ec_dip_fund_scikit_01

January 24, 2021

0.1 Digital Image Processing, ECE419

Ernesto Colon

The Cooper Union Department of Electrical Engineering

January 21st, 2020

References: - Digital Image Processing Using Matlab [DIPUM], 3rd ed - Digital Image Processing, 4th ed

The following set of notes and example scripts are part of my independent study sessions and adapted from the references outlined above. My main reference is DIPUM and supplemented with online documentation. As I go through the theory and textbook examples, I am translating the code and image processing techniques to Python. Note, I am not writing Matlab code, I am reading the textbook and working through the example problems directly in Python.

0.1.1 Fundamentals Notes/Exercises/Sandbox - Chapter 2 from DIPUM

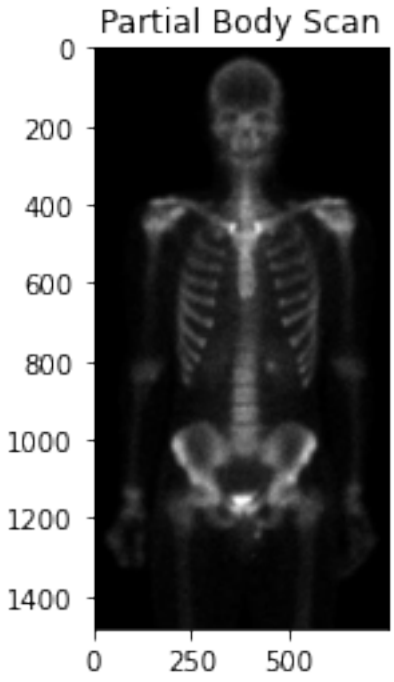
[1]:
```
#Importing the libraries
import numpy as np
import os
import matplotlib.pyplot as plt
from ec_img_utils import *      #this is a module that I am creating as I␣
 ↪progress through my IS. This module will grow in the upcoming weeks
from tabulate import tabulate
import skimage
import skimage.util
from skimage import io
```

0.1.2 Reading Image Files

[2]:
```
#using scikit-image to read image files - the images load as numpy arrays
img_body_scan = io.imread('./images/partial_body_scan.tif')

#Getting some basic information about the image - See my utilities package for␣
 ↪documentation
get_img_info(img_body_scan)
```

```
Image Information

 | Shape       | Data type   |      Bytes |
|-------------|-------------|------------|
| (1482, 750) | uint8       | 1.1115e+06 |
```

[3]:
```
#displaying the image using matplotlib since we are working with a notebook. I␣
 ↪can also use skimage.io.imshow but matplotlib will
#let me do side by side comparisons more easily

body_fig, body_ax = plt.subplots(1,1)
body_ax.imshow(img_body_scan, cmap = 'gray', vmin = 0, vmax = 255); body_ax.
 ↪set_title('Partial Body Scan');
```



The partial body scan image above has a relatively low dynamic range, which is defined as the difference between the minimum intensity value and the maximum intensity value. To enhance its visualization, we can threshold or range the image.

In Matlab, the imshow() function has an autorange feature that thresholds the image given input threshold values. We can accomplish the same task in matplotlib by passing in the vmin and vmax parameters. By default, matplotlib's imshow() uses the images' (min,max) properties for displaying.

[4]:
```
#printing min and max intensity values of our partial body scan image
print("min intensity value is: ", np.min(img_body_scan))
print("max intensity value is: ", np.max(img_body_scan))
```

```
min intensity value is:  0
max intensity value is:  255
```

Since our intensity values span the entire 2^8 -1 possible pixel values for a uint8 image, thresholding by 0 and 255 does not improve our displayed results.

Let's try different threshold values

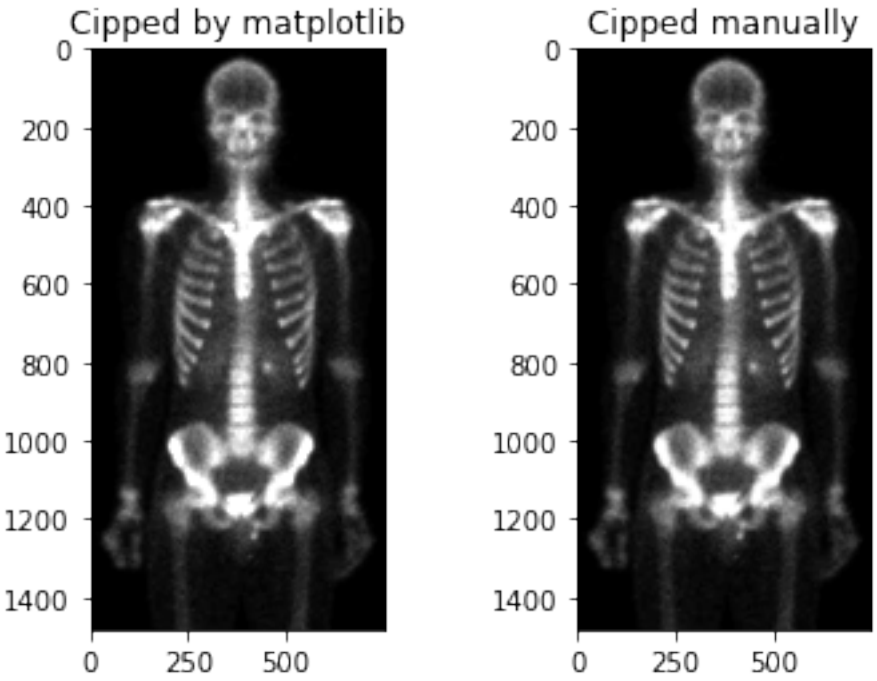
I am curious to see if my image has intensity values other than 0 and 255 (i.e., 2 discrete values)

[5]:
```
#finding unique intensity levels
body_scan_uniq_int = np.unique(img_body_scan)
print(body_scan_uniq_int)
```

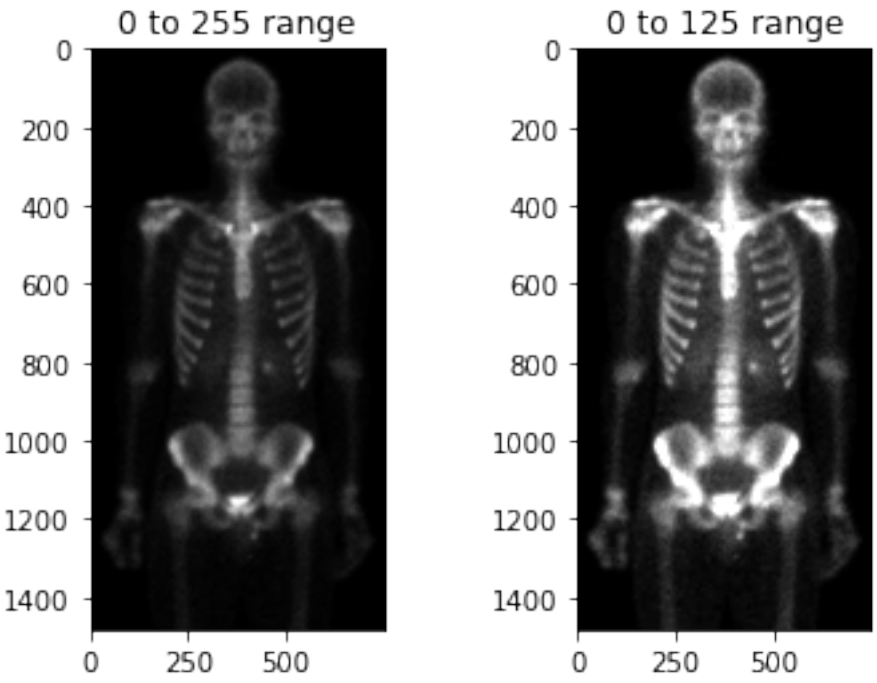
```
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 215 216
 217 218 219 220 221 222 223 224 225 226 228 229 230 231 234 235 236 237
 238 239 240 242 244 245 246 247 248 250 251 252 253 254 255]
```

Ok… so it does have more than just 0 and 255 after all. Let's threshold…

[6]:
```
#let me try clipping the image myself to verify that matplotlib is clipping to␣
 ↪125 and then normalizing to my new max when rendering
clip_val = 125
img_body_clipped = np.clip(img_body_scan, a_min = 0, a_max = clip_val)

#displaying the image
body_fig4, body_ax4 = plt.subplots(1,2)
body_ax4[0].imshow(img_body_scan, cmap = 'gray', vmin = np.min(img_body_scan),␣
 ↪vmax = clip_val) #letting matplotlib do the clipping
body_ax4[1].imshow(img_body_clipped, cmap = 'gray')  #my own clipped image
body_ax4[0].set_title('Cipped by matplotlib');
body_ax4[1].set_title('Cipped manually');
```



[7]:
```
#Now let's plot the original unclipped image next to the clipped image for␣
↪comparison

body_fig5, body_ax5 = plt.subplots(1,2)
body_ax5[0].imshow(img_body_scan, cmap = 'gray', vmin = np.min(img_body_scan),␣
↪vmax = 255)
body_ax5[1].imshow(img_body_scan, cmap = 'gray', vmin = np.min(img_body_scan),␣
↪vmax = clip_val)
body_ax5[0].set_title('0 to 255 range');
body_ax5[1].set_title('0 to 125 range');
```



Note that when matplotlib displays the image, it rescales the pixel values to fit witihn the data type's range. In other words, what we have done above is basically taken all the pixel values with intensities greater than our clip value (e.g., 125), and converted those to 255 when displaying the image. Alternatively, if one prefers to think of the intensity ranging from 0 to 1, we've taken all intensity values greater than or equal to 125/255 and made those equal to 1.

If we were to look at the individual intensity values of our clipped image, we'd see that our maximum value is 125.

[8]:
```
np.max(img_body_clipped)
```

[8]: 125

0.1.3 Writing Images

From scikit-image documentation page: https://scikit-image.org/docs/dev/user_guide/data_types.html#rescalin intensity-values

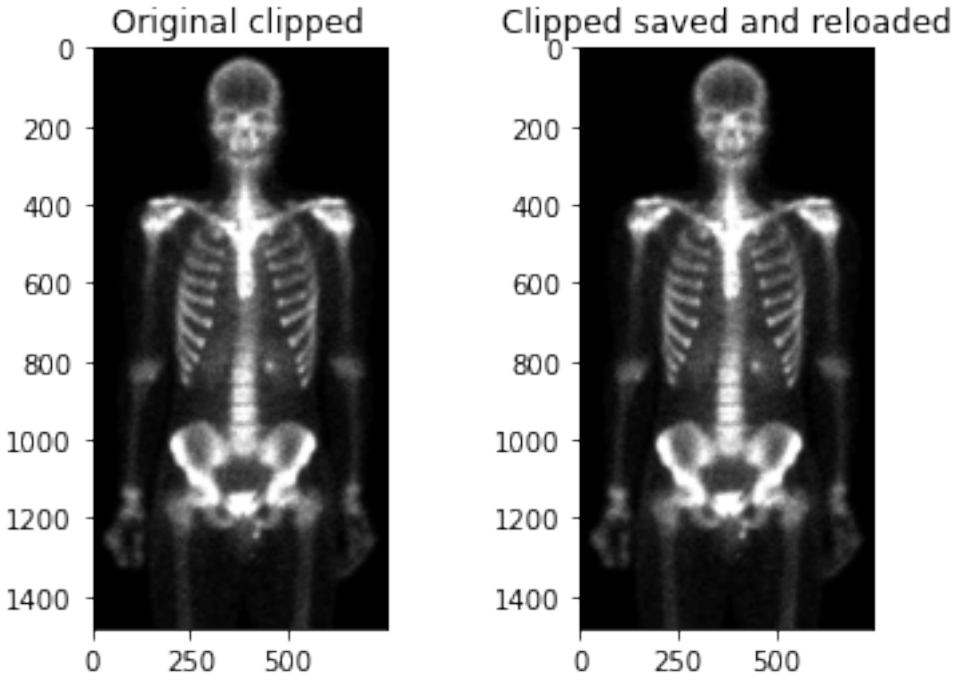
Supported data types

- uint8
- uint16
- uint32
- float (range -1 to 1 or 0 to 1)
- int8
- int16
- int32

scikit-image has a number of built-in utilities to convert from one data type to another which we will explore later

[9]:
```
#Our body scan image is an 8-bit single channel image. Let's save it as a tiff␣
 ↪file for consistency with the original format and reload it to check it out
#saving clipped image
io.imsave("./images/output/partial_body_scan_clipped_2.tif",img_body_clipped)
```

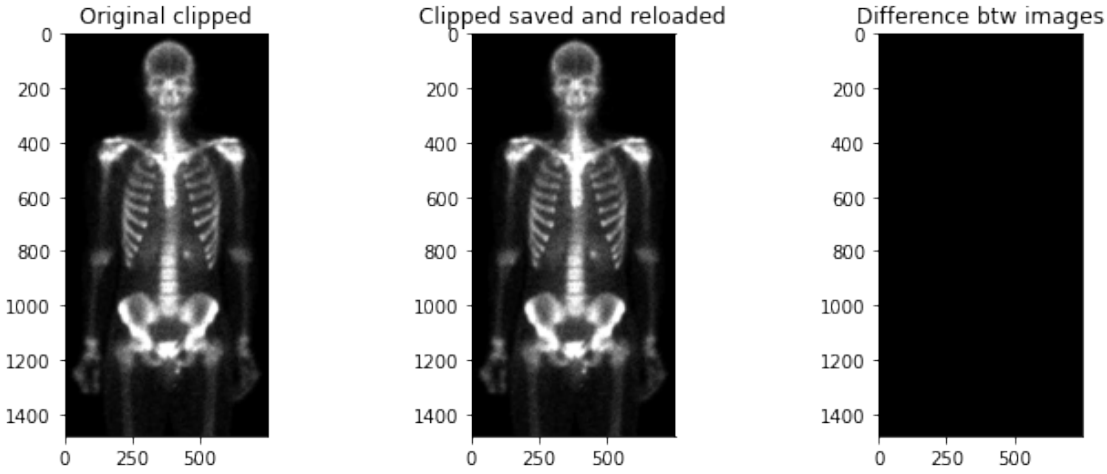
[10]:
```
####################
#SANITY CHECK
####################

#Reload image and display
img_body_clipped_reld = io.imread("./images/output/partial_body_scan_clipped_2.
 ↪tif")

body_fig6, body_ax6 = plt.subplots(1,2)
body_ax6[0].imshow(img_body_clipped, cmap = 'gray')
body_ax6[1].imshow(img_body_clipped_reld, cmap = 'gray')
body_ax6[0].set_title('Original clipped');
body_ax6[1].set_title('Clipped saved and reloaded');
```



0.1.4 Subtracting the images

Let's subtract my saved image from its original source image to see if they differ by much, if at all

[11]:
```
#subtracting
img_body_subtrct = img_body_clipped - img_body_clipped_reld

body_fig7, body_ax7 = plt.subplots(1,3,figsize = (10,10))
body_ax7[0].imshow(img_body_clipped, cmap = 'gray')
body_ax7[1].imshow(img_body_clipped_reld, cmap = 'gray')
body_ax7[2].imshow(img_body_subtrct, cmap = 'gray')
body_ax7[0].set_title('Original clipped');
body_ax7[1].set_title('Clipped saved and reloaded');
body_ax7[2].set_title('Difference btw images');
body_fig7.subplots_adjust(wspace = 1.0);
```



Our image write was succesful. There is no noticeable difference between the clipped/manipulated image and our saved image. I need to do more experimentation with image represented by floats...

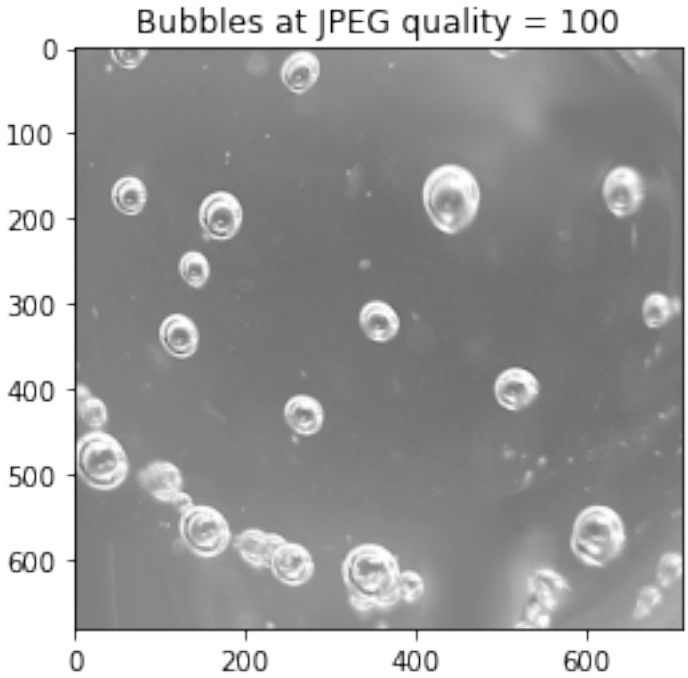
0.1.5 Example 2.2 JPEG Compression (writing images) - DIPUM

In this exercise, we will save a jpg image with increasing levels of compression and assess how much the image degrades visually. The degradation is measured by the level of **false contouring** present in the image

[12]:
```
#load bubbles image
img_bubbles = io.imread('./images/bubbles100.jpg')  #this image is at the␣
 ↪highest jpg quality available from the source DIPUM

get_img_info(img_bubbles)
```

```
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (682, 714) | uint8       |  486948 |
```

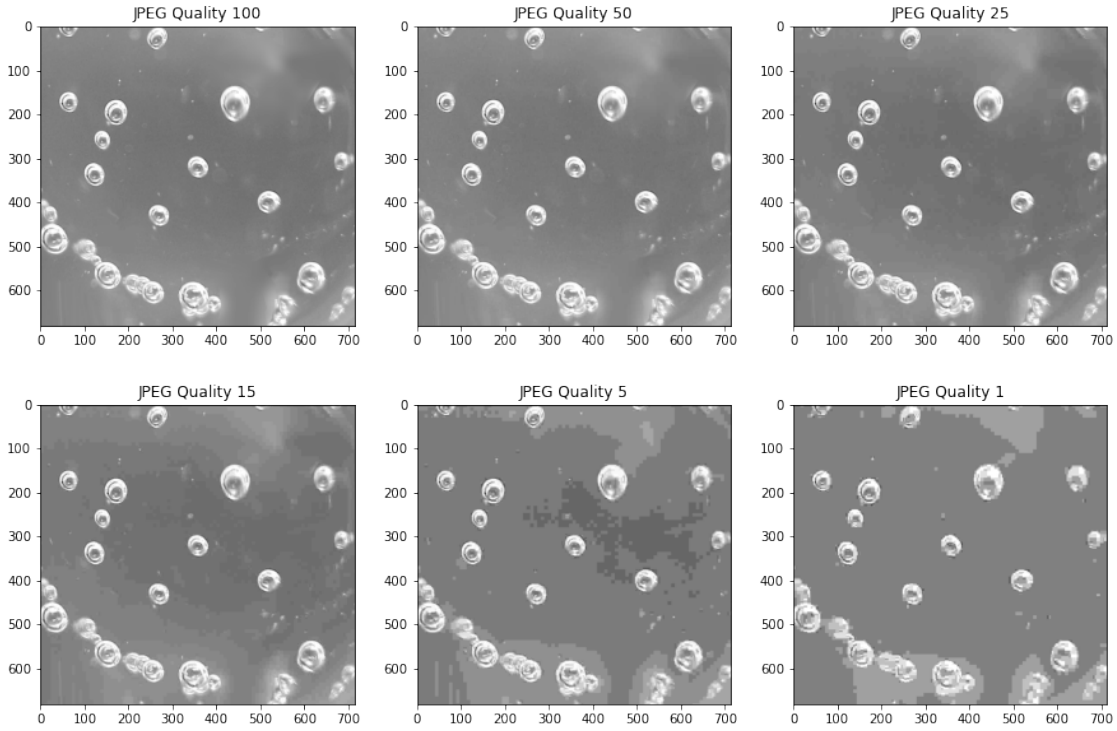
[13]:
```
#displaying the image
fig_bub, ax_bub = plt.subplots(1,1)
ax_bub.imshow(img_bubbles, cmap='gray'); ax_bub.set_title('Bubbles at JPEG␣
↪quality = 100');
```



[14]:
```
#Now, let's save the image with progressively increasing compression levels
jpg_qual_lst = [50,25,15,5,1]   #list containing jpg quality levels on a scale␣
↪from 0 to 100 with the highest quality being 100

for q in jpg_qual_lst:
    #cv2.imwrite('./images/output/bubbles'+str(q)+'.jpg',img_bubbles, [cv2.
↪IMWRITE_JPEG_QUALITY, q])
    io.imsave('./images/output/bubbles_'+str(q)+'.jpg',img_bubbles, quality = q)
```

[15]:
```
#display all compressed images
fig_bub2, ax_bub2 = plt.subplots(2,int((len(jpg_qual_lst)+1)/2),figsize =␣
↪(15,10))
ax_bub2 = ax_bub2.ravel()
ax_bub2[0].imshow(img_bubbles,cmap='gray'); ax_bub2[0].set_title('JPEG Quality␣
↪100');

for i, item in enumerate(jpg_qual_lst):
    #load image
    temp_img = io.imread('./images/output/bubbles_'+str(item)+'.jpg')
```

```
    ax_bub2[i+1].imshow(temp_img,cmap='gray'); ax_bub2[i+1].set_title('JPEG␣
↪Quality '+str(item))
```



Note the appearance of false countours starting with a JPEG compression level of 15.

Also note that the JPEG compresison quaility in scikit-image ranges from 1 to 100 as opposed to 0 to 100 as in Matlab or even OpenCV for Python.

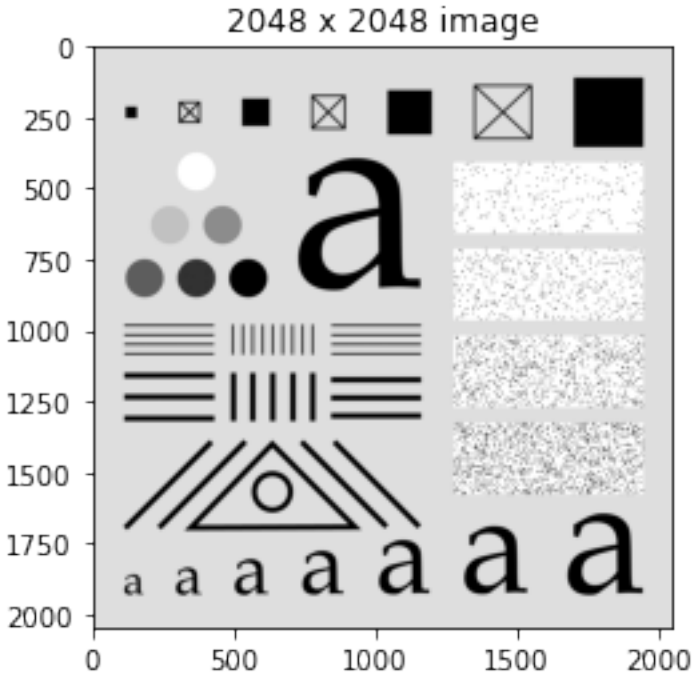
0.1.6 Changing the size of an image (tiff) without changing its resolution by manipulating the dots per inch (dpi) resolution

Example 2.3 from DIPUM - the original image has a resolution of 2048 x 2048 pixels at 800 dpi –> its size is 2.56 x 2.56 in We want to change the image size for printing from 2.56 x 2.56 in to 1.7 x 1.7 in without changing the number of pixels in the image.

We can accomplish this by saving the image as a tiff file with a different dpi resolution for both the rows and columns

[16]:
```
#load test image pattern
img_test_pattern = io.imread('./images/testpattern2048.tif')

#display original image
fig_tst_pttrn, ax_tst_pttrn = plt.subplots(1,1)
ax_tst_pttrn.imshow(img_test_pattern,cmap='gray'); ax_tst_pttrn.set_title('2048␣
 ↪x 2048 image');
```

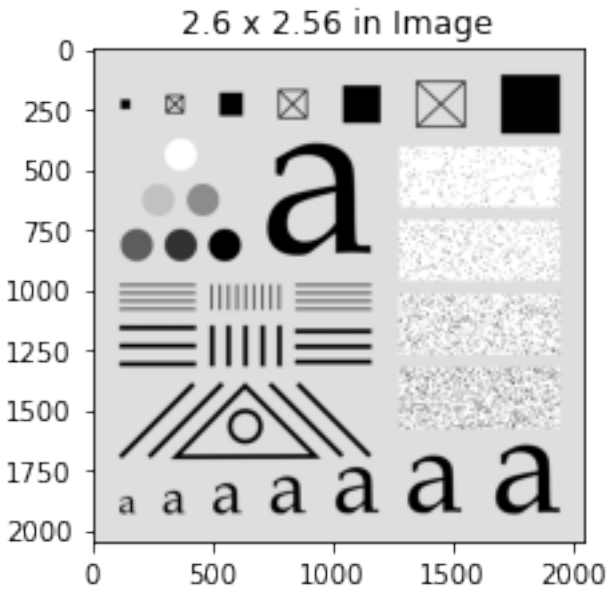
```
#To resize image to 1.7 x 1.7 in, we need to modify our DPI/resolution as␣
 ↪follows
res = np.round(800*((2.56/1.7)))
```

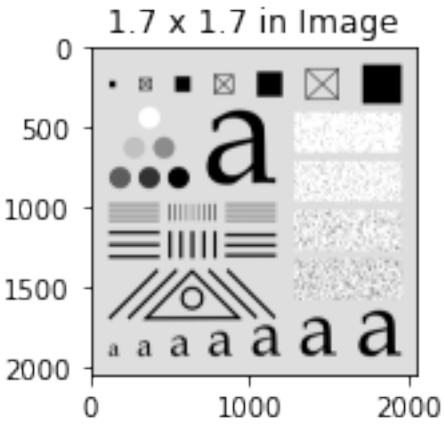


[17]:
```
#change dpi (resize image) while maintaining the same number of pixels
#img_pttrn_rszed = cv2.imwrite('./images/output/testpattern2048_rszed.tif',␣
 ↪img_test_pattern, [cv2.IMWRITE_TIFF_XDPI, res, cv2.IMWRITE_TIFF_YDPI, res])
img_pttrn_rszed = io.imsave('./images/output/testpattern2048_rszed_2.tif',␣
 ↪img_test_pattern,'tifffile', resolution = (res,res))
```

[18]:
```
#load resized image
pttrn_rszed_img = io.imread('./images/output/testpattern2048_rszed_2.tif')

#display both images
orig_dpi = 800
pttrn_img_ht, pttrn_img_wd = img_test_pattern.shape

#what size does the figure need to be in inches to fit the image?
figsize1 = pttrn_img_wd / float(orig_dpi), pttrn_img_ht / float(orig_dpi)

#create a figure of the right size with one axes that takes up the full figure
pttrn_fig1 = plt.figure(figsize = figsize1)
ax1 = pttrn_fig1.add_axes([0,0,1,1])
```

```
##################
#resized image
##################

pttrn_img_ht2, pttrn_img_wd2 = pttrn_rszed_img.shape

#what size does the figure need to be in inches to fit the image?
figsize2 = pttrn_img_wd2 / float(res), pttrn_img_ht2 / float(res)

#create a figure of the right size with one axes that takes up the full figure
pttrn_fig2 = plt.figure(figsize = figsize2)
ax2 = pttrn_fig2.add_axes([0,0,1,1])

#display image
ax1.imshow(img_test_pattern,cmap='gray')
ax1.set_title('2.6 x 2.56 in Image');

#display image
ax2.imshow(pttrn_rszed_img,cmap='gray')
ax2.set_title('1.7 x 1.7 in Image');
```





[19]:
```
#Verify that both images have the same number of pixels
print('Original Image Info\n')
get_img_info(img_test_pattern)
print('\nResized Image Info\n')
get_img_info(pttrn_rszed_img)
```

```
Original Image Info

Image Information

 | Shape        | Data type   |      Bytes |
|--------------|-------------|------------|
| (2048, 2048) | uint8       | 4.1943e+06 |

Resized Image Info

Image Information

 | Shape        | Data type   |      Bytes |
|--------------|-------------|------------|
| (2048, 2048) | uint8       | 4.1943e+06 |
```

0.1.7 Converting from one dtype to another using skimage

Reference: https://scikit-image.org/docs/dev/user_guide/data_types.html#image-data-types-and-what-they-mean

Warning: "You should **never use** *astype* on an image, because it violates [the] assumptions about the dtype range."

[20]:
```
image = np.arange(0,50,10, dtype = np.uint8)  #uint8 type image
```

```
print(image.astype(float))                    #these float values are out of␣
 ↪range for a float representation. Recall that the intensity values for a␣
 ↪float inmage are within [0,1] for unsigned.
```

```
[ 0. 10. 20. 30. 40.]
```

Float images are in the range [-1, 1] or [0, 1]

[21]:
```
#converting to float using skimage - skimage handles the range scaling
print(skimage.util.img_as_float(image))
```

```
[0.         0.03921569 0.07843137 0.11764706 0.15686275]
```

Note that our image has been properly scaled

[22]:
```
#converting to uint8 from float
img_float = np.array([0, 0.5, 1], dtype=float)
print("float image: ", img_float)
ubyte_img = skimage.util.img_as_ubyte(img_float)
print("uint8 image: ", ubyte_img)
```

```
float image:  [0.  0.5 1. ]
uint8 image:  [  0 128 255]
```

Again, note that skimage has properly scaled the images

0.1.8 Loss of precision example

[23]:
```
img_float2 = np.array([0, 0.5, 0.503, 1], dtype = float)
print(img_float2.dtype)
print("float image: ", img_float2)
```

```
float64
float image:  [0.    0.5   0.503 1.   ]
```

Converting to uint8 results in loss of precision since 8 bits cannot hold the same amount of information as 64 bits

[24]:
```
img_ubyte2 = skimage.util.img_as_ubyte(img_float2)
print(img_ubyte2.dtype)
print("ubyte image: ", img_ubyte2)
```

```
uint8
ubyte image:  [  0 128 128 255]
```

Note that pixel values 0.5 and 0.503 both got converted to 128. The conversion is as follows *round(pix_val 255)**

[25]:
```
#converting manually for a sanity check
print('check with pix = 0.5: ', np.round(255*0.5).astype(np.uint8))
print('check with pix = 0.503: ', np.round(255*0.503).astype(np.uint8))
```

```
check with pix = 0.5:  128
check with pix = 0.503:  128
```

0.1.9 Binary/boolean images

A binary image assigns the value 1 (true) to the upper half of the input dtype's positive range and 0 to the lower half. All negative values (if present) are False.

This is different from Matlab's binary image format where all non-zero pixels are set to 1 (positive or negative) and 0 otherwise

[26]:
```
img_float3 = np.array([[-0.5, 0.5],[0.75, 1.0]])
print('Float image')
print(img_float3,'\n\n')

#convert to binary image
img_bin = skimage.util.img_as_bool(img_float3)
print('Boolean image')
print(img_bin)
```

```
Float image
[[-0.5   0.5 ]
 [ 0.75  1.  ]]


Boolean image
[[False False]
 [ True  True]]
```

0.1.10 Other image conversion functions from skimage

- img_as_float32 –> single-precision 32-bit floating point format with range [0.0, 1.0] or [-1.0, 1.0] when converting from unsigned or signed respectively
- img_as_float64 –> double-precision 64-bit floating point format with range [0.0, 1.0] or [-1.0, 1.0] when converting from unsigned or signed respectively
- img_as_int –> 16-bit signed integer format with range -32768 to 32767
- img_as_ubyte –> 8-bit unsigned interger format with range 0 to 255. Negative inout values will be clipped to 0
- img_as_uint –> 16-bit unsigned integer format with range 0 to 65535. Negative input values will be clipped to 0

0.1.11 Binarizing images

If we want to binarize an image in Python, we can simply use a relational operator on the numpy array and get a binary image out. This is similar to Matlab's imbinarize() function which takes an input threshold parameter

[27]:
```
#binarize example
img_float2 = np.array([0, 0.5, 0.503, 1], dtype = float)
```

```
print("float image: ", img_float2,'\n')

img_binrzd = img_float2 > 0.2
print("binarized image: ",img_binrzd,'\n')

#check image type
print("image type: ", img_binrzd.dtype)
```

```
float image:  [0.    0.5   0.503 1.   ]

binarized image:  [False  True  True  True]

image type:  bool
```

0.1.12 Using Python's *isintance()* to determine image dtype

Note that the image itself is represented as a numpy array of class ndarray. Thus, we must examine an individual pixel to determine its type.

[28]:
```
#testing img_float from above
print("float32? ",isinstance(img_float2[0], np.float32))
print("bool? ",isinstance(img_float2[0], bool))
print("uint8? ",isinstance(img_float2[0], np.uint8))
print("float64? ",isinstance(img_float2[0], np.float64))
```

```
float32?  False
bool?  False
uint8?  False
float64?  True
```

0.1.13 Array Indexing with Numpy to Manipulate Images

Example 2.5 from DIPUM

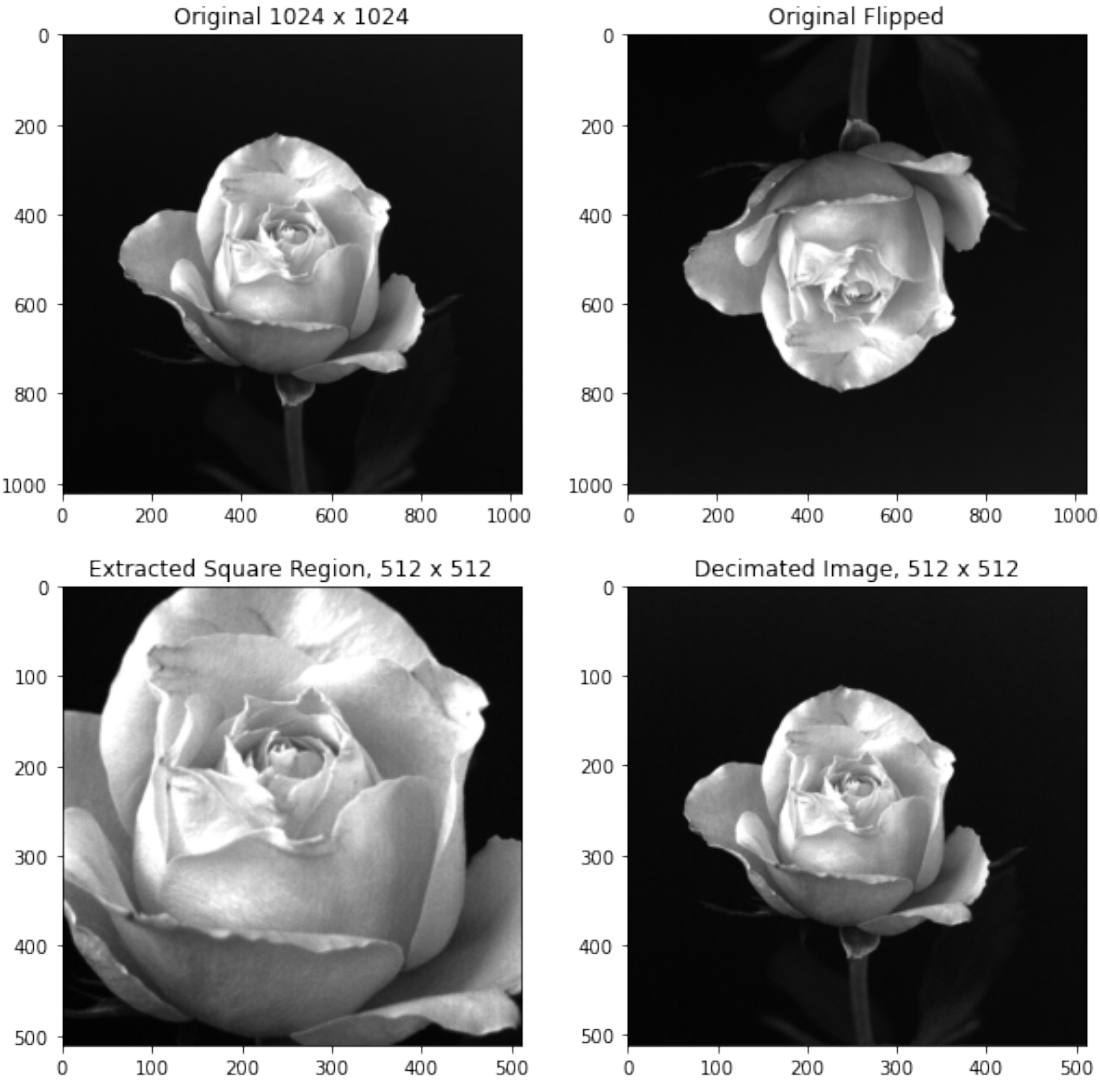
[29]:
```
#load rose image of size 1024 x 1024
img_rose = io.imread('./images/rose1024.tif')
get_img_info(img_rose)
```

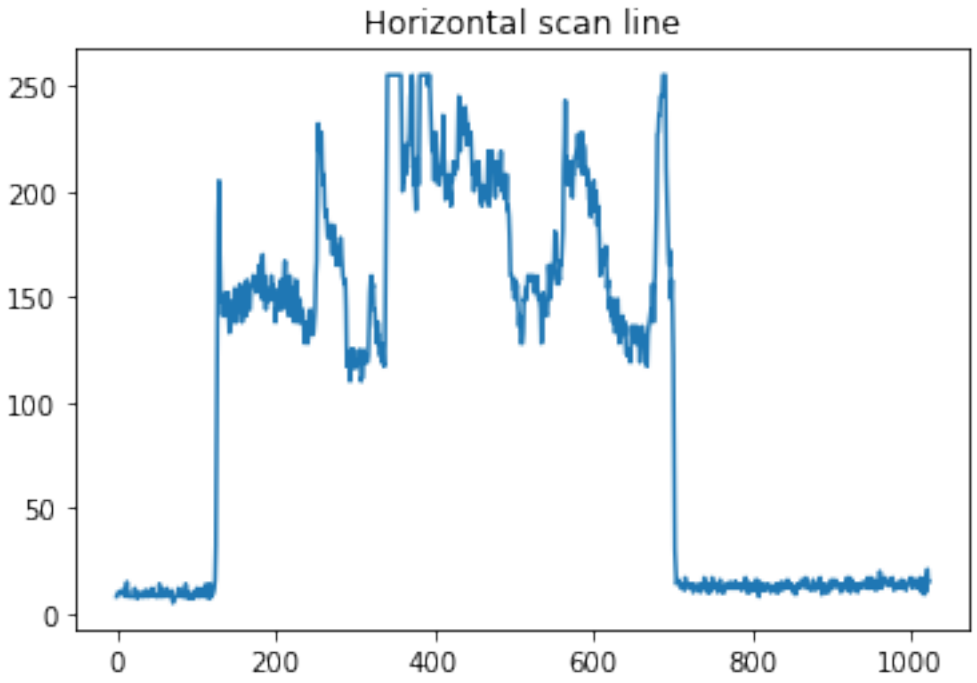
```
Image Information

 | Shape         | Data type   |       Bytes |
|---------------|-------------|-------------|
| (1024, 1024) | uint8       | 1.04858e+06 |
```

[30]:
```
#Let's flip the image vertically by flipping the rows
img_rose_vflip = img_rose[::-1]  #flip all rows

#Extract square region of the image
img_rose_sq_reg = img_rose[257:768, 257:768]
```

```
#subsample rose image by decimating by 2
img_rose_subsamp = img_rose[0:img_rose.shape[0]:2, 0:img_rose.shape[1]:2]

#Let's extract a horizontal scan line through the middle of the image
img_rose_scan_ln = img_rose[512,:]
```

[31]:
```
#displaying the processed images from the steps above
fig_rose, ax_rose = plt.subplots(2,2,figsize = (10,10))
ax_rose = ax_rose.ravel()
ax_rose[0].imshow(img_rose,cmap = 'gray'); ax_rose[0].set_title("Original 1024␣
 ↪x 1024");
ax_rose[1].imshow(img_rose_vflip,cmap = 'gray'); ax_rose[1].set_title("Original␣
 ↪Flipped");
ax_rose[2].imshow(img_rose_sq_reg,cmap = 'gray'); ax_rose[2].
 ↪set_title("Extracted Square Region, 512 x 512");
ax_rose[3].imshow(img_rose_subsamp,cmap = 'gray'); ax_rose[3].
 ↪set_title("Decimated Image, 512 x 512");

fig_rose1, ax_rose1 = plt.subplots(1,1)
ax_rose1.plot(img_rose_scan_ln); ax_rose1.set_title('Horizontal scan line');
```

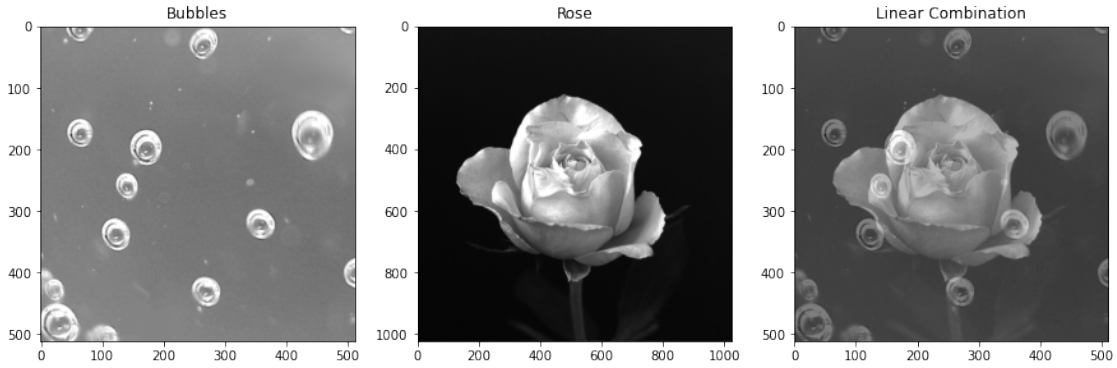

Original 1024 x 1024
Original Flipped
Extracted Square Region, 512 x 512
Decimated Image, 512 x 512



0.1.14 Linear Combination of Images

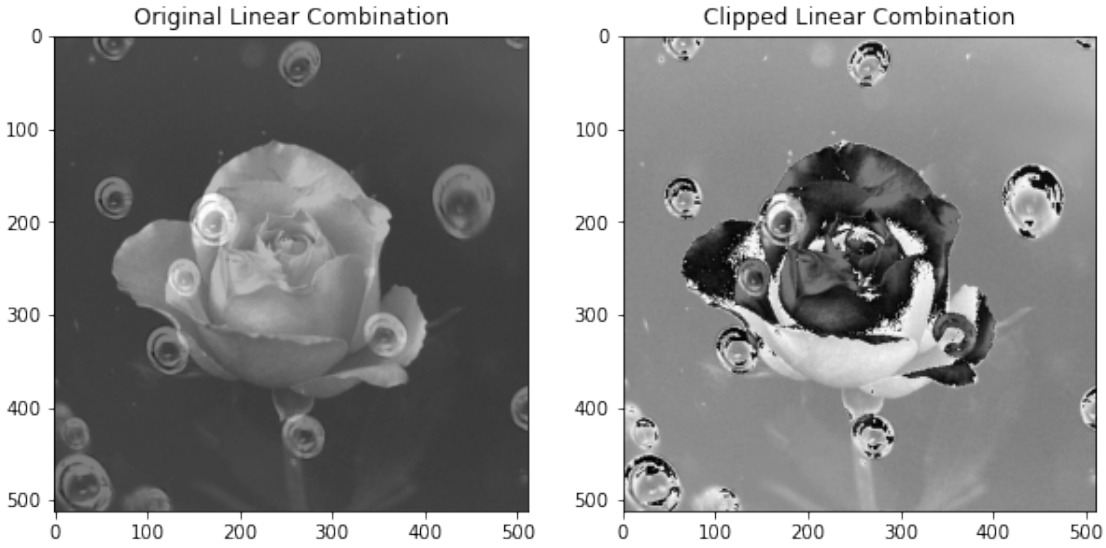
Let's experiment with linearly combining two images while watching out for overflow possibilities

[32]:
```
#let's crop a 512 x 512 region from the bubbles image
img_bub_512 = img_bubbles[0:512,0:512]

#linearly combine both images with coefficients [0.5 0.5]
img_lin_comb = 0.5*img_bub_512 + 0.5*img_rose_subsamp

#convert image to uint8 since numpy multiplication operations result in floats.␣
 ↪Float images must be in the range [-1,1] or [0,1] for unsigned types, so
#I must scale appropriately before converting
img_lin_comb = skimage.img_as_ubyte(img_lin_comb/np.max(img_lin_comb))

#display the linear combination
fig_lin_comb, ax_lin_comb = plt.subplots(1,3,figsize = (15,5)); ax_lin_comb =␣
 ↪ax_lin_comb.ravel()
ax_lin_comb[0].imshow(img_bub_512, cmap='gray'); ax_lin_comb[0].
 ↪set_title('Bubbles');
ax_lin_comb[1].imshow(img_rose, cmap='gray'); ax_lin_comb[1].set_title('Rose');
ax_lin_comb[2].imshow(img_lin_comb, cmap='gray'); ax_lin_comb[2].
 ↪set_title('Linear Combination');
```



Note that multiplying each image by its coefficient is imperative in this case as opposed to 0.5*(img1 + img2). The latter approach will cause pixels with intensity values over 255 to clip (overflow) for this uint8 operation. See below for a comparison

[33]:
```
#causing the linear combination to overflow
img_lin_comb_ovrflw = 0.5*(img_bub_512 + img_rose_subsamp)

#convert image back to uint8
img_lin_comb_ovrflw = skimage.img_as_ubyte(img_lin_comb_ovrflw/np.
 ↪max(img_lin_comb_ovrflw))

#display images
fig_lin_comb2, ax_lin_comb2 = plt.subplots(1,2,figsize = (10,5)); ax_lin_comb2␣
 ↪= ax_lin_comb2.ravel()
ax_lin_comb2[0].imshow(img_lin_comb, cmap='gray'); ax_lin_comb2[0].
 ↪set_title('Original Linear Combination');
ax_lin_comb2[1].imshow(img_lin_comb_ovrflw, cmap='gray'); ax_lin_comb2[1].
 ↪set_title('Clipped Linear Combination');
```



[34]:
```
#confirm we have uint8 images
print("Origianl Linear Combination Information")
get_img_info(img_lin_comb)

print("Overflowed Linear Combination Information")
get_img_info(img_lin_comb_ovrflw)
```

```
Origianl Linear Combination Information
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (512, 512) | uint8       |  262144 |
Overflowed Linear Combination Information
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (512, 512) | uint8       |  262144 |
```

We can see a significant amount of degradation in our image's appearance due to having oveflowed the intensity values.

What if we converted our image to uint16 first and then linearly combined the images? We will still have the same issue as before because first converting to uint16 essentially rescales my pixel intensity values to the range [0, 2^16 -1]. Let's see...

[35]:
```
#converting images to uint16 prior to linear combination
img_rose_uint = skimage.img_as_uint(img_rose_subsamp)
img_bub_512_uint = skimage.img_as_uint(img_bub_512)
```

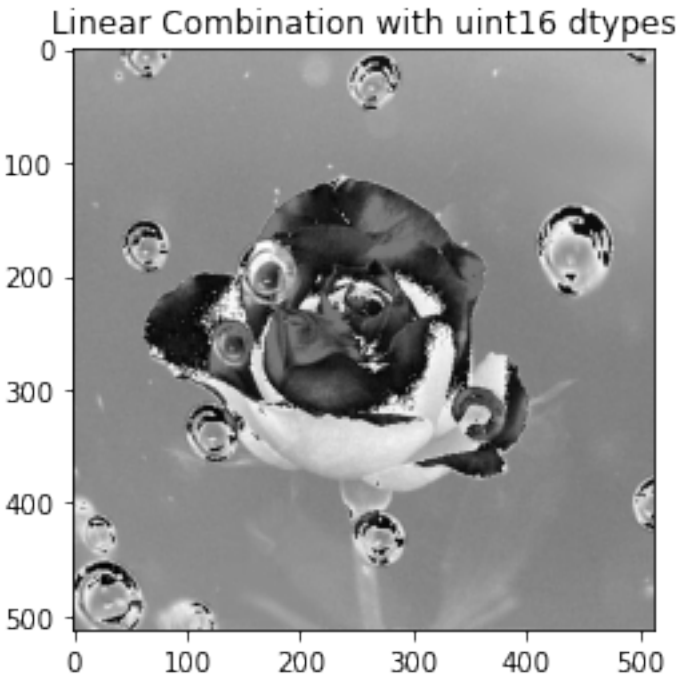
```
#verify dtype conversion
get_img_info(img_rose_uint)
get_img_info(img_bub_512_uint)
```

```
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (512, 512) | uint16      |  524288 |
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (512, 512) | uint16      |  524288 |
```

[36]:
```
#linearly combine the uint16 images
img_lin_comb_uint = 0.5*(img_rose_uint + img_bub_512_uint)

#rescale and convert back to uint16 from float
img_lin_comb_uint = skimage.img_as_uint(img_lin_comb_uint/np.
 ↪max(img_lin_comb_uint))

#confirm converstion worked
get_img_info(img_lin_comb_uint)

#print max value just to check
print(np.max(img_lin_comb_uint))
```

```
Image Information

 | Shape      | Data type   |   Bytes |
|------------|-------------|---------|
| (512, 512) | uint16      |  524288 |
65535
```

[37]:
```
#display uint16 image
plt.imshow(img_lin_comb_uint,cmap='gray'); plt.title("Linear Combination with␣
 ↪uint16 dtypes");
```



We encounter the same problem

0.1.15 Working in float space is the key...

[38]:
```
#converting images to float64 prior to linear combination
img_rose_float64 = skimage.img_as_float64(img_rose_subsamp)
img_bub_512_float64 = skimage.img_as_float64(img_bub_512)

#verify dtype conversion
get_img_info(img_rose_float64)
get_img_info(img_bub_512_float64)
```

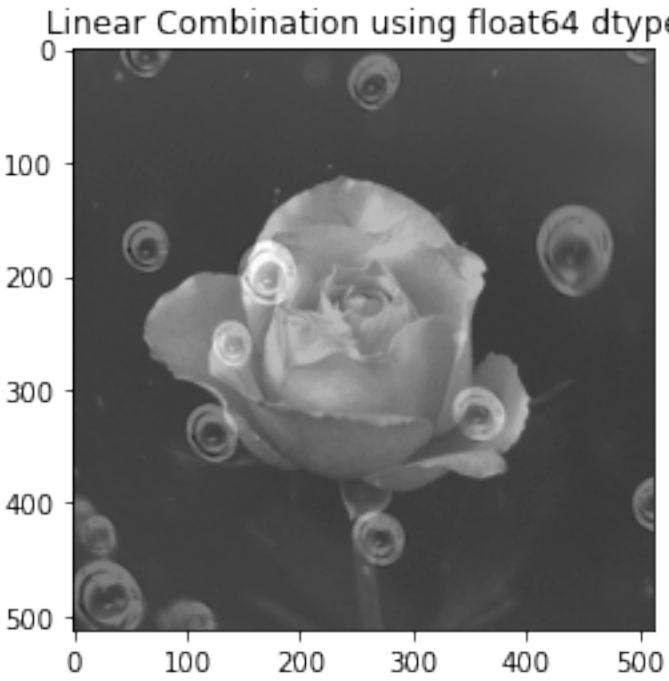
```
Image Information

 | Shape      | Data type   |       Bytes |
|------------|-------------|-------------|
| (512, 512) | float64     | 2.09715e+06 |
Image Information

 | Shape      | Data type   |       Bytes |
|------------|-------------|-------------|
| (512, 512) | float64     | 2.09715e+06 |
```

[39]:
```
print(np.max(img_rose_float64))
```

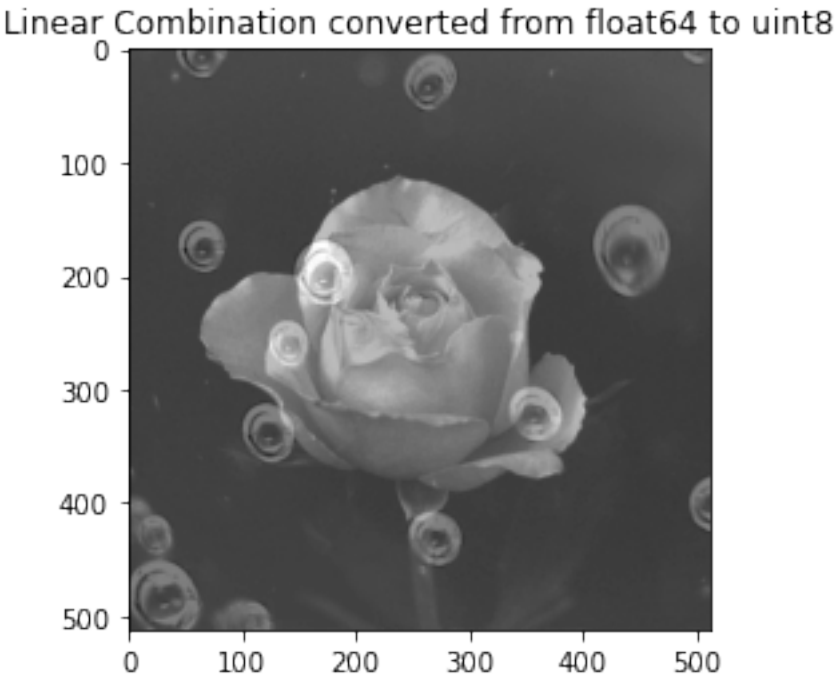
```
1.0
```

[40]:
```
#perform linear combination
img_lin_comb_float64 = 0.5*(img_rose_float64 + img_bub_512_float64)

print(np.max(img_lin_comb_float64))

#rescale image and plot
plt.imshow(img_lin_comb_float64/np.max(img_lin_comb_float64),cmap='gray'); plt.
↪title("Linear Combination using float64 dtype");
```

```
1.0
```



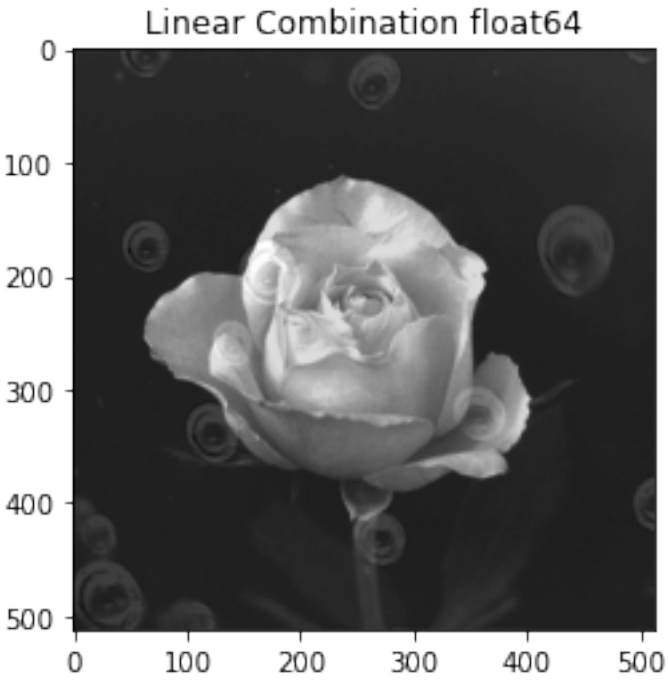
[41]:
```
#what if I now convert my image back to uint8?
img_lin_comb_back_to_uint8 = skimage.img_as_ubyte(img_lin_comb_float64)

#display the image
plt.imshow(img_lin_comb_back_to_uint8,cmap='gray'); plt.title('Linear␣
↪Combination converted from float64 to uint8');
```



[42]:
```
#what if I oveflow even the float range? That is, what if my coefficients␣
 ↪result in intensities > 1?
img_lin_comb_grtr1 = 1.5*img_rose_float64 + 0.5*img_bub_512_float64

#print max intensity
print(np.max(img_lin_comb_grtr1))
```

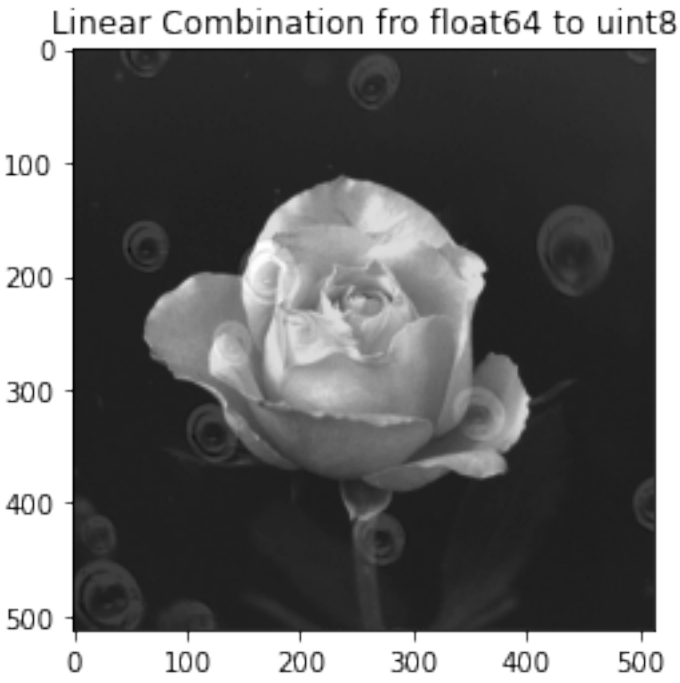
```
2.0
```

[43]:
```
#display the image - note that the imshow function scales appropriately
plt.imshow(img_lin_comb_grtr1, cmap='gray'); plt.title('Linear Combination␣
 ↪float64');
```



[44]:
```
#what if I try to convert this image without first rescaling? I think the␣
 ↪built-in function to convert between dtypes takes care of the scaling?
#img_lin_comb_grtr1_uint8 = skimage.img_as_ubyte(img_lin_comb_grtr1)

#NOPE... I got the following error:

# ValueError: Images of type float must be between -1 and 1.
```

[45]:
```
#before converting my overflowed image, I need to rescale
img_lin_comb_grtr1_uint8 = skimage.img_as_ubyte(img_lin_comb_grtr1/np.
 ↪max(img_lin_comb_grtr1))

#display image
plt.imshow(img_lin_comb_grtr1,cmap = 'gray'); plt.title('Linear Combination fro␣
 ↪float64 to uint8');
```


Linear Combination fro float64 to uint8
0
100
200
300
400
500
0
100
200
300
400
500