HW1

October 19, 2020

Name: Chaitanya Sharadchandra Patil
PID: A53311586

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind.
By including this in my report, I agree to abide by the Academic Integrity Policy mentioned above.

0.0.1 Question 1

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

%matplotlib inline
```

[2]:
```
A = [[3,9,5,1], [4, 25, 4, 3], [63, 13, 23, 9], [6, 32, 77, 0], [12, 8, 6, 1]]
```

[3]:
```
A = np.asarray(A)
```

[4]:
```
A
```

[4]:
```
array([[ 3,  9,  5,  1],
       [ 4, 25,  4,  3],
       [63, 13, 23,  9],
       [ 6, 32, 77,  0],
       [12,  8,  6,  1]])
```

[5]:
```
B = [[0,1,0,1],[0,1,1,0],[0,0,0,1],[1,1,0,1],[0,1,0,0]]
```

[6]:
```
B = np.asarray(B)
```

[7]:
```
B
```

[7]:
```
array([[0, 1, 0, 1],
       [0, 1, 1, 0],
       [0, 0, 0, 1],
       [1, 1, 0, 1],
       [0, 1, 0, 0]])
```

[8]:
```
# (i)

C = A*B
```

[9]:
```
C
```

[9]:
```
array([[ 0,  9,  0,  1],
       [ 0, 25,  4,  0],
       [ 0,  0,  0,  9],
       [ 6, 32,  0,  0],
       [ 0,  8,  0,  0]])
```

[10]:
```
# (ii)

inner_2rC_3rC = np.dot(C[1,:], C[2,:])
```

[11]:
```
inner_2rC_3rC
```

[11]:
```
0
```

[12]:
```
# (iii)

max_element = C.max()
print('The max element is ', max_element)
temp = np.where(C == C.max())
print('The positions of the max element are ', np.array(temp))
```

```
The max element is  32
The positions of the max element are  [[3]
 [1]]
```

[13]:
```
min_element = C.min()
print('The max element is ', min_element)
temp = np.where(C == C.min())
print('The positions of the min element are ', np.array(temp))
```

```
The max element is  0
The positions of the min element are  [[0 0 1 1 2 2 2 3 3 4 4 4]
 [0 2 0 3 0 1 2 2 3 0 2 3]]
```

np.where returns a list of arrays that represent the position of condition satisfying elements in the parameter passed to it. So the way to interpret these positions is: the min elements are present at

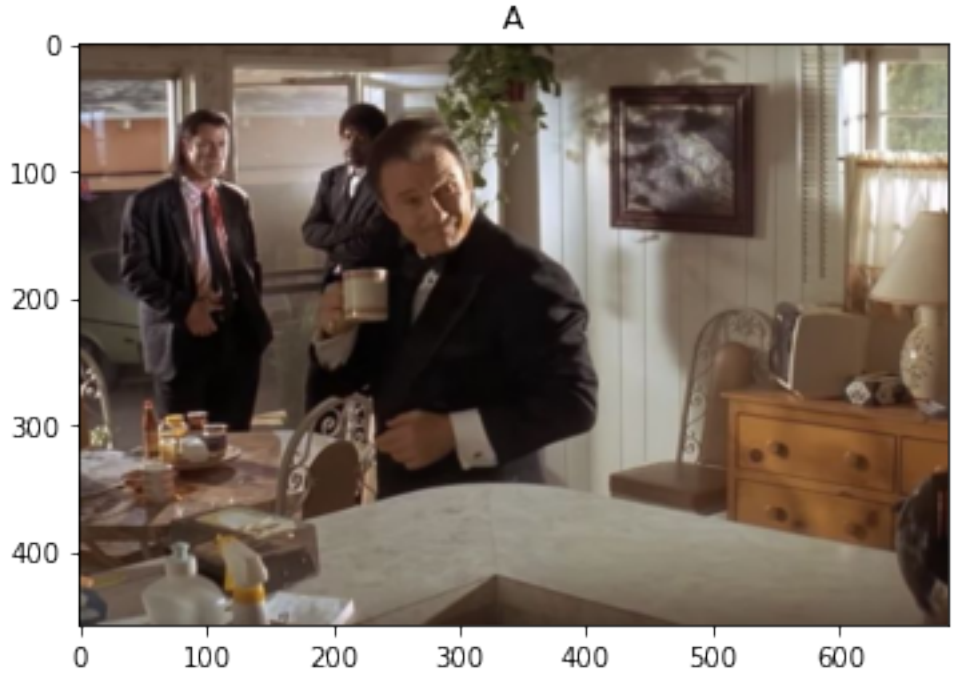
(0,0), (0, 2), (1, 0) and so on, taking the row position from the first array returned and the column position from the second array returned.

0.0.2 Question 2

[14]:
```
#(i)
A = image.imread('wolf.jpg')
```

[15]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(A);
ax.set_title('A')
```

[15]:
```
Text(0.5, 1.0, 'A')
```



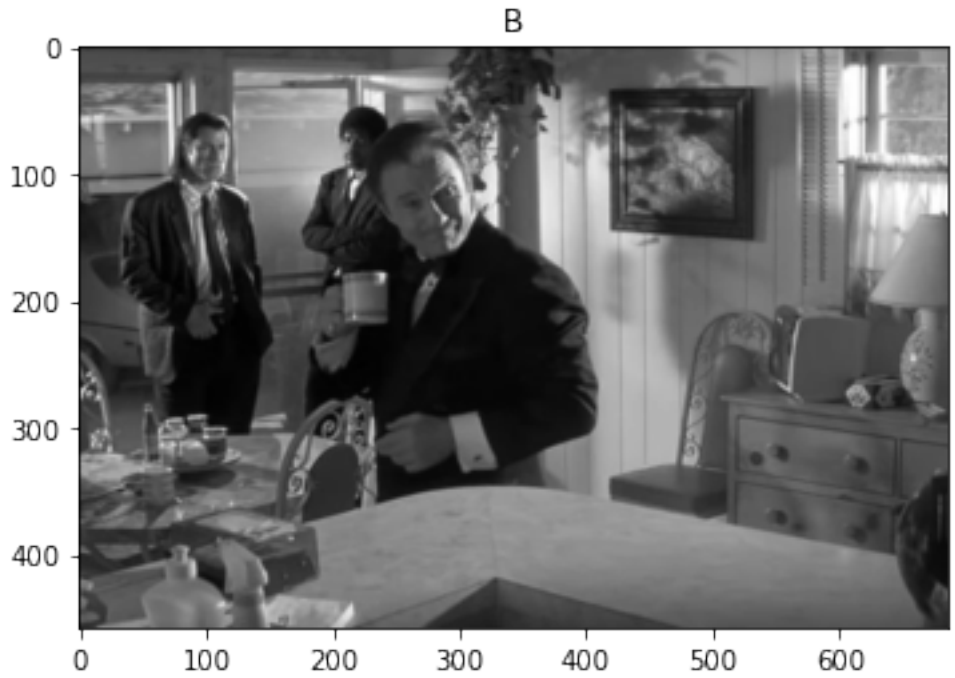
[16]:
```
# (ii)

B = np.mean(A, axis = 2) # this is just an approximation.

# the values of B cannot possibly be more than 255 or less than 0
# if they were, this would imply that one of the red, green or blue values did
# not belong to this range, which we know to be false.
```

[17]:
```
# nevertheless we will verify it
print('The max element is ', B.max(), 'and the min element is', B.min())
```

```
The max element is  255.0 and the min element is 0.3333333333333333
```

[18]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(B, cmap = 'gray');
ax.set_title('B')
```

[18]:
```
Text(0.5, 1.0, 'B')
```

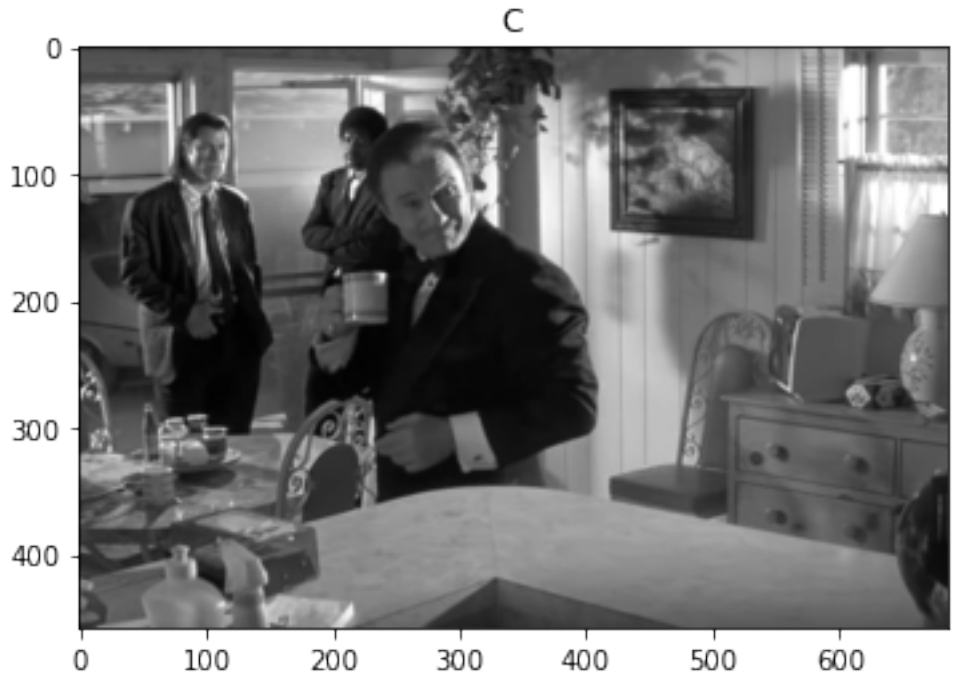


[19]:
```
# (iii)

C_dash = B + 15
```

[20]:
```
C = np.where(C_dash>255,255, C_dash)
```

[21]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(C, cmap = 'gray');
ax.set_title('C')
```

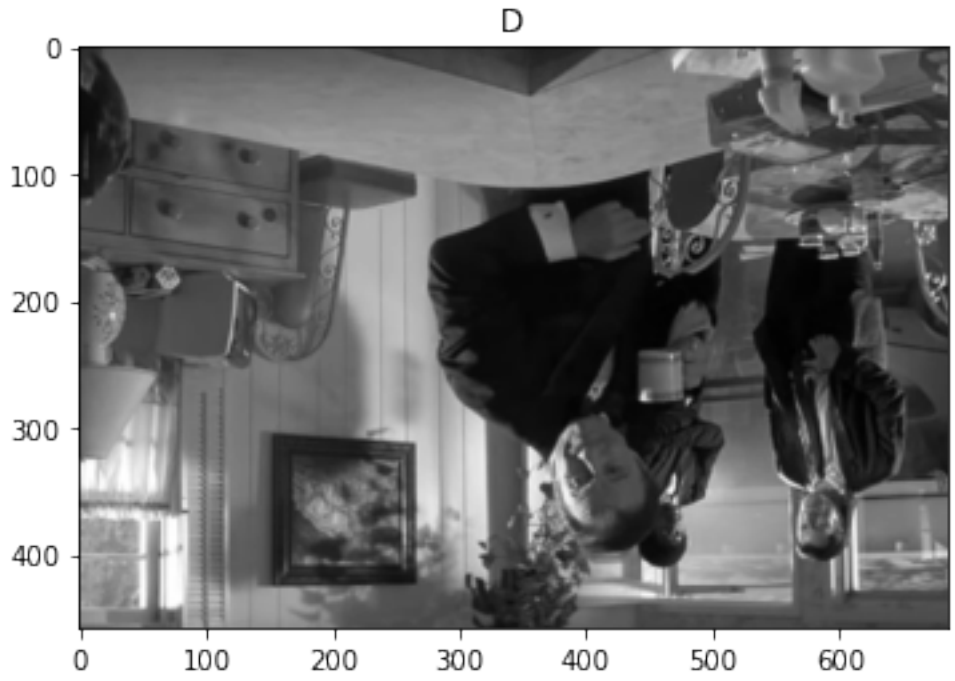
[21]:
```
Text(0.5, 1.0, 'C')
```



[22]:
```
D_dash = B[::-1]
D = D_dash.T[::-1].T
```

[23]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(D, cmap = 'gray');
ax.set_title('D')
```

[23]: Text(0.5, 1.0, 'D')

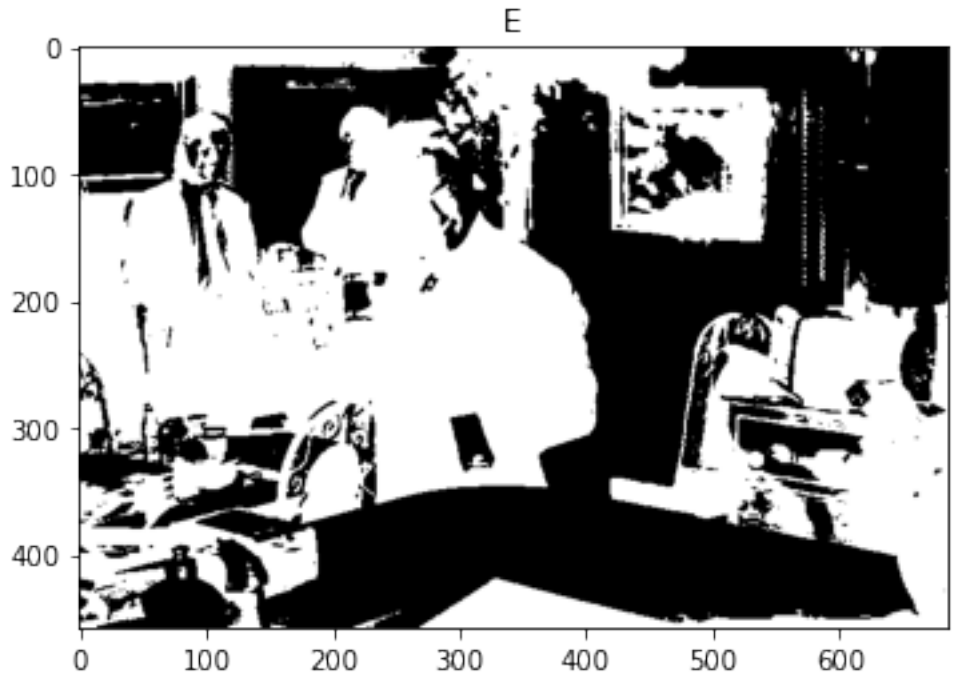


[24]:
```
# (v)

med = np.median(B)
print(med)
```

```
91.33333333333333
```

[25]:
```
E = np.where(B>med,0,1)
```

[26]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(E, cmap = 'gray')
ax.set_title('E');
```



0.0.3 Problem 3 : Histograms

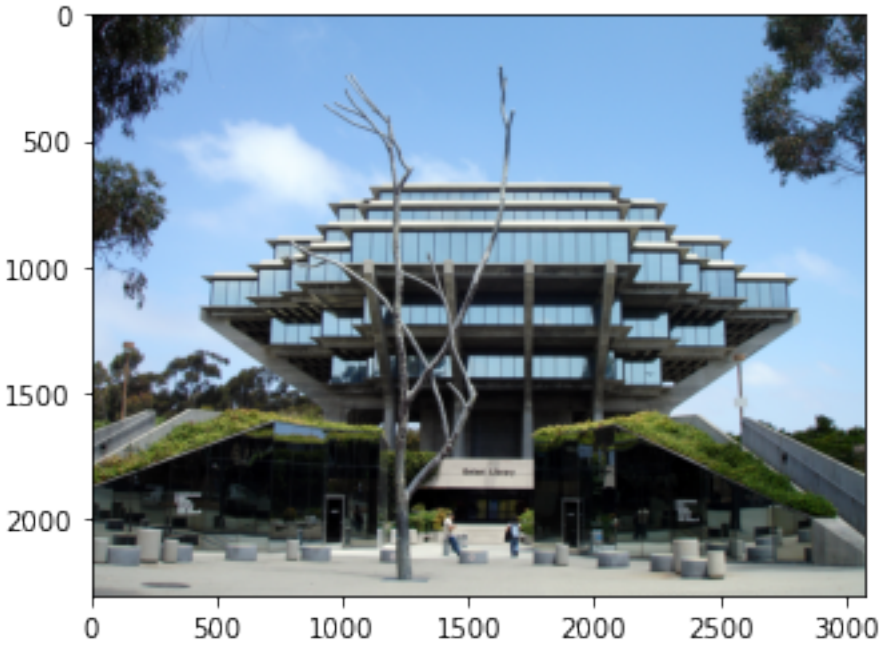
[27]:
```
def compute_norm_rgb_histogram(A):
    """computes the RGB color histogram"""
    R_hist = np.zeros((256))
    G_hist = np.zeros((256))
    B_hist = np.zeros((256))
    for i in A:
        for j in i:
            R_hist[j[0]] = R_hist[j[0]] + 1
            G_hist[j[1]] = G_hist[j[1]] + 1
            B_hist[j[2]] = B_hist[j[2]] + 1
    R_hist_32 = np.zeros((32))
    G_hist_32 = np.zeros((32))
    B_hist_32 = np.zeros((32))
    for i in range(256):
        R_hist_32[i//8] = R_hist_32[i//8] + R_hist[i]
        G_hist_32[i//8] = G_hist_32[i//8] + G_hist[i]
        B_hist_32[i//8] = B_hist_32[i//8] + B_hist[i]
    R_hist_32_norm = R_hist_32/np.sum(R_hist_32)
    G_hist_32_norm = G_hist_32/np.sum(G_hist_32)
    B_hist_32_norm = B_hist_32/np.sum(B_hist_32)
    RGB_FullHistogram = np.hstack((R_hist_32_norm, G_hist_32_norm,␣
↪B_hist_32_norm))
```

```
    return(RGB_FullHistogram)
```

[28]:
```
A = image.imread('geisel.jpg')
```

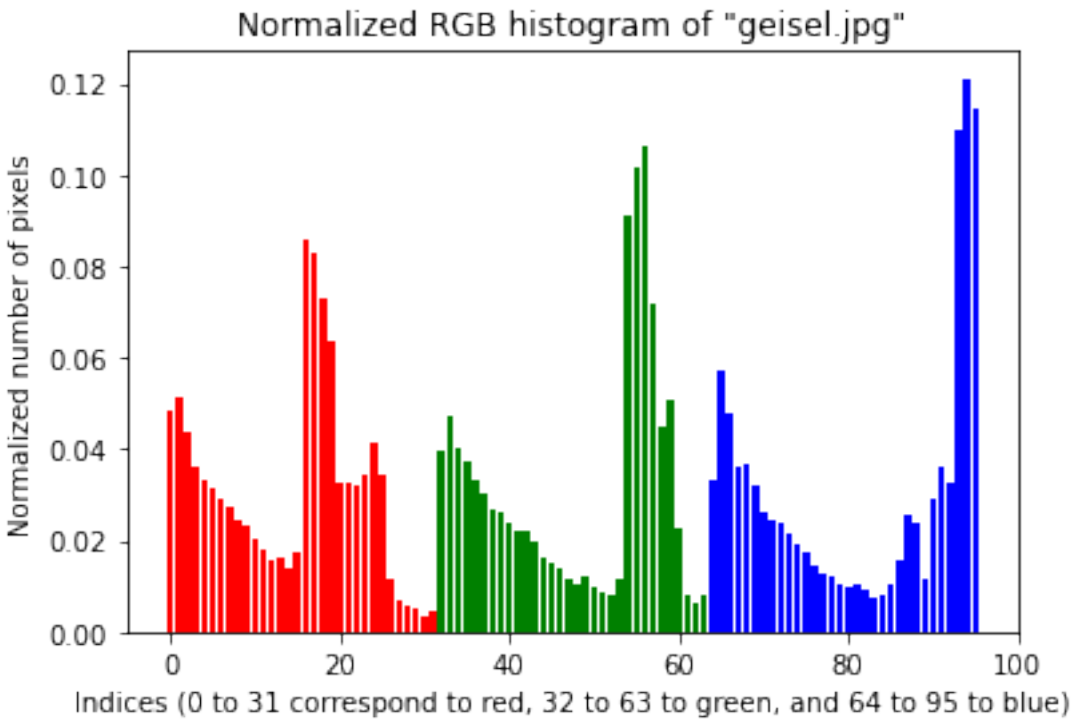
[29]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(A)
```

[29]:
```
<matplotlib.image.AxesImage at 0x23a1ed8eeb0>
```



[30]:
```
bar_plot = compute_norm_rgb_histogram(A)
```

[31]:
```
hist_fig = plt.figure()
hist_ax = hist_fig.add_subplot()
hist_ax.bar(np.arange(32), bar_plot[32*0:32*1], color = 'r')
hist_ax.bar(np.arange(32)+32, bar_plot[32*1:32*2], color = 'g')
hist_ax.bar(np.arange(32)+32*2, bar_plot[32*2:32*3], color = 'b')
hist_ax.set_title('Normalized RGB histogram of "geisel.jpg"')
hist_ax.set_xlabel('Indices (0 to 31 correspond to red, 32 to 63 to green, and␣
 ↪64 to 95 to blue)');
hist_ax.set_ylabel('Normalized number of pixels');
```



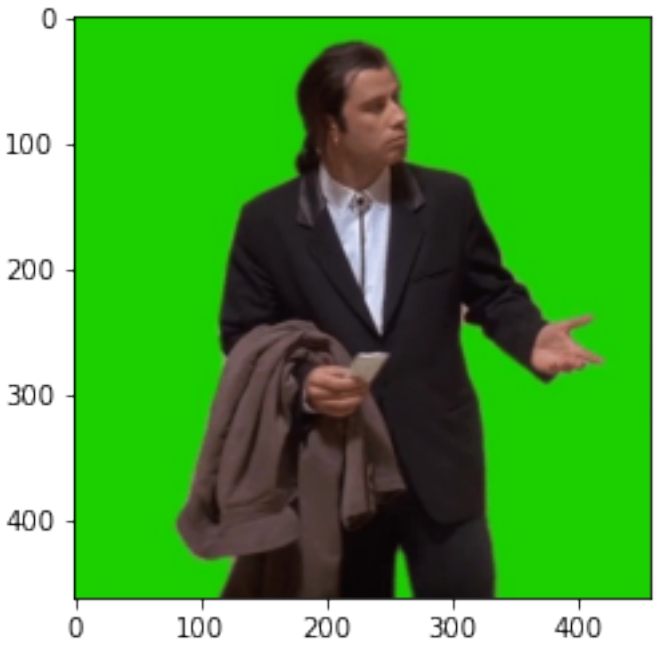
0.0.4 Problem 4 : Chroma Keying

[32]:
```
travolta_mpl = image.imread('travolta.jpg')
```

[33]:
```
travolta_mpl.shape
```

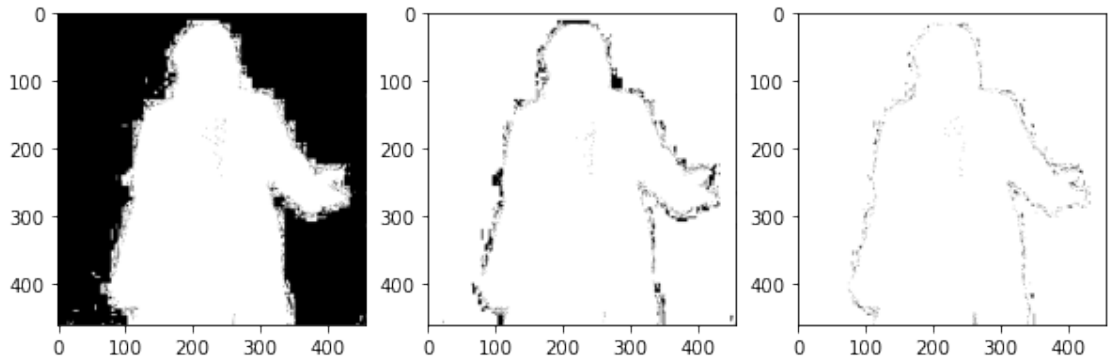
[33]:
```
(462, 458, 3)
```

[34]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(travolta_mpl);
```



[35]:
```
# (i) : a binary image showing the foreground mask, that is, all foreground␣
 ↪pixels set
# to 1 and all background pixels set to 0

fg_1 = np.where(travolta_mpl[:,:,1]==208,0,1)
fg_2 = np.where(travolta_mpl[:,:,1]==207,0,1)
fg_3 = np.where(travolta_mpl[:,:,1]==206,0,1)
```

[36]:
```
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot(131)
ax.imshow(fg_1, cmap = 'gray');
ax2 = fig.add_subplot(132)
ax2.imshow(fg_2, cmap = 'gray');
ax3 = fig.add_subplot(133)
ax3.imshow(fg_3, cmap = 'gray');
```

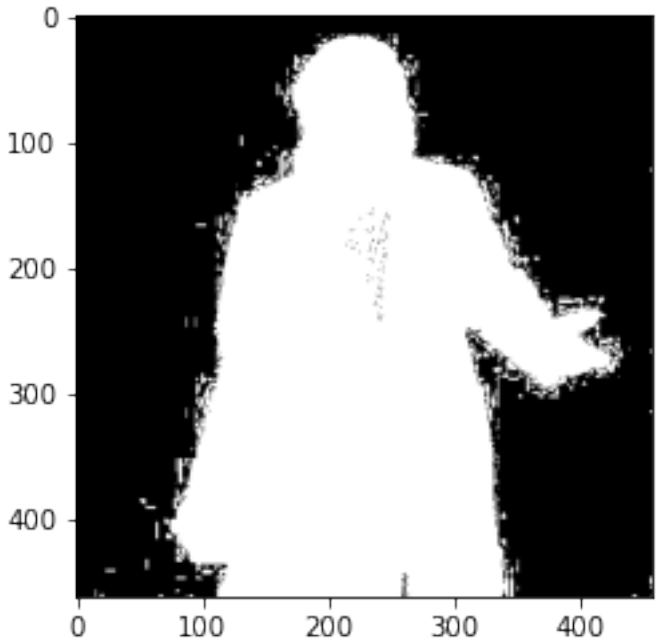


If we and these two images, we will get a result image where both images are white.

[37]:
```
fgtemp = np.logical_and(fg_1, fg_2)
fg = np.logical_and(fgtemp, fg_3)
```

[38]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(fg, cmap = 'gray')
```

[38]:
```
<matplotlib.image.AxesImage at 0x23a2038be20>
```



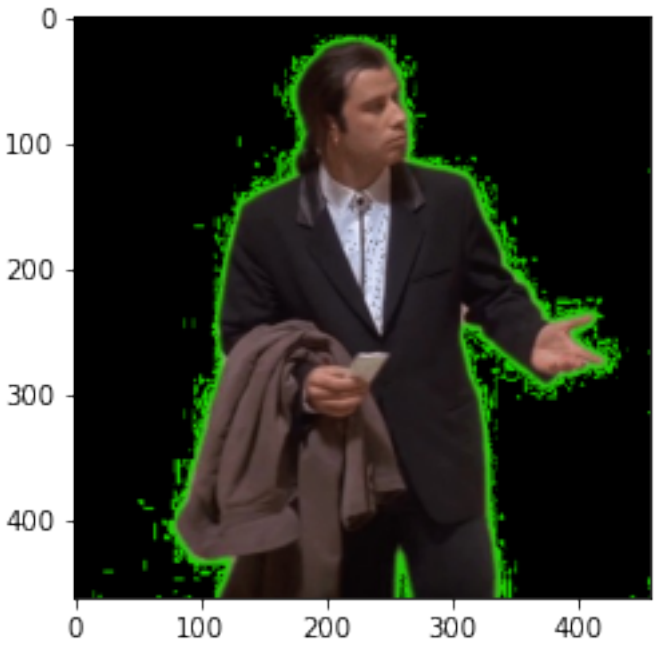
[39]:
```
# (ii) An image with the background pixels set to 0 and the foreground pixels␣
 ↪set
# to their original values.
```

[40]:
```
# above does not work because fg has shape (462,458). We need to make this␣
 ↪(462,458,3).


fg_3d = np.dstack((fg,fg,fg))
```

[41]:
```
fg_orig2 = travolta_mpl * fg_3d
```

[42]:
```
fig = plt.figure()
ax = fig.add_subplot()
ax.imshow(fg_orig2)
```

[42]:
```
<matplotlib.image.AxesImage at 0x23a20558340>
```



[43]:
```
# References: https://pythontic.com/image-processing/pillow/blend
# https://note.nkmk.me/en/python-pillow-paste/

# (iii) :  An image with the foreground overlayed on a background of your choice.
```

[44]:
```
adam = image.imread('adam_resized_online.jpg')
```

[45]:
```
adam.shape, fg_orig2.shape
```

[45]:
```
((512, 1128, 3), (462, 458, 3))
```

make john travolta sized hole and then add john travolta to it

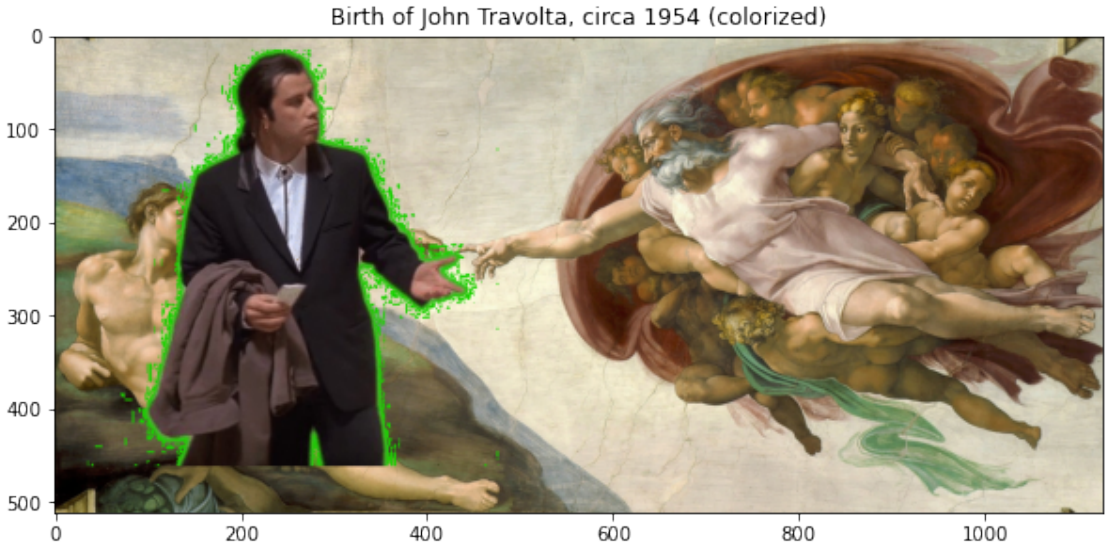
a. make john travolta sized hole
b. multiply a part of adam with john travolta hole

[46]:
```
john_travolta_hole = np.logical_not(fg)
john_travolta_hole_3d = np.
 ↪dstack((john_travolta_hole,john_travolta_hole,john_travolta_hole))
```

[47]:
```
row_offset = 0
col_offset = 20
```

[48]:
```
a = np.zeros(adam.shape)
a = a + adam
a[row_offset:row_offset+fg_orig2.shape[0],col_offset:col_offset+fg_orig2.
 ↪shape[1],:] = adam[row_offset:row_offset+fg_orig2.shape[0],col_offset:
 ↪col_offset+fg_orig2.shape[1],:] * john_travolta_hole_3d + fg_orig2
```

[49]:
```
fig = plt.figure(figsize = (10,5))
ax = fig.add_subplot()
ax.imshow(a.astype(np.uint8))
ax.set_title('Birth of John Travolta, circa 1954 (colorized)');
```



0.0.5 Problem 5: Upsampling and Downsampling

0.0.6 (i) List (and describe in a short paragraph) 3 interpolation methods.

(Source for the entire answer: Digital Image Processing, Gonzalez, Woods)

Interpolation is the process of guessing the unknown values of a signal based on values of a signal based on its known values. In signal processing, this is often done after upsampling a signal, and interpolation filters are used to determine how to calculate the unknown values of the signal.

The three interpolation methods we are going to look at are:

A. Nearest neighbour interpolation B. Bilinear interpolation C. Bicubic interpolation

A. Nearest Neighbour interpolation: In this method of interpolation, the value that is equal to the value of the nearest neighbouring pixel is assigned to the unknown pixel. The nearness in the case of an image means nearness in quantized space. Nearest neighbour interpolation is the simplest interpolation method in signal processing.

B. Bilinear Interpolation: In this method, the value of an unknown pixel at x, y is calculated based on its four neighbours. The equation used is:

$$v(x, y) = ax + by + cxy + d$$

(Source: Digital Image Processing, Gonzalez, Woods)

Here, the values of a, b, c and d are determined using four nearest neighbours of the pixel under consideration (Source: Digital Image Processing, Gonzalez, Woods)

C. Bicubic Interpolation: In bicubic interpolation, the values of a cell are determined using its sixteen nearest neighbours. The equation to calculate intensities is:

$$v(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j$$

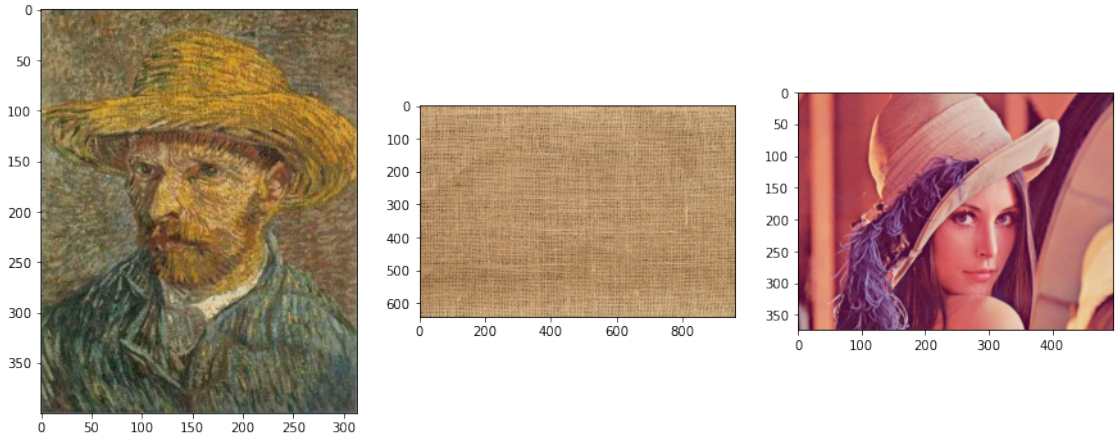
(Source: Digital Image Processing, Gonzalez, Woods)

The way to obtain the sixteen coefficients in this equation is: 1. There will be one such equation for each of the sixteen neighbours for an image 2. Each equation has sixteen unknowns 3. When you solve sixteen equations with sixteen unknowns, you can obtain the values of the unknowns. 4. You use the values so obtained to get v(x, y)
(Source: Digital Image Processing, Gonzalez, Woods)

0.0.7 The three images selected are displayed below:

[50]:
```
# select 3 color images
vincent = image.imread('vincent.jpg')
texture = image.imread('texture.jpg')
lenna = image.imread('lenna.jpg')

# displaying images in a subplot
fig = plt.figure(figsize=(15,10))
```

```
ax_0 = fig.add_subplot(131)
ax_0.imshow(vincent)
ax_1 = fig.add_subplot(132)
ax_1.imshow(texture)
ax_2 = fig.add_subplot(133)
ax_2.imshow(lenna);
```



0.0.8 (ii) Select 3 color images and downsample using the different methods with the 3 ratios below. What differences do you observe? Which interpolation method do you think works best?

[51]:
```
# setup
images = [vincent, texture, lenna]
methods = [cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC]
dr = [0.3, 0.5, 0.7]
methods_desc = ['Nearest Nb.', 'Bilinear', 'Bicubic']
```

[52]:
```
# first, we choose vincent
curr_im = vincent

fig, axs = plt.subplots(3,4, figsize = (20,20))

for ind, ratio in enumerate(dr):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(vincent, dsize=(round(vincent.shape[1]*ratio),␣
 ↪round(vincent.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind
```

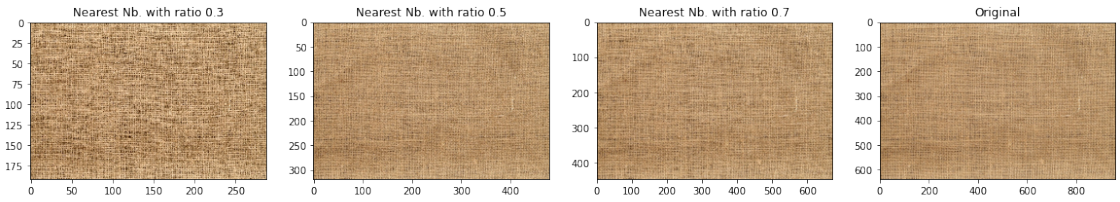
```
        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```

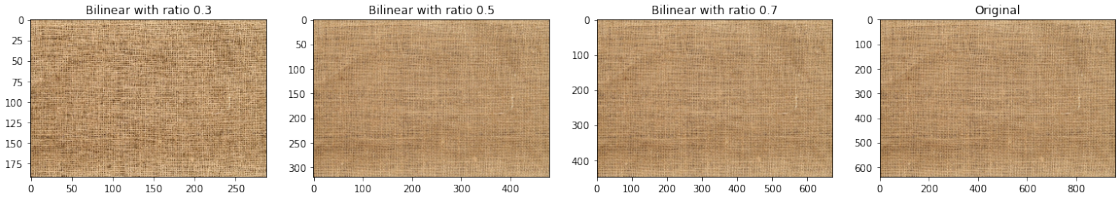
Based on visual inspection, I think that for this image, the Bicubic interpolation with ratio 0.7 looks closest to the original. Keeping the downsampling method the same, as the ratio downsampling ratio increases, the image becomes closer to the original. Keeping downsampling ratio the same, nearest neighbour interpolation gives us the most noisy image according to me, while bilinear interpolation is slightly better. Bicubic interpolation seems to be performing the best here.

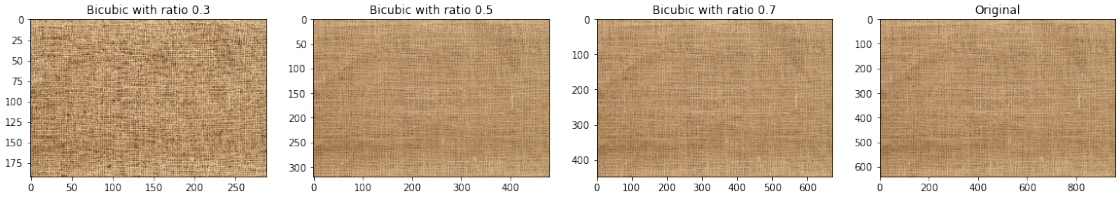
We also observe that in the original image, the face of the person in the image has some dark spots on the cheek to the right of the viewer. Comparing this original to the highest ratio of 0.7 in the downsampled images, we see that the bilinear image smooths out these dark spots in the original, while the bicubic retains them. Hence for this image, bicubic with 0.7 ratio of interpolation seems

to have the most fidelity to the original image.

[53]:
```
# second, we choose texture
curr_im = texture

fig, axs = plt.subplots(3,4, figsize = (20,20))

for ind, ratio in enumerate(dr):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(curr_im, dsize=(round(curr_im.shape[1]*ratio),␣
 ↪round(curr_im.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind

        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```

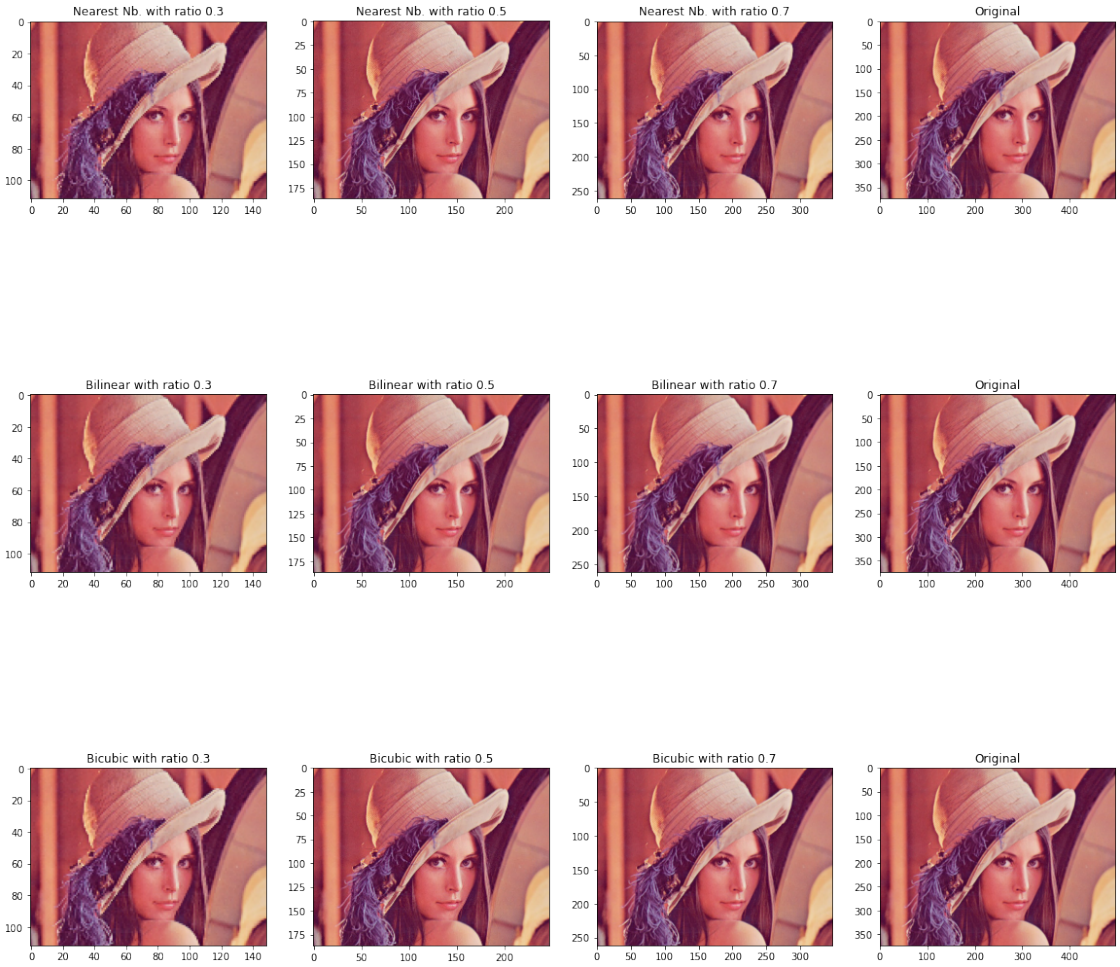






Although this image has fewer features than the rest, I am using it here to understand the effect different downsampling ratios and methods have on the texture in an image. We find that Nearest neighbour approach even with 0.7 ratio changes the texture of the image, probably due to high frequency noise being introduced into the image. Between bilinear and bicubic interpolation for this particular image, it is hard to tell which one is better for the 0.7 interpolation ratio.

[54]:
```
# lastly, we choose our third image, lenna
curr_im = lenna

fig, axs = plt.subplots(3,4, figsize = (20,20))

for ind, ratio in enumerate(dr):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(curr_im, dsize=(round(curr_im.shape[1]*ratio),␣
↪round(curr_im.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind

        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```



Here too, we observe that lower ratios leads to images with less resemblence to the original. It seems that in the nearest neibhbor interpolation image with 0.7 ratio, spots are introduced at some locations and existing spots are made larger. For example, two whitish spots appear in the background to the right of the viewer. Between the x coordinates 250 to 300, and around y coordinate 150. Similarly, lines in the bilinear interpolated image seem less pronounced (more blurred) than those in bicubic and original images. For example, a shiny hair strand in the 0.7 ratio image between y coordinates 100 to 150 and x coordinates 50 to 100 is less shiny in the bilinear image than the bicubic image.

0.0.9 (iii) Repeat the previous step, but this time use upsampling with the ratios below. What differences do you observe? What interpolation method works best?

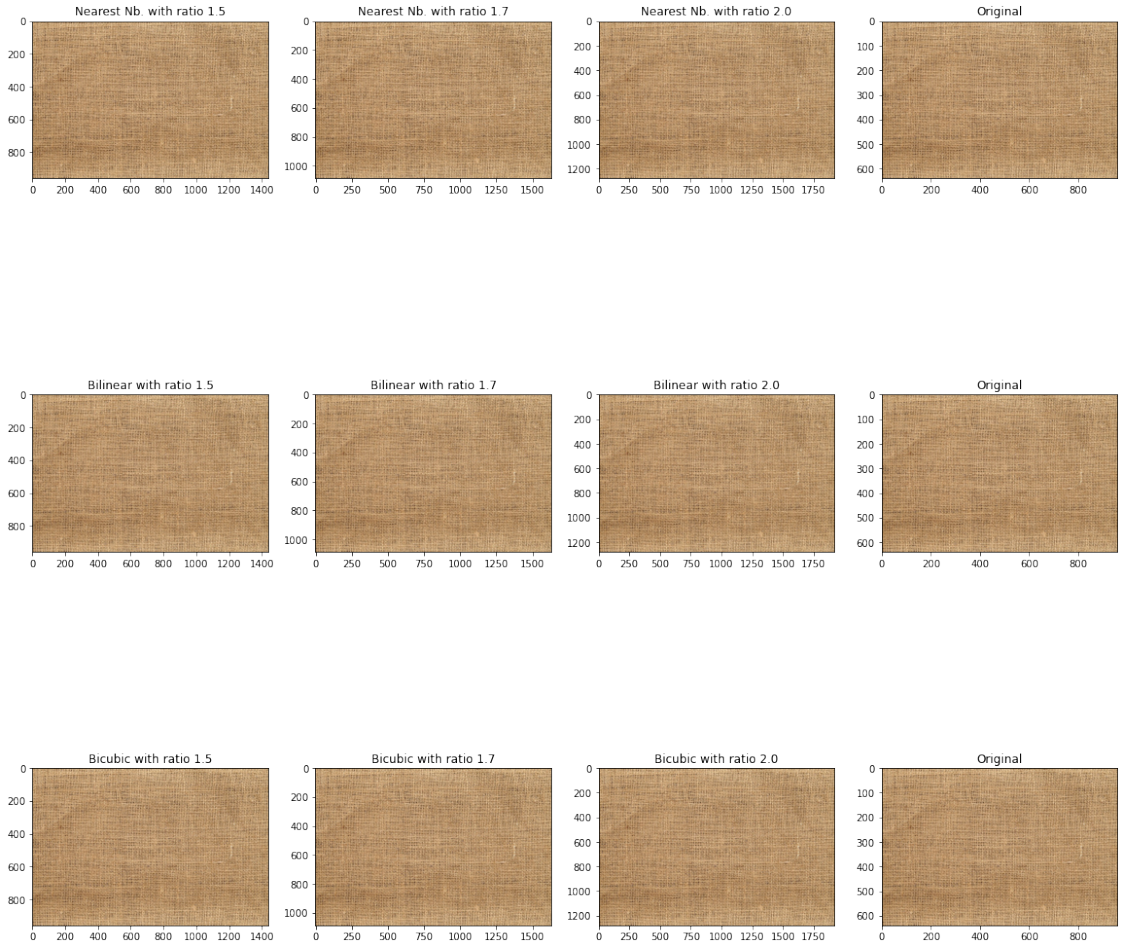
[55]:
```
ur = [1.5, 1.7, 2.0]
```

[56]:
```
curr_im = vincent

fig, axs = plt.subplots(3,4, figsize = (20,20))
```

```
for ind, ratio in enumerate(ur):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(curr_im, dsize=(round(curr_im.shape[1]*ratio),␣
 ↪round(curr_im.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind

        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```

For this image: 1. Every other condition remaining the same, I think a higher ratio of interpolation gives a more detailed image. 2. Across interpolation methods and keeping interpolation ratios the same, I think the bicubic image looks much closer to a zoomed in version of the original than the rest. It seems that for interpolation ratio two, if we zoom into the region below the eye of the subject, we see that again the nearest neighbour approach introduces salt and pepper noise. The bilinear interpolation image with ratio 2.0 appears smoother than the original in some parts.
The bicubic image is by far the most similar to the original.

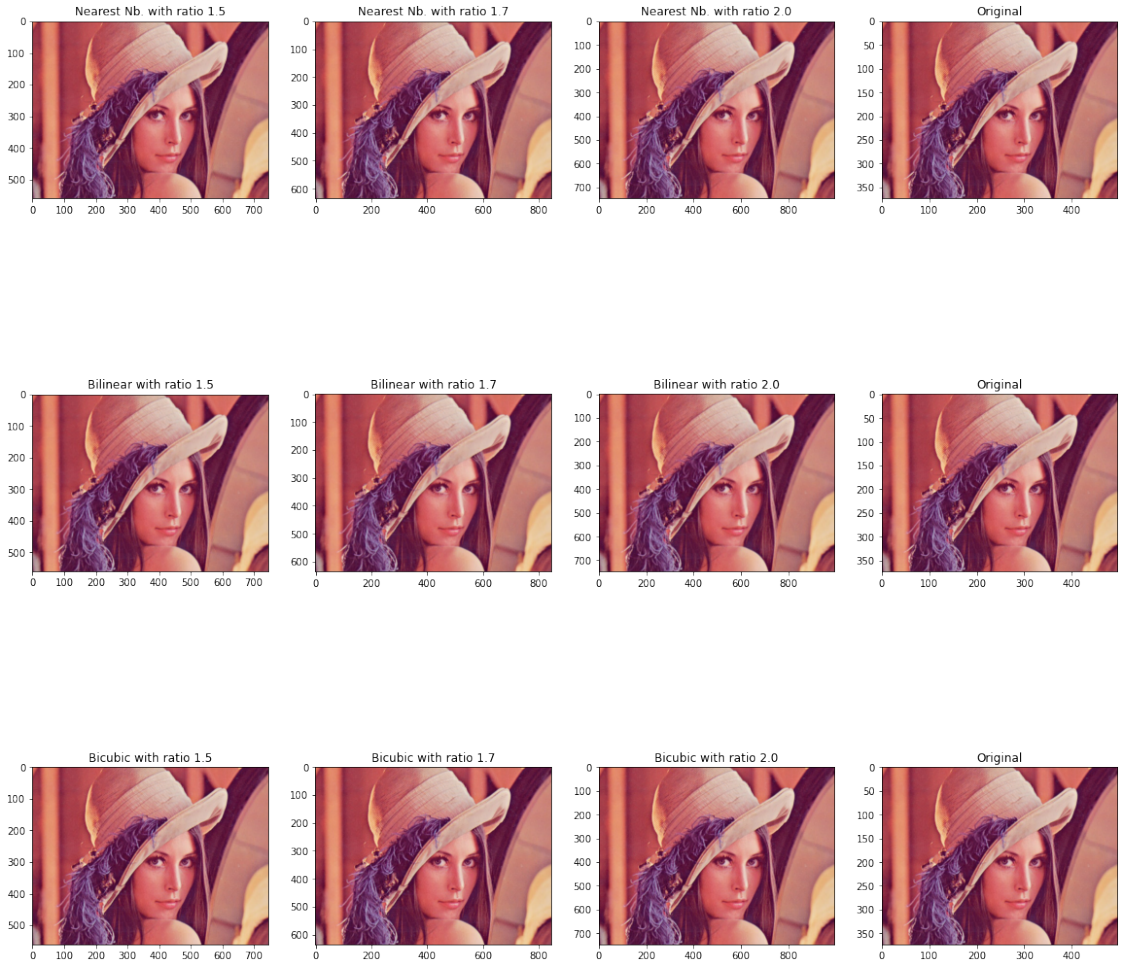
[57]:
```
curr_im = texture

fig, axs = plt.subplots(3,4, figsize = (20,20))

for ind, ratio in enumerate(ur):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(curr_im, dsize=(round(curr_im.shape[1]*ratio),␣
 ↪round(curr_im.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind

        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```



We see that the bilinear images are slightly blurry compared to the corresponding bicubic and original images. With an increase in ratio, the resultant interpolated image starts looking closer to the original. The bicubic with 2.0 ratio seems to look closer to the original than the rest.

[58]:
```
curr_im = lenna

fig, axs = plt.subplots(3,4, figsize = (20,20))

for ind, ratio in enumerate(ur):
    axs[ind, 3].imshow(curr_im)
    axs[ind, 3].set_title('Original')
    for ind2, methodd in enumerate(methods):
        im = cv2.resize(curr_im, dsize=(round(curr_im.shape[1]*ratio),␣
 ↪round(curr_im.shape[0]*ratio)), interpolation=methodd)

        row = ind2
        col = ind
```

```
        axs[row,col].imshow(im)
        axs[row,col].set_title(methods_desc[ind2] + ' with ratio ' + str(ratio))
```



In general, an increasing ratio seems to contribute towards a more faithful representation of the original image. The bicubic with the 2.0 ratio seems more like the original than the others.

0.0.10 (iv) Using 3 color images, downsample the images with scale 0.1 and upsample back to the original size, using all combinations of the three chosen interpolation methods. Which interpolation combination do you think works best to reconstruct the original image?

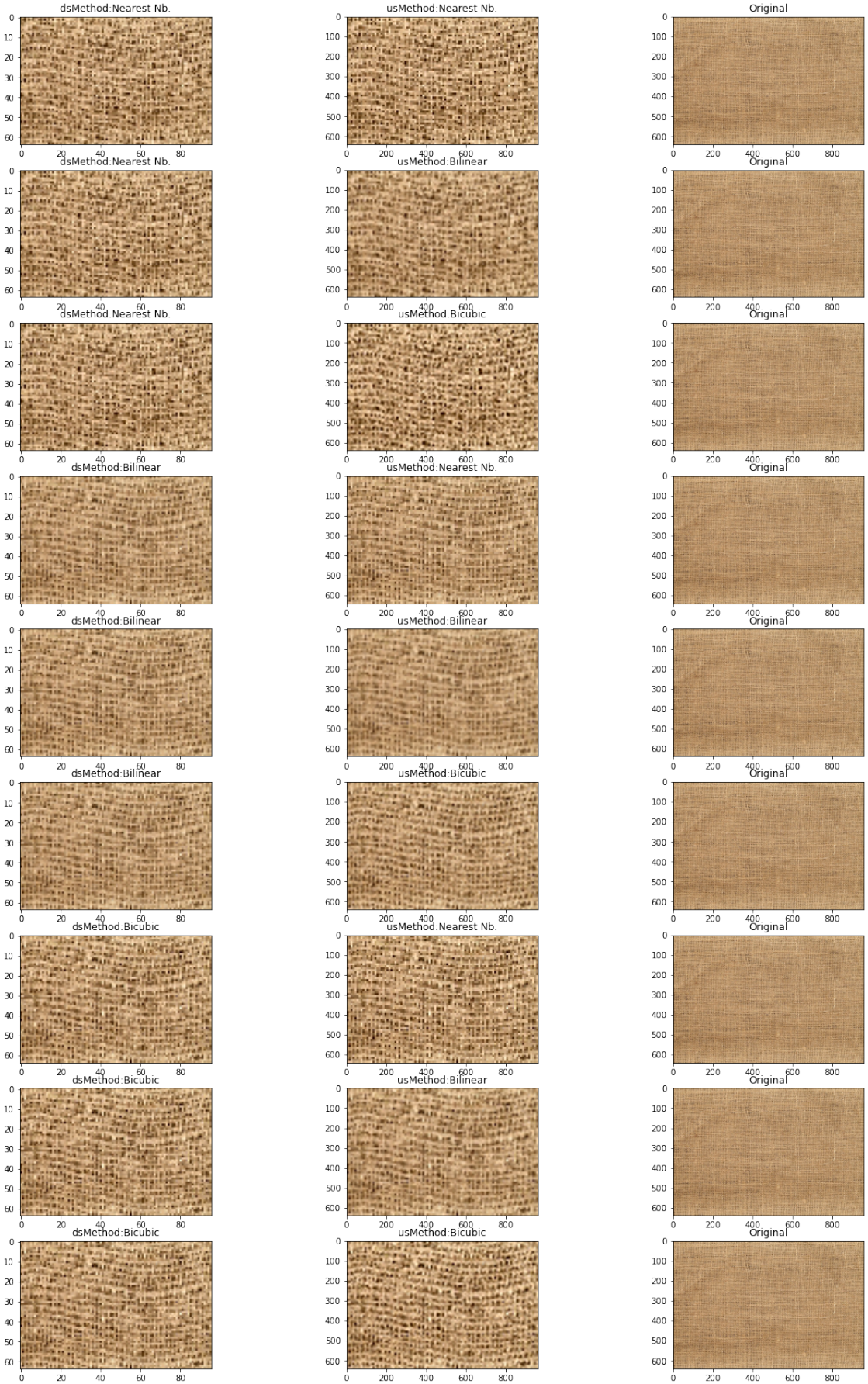
[59]:
```
dr = 0.1

# setup
images = [vincent, texture, lenna]
methods = [cv2.INTER_NEAREST, cv2.INTER_LINEAR, cv2.INTER_CUBIC]
methods_desc = ['Nearest Nb.', 'Bilinear', 'Bicubic']
```

[60]:
```
# first we downsample with scale 0.1

fig, ax = plt.subplots(9,3, figsize=(20,30))

image = vincent
dr = 0.1
plot_row_num = 0

for indx_ds, dsMethod in enumerate(methods):

    vincent_down = cv2.resize(vincent, dsize=(round(vincent.shape[1]*dr),␣
 ↪round(vincent.shape[0]*dr)), interpolation=dsMethod)

    for indx_us, usMethod in enumerate(methods):

#         print('downsampling method: ', dsMethod, ', Upsampling method: ',␣
 ↪usMethod)
        ax[plot_row_num, 0].imshow(vincent_down)
        ax[plot_row_num, 0].set_title('dsMethod:'+methods_desc[indx_ds])
        vincent_reconstruct = cv2.resize(vincent_down, dsize=(round(vincent.
 ↪shape[1]), round(vincent.shape[0])), interpolation=usMethod)
        ax[plot_row_num, 1].imshow(vincent_reconstruct)
        ax[plot_row_num, 1].set_title('usMethod:'+methods_desc[indx_us])
        # also plotting the original
        ax[plot_row_num, 2].imshow(vincent)
        ax[plot_row_num, 2].set_title('Original')
        plot_row_num = plot_row_num + 1
```


dsMethod:Nearest Nb.
usMethod:Nearest Nb.
Original
dsMethod:Nearest Nb.
usMethod:Bilinear
Original
dsMethod:Nearest Nb.
usMethod:Bicubic
Original
dsMethod:Bilinear
usMethod:Nearest Nb.
Original
dsMethod:Bilinear
usMethod:Bilinear
Original
dsMethod:Bilinear
usMethod:Bicubic
Original
dsMethod:Bicubic
usMethod:Nearest Nb.
Original
dsMethod:Bicubic
usMethod:Bilinear
Original
dsMethod:Bicubic
usMethod:Bicubic
Original

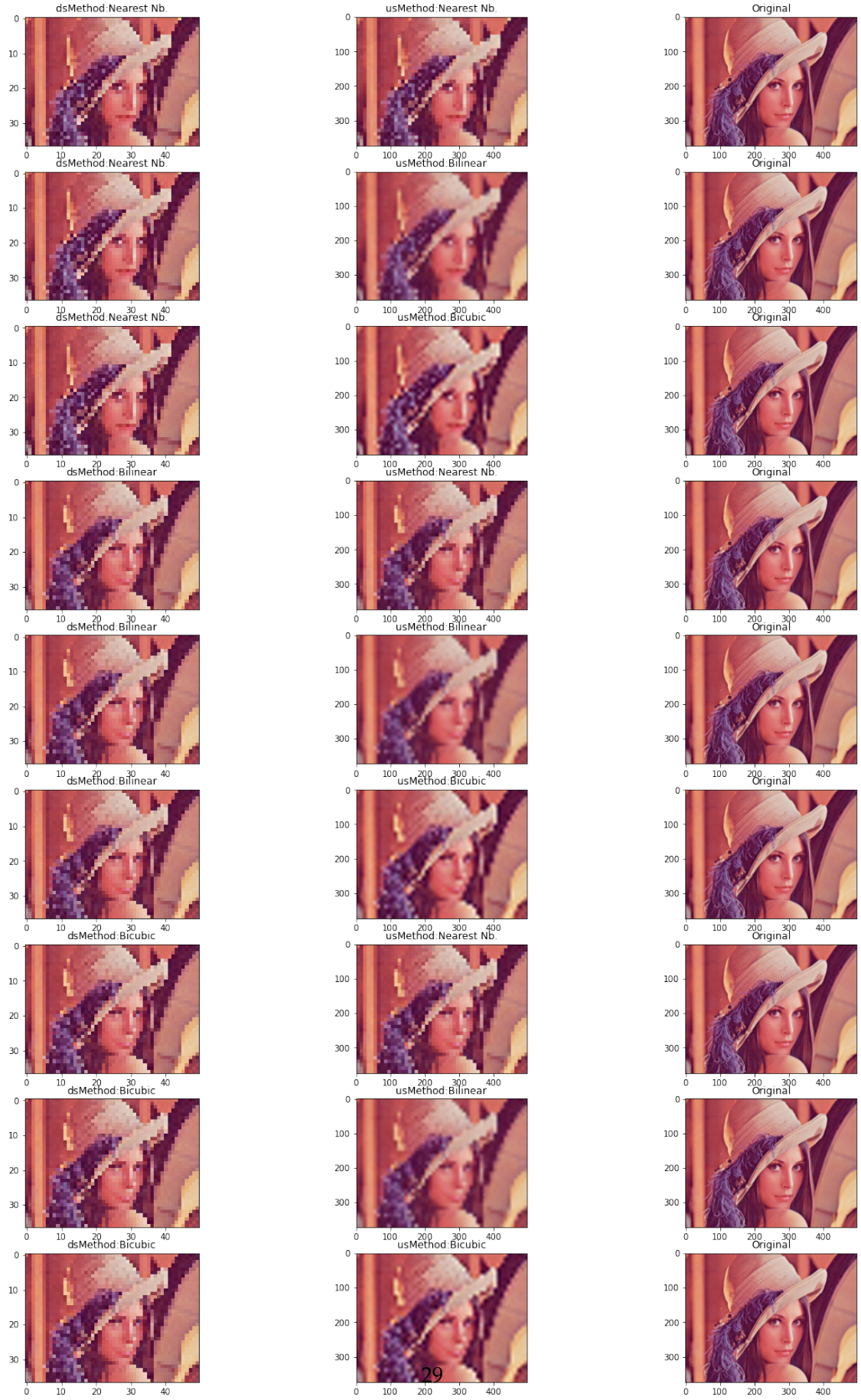
For this image of Vincent van Gogh, I think the combination of downsampling using Bicubic and upsampling using Bilinear gives a resultant image closest to the original. I initially expected the Bicubic, Bicubic to perform better than Bicubic, Bilinear, but the resultant image in that case has a lot of salt and pepper noise. The Bicubic, Bilinear image is smoother according to me.

[61]:
```
# first we downsample with scale 0.1

fig, ax = plt.subplots(9,3, figsize=(20,30))

image = texture
dr = 0.1
plot_row_num = 0

for indx_ds, dsMethod in enumerate(methods):

    vincent_down = cv2.resize(image, dsize=(round(image.shape[1]*dr),␣
 ↪round(image.shape[0]*dr)), interpolation=dsMethod)

    for indx_us, usMethod in enumerate(methods):

#         print('downsampling method: ', dsMethod, ', Upsampling method: ',␣
 ↪usMethod)
        ax[plot_row_num, 0].imshow(vincent_down)
        ax[plot_row_num, 0].set_title('dsMethod:'+methods_desc[indx_ds])
        vincent_reconstruct = cv2.resize(vincent_down, dsize=(round(image.
 ↪shape[1]), round(image.shape[0])), interpolation=usMethod)
        ax[plot_row_num, 1].imshow(vincent_reconstruct)
        ax[plot_row_num, 1].set_title('usMethod:'+methods_desc[indx_us])
        # also plotting the original
        ax[plot_row_num, 2].imshow(image)
        ax[plot_row_num, 2].set_title('Original')
        plot_row_num = plot_row_num + 1
```


dsMethod:Nearest Nb.
usMethod:Nearest Nb.
Original
dsMethod:Nearest Nb.
usMethod:Bilinear
Original
dsMethod:Nearest Nb.
usMethod:Bicubic
Original
dsMethod:Bilinear
usMethod:Nearest Nb.
Original
dsMethod:Bilinear
usMethod:Bilinear
Original
dsMethod:Bilinear
usMethod:Bicubic
Original
dsMethod:Bicubic
usMethod:Nearest Nb.
Original
dsMethod:Bicubic
usMethod:Bilinear
Original
dsMethod:Bicubic
usMethod:Bicubic
Original

For this image of a texture, I think none of the downsampled and interpolated images come even close to the original, as the original has a much higher amount of detail than the interpolated images. I think the combination of downsampling method: bicubic, and upsampling method: bicubic preserves some of the visual characteristics. For example, at the top left of the image, there is a fold. This is represented to some extent in the resultant image of the aforementioned combination.

[62]:
```
# first we downsample with scale 0.1

fig, ax = plt.subplots(9,3, figsize=(20,30))

image = lenna
dr = 0.1
plot_row_num = 0

for indx_ds, dsMethod in enumerate(methods):

    vincent_down = cv2.resize(image, dsize=(round(image.shape[1]*dr),␣
 ↪round(image.shape[0]*dr)), interpolation=dsMethod)

    for indx_us, usMethod in enumerate(methods):

#         print('downsampling method: ', dsMethod, ', Upsampling method: ',␣
 ↪usMethod)
        ax[plot_row_num, 0].imshow(vincent_down)
        ax[plot_row_num, 0].set_title('dsMethod:'+methods_desc[indx_ds])
        vincent_reconstruct = cv2.resize(vincent_down, dsize=(round(image.
 ↪shape[1]), round(image.shape[0])), interpolation=usMethod)
        ax[plot_row_num, 1].imshow(vincent_reconstruct)
        ax[plot_row_num, 1].set_title('usMethod:'+methods_desc[indx_us])
        # also plotting the original
        ax[plot_row_num, 2].imshow(image)
        ax[plot_row_num, 2].set_title('Original')
        plot_row_num = plot_row_num + 1
```


dsMethod:Nearest Nb.
usMethod:Nearest Nb.
Original
dsMethod:Nearest Nb.
usMethod:Bilinear
Original
dsMethod:Nearest Nb.
usMethod:Bicubic
Original
dsMethod:Bilinear
usMethod:Nearest Nb.
Original
dsMethod:Bilinear
usMethod:Bilinear
Original
dsMethod:Bilinear
usMethod:Bicubic
Original
dsMethod:Bicubic
usMethod:Nearest Nb.
Original
dsMethod:Bicubic
usMethod:Bilinear
Original
dsMethod:Bicubic
usMethod:Bicubic
Original

For this image of lenna, I think the combination of downsampling: bicubic, and upsampling: bilinear looks closest to the original image. The reasosn is again because it looks smoother than the bicubic, bicubic, which I initially expected to be closest to the original.