter(mse_for_b_bits_ML_series.index + 1, mse_for_b_bits_ML_series)

# clause b: plot of the decision and represantation levels
plot_quantization(d_list, r_list)
```

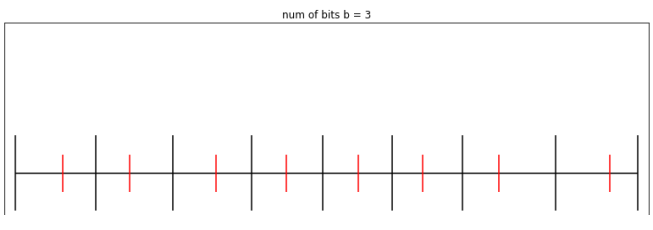
```
for b = 1: 1232.1120718319123
for b = 2: 297.5497108912991
for b = 3: 76.02411774859655
for b = 4: 20.841106053081774
for b = 5: 5.175351900529969
for b = 6: 1.2373538078451563
for b = 7: 0.24943846163111635
for b = 8: 2.547388693050799e-29
```

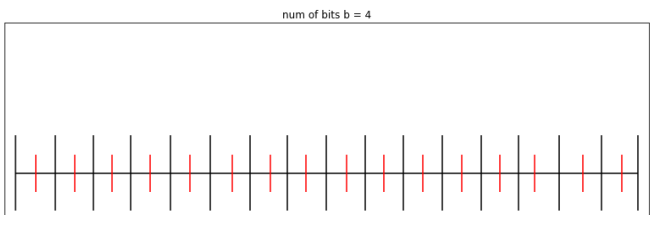


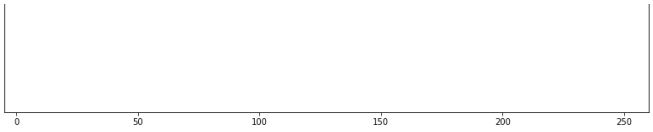
Decision and Representation levels colored blue and red respectively.

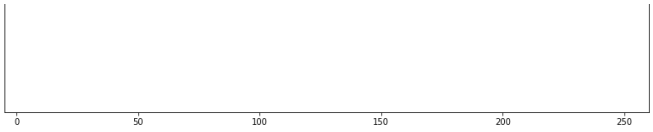


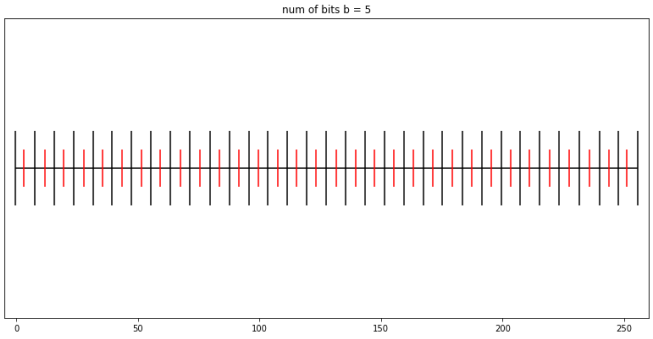


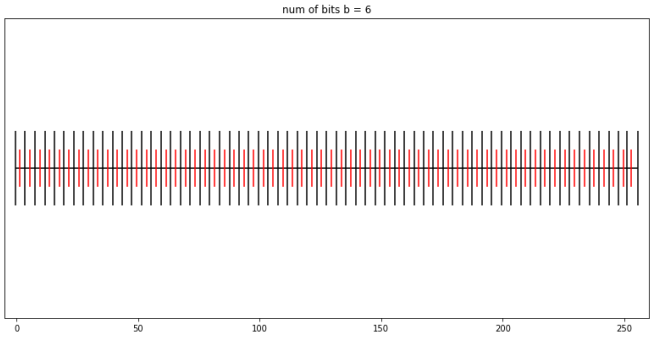


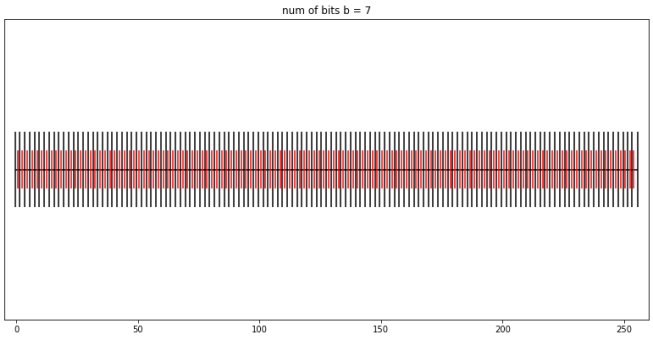


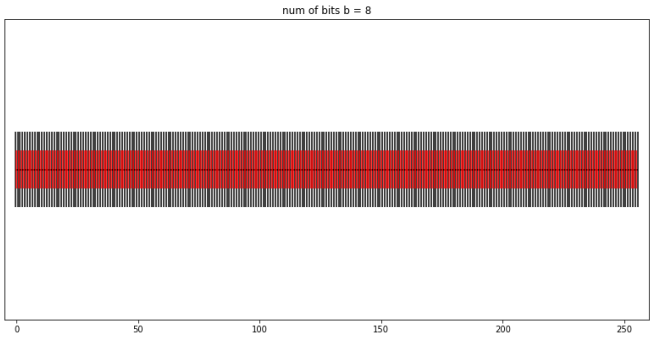








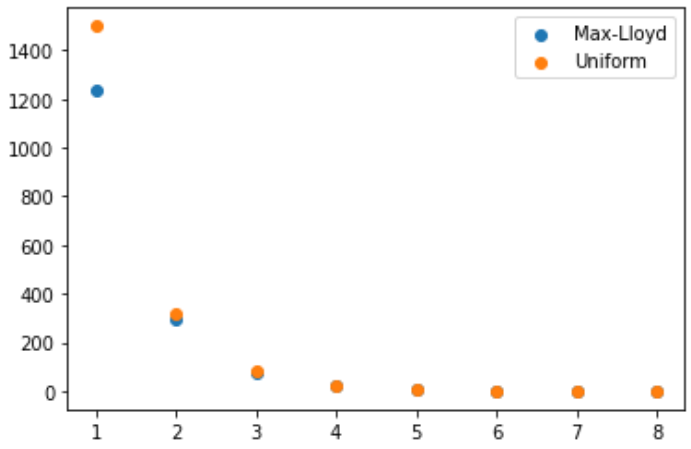




Q4. c)

In [11]:

```
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.scatter(mse_for_b_bits_ML_series.index + 1, mse_for_b_bits_ML_series, label='Max-Lloyd')
ax1.scatter(mse_for_b_bits_uni_series.index + 1, mse_for_b_bits_uni_series, label='Uniform')
plt.legend(loc='upper right')
plt.show()
```



In [1]:

```
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Question 2

Q1. a+b)

In [2]:

```
im = Image.open(r"./Truman_receives_menorah_square.jpg").convert('L')
pix = np.array(im)
side_len = pix.shape[0]
```

In [3]:

```
def get_uni_boundries(b, func_range):
    """ Returns an array with the uniform boundaried, with the lowest boundary at 0."""
    uni_boundries = [0]
    num_reps = np.power(2,b)
    interval_size = func_range / num_reps

    for i in range(num_reps):
        uni_boundries.append(int(uni_boundries[i] + interval_size))

    return uni_boundries
```

In [4]:

```
def get_local_mse(matrix, rep):
    """ Returns the local MSE for the given sample region considering it is represented b
y rep."""
    curr_matrix = pd.DataFrame(matrix)
    curr_matrix -= rep
    curr_matrix = curr_matrix.pow(2)
    return np.mean(curr_matrix.to_numpy().flatten())

def get_local_mad(matrix, rep):
    """ Returns the local MAD for the given sample region considering it is represented b
y rep."""
    curr_matrix = pd.DataFrame(matrix)
    curr_matrix = np.abs(curr_matrix - rep)
    return np.mean(curr_matrix.to_numpy().flatten())
```

In [8]:

```
function_range = 255
MSE_reps = []  # will be list of 2D arrays of represantators of every grid sample region
MSE_errors = []  # MSE error for every b
MAD_reps = []  # will be list of 2D arrays of represantators of every grid sample region
MAD_errors = []  # MSE error for every b
for b in range(1,9):
    MSE_errors.append(0)
    MAD_errors.append(0)
    D_value = np.power(2,b)
    MSE_rep = np.ndarray(shape=(D_value,D_value), dtype=float)
    MAD_rep = np.ndarray(shape=(D_value,D_value), dtype=float)
    list_of_xboundries = get_uni_boundries(b, side_len)
    list_of_yboundries = get_uni_boundries(b, side_len)
    for i in range(len(list_of_xboundries) - 1):
        for j in range(len(list_of_yboundries) - 1):
            lower_x = list_of_xboundries[i]
```

```
            upper_x = list_of_xboundries[i+1]
            lower_y = list_of_yboundries[j]
            upper_y = list_of_yboundries[j+1]
            local_pix = pix[lower_x:upper_x,lower_y:upper_y]

            # in the MSE sense: our represantator is the sample's mean
            sample_mean = np.mean(local_pix)
            MSE_rep[i][j] = sample_mean / function_range
            MSE_errors[b - 1] += get_local_mse(local_pix, sample_mean)

            # in the MAD sense: our represantator is the sample's median
            sample_median = np.median(local_pix)
            MAD_rep[i][j] = sample_median / function_range
            MAD_errors[b - 1] += get_local_mad(local_pix, sample_median)
    MSE_errors[b - 1] /=  np.power(D_value, 2)
    MAD_errors[b - 1] /=  np.power(D_value, 2)
    MSE_reps.append(MSE_rep)
    MAD_reps.append(MAD_rep)
```

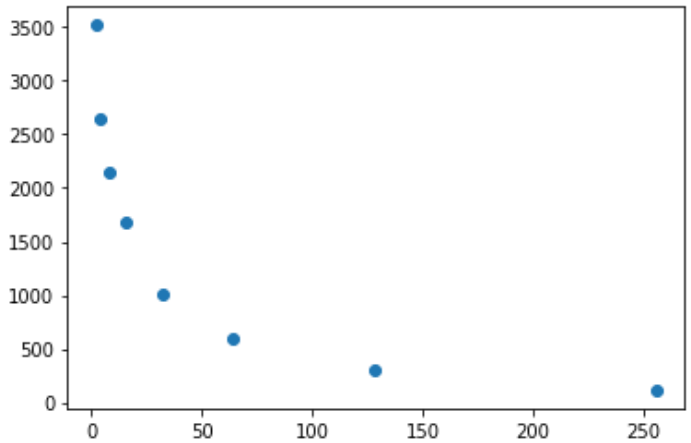
In [9]:

```
for b in range(1,9):
    D_value = np.power(2,b)
    print("mse for D = %s: %s" % (D_value, MSE_errors[b-1]) )

mse_for_b_series = pd.Series(MSE_errors)
plt.scatter(2**(mse_for_b_series.index + 1), mse_for_b_series)
```

```
mse for D = 2: 3519.23942785681
mse for D = 4: 2653.6322570794728
mse for D = 8: 2152.343625162728
mse for D = 16: 1677.3657508902252
mse for D = 32: 1016.9303231984377
mse for D = 64: 599.3637393116951
mse for D = 128: 307.53233790397644
mse for D = 256: 115.3246603012085
```

Out[9]:

```
<matplotlib.collections.PathCollection at 0x7f70aba8b950>
```



In [10]:

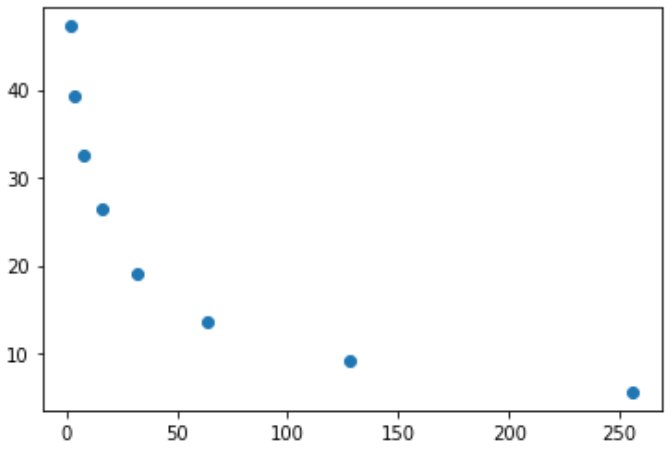
```
for b in range(1,9):
    D_value = np.power(2,b)
    print("mad for D = %s: %s" % (D_value, MAD_errors[b-1]) )

mad_for_b_series = pd.Series(MAD_errors)
plt.scatter(2**(mad_for_b_series.index + 1), mad_for_b_series)
```

```
mad for D = 2: 47.21493911743164
mad for D = 4: 39.22738265991211
mad for D = 8: 32.5818977355957
mad for D = 16: 26.33602523803711
mad for D = 32: 19.047534942626953
mad for D = 64: 13.544551849365234
mad for D = 128: 9.19363784790039
```

```
mad for D = 256: 5.628452301025391
```

Out[10]:

```
<matplotlib.collections.PathCollection at 0x7f70aa9d4e10>
```

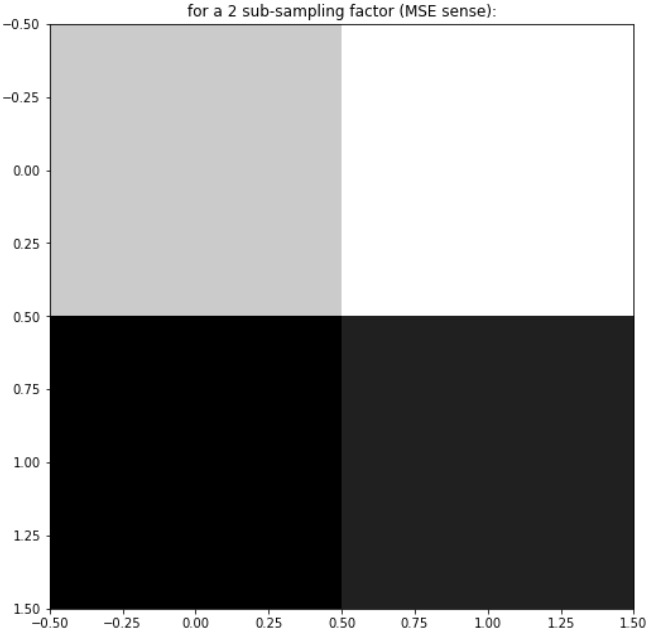


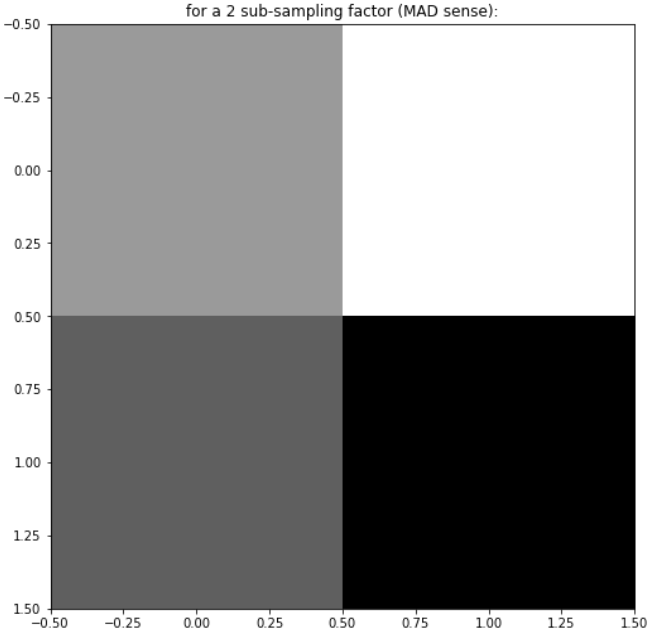
In [11]:

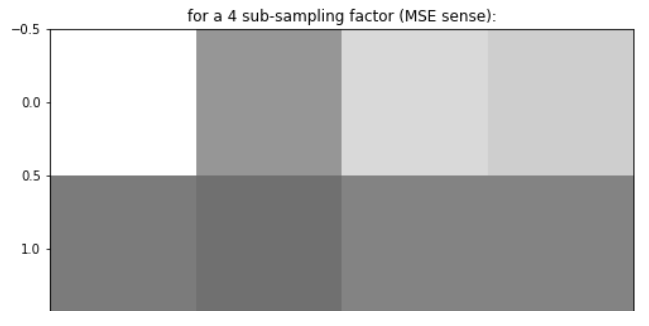
```
def plot_MSE_and_MAD_images(MSE_matrix_list, MAD_matrix_list):
    """ Plots the recieved matrices side by side. """
    plt.figure(figsize=(20,80))
    for b in range(1,9):
        plt.subplot(8,2,2*b-1)
        plt.title(f"for a {np.power(2,b)} sub-sampling factor (MSE sense):")
        plt.imshow(MSE_matrix_list[b-1], cmap='gray')
        plt.subplot(8,2,2*b)
        plt.title(f"for a {np.power(2,b)} sub-sampling factor (MAD sense):")
        plt.imshow(MAD_matrix_list[b-1], cmap='gray')
```

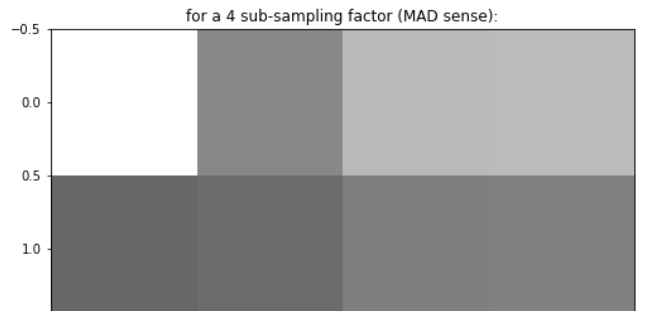
In [12]:

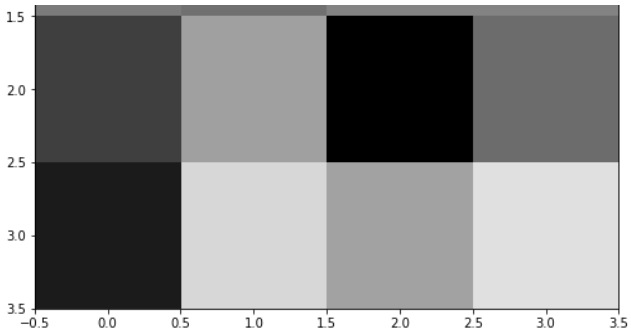
```
# Here we plot the sampled matrices, without reconstruction.
plot_MSE_and_MAD_images(MSE_reps, MAD_reps)
```

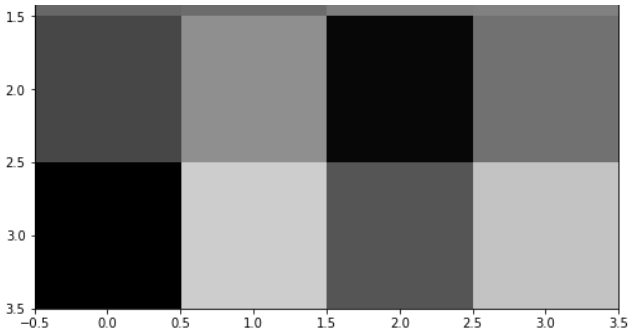




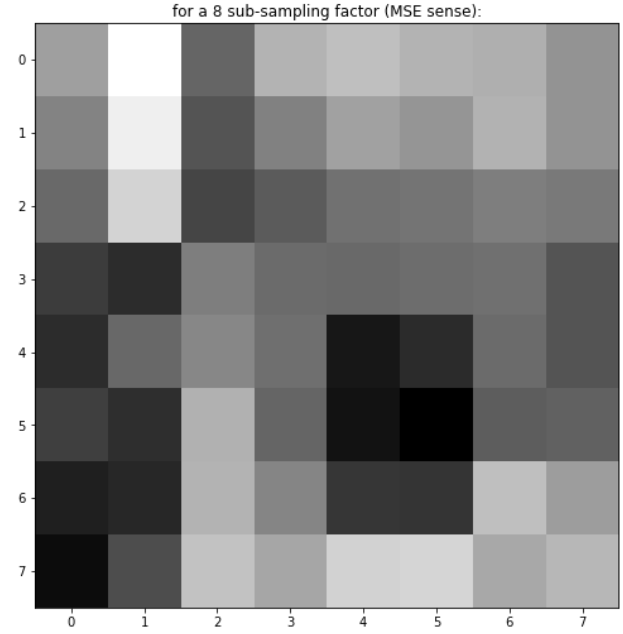




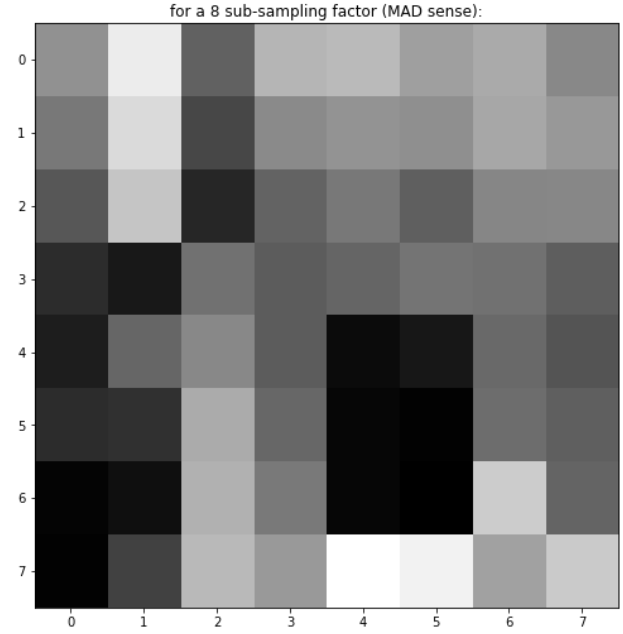




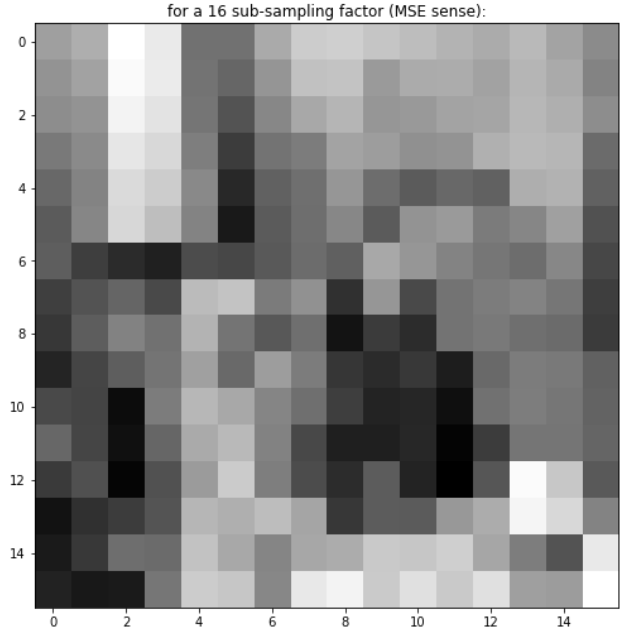
for a 8 sub-sampling factor (MSE sense):



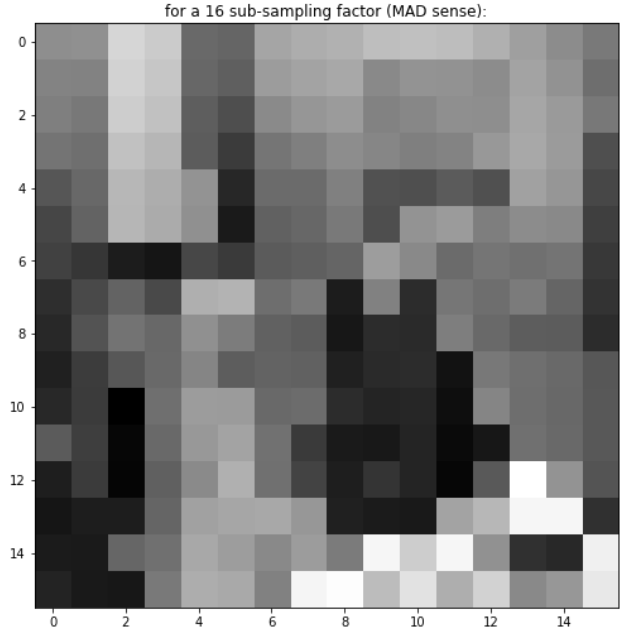
for a 8 sub-sampling factor (MAD sense):



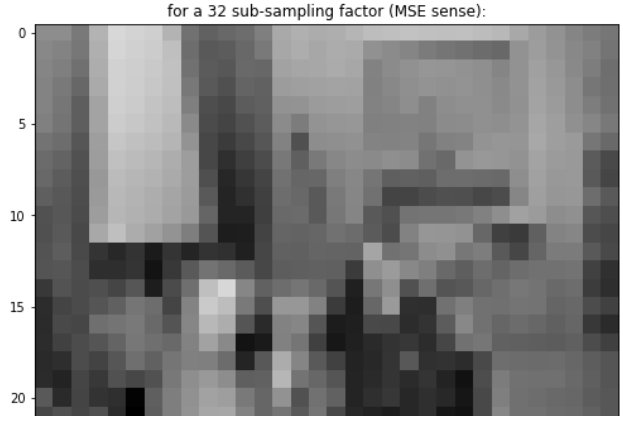
for a 16 sub-sampling factor (MSE sense):



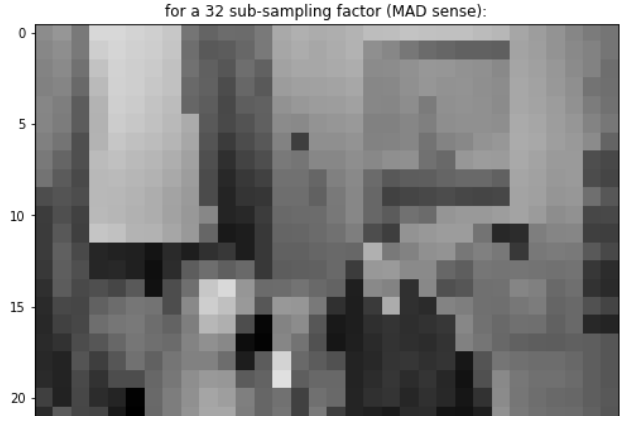
for a 16 sub-sampling factor (MAD sense):

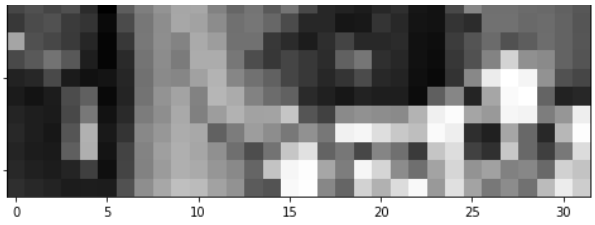


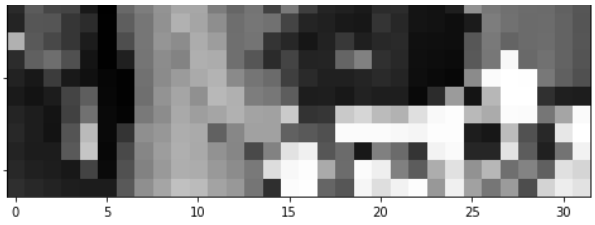
for a 32 sub-sampling factor (MSE sense):



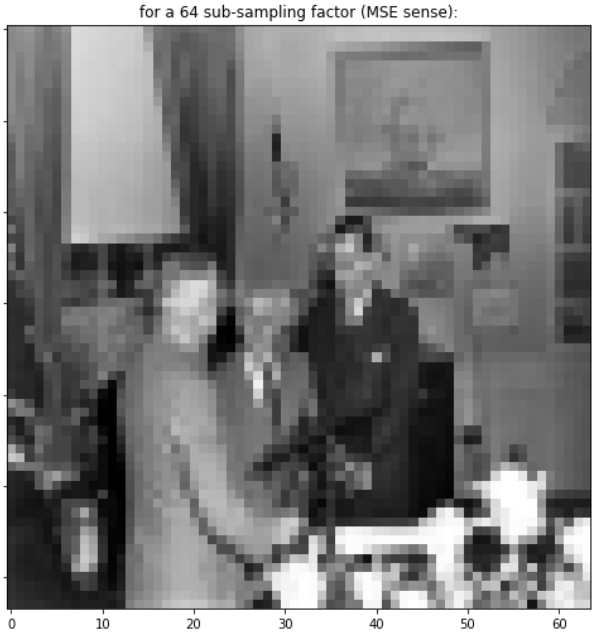
for a 32 sub-sampling factor (MAD sense):



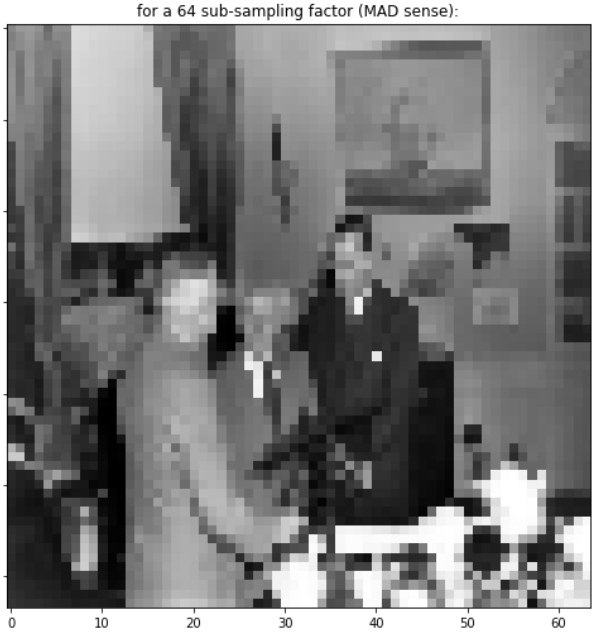




for a 64 sub-sampling factor (MSE sense):



for a 64 sub-sampling factor (MAD sense):



for a 128 sub-sampling factor (MSE sense):

for a 128 sub-sampling factor (MAD sense):

for a 256 sub-sampling factor (MSE sense):

for a 256 sub-sampling factor (MAD sense):

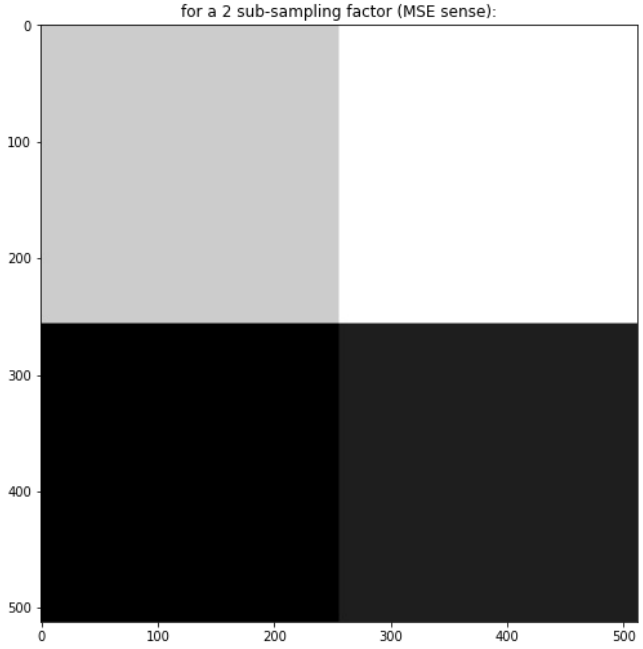
Q2.

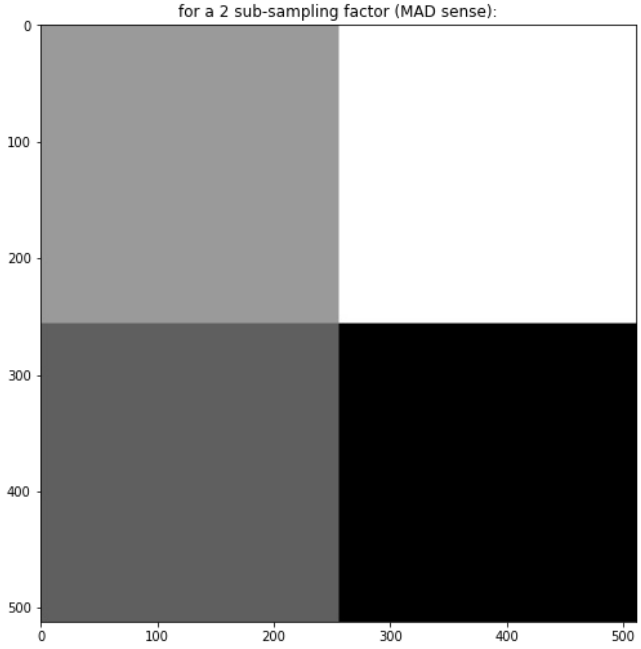
In [13]:

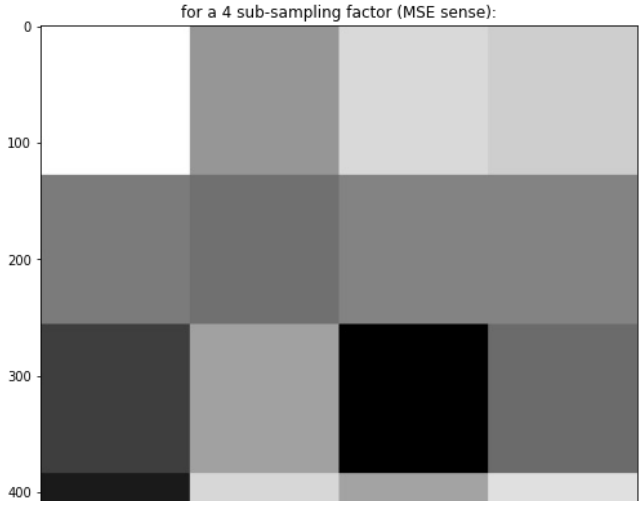
```
# reconstructions are rounded at the offset to the nearest whole
# integer because both for MSE and MAD this minimizes error. For MSE it minimizes error s
ince
# the error can be seperated to representation and quantization errors, and the minium
# quantization error is achieved for the closest quantized value. For MAD it is obvious
# from the MAD's form.

def get_reconstructed_from_reps(reps_list):
    reconstructed_list = []
    for b in range(1,9):
        rep_matrix = reps_list[b-1]
        D_value = np.power(2,9-b)
        new_picture = np.round(np.kron(rep_matrix,np.ones((D_value,D_value))) * function
_range)
        reconstructed_list.append(new_picture)
    return reconstructed_list
```

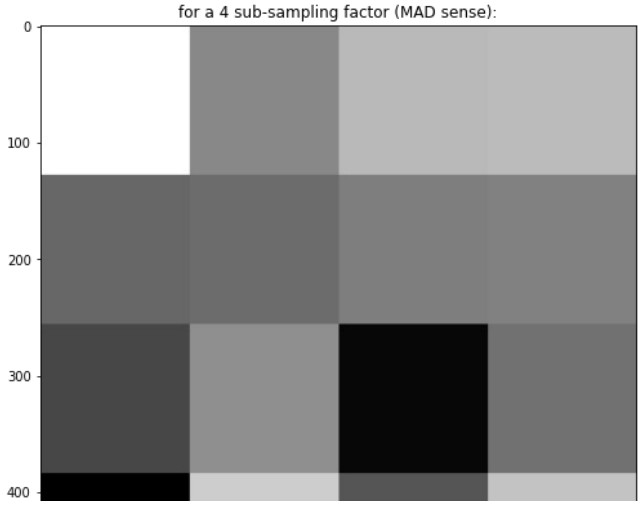
In [14]:

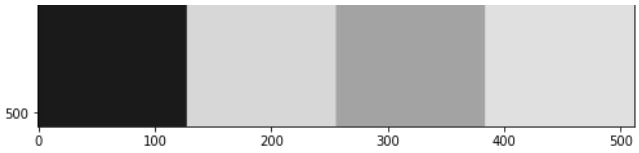
```
# Here we plot the reconstructed images in their full size and integer values.
plot_MSE_and_MAD_images(get_reconstructed_from_reps(MSE_reps),get_reconstructed_from_reps
(MAD_reps))
```

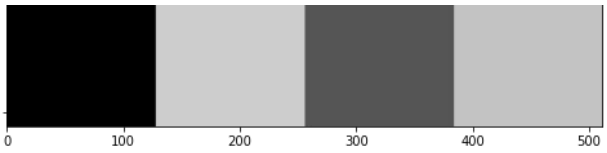




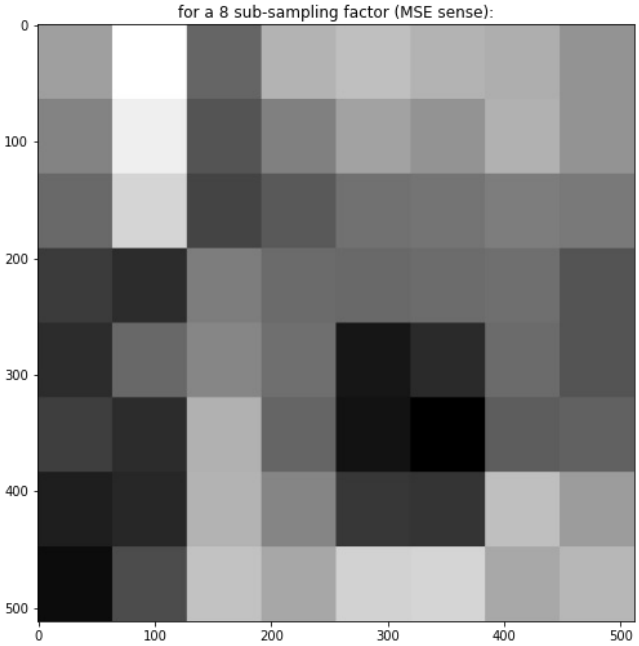




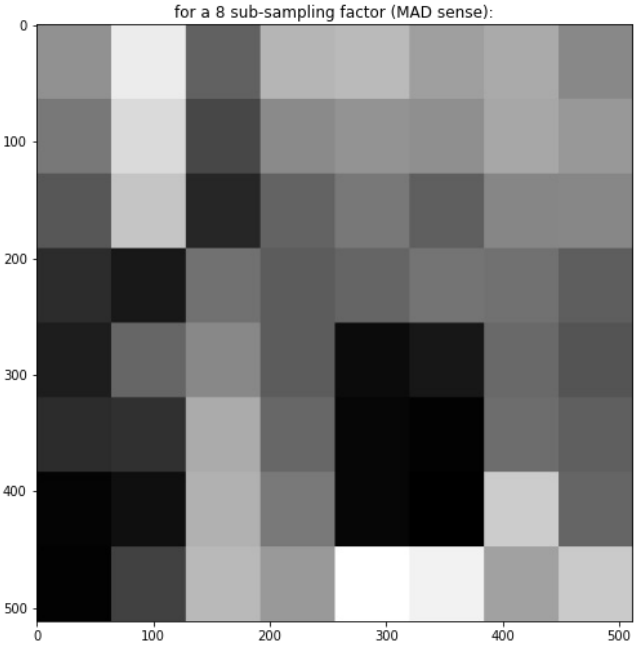




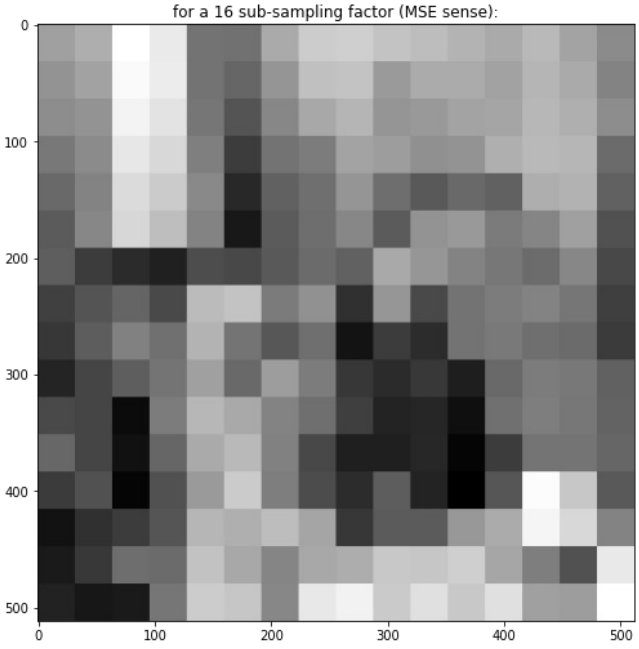
for a 8 sub-sampling factor (MSE sense):



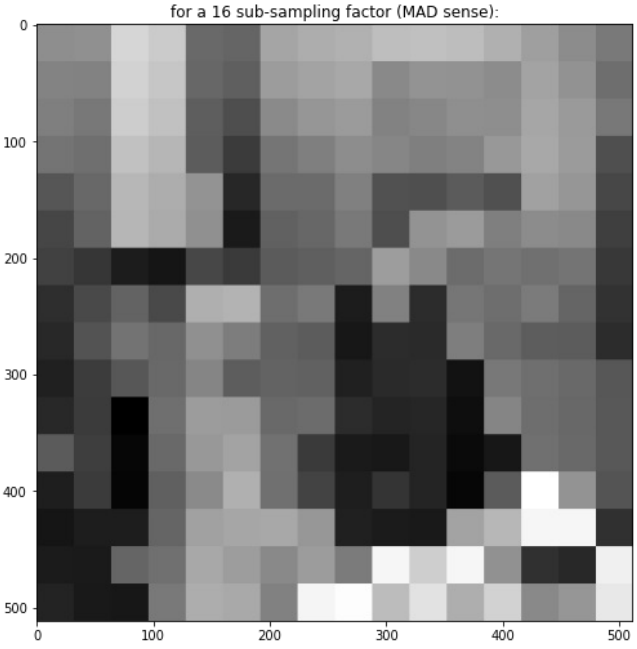
for a 8 sub-sampling factor (MAD sense):



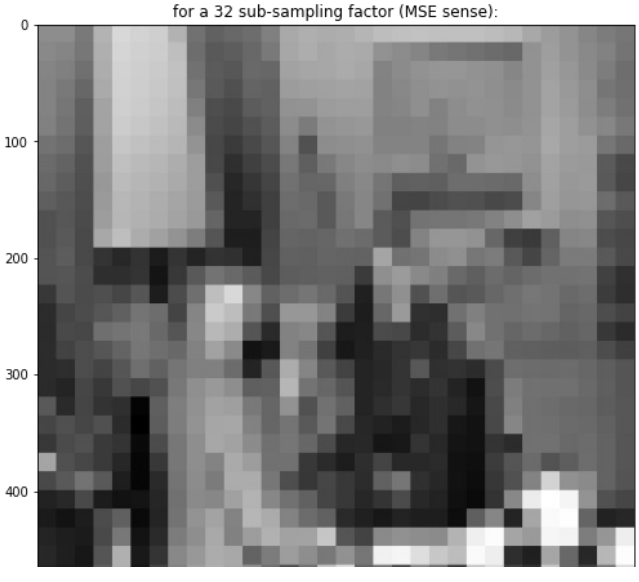
for a 16 sub-sampling factor (MSE sense):



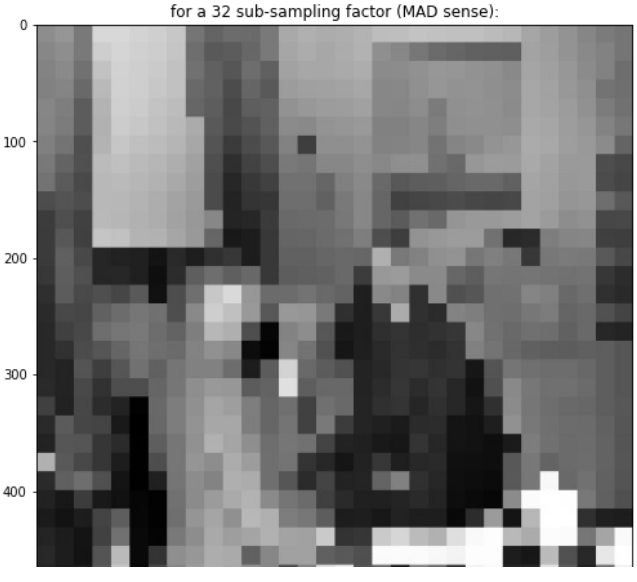
for a 16 sub-sampling factor (MAD sense):

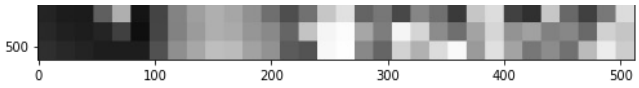


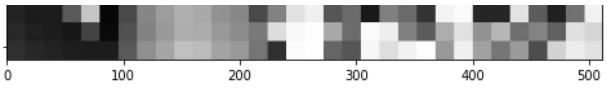
for a 32 sub-sampling factor (MSE sense):



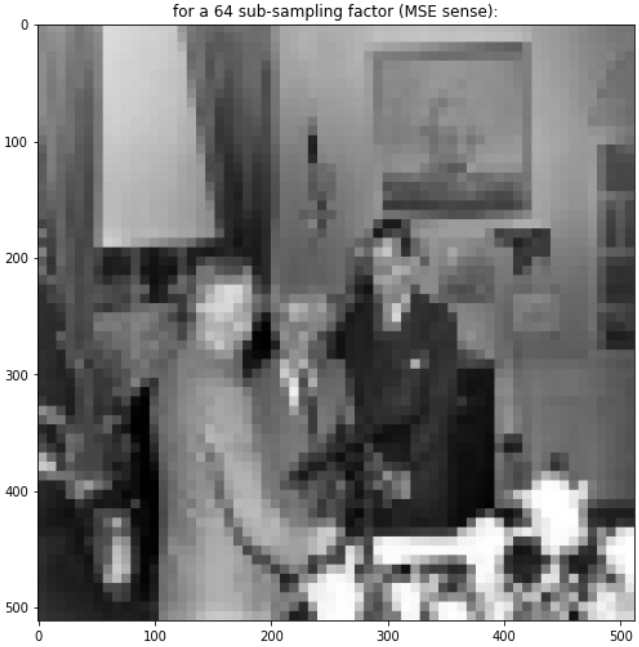
for a 32 sub-sampling factor (MAD sense):



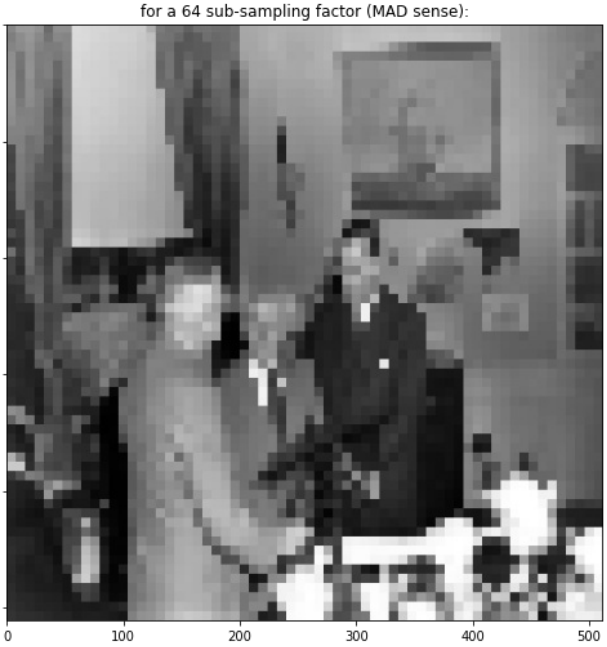




for a 64 sub-sampling factor (MSE sense):



for a 64 sub-sampling factor (MAD sense):



for a 128 sub-sampling factor (MSE sense):

for a 128 sub-sampling factor (MAD sense):

for a 256 sub-sampling factor (MSE sense):

for a 256 sub-sampling factor (MAD sense):

Q2. 3.

As we stated in clause 1, when D values (the number of sub-samples we make) increase, the reconstructed picture is clearer. When D value is lower than 32, it is not clear what the picture shows. For D value of 32 it is hard to understand but possible, and for D values higher than 32 it is easy. This is true for both the MSE and MAD cases.

Q3.1

1. The algorithm:

We will mark Δ_i as a domain $[i', i'+1) \times [j, j+1)$ inside the unit square, one from the $N_x \times N_y$ domains.

Input: $f(x,y), \delta$.

Initialization: for each 2D domain Δ_i, we have $f_i(x,y)$, and we initialize:

$$\hat{f}_i(x,y) \leftarrow 120 \cdot \mathbf{1}_{\Delta_i}(x,y)$$

$$w_i'(x,y) \leftarrow \min\left\{\frac{1}{\epsilon}, \left|f_i(x,y) - \hat{f}_i(x,y)\right|^{p-2} w(x,y)\right\}$$, while $\epsilon > 0$ is a small, fixed number.

Iterate (until the error of the L^p optimization problem changes by some small value smaller than the given δ):

4. $\gamma_i^* \leftarrow \frac{\int_{\Delta_i} f(x,y) w_i'(x,y) dxdy}{\int_{\Delta_i} w_i'(x,y) dxdy}$
5. $\hat{f}_i \leftarrow \gamma_i^* \cdot \mathbf{1}_{\Delta_i}(x,y)$
6. $w_i'(x,y) \leftarrow \min\left\{\frac{1}{\epsilon}, \left|f_i(x,y) - \hat{f}_i(x,y)\right|^{p-2} w(x,y)\right\}$

In [1]:

```
from PIL import Image
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Question 3

Q2.

In [2]:

```
im = Image.open(r"./Truman_receives_menorah_square.jpg").convert('L')
pix = np.array(im)
side_len = pix.shape[0]
```

In [3]:

```
def get_weight_at_point(x,y,epsilon, subsample, curr_coefficiant, p, weight_subsample):
    """ This function calculates the w' of the point x,y in the subsample of the image.
        Meaning, it returns the current updated weight for the point.
        This is done instead of calculating all the values of w' in each iteration and ke
eping thiem in memory."""
    subsample_df = pd.DataFrame(subsample)
    if float(subsample_df[x][y])-curr_coefficiant < 1E-7 and p<2:
        return np.power(epsilon, -1)
    return np.min([np.power(epsilon, -1),
                   np.power(np.abs(float(subsample_df[x][y])-curr_coefficiant), p-2.0) *
weight_subsample[x][y]])
```

In [4]:

```
def get_uni_boundries_from_num_of_reps(num_reps, func_range):
    """ Returns an array with the uniform boundaried, with the lowest boundary at 0."""
    uni_boundries = [0]
    interval_size = func_range / num_reps

    for i in range(num_reps):
        uni_boundries.append(int(uni_boundries[i] + interval_size))

    return uni_boundries
```

In [5]:

```
def get_rep_coefficiant(epsilon, subsample, curr_coefficiant, p, weight_subsample):
    """ Returns the represenative value (gamma_i* in our pseudo-code) of the subsample (d
elta_i in our pseudo-code)."""
    weighted_subsample = np.zeros(shape= subsample.shape)
    total_weight = 0
    for i in range(subsample.shape[0]):
        for j in range(subsample.shape[1]):
            curr_weight = get_weight_at_point(i,j,epsilon, subsample,
                                              curr_coefficiant, p, weight_subsample)
            weighted_subsample[i,j] = subsample[i][j] * curr_weight
            total_weight += curr_weight
    return float(weighted_subsample.sum()) / total_weight
```

In [6]:

```
def get_local_LP_error(pix_submatrix, coefficient, p, weights):
    """ Returns the LP error term for the current coefficient for the sub-image (delta_i
in our pseudo-code)."""
    curr_weights = pd.DataFrame(weights)
    curr_matrix = pd.DataFrame(pix_submatrix)
```

```
    return np.sum((np.power(np.abs(curr_matrix - coefficient), p) * curr_weights).sum())
```

In [7]:

```
def get_LP_error(pix, rep_coefficients, weight_matrix , p, list_of_xboundries, list_of_yb
oundries):
    """ Returns the LP error for the whole image for the current recunstruction.
    In each iteration we add the LP error term of a sub-domain and its current representa
tive value."""
    total_error = 0
    total_weight = weight_matrix.sum()
    for i in range(len(rep_coefficients)):
        for j, gamma in enumerate(rep_coefficients[i]):
            lower_x = list_of_xboundries[i]
            upper_x = list_of_xboundries[i+1]
            lower_y = list_of_yboundries[j]
            upper_y = list_of_yboundries[j+1]
            total_error += get_local_LP_error(pix[lower_x:upper_x,lower_y:upper_y],
                                              rep_coefficients[i,j], p,
                                              weight_matrix[lower_x:upper_x,lower_y:u
pper_y])
    return float(total_error) / total_weight
```

In [8]:

```
# step 1 in the algorithm: pray to god N is a power of two

def LP_rep(pix, sub_sampling_factor, epsilon, p, stopping_diff, weight_func):
    """ Implementation of the LP solver algorithm."""
    epsilon = float(epsilon)
    rep_coefficients = (np.ones(shape=(sub_sampling_factor, sub_sampling_factor)) * 120.
0).astype('float64')
    list_of_xboundries = get_uni_boundries_from_num_of_reps(sub_sampling_factor, pix.shap
e[0])
    list_of_yboundries = get_uni_boundries_from_num_of_reps(sub_sampling_factor, pix.shap
e[1])
    last_diff = stopping_diff
    weight_matrix = np.asarray([[float(weight_func(x,y)) for x in range(pix.shape[0])] f
or y in range(pix.shape[0])])
    last_error = get_LP_error(pix, rep_coefficients,
                              weight_matrix , p, list_of_xboundries, list_of_yboundries)
    while(last_diff >= stopping_diff):
        print("new iteration")
        for i in range(sub_sampling_factor):
            for j in range(sub_sampling_factor):
                lower_x = list_of_xboundries[i]
                upper_x = list_of_xboundries[i+1]
                lower_y = list_of_yboundries[j]
                upper_y = list_of_yboundries[j+1]
                rep_coefficients[i][j] = get_rep_coefficiant(epsilon=epsilon,
                                                             subsample=pix[lower_x:upper
_x,lower_y:upper_y],
                                                             curr_coefficiant=rep_coeffi
cients[i][j],
                                                             p=p,
                                                             weight_subsample=weight_mat
rix[lower_x:upper_x,lower_y:upper_y])
        curr_error = get_LP_error(pix, rep_coefficients, weight_matrix, p, list_of_xboun
dries, list_of_yboundries)
        last_diff = np.abs(last_error - curr_error)
        last_error = curr_error
    return rep_coefficients, last_error
```

Q3.

In [9]:

```
def uni_weight(x,y):
    """ The uniform weight function we give as input to the LP solver algorithm in Q4."""
    return 1
```

```
im = Image.open(r"./Truman_receives_menorah_square.jpg").convert('L')
pix = np.array(im)
```

In [10]:

```
def analytical_MAD(rep, pix_shape):
    """ Returns the MAD term for the given image with its representatives in a matrix.
        Size of the representatives matrix is DxD from the algorithm below."""
    MAD = np.kron(rep, np.ones(shape = (pix_shape[0] // rep.shape[0],
                                        pix_shape[1] // rep.shape[1])))
    return np.mean(np.abs(pix - MAD))
```

In [11]:

```
""" Here we copied part of the algorithm from question 2, which does subsampling and reco
nstruction
    using the MAD sense."""
analytical_MAD_reps = []  # will be list of 2D arrays of represantators of every grid sam
ple region
analytical_MAD_errors = []  # MAD error for every b
for b in range(1,9):
    D_value = np.power(2,b)
    MAD_rep = np.ndarray(shape=(D_value,D_value), dtype=float)
    list_of_xboundries = get_uni_boundries_from_num_of_reps(D_value, pix.shape[0])
    list_of_yboundries = get_uni_boundries_from_num_of_reps(D_value, pix.shape[1])
    for i in range(len(list_of_xboundries) - 1):
        for j in range(len(list_of_yboundries) - 1):
            lower_x = list_of_xboundries[i]
            upper_x = list_of_xboundries[i+1]
            lower_y = list_of_yboundries[j]
            upper_y = list_of_yboundries[j+1]

            # in the MAD sense: our represantator is the sample's median
            sample_median = np.median(pix[lower_x:upper_x,lower_y:upper_y])
            MAD_rep[i][j] = sample_median
    analytical_MAD_errors.append(analytical_MAD(MAD_rep, pix.shape))
    analytical_MAD_reps.append(MAD_rep)
```

Q4.

In [12]:

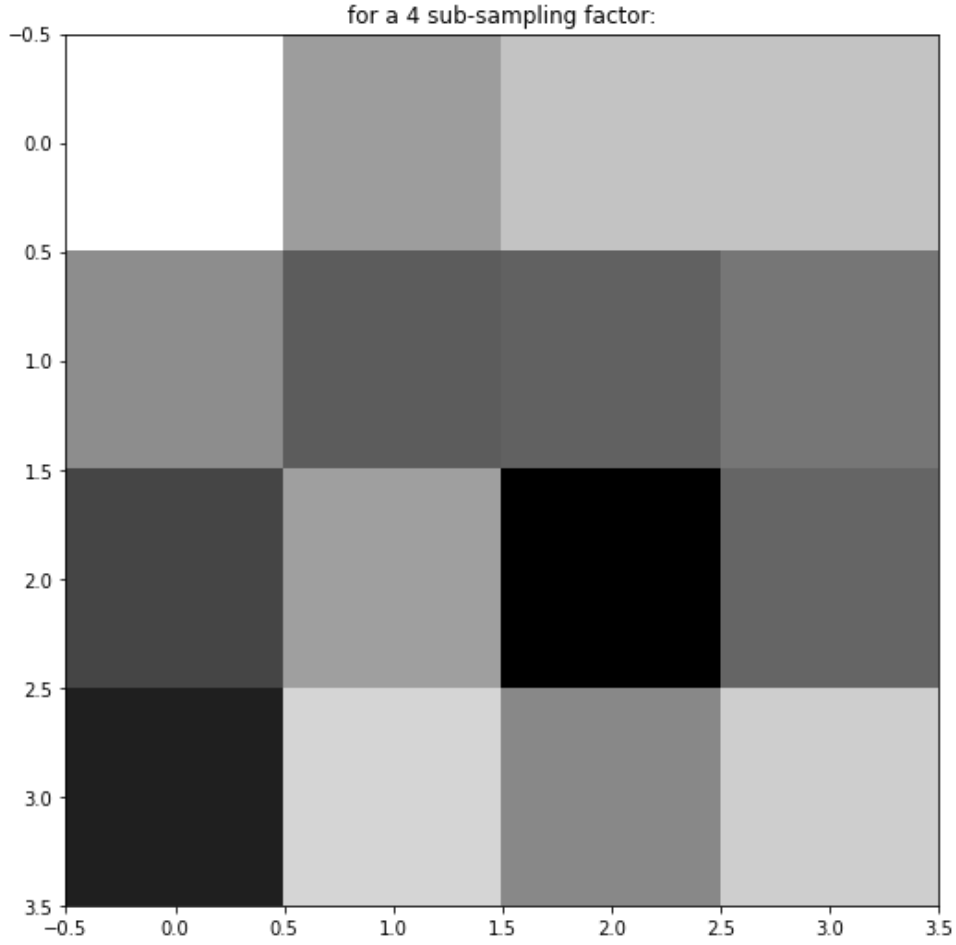
```
""" Here we computed the LP representations of the image given p = 1 for varying epsilon
and N values."""
irls_MAD_reps = []
irls_estimated_errors = []
irls_analytical_errors = []
for b in [2,4,6]:
    for epsilon in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
        D_value = np.power(2,b)
        irls_rep, irls_estimated_error = LP_rep(pix=pix, sub_sampling_factor=D_value, ep
silon=epsilon, p=1,
                                                stopping_diff=0.5, weight_func=uni_weigh
t)
        irls_MAD_reps.append(irls_rep)
        irls_estimated_errors.append(irls_estimated_error)
        irls_analytical_errors.append(analytical_MAD(irls_rep, pix.shape))
```

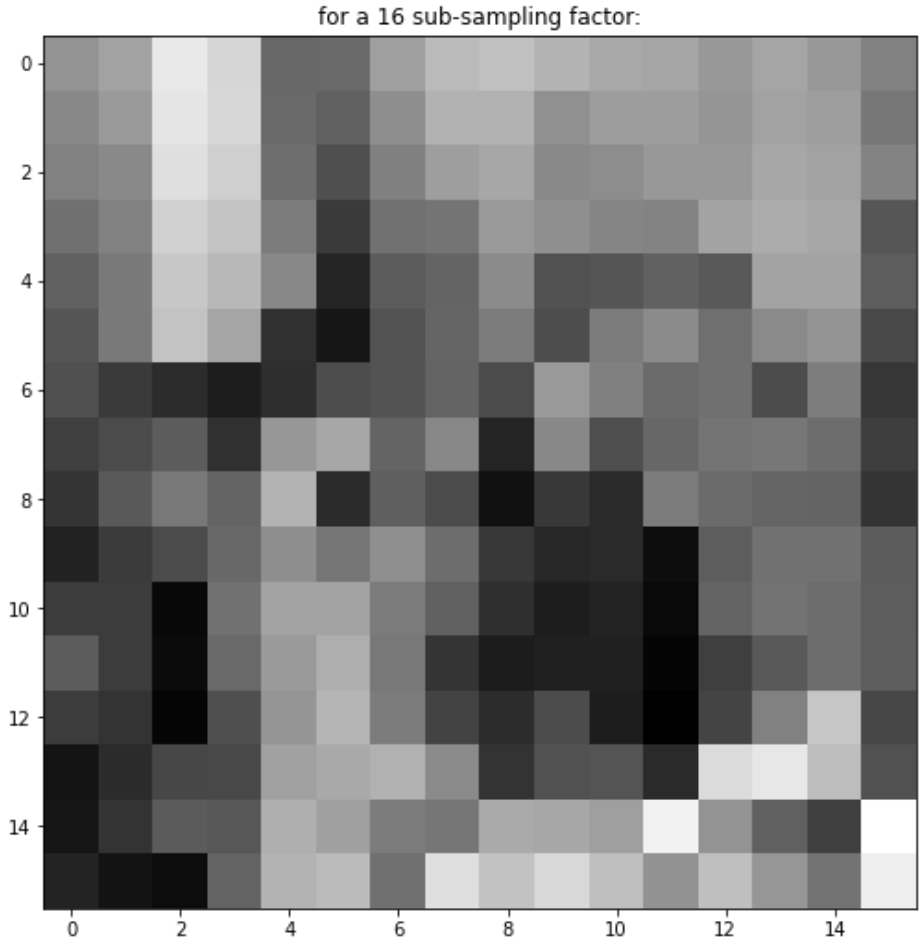
4. Comparing the errors:

N	Exact algorithm error	ϵ	LP solver for $p = 1$
$N = 4$	40.5274	1	44.8908
		0.1	48.7407
		0.01	48.8927
		0.001	48.9024
		0.0001	48.9097
		0.00001	48.9098
$N = 16$	19.9914	1	22.2488
		0.1	22.6895
		0.01	23.0258
		0.001	23.1172
		0.0001	23.1221
		0.00001	23.1225
$N = 64$	12.0437	1	13.2726
		0.1	13.7087
		0.01	13.8169
		0.001	13.8276
		0.0001	13.8286
		0.00001	13.8287

We can see that generally the exact algorithm was better at minimizing the MAD error than the iterative LP solver. Also, we notice that for higher epsilon values, the MAD of the LP solver was lower. That's because low epsilon values give a lot of weight to the color values that are close to the current representative in each subsample, so for low epsilon we have more error.

```
plotimages(irls_MAD_reps[::6])  # plotting the sampled images from using the LP solver.
```





In [15]:

```
""" Obtaining the L4 representations through our iterative algorithm"""
```

```
print("irls_L1_and_a_half_errors: \n", irls_L1_and_a_half_errors)
```

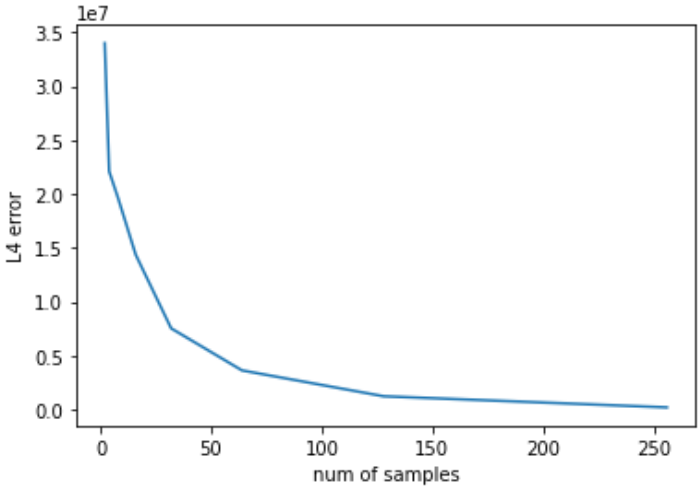
```
irls_L4_errors:
 [33973410.72299675, 22087766.752701778, 19635318.920403257, 14360223.383708533, 7542443.
080649059, 3642802.63412069, 1245457.5152016566, 212811.86304469086]
irls_L1_and_a_half_errors:
 [395.8834481501708, 314.45098825264483, 260.18814966685864, 211.119894192618, 138.389333
43704272, 88.77382461939506, 51.54016788187601, 23.98814683263534]
```

In [19]:

```
sns.lineplot(y = irls_L4_errors, x = [np.power(2,b) for b in range(1,9)]).set(xlabel="nu
m of samples",
                                                                              ylabel="L
4 error")
```

Out[19]:

```
[Text(0, 0.5, 'L4 error'), Text(0.5, 0, 'num of samples')]
```

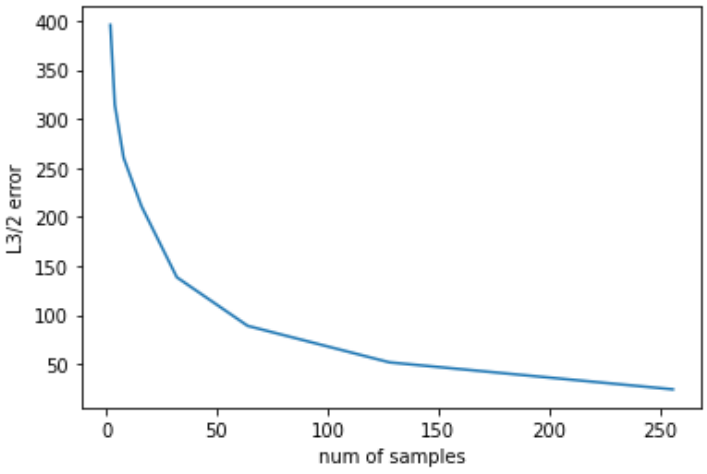


In [20]:

```
sns.lineplot(y = irls_L1_and_a_half_errors, x = [np.power(2,b) for b in range(1,9)]).set
(xlabel="num of samples",

ylabel="L3/2 error")
```

Out[20]:

```
[Text(0, 0.5, 'L3/2 error'), Text(0.5, 0, 'num of samples')]
```



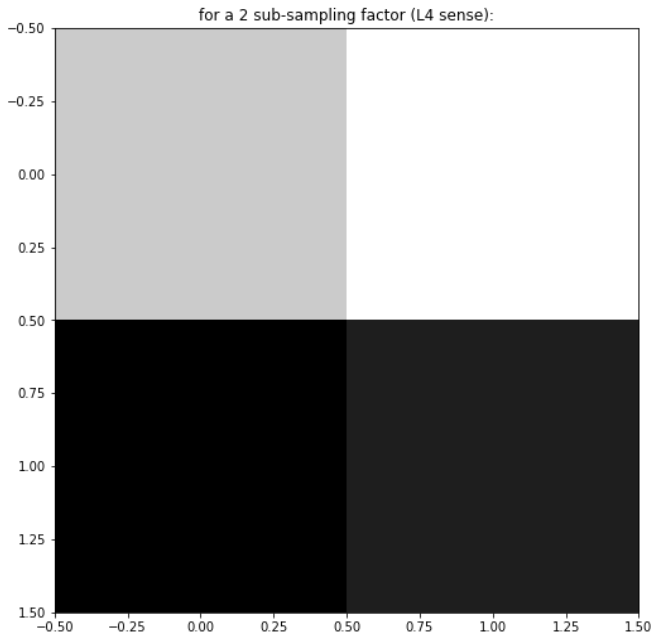
In [21]:

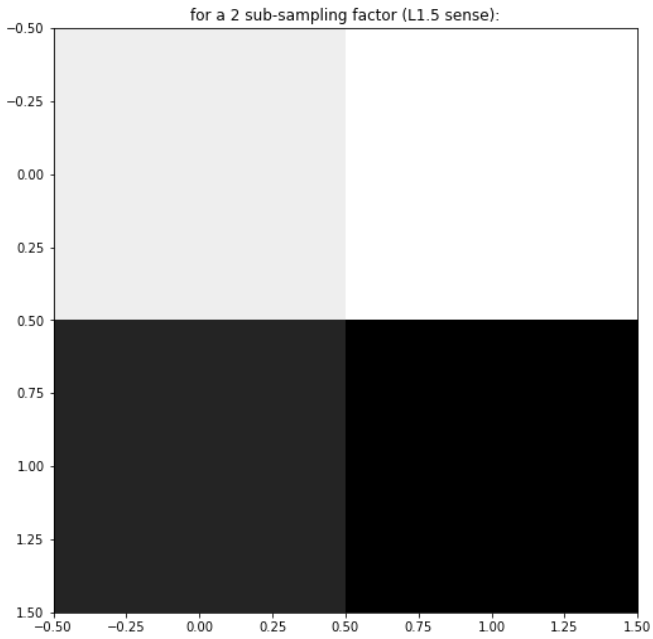
```
def plot_image_lists(matrix_list1, matrix_list2):
    """ Plots the recieved matrices side by side. """
    plt.figure(figsize=(20,80))
    for b in range(1,8):
        plt.subplot(8,2,2*b-1)
        plt.title(f"for a {np.power(2,b)} sub-sampling factor (L4 sense):")
```

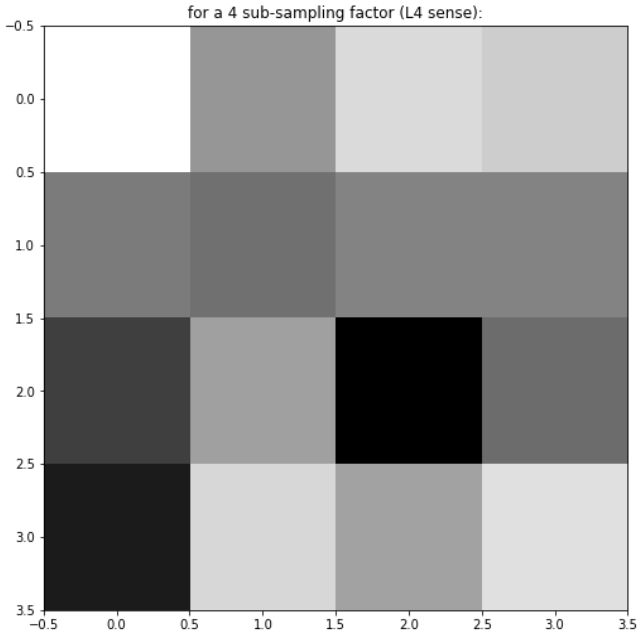
```
        plt.imshow(matrix_list1[b-1], cmap='gray')
        plt.subplot(8,2,2*b)
        plt.title(f"for a {np.power(2,b)} sub-sampling factor (L1.5 sense):")
        plt.imshow(matrix_list2[b-1], cmap='gray')
```

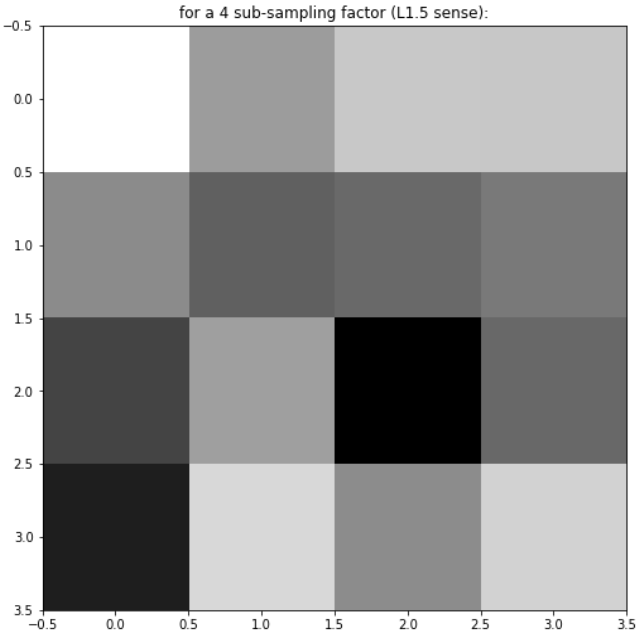
In [22]:

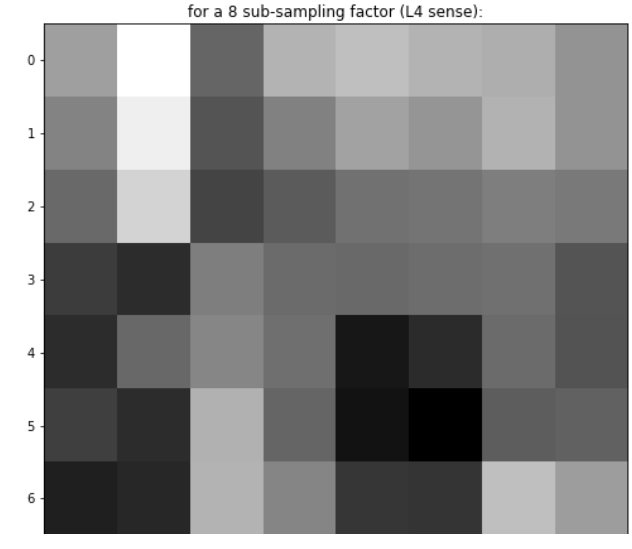
```
plot_image_lists(irls_L4_reps, irls_L1_and_a_half_reps)
```

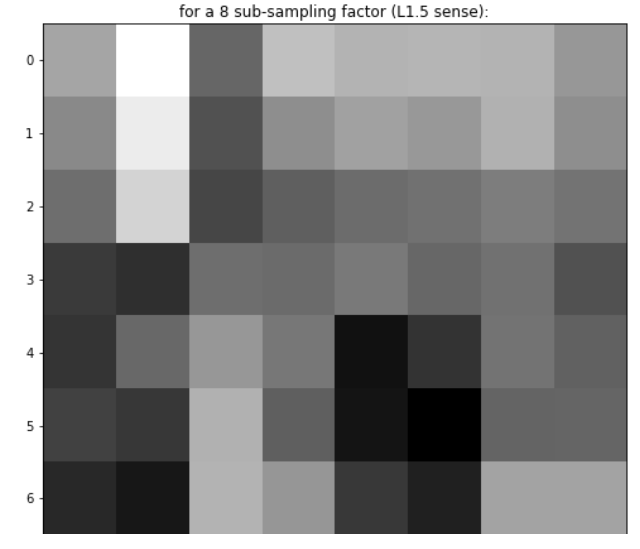


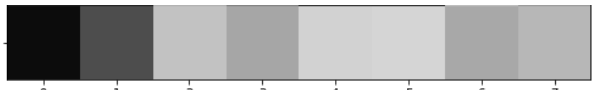


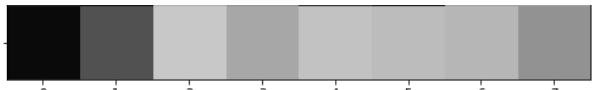




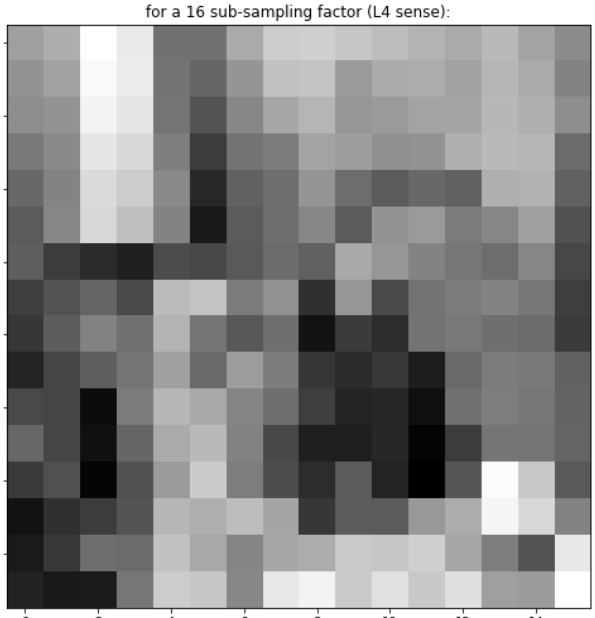




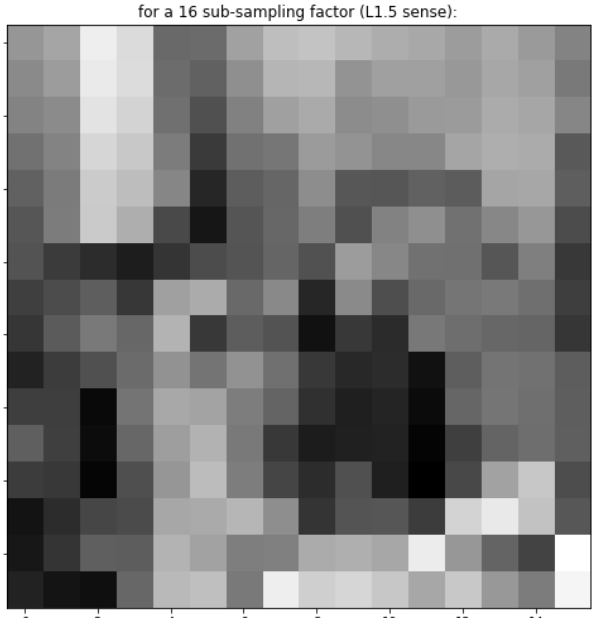




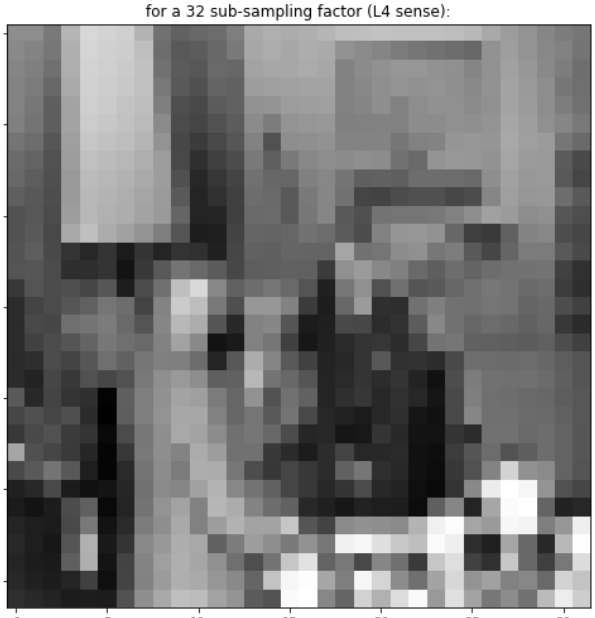
for a 16 sub-sampling factor (L4 sense):



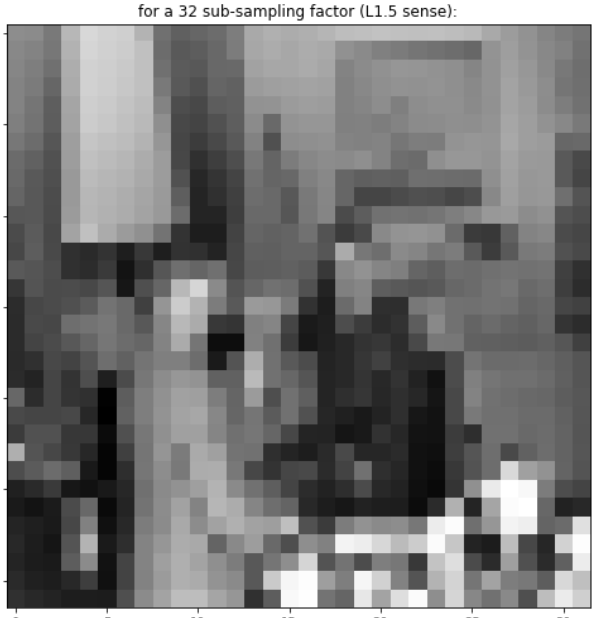
for a 16 sub-sampling factor (L1.5 sense):



for a 32 sub-sampling factor (L4 sense):



for a 32 sub-sampling factor (L1.5 sense):



for a 64 sub-sampling factor (L4 sense):

for a 64 sub-sampling factor (L1.5 sense):


for a 128 sub-sampling factor (L4 sense):
0
20
40
60
80
100
120
0
20
40
60
80
100
120


for a 128 sub-sampling factor (L1.5 sense):
0
20
40
60
80
100
120
0
20
40
60
80
100
120