Cornelio, Andrew HW1

February 15, 2020

1 Mathematical Image Analysis (553.493)

Cornelio, Andrew

February 16, 2020

1.0.1 Problem 1.1

We can use an affine transformation which maps a to 0 and b to $L-1$. Specifically, we can use:

$$S(i,j) = (L-1)\left(\frac{I(i,j)-a}{b-a}\right)$$

This will result in a pixel range that is stretched over all values of pixels. However, one potentially undesirable side effect of this method is that since the both the domain is finite, this transformation may produce noticable jumps color from pixel to pixel as it amplifies the (finite) differences.

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

1.0.2 Problem 1.2.a

We create a list of all the gammas and then loop through then, additing them to a figure and creating a label:

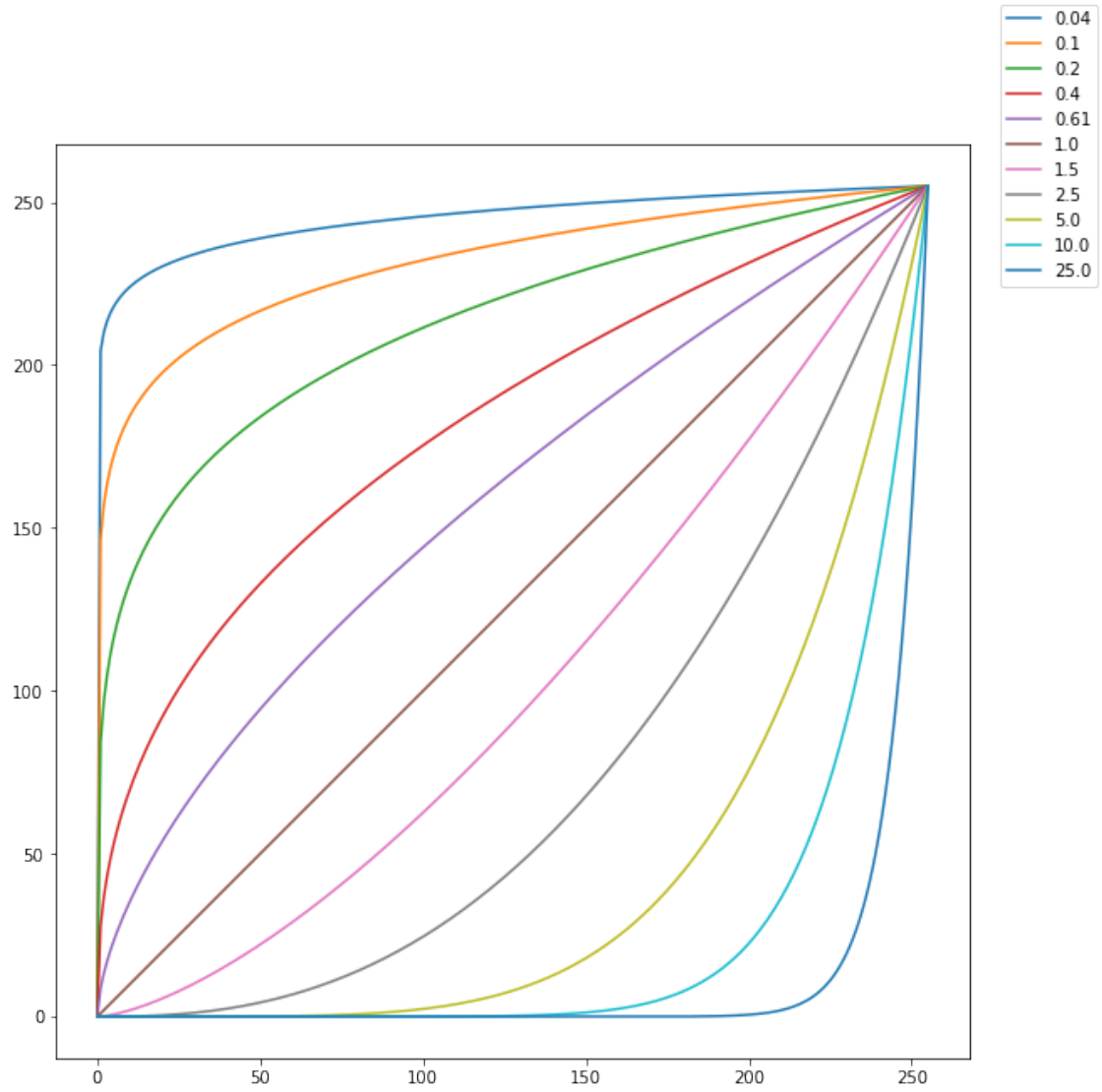
[2]:
```
gammas = [0.04, 0.1, 0.2, 0.4, 0.61, 1.0, 1.5, 2.5, 5.0, 10.0, 25.0]
fig = plt.figure()
L = 256.0
x = np.linspace(0,255,255)

for gamma in gammas:
    plt.plot(x,(L - 1)* ((x / (L - 1)) ** gamma), label=str(gamma))

plt.gca().set_aspect('equal', adjustable='box')
```

```
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
fig.legend()
```

[2]: <matplotlib.legend.Legend at 0x7f5d6950ed50>



1.0.3 Problem 1.2.b

Using simple algebra, we can see that the inverse gamma is:

$$S_\gamma^{-1}(i,j) = (L-1)\left(\frac{J(i,j)}{L-1}\right)^{\frac{1}{\gamma}}$$

We see that this is simply the gamma function that uses the reciprocal value as the original.

1.0.4 Problem 1.2.c

Rather than looping through the image, we can use the OpenCV look up table feature. First we create an array that maps the original pixel values to transformed pixel values using the gamma function and a list comprehension. Then we use use the cv2.LUT function to actually map the old pixels onto the new ones. Finally, we convert the transformed image to a numpy array so we can more easily do manipulations of it, such as unraveling it into a 1D array and creating a pixel histogram.

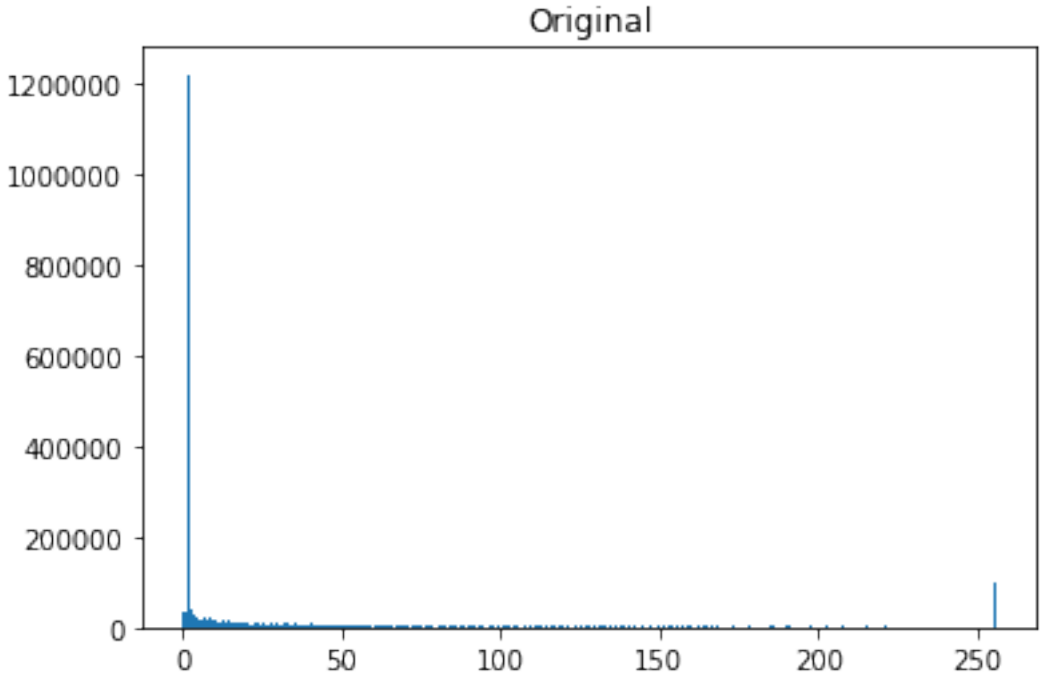
[3]:
```
# easy way of performing discrete pixel transformation on image is to create a␣
↪look up table
def gamma_func(image, gamma):
    table = np.array([((i / 255.0) ** gamma) * 255
        for i in np.arange(0, 256)]).astype("uint8")
    return np.array(cv2.LUT(image, table))
```

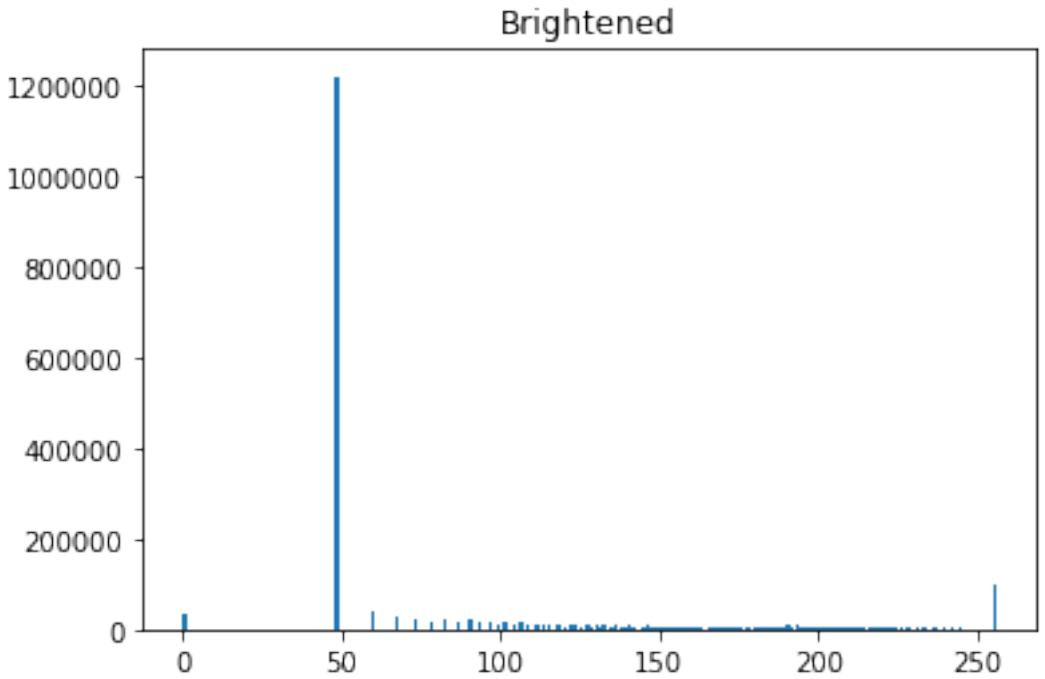
[4]:
```
image_spine = cv2.imread('image-spine.tif')
image_spine_bright = gamma_func(image_spine, 0.3)

img2 = image_spine_bright[:,:,::-1]
plt.imshow(img2)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
```

[5]:
```
plt.hist(image_spine.ravel(),256,[0,256])
plt.title('Original')
plt.show()
```

```
plt.hist(image_spine_bright.ravel(),256,[0,256])
plt.title('Brightened')
plt.show()
```



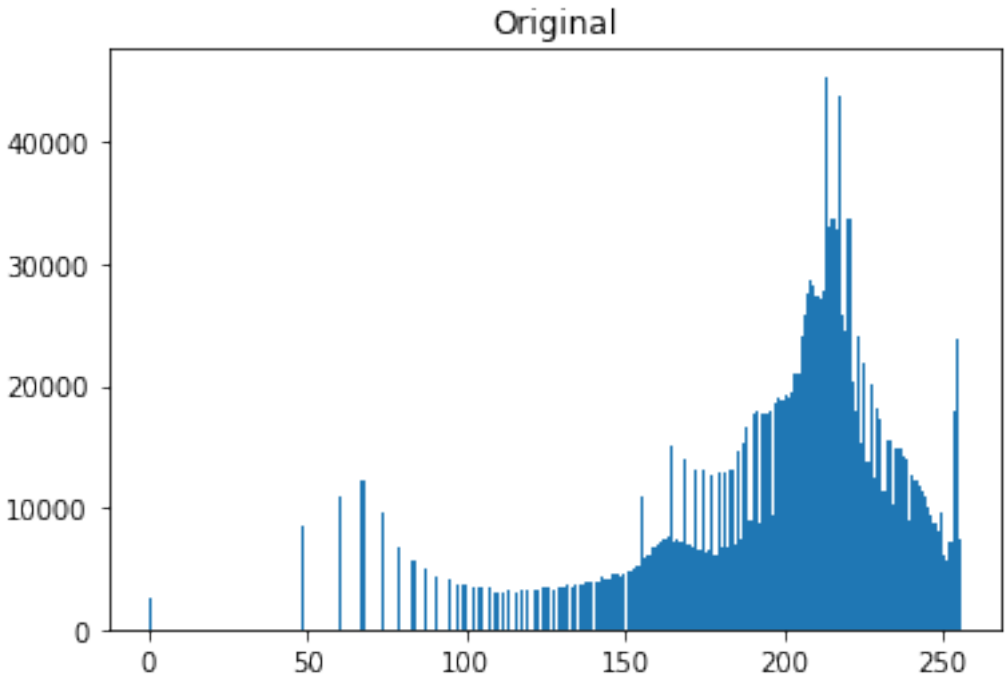


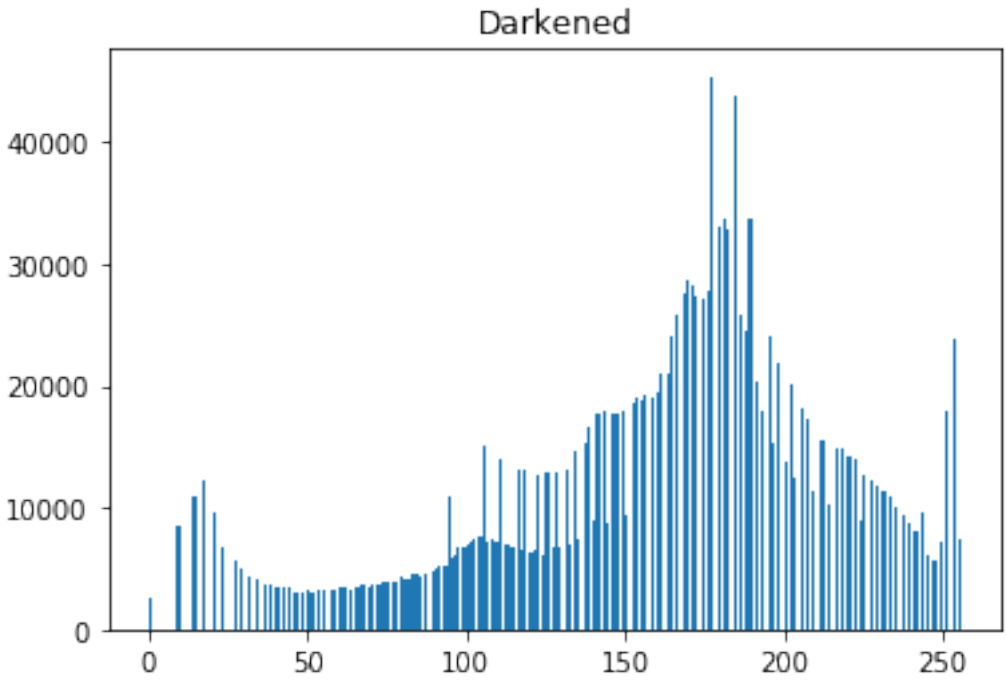
[6]:
```
image_airport = cv2.imread('image-airport.tif')
image_airport_dark = gamma_func(image_airport, 2)

img2 = image_airport_dark[:,:,::-1]
plt.imshow(img2)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
```

[7]:
```
plt.hist(image_airport.ravel(),256,[0,256])
plt.title('Original')
plt.show()
```

```
plt.hist(image_airport_dark.ravel(),256,[0,256])
plt.title('Darkened')
plt.show()
```





1.0.5 Problem 1.3

We can use opencv's filter2D function to do the convolution. We just need to define a normalized, 3x3 numpy array of 1's for the first part and a 5x5 part for the second part. For the third part, we can use opencv's getGaussianKernel to generate the coefficients of a 1D kernel of length 5. We pick sigma to be 5. Then we multiply it by it's transpose to get a 2D kernel (this works because the gaussians are independent so their joint density is the product of their densities). I noticed that the image convolved with the 3x3 filter is a lot less noisy than the original, but still has a lot of noise. The 5x5 averaging filter is less noisy but is still grainy and there is less continuity between the pixels. The gaussian blur shows the least noise and the most continuity.

[8]:
```
image_lena_noisy = cv2.imread('image-lena-noisy.png')
```

[9]:
```
kernel1 = 1/9 * np.ones((3,3))
image_lena_blur1 = cv2.filter2D(image_lena_noisy, -1, kernel1)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
plt.imshow(image_lena_blur1)
```

[9]:
```
<matplotlib.image.AxesImage at 0x7f5d637ab210>
```

[10]:
```
kernel2 = 1/25 * np.ones((5, 5))
image_lena_blur2 = cv2.filter2D(image_lena_noisy, -1, kernel2)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
plt.imshow(image_lena_blur2)
```

[10]:
```
<matplotlib.image.AxesImage at 0x7f5d68015510>
```

[11]:
```
row = cv2.getGaussianKernel(5, sigma=5)
kernel3 = np.outer(row, row.transpose())
image_lena_blur3 = cv2.filter2D(image_lena_noisy, -1, kernel3)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
plt.imshow(image_lena_blur3)
```

[11]: <matplotlib.image.AxesImage at 0x7f5d6808a5d0>

1.0.6 Problem 1.4.a

A non linear filter is simply a filter that uses a nonlinear function. It is reasonably easy to prove that the median function is nonlinear. Say we have the positive integer values $a_1 < a_2 < a_3$. We have:

$$med(a_1, a_2, a_3) = a_2$$

If the median function was linear, we'd expect that when we add a_3 to the second argument we would get a linear combination of the two functions. However, this is not what we get:

$$med(a_1, a_2 + a_3, a_3) = a_3 \neq a_2 = med(a_1, a_2, a_3) + med(a_1, a_2, a_3)$$

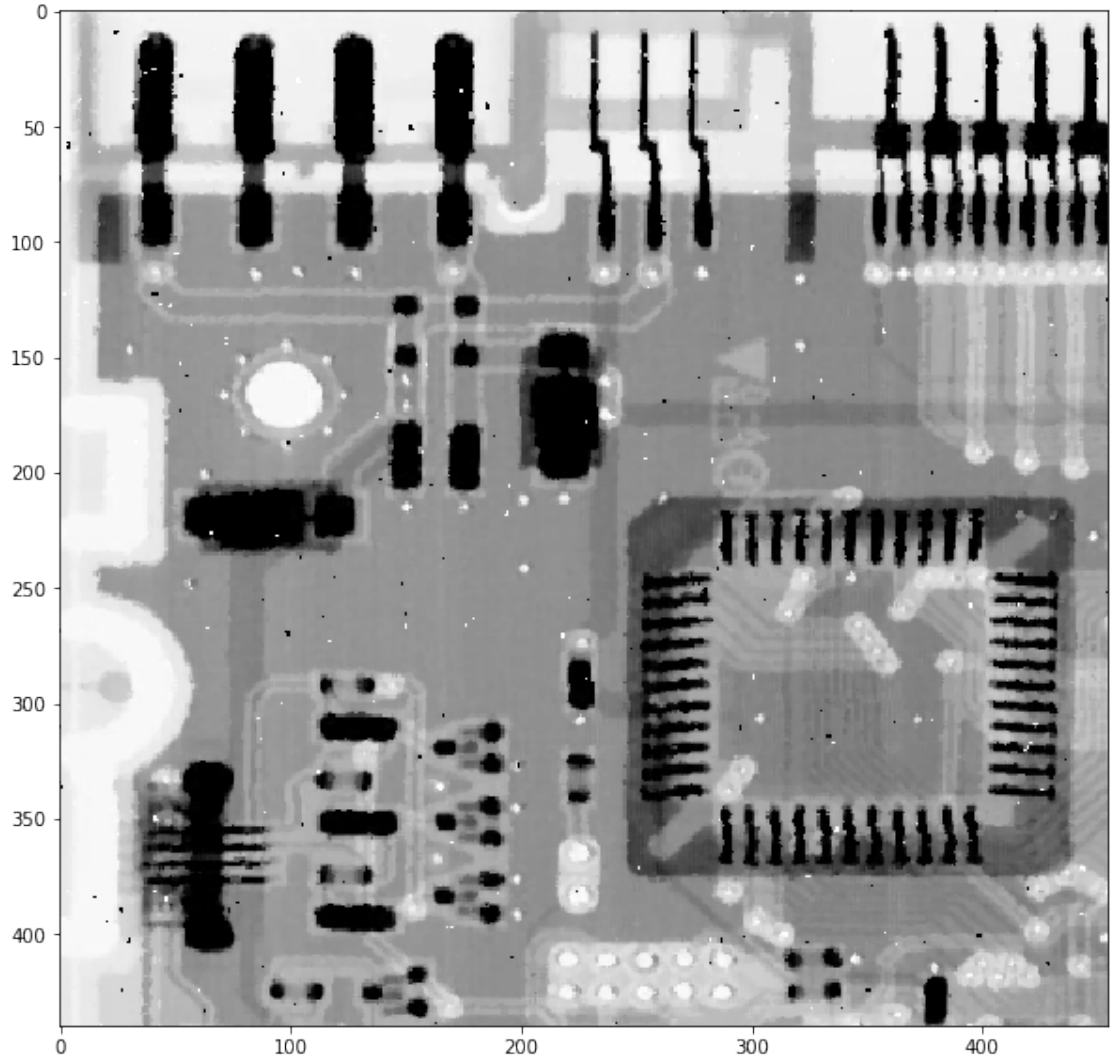
The scaling property of linear functions also does not hold, but showing this is enough.

1.0.7 Problem 1.4.b

We first use opencv's medianBlur function with a kernel size 3. Then we use the lnear averaging filter that we used in the previous problem. We can see such a noticable difference due to the fact that the median filter mostly elimiates the salt and pepper noise (since 0 and 255 are at the lowest and highest ends of the pixel range so they most likely won't be a median unless the pixels we are considering are in the middle of a salt or pepper spot) whereas the linear average spreads the salt and pepper noise around.

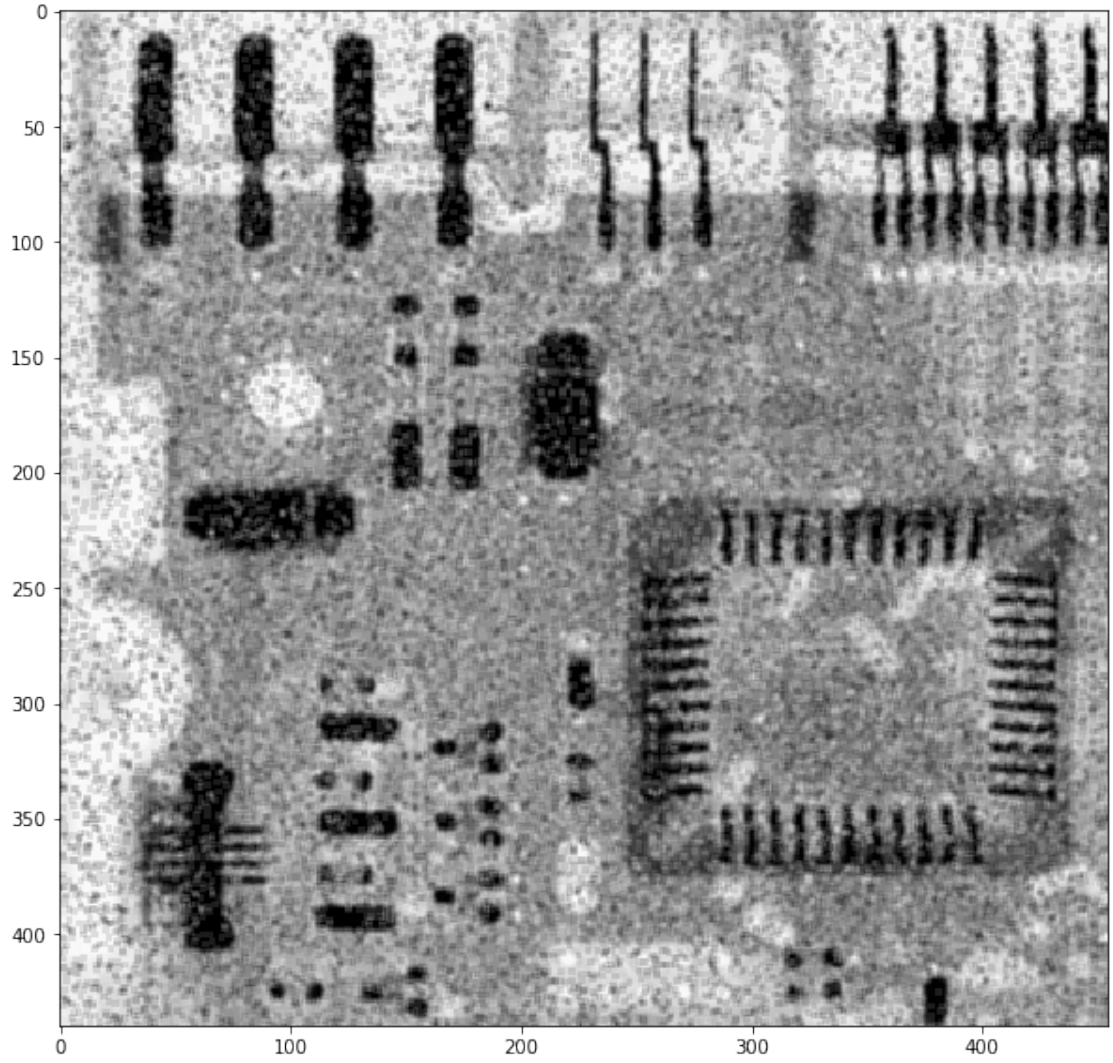
[12]:
```
image_circuit = cv2.imread('image-circuit.jpg')

image_circuit_med = cv2.medianBlur(image_circuit, 3);
plt.imshow(image_circuit_med)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
```



[13]:
```
kernel1 = 1/9 * np.ones((3,3))
image_circuit_blur1 = cv2.filter2D(image_circuit, -1, kernel1)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
plt.imshow(image_circuit_blur1)
```

[13]:
```
<matplotlib.image.AxesImage at 0x7f5d69338b10>
```



1.0.8 Problem 1.5

We are trying to prove the associativity of the convolution. We have that:

$$(f * (g * h))[k] = \sum_{j=-\infty}^{\infty} f[j](g * h)[k-j] = \sum_{j=-\infty}^{\infty} f[j] \sum_{i=-\infty}^{\infty} g[i]h[k-j-i]$$

Now if we make the substitution $u = i + j$.

$$\sum_{u=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[j]g[u-j]h[k-u] = \sum_{u=-\infty}^{\infty} (f * g)[u]h[k-u] = ((f * g) * h)[k]$$

Hence, we have shown associativity.

1.0.9 Problem 1.6

We can combine the two difference formulas to get a formula for the second order difference:

$$\begin{aligned} f''(x) &\approx \frac{f'_F(x) - f'_B(x)}{\Delta x} \\ &= \frac{\frac{f(x+\Delta x)-f(x)}{\Delta x} - \frac{f(x)-f(x-\Delta x)}{\Delta x}}{\Delta x} \\ &= \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2} \end{aligned}$$

This is the second order difference formula we seek.

1.0.10 Problem 1.7

There are 3 options for the first kernel and three options for the secondkernel, so there should be 9 total kernels. The difference between $h_x * h_x$ and $h_x * h_x^P$ should be that the first one will only have one element in any one row where as the second should have three, because h_x^P has a full row of entries. The following cell performs the computations.

[24]:
```
Kernel_hx = 1/2* np.array([[0,  1, 0],
                           [0,  0, 0],
                           [0, -1, 0]])

Kernel_hxP = 1/6 * np.array([[ 1,  1,  1],
                             [ 0,  0,  0],
                             [-1, -1, -1]])


Kernel_hxS = 1/8 * np.array([[ 1,  2,  1],
                             [ 0,  0,  0],
                             [-1, -2, -1]])


hxx = cv2.filter2D(Kernel_hx, -1, Kernel_hx)
print('hx*hx = ')
print(hxx)
print()

hxx2 = cv2.filter2D(Kernel_hx, -1, Kernel_hxP)
print('hx*hx^P = ')
print(hxx2)
print()
```

```
hxx3 = cv2.filter2D(Kernel_hx, -1, Kernel_hxS)
print('hx*hx^S = ')
print(hxx3)
print()
```

```
hx*hx =
[[0.  0.  0. ]
 [0.  0.5 0. ]
 [0.  0.  0. ]]

hx*hx^P =
[[0.         0.         0.        ]
 [0.33333333 0.16666667 0.33333333]
 [0.         0.         0.        ]]

hx*hx^S =
[[0.   0.   0.  ]
 [0.25 0.25 0.25]
 [0.   0.   0.  ]]
```

1.0.11 Problem 1.8

We can take the y partial derivative of the 3-point filter:

$$
\begin{aligned}
f_{yx}(x) &\approx \frac{f_x(x, y+\Delta y) - f_x(x, y-\Delta y)}{2(\Delta y)} \\
&\approx \frac{\frac{f(x+\Delta x, y+\Delta y) - f(x-\Delta x, y+\Delta y)}{2(\Delta x)} - \frac{f(x+\Delta x, y-\Delta y) + f(x-\Delta x, y-\Delta y)}{2(\Delta x)}}{2(\Delta y)} \\
&= \frac{f(x+\Delta x, y+\Delta y) - f(x-\Delta x, y+\Delta y) - f(x+\Delta x, y-\Delta y) + f(x-\Delta x, y-\Delta y)}{4(\Delta x)(\Delta y)}
\end{aligned}
$$

We can see that this translates to a the following matrix:

$$
h_{yx} = \frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}
$$

1.0.12 Problem 1.9

We can show this trivially by taking the partial derivatives w.r.t. u and v and using the chain rule.

$$\frac{\partial F}{\partial u} = \frac{\partial f}{\partial x} cos\,\theta + \frac{\partial f}{\partial y} sin\,\theta$$
$$\frac{\partial^2 F}{\partial u^2} = \frac{\partial^2 f}{\partial x^2} cos^2\,\theta + \frac{\partial^2 f}{\partial y^2} sin^2\,\theta$$

$$\frac{\partial F}{\partial v} = -\frac{\partial f}{\partial x} sin\,\theta + \frac{\partial f}{\partial y} cos\,\theta$$
$$\frac{\partial^2 F}{\partial v^2} = \frac{\partial^2 f}{\partial x^2} sin^2\,\theta + \frac{\partial^2 f}{\partial y^2} cos^2\,\theta$$

$$\begin{aligned}\frac{\partial^2 F}{\partial u^2} + \frac{\partial^2 F}{\partial v^2} &= \frac{\partial^2 f}{\partial x^2}(sin^2\,\theta + cos^2\,\theta) + \frac{\partial^2 f}{\partial y^2}(sin^2\,\theta + cos^2\,\theta) \\ &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\end{aligned}$$

So we see that these are equal, and thus rotation is invariant unter the laplacian.

1.0.13 Problem 1.10

We can manipulate the expression to get a formula for the desired kernel:

$$J(x,y) = I - \nabla^2 I = (\delta - \nabla^2) I = h^{Sh} * I = I * h^{Sh}$$

So we have:

$$h^{Sh} = \delta - \nabla^2$$

Now that we have an expression for the kernel in terms of the delta and the laplacian kernels, both of which we know the formula for.

[14]:
```
delta = np.array([[0, 0, 0],
                  [0, 1, 0],
                  [0, 0, 0]])

laplacian = np.array([[0, 1, 0],
                      [1, -4, 1],
                      [0, 1, 0]])

h_Sh = delta - laplacian


image_moon = cv2.imread('image-moon.jpg')
image_moon_lap = cv2.filter2D(image_moon, -1, h_Sh)

plt.imshow(image_moon_lap)
```

```
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(10, 10)
```