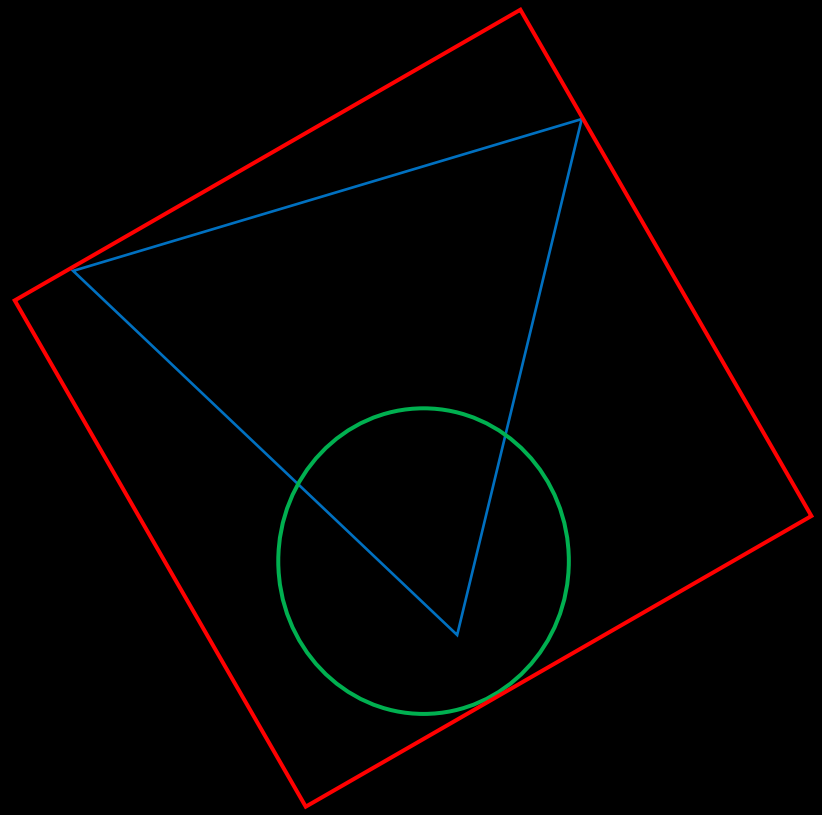
Module 2

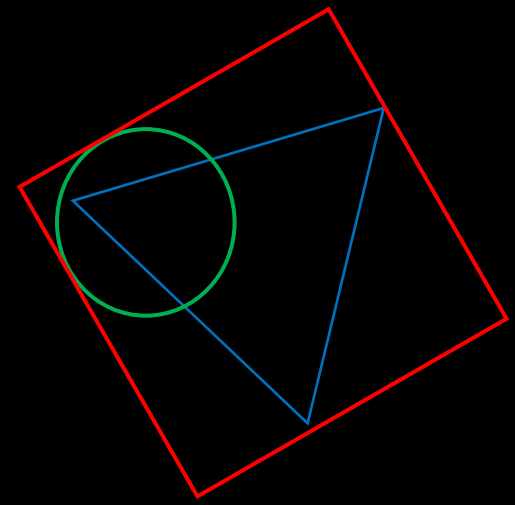
Fourier Transform Model of Image Formation (Part I)

Ron Michael V. Acda
AP 157 WFY-FX2



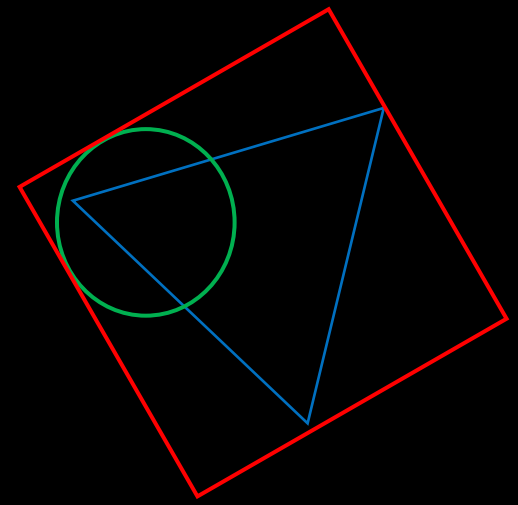
Objectives

1. To understand the numpy implementation 2D Fast Fourier Transform (2D-FFT) in images.
2. To simulate an imaging system by convolution of an image with an aperture.
3. To perform basic template matching using function correlation.

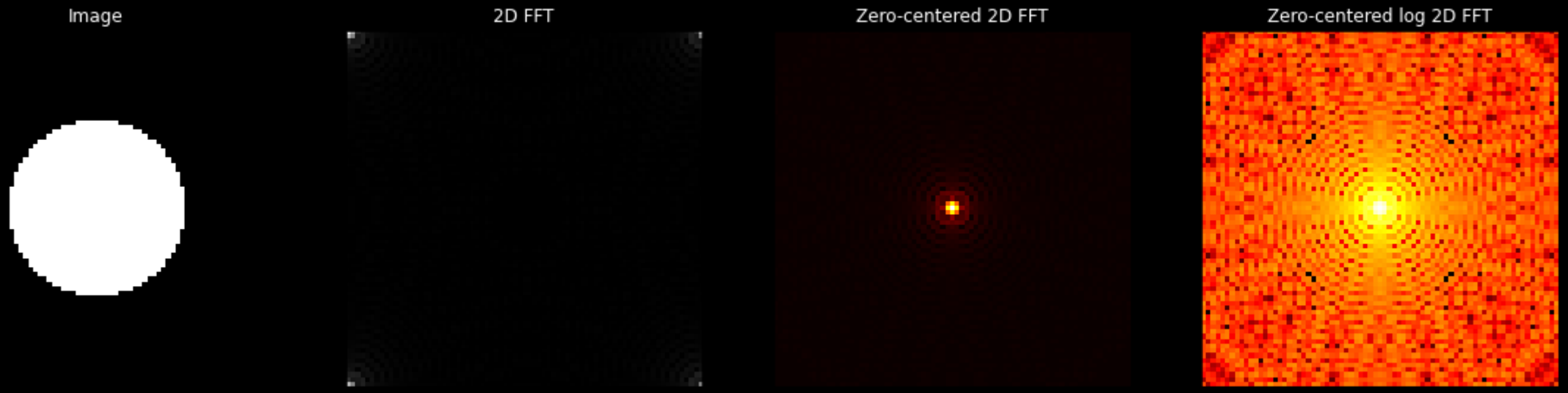


Objectives

1. To understand the numpy implementation 2D Fast Fourier Transform (2D-FFT) in images.
2. To simulate an imaging system by convolution of an image with an aperture.
3. To perform basic template matching using function correlation.



Understanding 2D FFT



The white spots on the corners of the 2D FFT of the image are the zero-frequency components (DC offset). The high frequencies (Nyquist frequencies) are located in the middle of the 2D FFT image.

Shifting the zero-frequency in the middle and the Nyquist frequencies on the edges is customary, producing a zero-centered 2D-FFT image as shown above.

Understanding 2D FFT

```
N = 75 #the higher, the finer. Lower the number to better see the patterns.
x = np.linspace(-1,1,num = N)
y= np.linspace(-1,1,num=N)
X,Y = np.meshgrid(x,y)
R = np.sqrt(X**2 + Y**2) #Setup a circle, centered at the origin.
A = np.zeros(np.shape(R))
A[np.where(R<0.5)] = 1.0 #Circle of radius 0.5

#2D FFT of the circle
FA = np.fft.fft2(A)
abs_FA = abs(FA)

#display as a white circle in 2D coordinates
fig, ax = plt.subplots(1,4,figsize=(20,5))
ax[0].imshow(A, cmap='gray')
ax[0].set_title('Image', color='white')
ax[1].imshow(abs_FA, cmap='gray')
ax[1].set_title('2D FFT', color='white')

#Zero-center the 2D FFT
FAshifted = np.fft.fftshift(FA)
ax[2].imshow(abs(FAshifted), cmap='hot')
ax[2].set_title('Zero-centered 2D FFT', color='white’)

#Contrast stretch the Airy pattern by plotting the log
ax[3].imshow(np.log(abs(FAshifted)), cmap='hot')
ax[3].set_title('Zero-centered log 2D FFT', color='white')
```

Understanding 2D FFT

Image

2D FFT

The Fourier coefficients are plotted in the 2D FFT image, which are given by:

$$A_{kl} = \sum_{r=0}^{R-1} \sum_{c=0}^{C-1} a_{rc} \exp\left[-2\pi i \left(\frac{mk}{M} + \frac{nl}{N}\right)\right]$$

Where $k = 0,1, \dots, R-1,\ l = 0,1, \dots, C-1$. The output of the 2D FFT are complex coefficients.

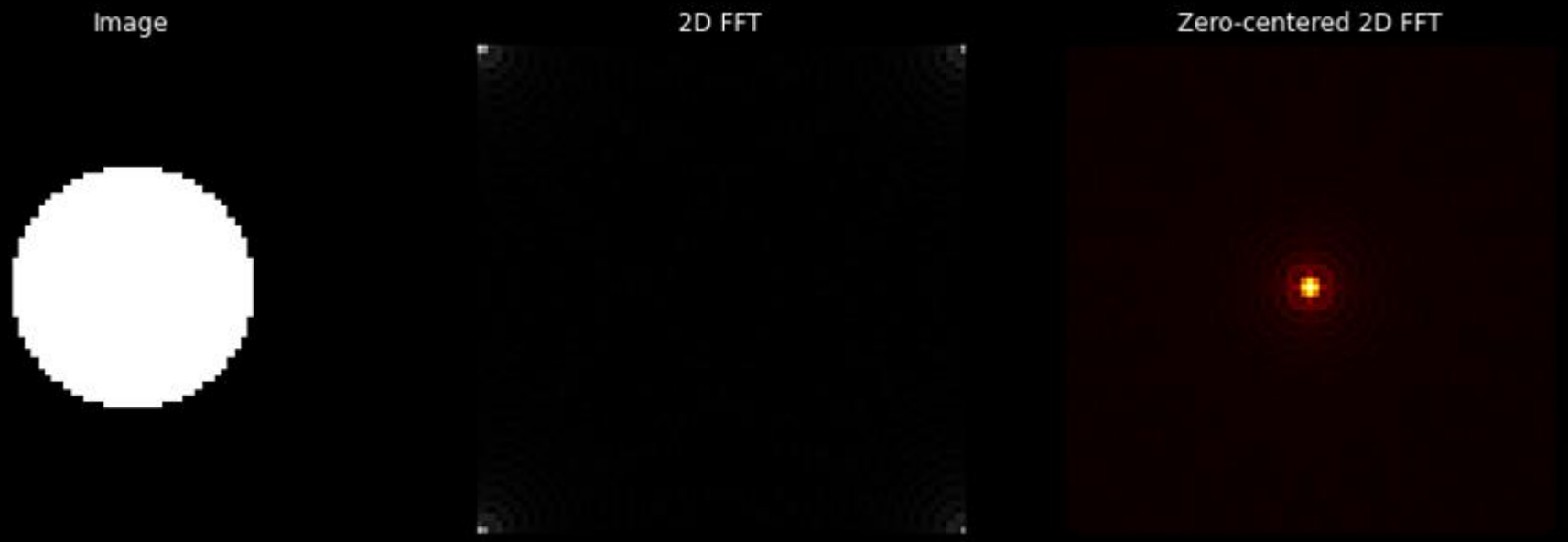
Essentially, we perform a 1D FFT on the columns first, then perform 1D FFT on the rows next. To show the plot, we need to take the modulus of the fft array because the Fourier coefficients are complex numbers.

Recall that in a 1D FFT, the frequency plot is symmetric about the middle, and the zero-frequency bins (with high amplitude) are located on the sides of the plot.

This explains why the low-frequency, high-amplitude components are located on the corners upon performing 2D FFT on a highly-symmetric image such as the circle.

Understanding 2D FFT

By performing np.fft.fftshift() on the 2D FFT image, we can center the low-frequency components in the middle of the plot. Note that it looks like an Airy diffraction pattern.



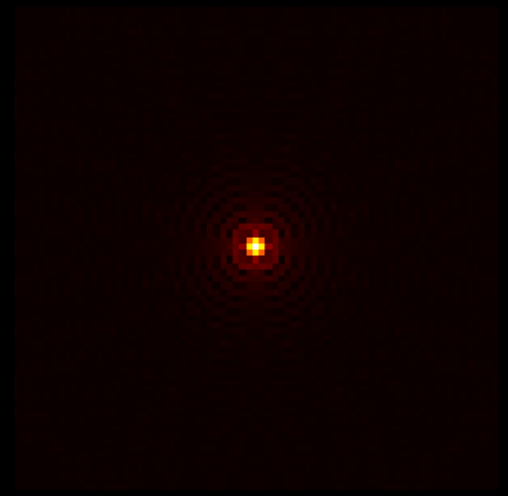
Understanding 2D FFT

We can plot the logarithm of the original zero-centered FFT image (or apply contrast stretching methods from the previous module) to see more detail.

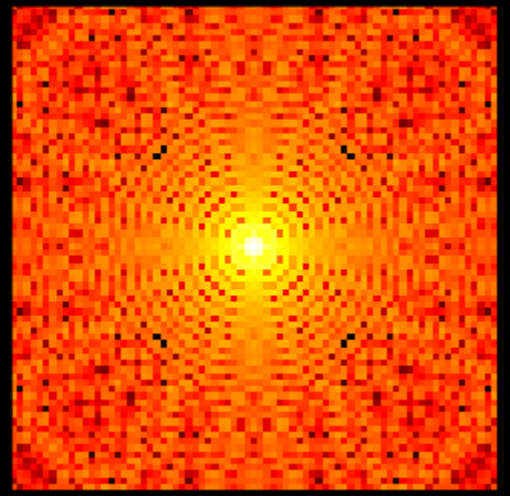
Image

2D FFT

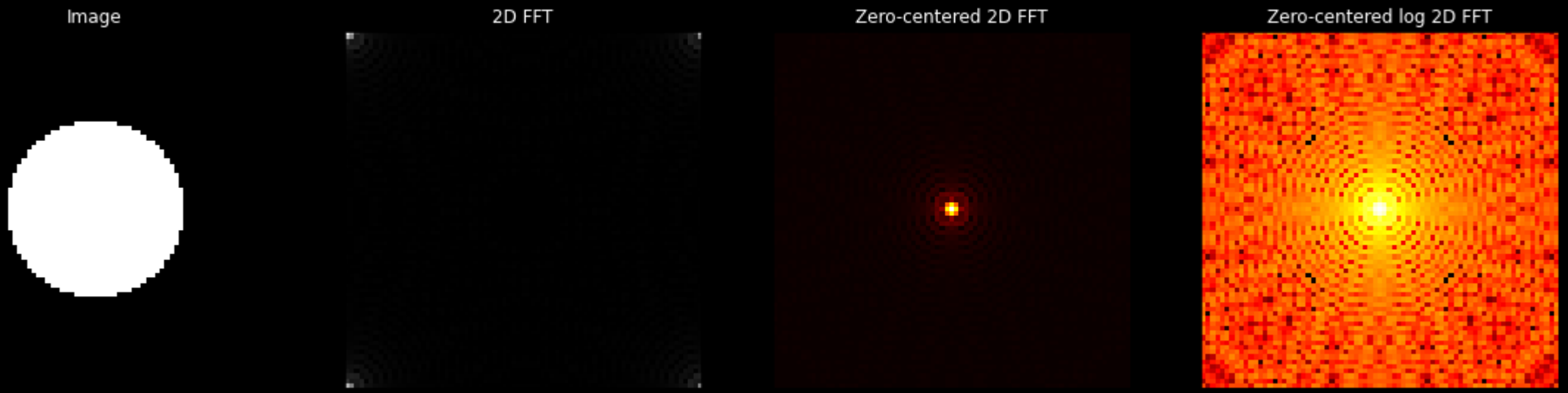
Zero-centered 2D FFT



Zero-centered log 2D FFT

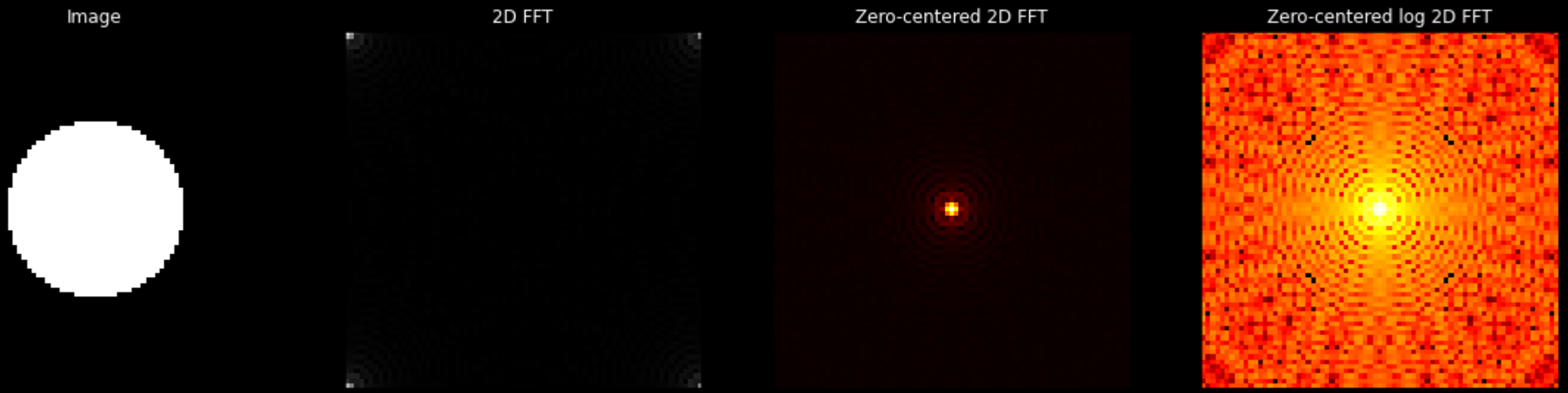


Understanding 2D FFT



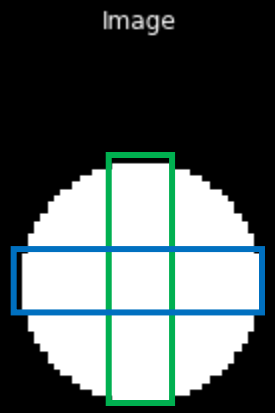
Note the dominance of the low-frequency components. This means that the Fourier series expansion of the 2D circular image is dominated by 2D sinusoids with low-frequency terms. Thus, if we were to compress the image, all we need to do is take a sample of Fourier coefficients near the center, and perform inverse FFT in the image (np.fft.ifft2()).

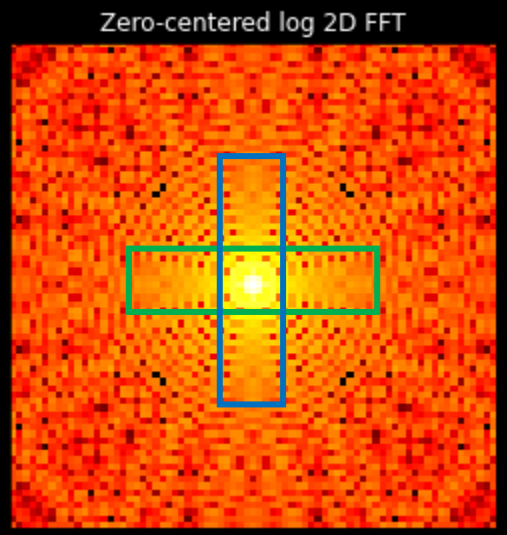
Understanding 2D FFT

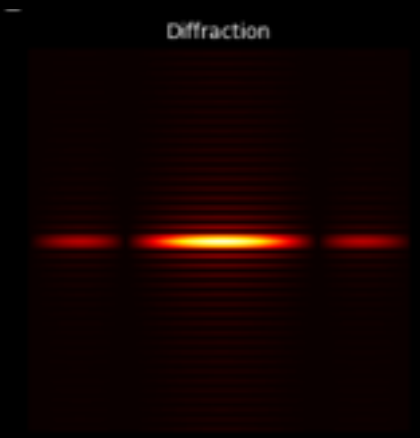


Another interesting feature is the presence of a somewhat high-amplitude “cross” near the center. This can be attributed to the four flat edges of the circle in the image.

Understanding 2D FFT







Here, I provide a qualitative and approximate justification for why the cross pattern appears. I've indicated the diametrically opposite, flat edges of the image. These colored rectangles can be treated as rectangular slits. Their contribution to the 2D FFT is the "diffraction patterns" we observe for single slits, producing a cross-shaped pattern.

Understanding 2D FFT

This provides an avenue for analyzing the features of the 2D FFT. Mathematically, an $m \times n$ array a can be expressed as a sum of $m \times n$ "subarrays" a_i (each with a certain "feature" such as the rectangular slit on the previous slide). If A is the FT of the array a, and A is the FT of the subarray a_i, then it follows from the linearity of FT that:

$$a = \sum_{i}^{N} a_i \Rightarrow A = \sum_{i}^{N} A_i$$

where N is the number of "features" that need to be analyzed. In layman's terms, we can analyze the 2D FFT image "piece-wise".

Understanding FFT

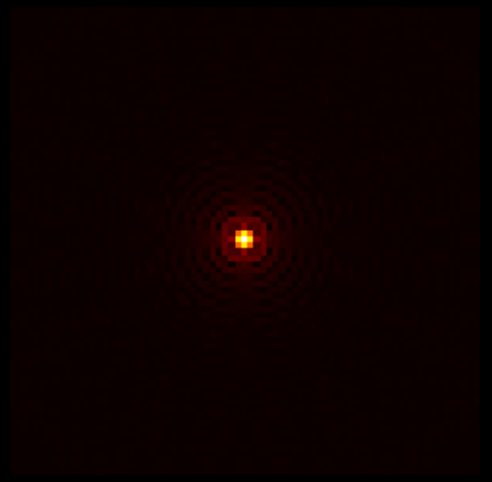
Image

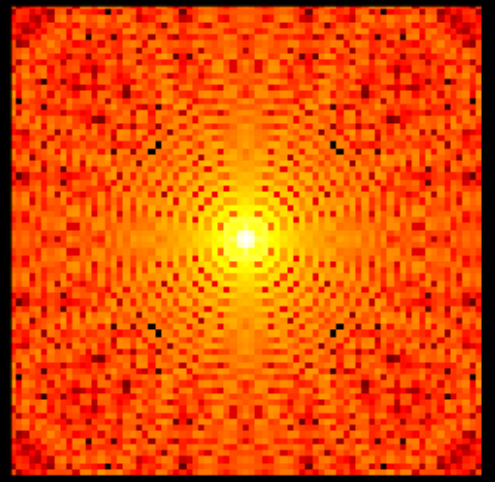
2D FFT

Zero-centered 2D FFT

Zero-centered log 2D FFT

75x75





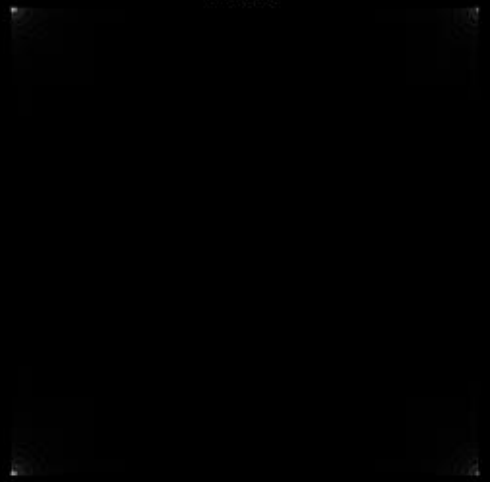
Image

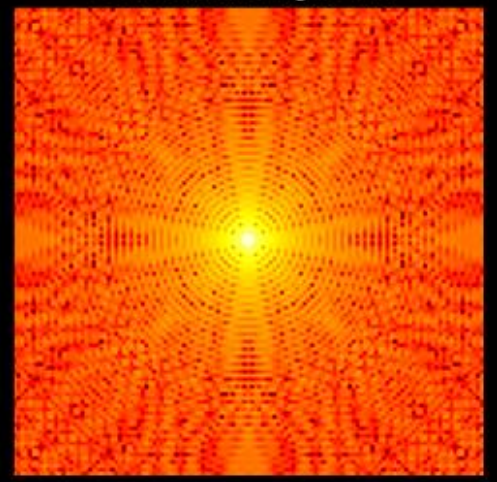
2D FFT

Zero-centered 2D FFT

Zero-centered log 2D FFT

125x125



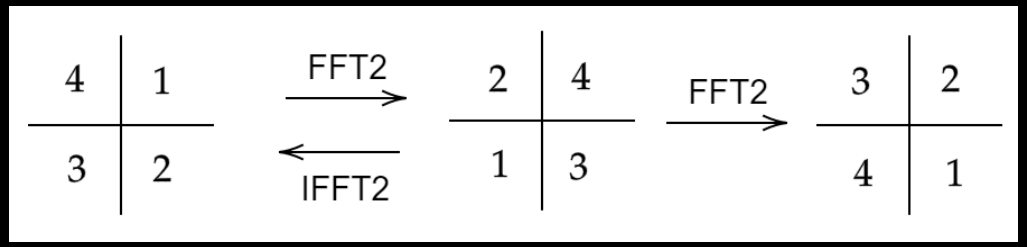


Understanding 2D FFT

Note that increasing the number of pixels (without reducing the relative diameter of the aperture) reduces the overall granularity or noise of the FFT images. It can be seen that the "intensity" maximum appears to shrink as the number of pixels increases. One possible explanation for this is since the relative size of the aperture remains unchanged, the "spread" of the low-frequency Fourier coefficients remains unchanged. Since the overall plot size increases, this constant "spread" would appear smaller. If we zoom in; however, the granularity of the 2D FFT is reduced.

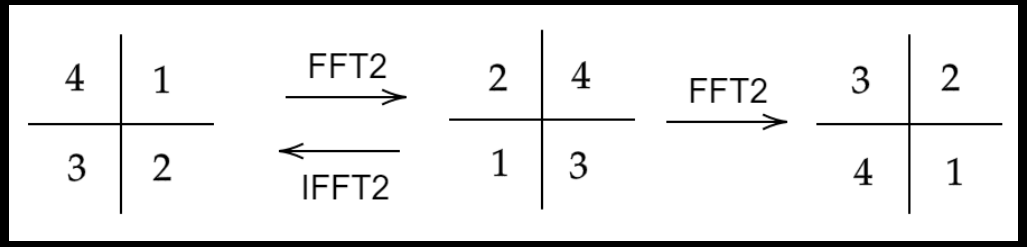
Understanding FFT

By performing np.fft.ifft2() on the 2D FFT of the original image, the original image is reconstructed without inversion or flipping of axes. Mathematically, performing Fourier transform twice should give the original function, but the FFT implementation is quite more subtle. 2D FFT interchanges the quadrants of the array along the diagonal, so that:



Understanding FFT

By performing np.fft.ifft2() on the 2D FFT of the original image, the original image is reconstructed without inversion or flipping of axes. Mathematically, performing Fourier transform twice should give the original function, but the FFT implementation is quite more subtle. 2D FFT interchanges the quadrants of the array along the diagonal, so that:



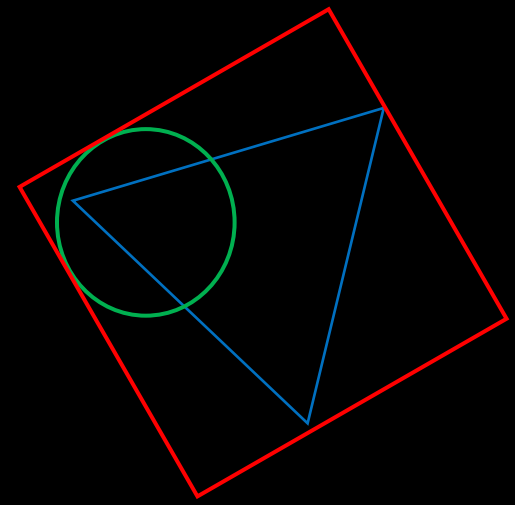
Note that performing FFT2 twice on the image flips the image about the horizontal axis.

Understanding FFT

```
fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].imshow(mr_incredible, cmap='gray')
ax[0].set_title('Original', color='white')


#Reconstruct the image from the FFT using ifft2 (inverse FFT)
rec= np.fft.ifft2(np.fft.fft2(mr_incredible))

ax[1].imshow(abs(rec), cmap='gray')
ax[1].set_title('Reconstruction (ifft2)', color='white')

#Reconstruct image by applying fft2 twice
FA = np.fft.fft2(mr_incredible)
rec1 = np.fft.fft2(FA)
ax[2].imshow(abs(rec1), cmap='gray')
ax[2].set_title('Reconstruction (two fft2)', color='white')
```

Objectives

1. To understand the numpy implementation 2D Fast Fourier Transform (2D-FFT) in images.
2. To simulate an imaging system by convolution of an image with an aperture.
3. To perform basic template matching using function correlation.



2D Image Convolution

Suppose f is the image "array" and g is the convolving kernel. Mathematically, a 2D convolution is defined as

$$h(x,y) = f \odot g = \iint f(x,y)g(x-x', y-y')dx'dy'$$

where, of course, the integral turns into a double sum since the arrays are discrete. This operation can be done by brute force, but it can be mathematically shown that convolution is just multiplication in Fourier space; i.e.,

$$H = FG$$

where H, F, G are the Fourier transforms of the convolved image, the original image, and the kernel, respectively.

2D Image Convolution

Suppose f is the image "array" and g is the convolving kernel. Mathematically, a 2D convolution is defined as

$$h(x,y) = f \odot g = \iint f(x,y)g(x-x', y-y')dx'dy'$$

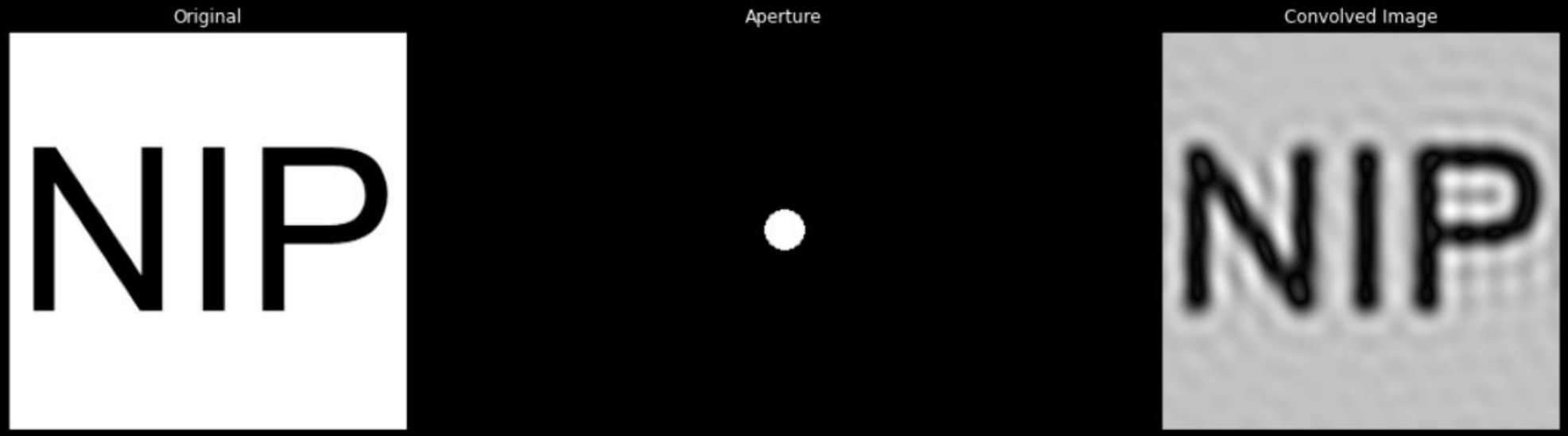
where, of course, the integral turns into a double sum since the arrays are discrete. This operation can be done by brute force, but it can be mathematically shown that convolution is just multiplication in Fourier space; i.e.,

$$H = FG$$

where H, F, G are the Fourier transforms of the convolved image, the original image, and the kernel, respectively.

However, it is important to realize that since F and G are arrays (matrices), multiplication here means "element-wise" multiplication, and not the standard matrix multiplication we use in linear algebra.

Image Convolution with Circular Aperture



Here, we can simulate what the original image would look like when viewed through a circular aperture. To perform this operation, we take the FFT of the original image (without taking the magnitude), multiply element-wise with the aperture array, and perform ifft2() on the resulting array. Note that F = fft of the original image, G is the aperture, and ifft2(FG) is the deconvolved image.

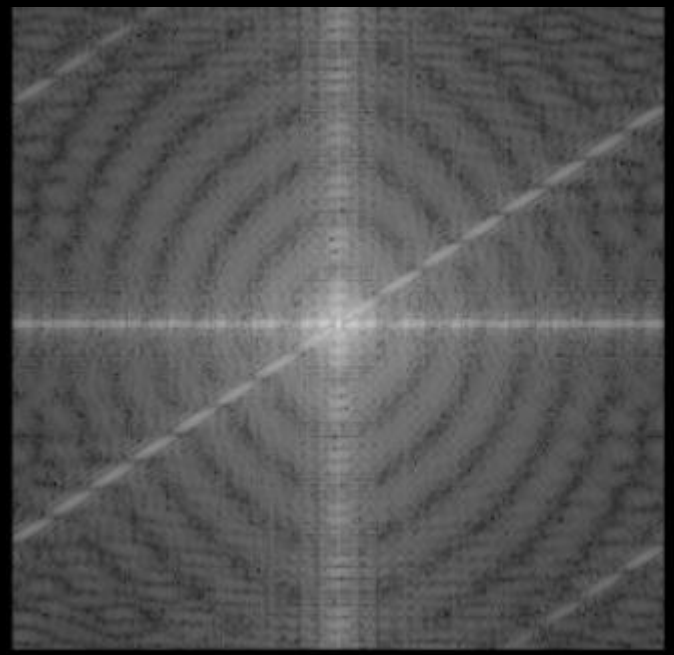
Note that the resulting deconvolved image looks like the diffraction patterns we can see, for example, in electron microscopy. Thus, digital image convolution can be used to simulate imaging systems.

Image Convolution with Circular Aperture

```
def circle_aperture(D, N=256):
    x = np.linspace(-1,1,num = N)
    y= np.linspace(-1,1,num=N)
    x= y
    radius = (D*2)/2
    X,Y = np.meshgrid(x,y)
    R = np.sqrt(X**2 + Y**2) #Setup a circle, centered at the origin.
    A = np.zeros(np.shape(R))
    A[np.where(R<radius)] = 1.0
    return A

NIP = cv2.imread(pic2, 0)
aperture = np.fft.fftshift(circle_aperture(D=0.1))
conv = np.fft.fft2(NIP)*aperture
deconv = np.fft.ifft2(conv)
NIP_fft = np.fft.fftshift(np.fft.fft2(NIP))

fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].imshow(NIP, cmap='gray')
ax[0].set_title('Original', color='white')
ax[1].imshow(circle_aperture(D=0.1), cmap='gray')
ax[1].set_title('Aperture', color='white')
ax[2].imshow(abs(deconv), cmap='gray')
ax[2].set_title('Convolved Image', color='white')
```

Image Convolution with Circular Aperture

Image FFT (contrast stretched)



Aperture

FFT of the
"compressed" image

In this case, the convolution of the original image can be also thought of as digital image compression. For a circular aperture G, the convolution "crops" or filters out only the FFT frequencies within the circle. In this case, the circle is in the center, so the convolution selects only the low frequencies. The image is reconstructed using a subset of the 2D FFT of the original image.

Image Convolution with Circular Aperture

Original

Aperture

Convolved Image

This is also why the circle needs to be centered, as the Fourier expansion of the image is dominated by low frequencies (as evidenced by the high amplitudes). If the circular aperture were off-center, the reconstructed image would look nowhere like the original image.

Image Convolution with Circular Aperture

```
def circle_aperture_offcenter(D, N=256):
    x = np.linspace(-1,1,num = N)
    y= np.linspace(-1,1,num=N)
    x= y
    radius = (D*2)/2
    X,Y = np.meshgrid(x,y)
    R = np.sqrt((X-0.5)**2 + (Y-0.5)**2) #Setup an offcenter circle.
    A = np.zeros(np.shape(R))
    A[np.where(R<radius)] = 1.0
    return A

NIP = cv2.imread(pic2, 0)
aperture = np.fft.fftshift(circle_aperture_offcenter(D=0.2))
conv = np.fft.fft2(NIP)*aperture
deconv = np.fft.ifft2(conv)
NIP_fft = np.fft.fftshift(np.fft.fft2(NIP))

fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].imshow(NIP, cmap='gray')
ax[0].set_title('Original', color='white')
ax[1].imshow(circle_aperture_offcenter(D=0.2), cmap='gray')
ax[1].set_title('Aperture', color='white')
ax[2].imshow(abs(deconv), cmap='gray')
ax[2].set_title('Convolved Image', color='white')
```

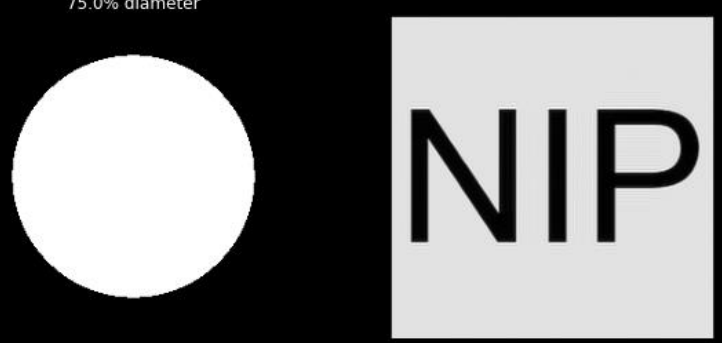
Image Convolution with Circular Aperture

10.0% diameter

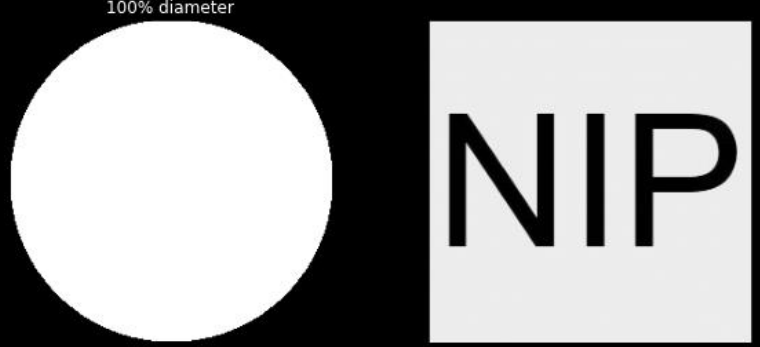
25.0% diameter

50.0% diameter

75.0% diameter



100% diameter



When viewed with an increasingly large aperture, more Fourier coefficients are used, resulting to an image that increasingly looks like the original.

Image Convolution with Circular Aperture

```
diams = [0.1, .25, .50, .75, 1]
fig, ax = plt.subplots(len(diams), 2, figsize=(10, 5*len(diams)))
ax = ax.flatten()
NIP_fft = np.fft.fft2(NIP)
for i in range(len(diams)):
    circle = circle_aperture(N=np.shape(NIP)[0], D=diams[i])
    aperture = np.fft.fftshift(circle)
    conv = NIP_fft*aperture
    deconv = np.fft.ifft2(conv)
    ax[2*i].imshow(circle, cmap='gray')
    ax[2*i].set_title('{percent}% diameter'.format(percent=diams[i]*100),
color='white')
    ax[2*i].set_xticks([])
    ax[2*i].set_yticks([])
    ax[2*i+1].imshow(abs(deconv), cmap='gray')
    ax[2*i+1].set_title('Image')
    ax[2*i+1].set_xticks([])
    ax[2*i+1].set_yticks([])
```

Image Convolution with Circular Aperture

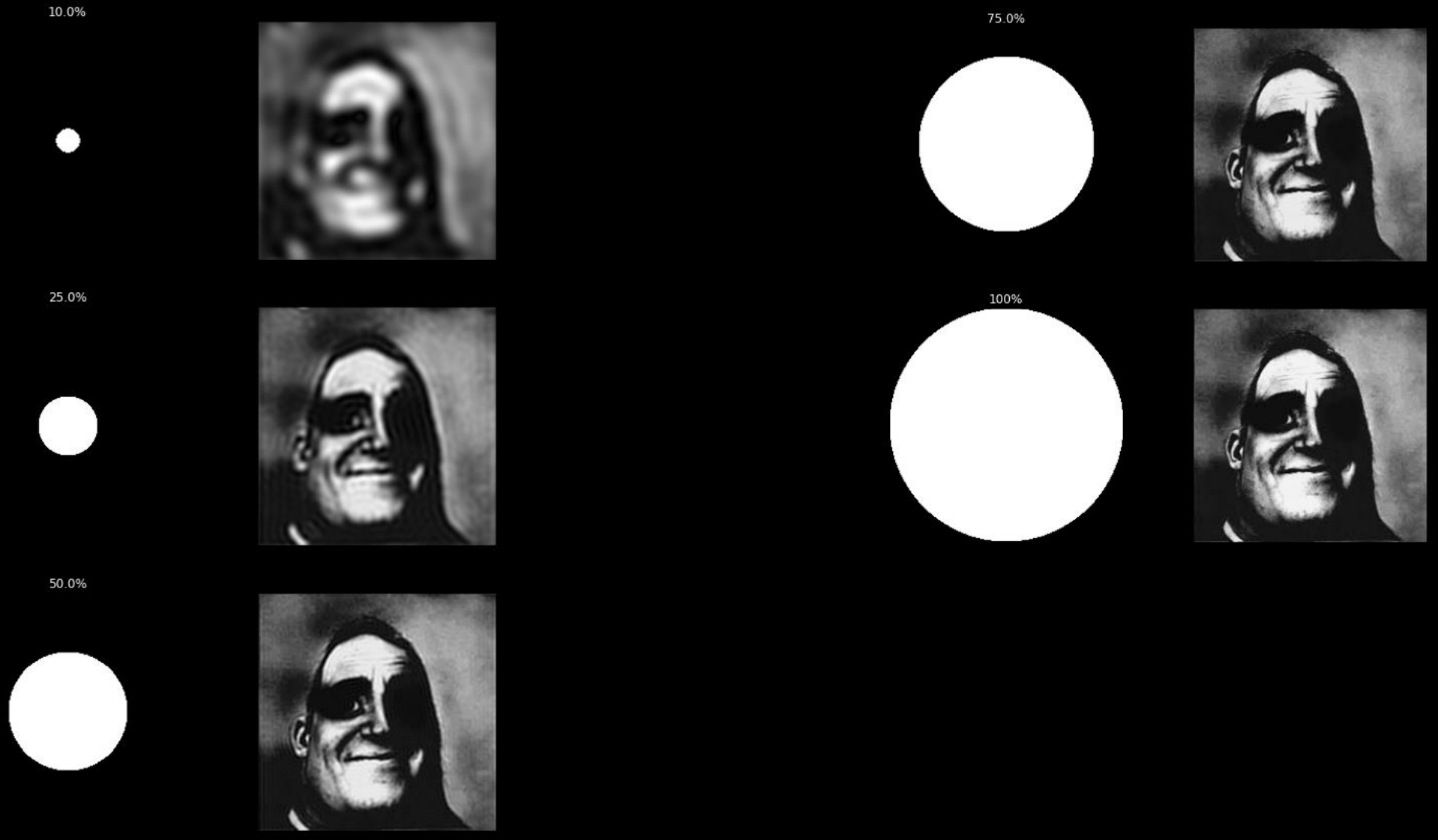


Image Convolution with Polygonal Aperture*

I also experimented with various polygonal apertures and their effects on the reconstructed image.

3-gon

Image

4-gon

Image

5-gon

Image

6-gon

Image

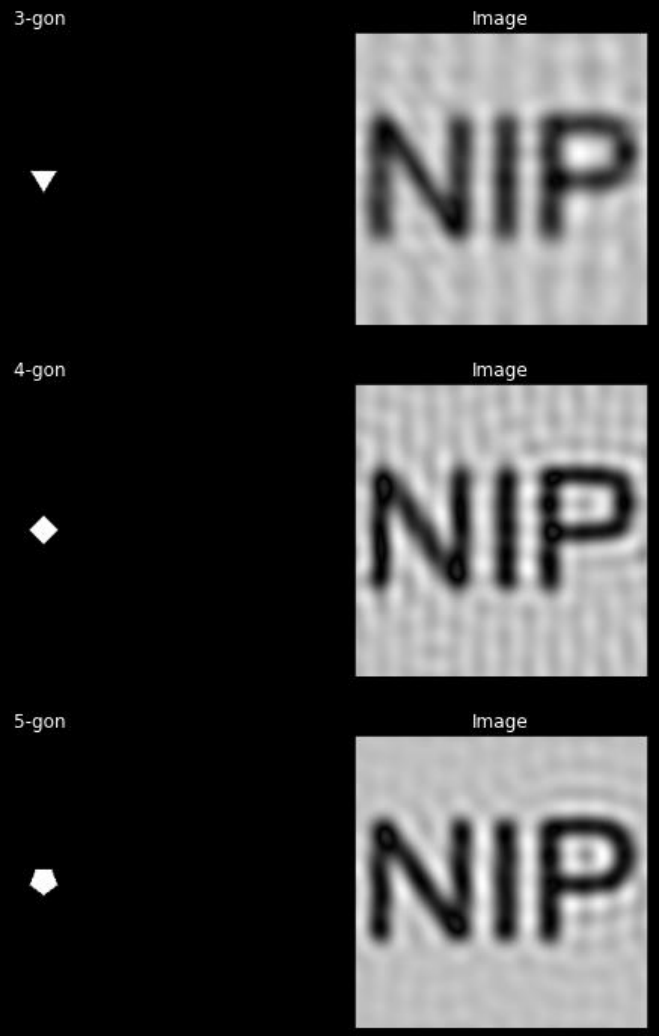
7-gon

Image

8-gon

Image

Image Convolution with Polygonal Aperture*



One possible use I can see with these polygonal apertures is when the 2D FFT of the original image has an n-fold (polygonal) symmetry at the center. However, I have yet to see an image with such a unique 2D FFT. Nevertheless, they are interesting to look at.

Image Convolution with Polygonal Aperture*

Code for creating polygonal aperture.

```
from skimage.draw import polygon

def vertices(sides, r):
    vertices = np.array([[r*np.cos(2*np.pi*i/sides),
r*np.sin(2*np.pi*i/sides)] for i in range(sides+1)])
    return vertices

def polygon_aperture(sides, r, N=200):
    img = np.zeros((N,N))
    v = np.array(vertices(sides, r)*N, dtype='int') + int(N/2)
    rr, cc = polygon(v[:,0], v[:,1], img.shape)
    img[rr,cc] = 1
    return img
```

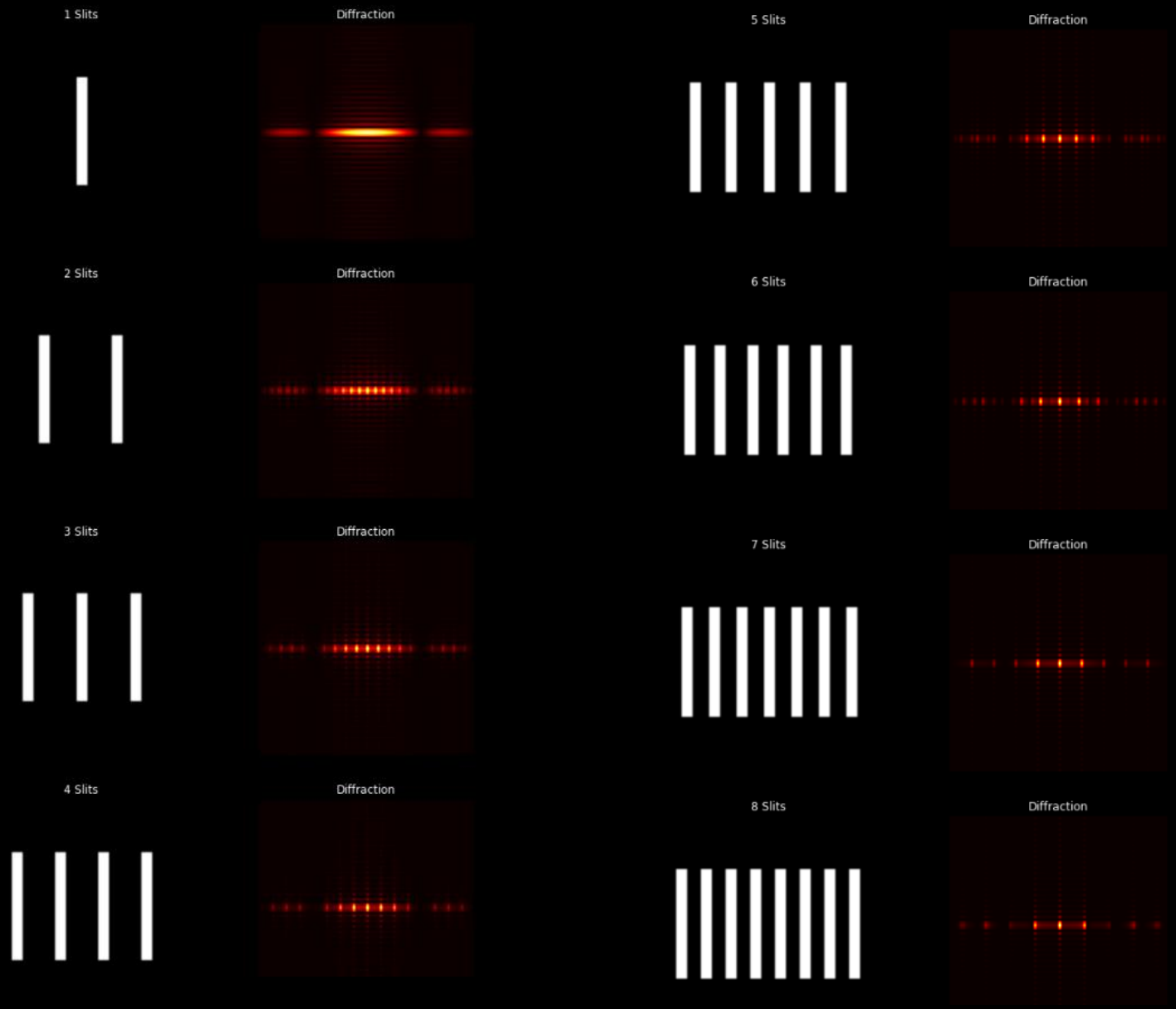
Convolution vis-à-vis Fraunhofer (Far-Field) Diffraction

Here, I try to establish a connection between image convolution and Fraunhofer diffraction. By making this connection, we can perhaps see a bit more mathematically why diffraction patterns are produced when performing 2D FFT.

1. In Fraunhofer diffraction, the light source is far enough so that the incident light rays to an aperture are parallel. This is a good approximation for distant point light sources.
2. This incident light rays can be thought of as an array of 1's in Fourier space. In the convolution formula, this is represented by F.
3. The aperture is represented by G.
4. Since $F = 1$, $H = G$, so performing ifft2 on H is equivalent to performing ifft2 on G. Since the aperture is already in the Fourier space, this is also equivalent to performing an fft2 on the aperture. Hence, the resulting image is what the distant point light source would look like under Fraunhofer diffraction.

Moral: FFT2 can be used to simulate diffraction.

Fraunhofer Diffraction Pattern of N-Slits




```
1 Slit Amplitude:160.0
2 Slit Amplitude:320.0
3 Slit Amplitude:480.0
4 Slit Amplitude:640.0
5 Slit Amplitude:800.0
6 Slit Amplitude:960.0
7 Slit Amplitude:1120.0
8 Slit Amplitude:1280.0
9 Slit Amplitude:1440.0
10 Slit Amplitude:1600.0
```

The simulation of diffraction using FFT2 agrees with one prediction of diffraction theory; that is, as the number of slit increases, the amplitude of the bright fringes increases. That is,

$$A_n = nA_1$$

Where A_n is the amplitude at n slits.

Fraunhofer Diffraction Pattern of N-Slits

Code for creating N vertical slits

```
def slit(slits=1, width = 0.05, N=256):
    y = np.linspace(0,N, N+1)
    x = y
    X,Y = np.meshgrid(x,y)
    vals = np.zeros((N,N))
    cen = np.array([(1/2, (i+1)/(slits+1)) for i in range(slits)])
    cen_trans = np.array(cen*N, dtype='int') #index of the rectangle center, translated
for a N x N square array
    for c in cen_trans: #code each vertical slit
        low_y = int(c[1] - np.round(width*N/2))
        high_y = int(c[1] + np.round(width*N/2))
        left_x = int(c[0] - np.round(N/4))
        right_x = int(c[0] + np.round(N/4))
        vals[left_x : right_x, low_y : high_y] = 1
    return vals
```

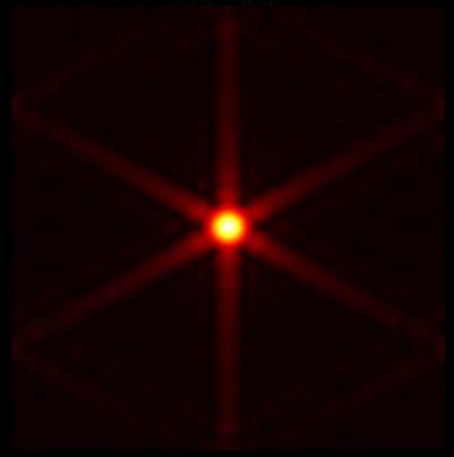
Fraunhofer Diffraction Pattern of N-Slits

```
slits = [i+1 for i in range(10)]
fig, ax = plt.subplots(len(slits), 2, figsize=(10, 5*len(slits)))
ax = ax.flatten()
for i in range(len(slits)):
    my_slit = slit(slits[i], width = 0.05, N=80)
    fft_slit= np.fft.fftshift(np.fft.fft2(my_slit))
    abs_fftslit = abs(fft_slit)
    print(str(i+1)+' Slit Amplitude:' + str(np.max(abs_fftslit)))
    ax[2*i].imshow(my_slit, cmap='gray')
    ax[2*i].set_title('{s} Slits'.format(s=slits[i]), color='white')
    ax[2*i].set_xticks([])
    ax[2*i].set_yticks([])
    ax[2*i+1].imshow(abs_fftslit, cmap='hot')
    ax[2*i+1].set_title('Diffraction', color='white')
    ax[2*i+1].set_xticks([])
    ax[2*i+1].set_yticks([])
```

Fraunhofer Diffraction Pattern of Regular Polygons*

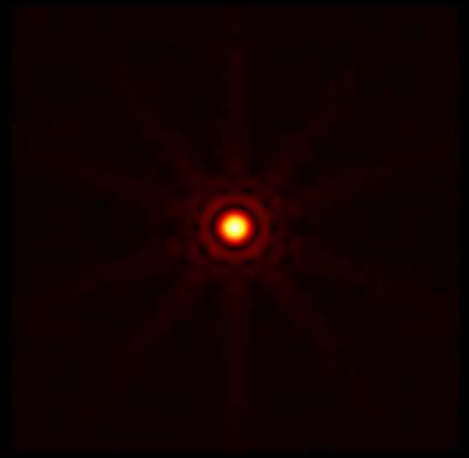
3-gon

Diffraction



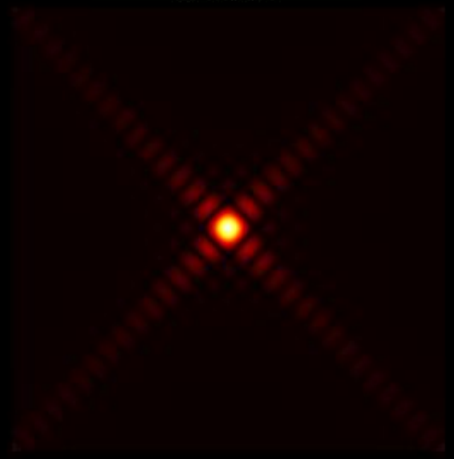
5-gon

Diffraction



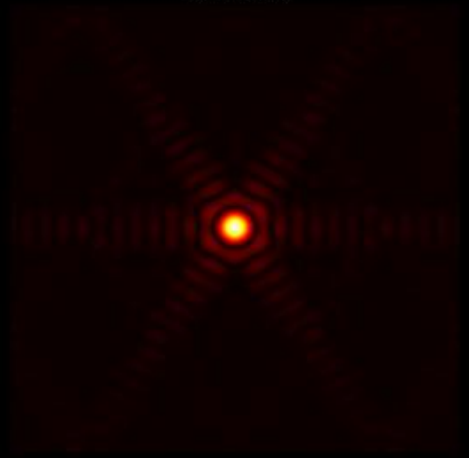
4-gon

Diffraction



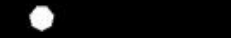
6-gon

Diffraction

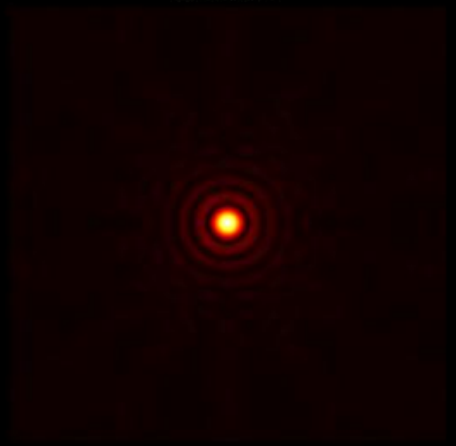


Fraunhofer Diffraction Pattern of Regular Polygons*

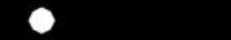
7-gon



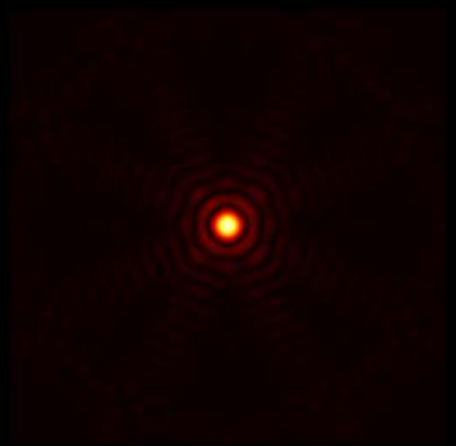
Diffraction



8-gon

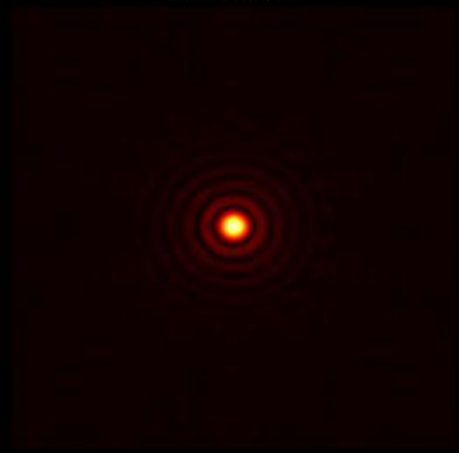


Diffraction



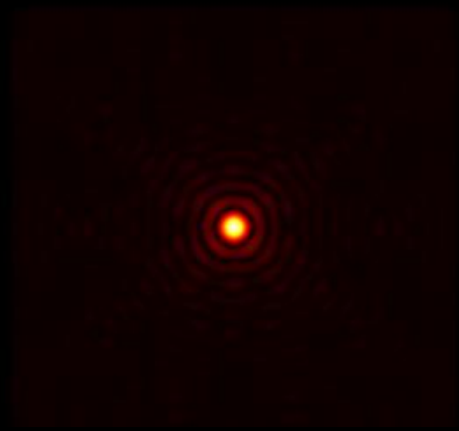
9-gon

Diffraction



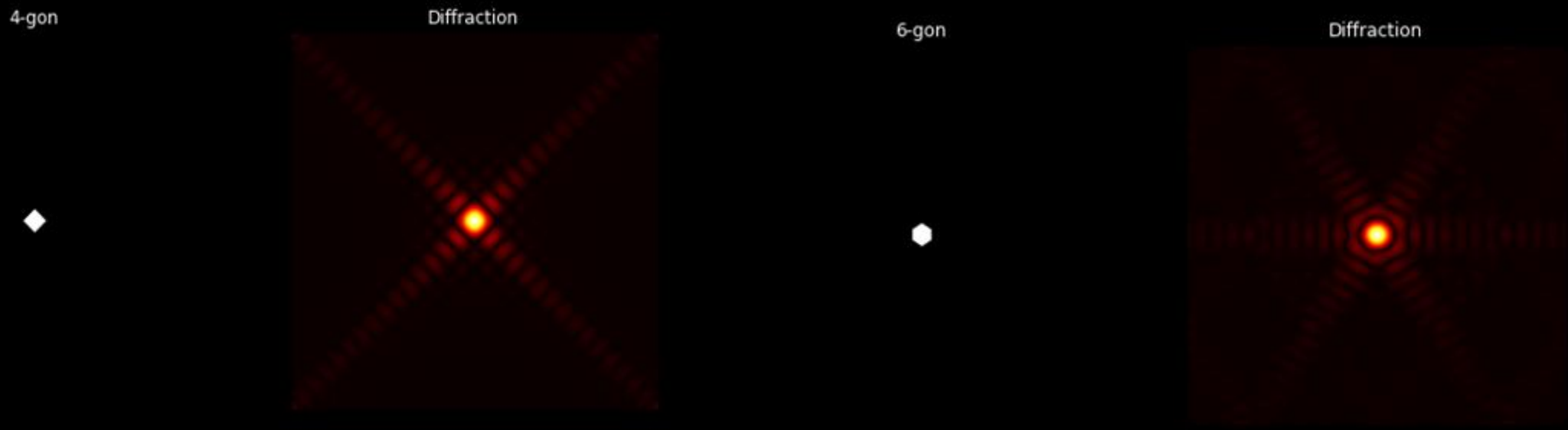
10-gon

Diffraction



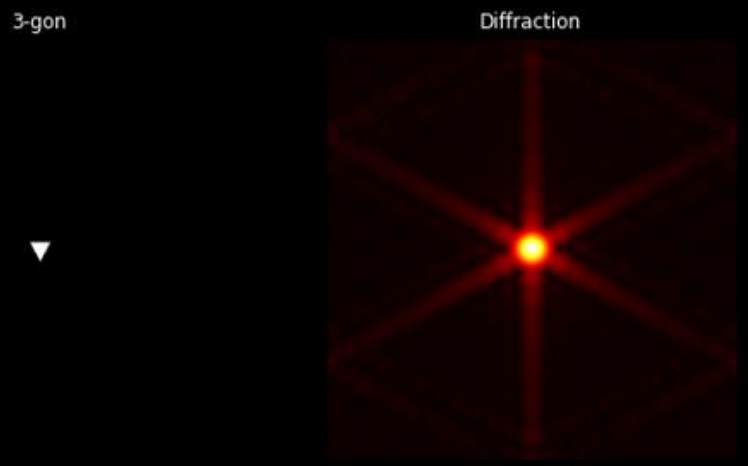
Fraunhofer Diffraction Pattern of Regular Polygons*

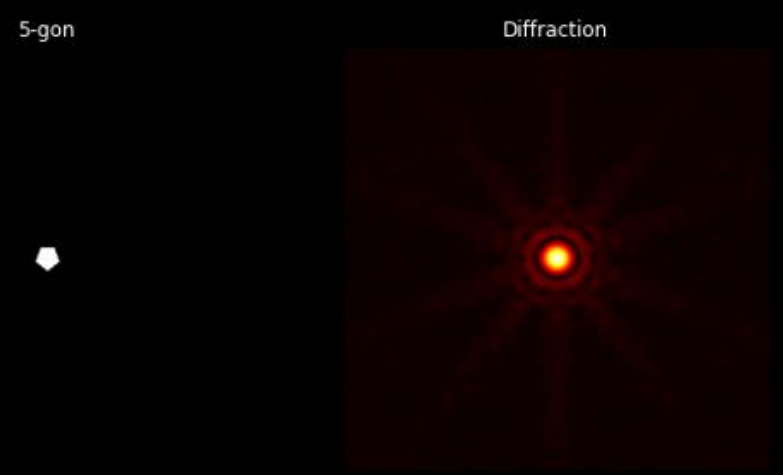
The diffraction pattern for even-sided polygons can be partially explained using the "superimposition" of rectangular-slit features as I've done previously. In such polygons, each side has a diametrically opposite side that together forms a "rectangular slit" e.g. 2 rectangular slits in a square, 3 slits in a hexagon, and so on.



Fraunhofer Diffraction Pattern of Regular Polygons*

For odd-sided polygons, the patterns are a tad bit more complicated. However, they agree quite well with the theoretical result by J. Kormska (1973).



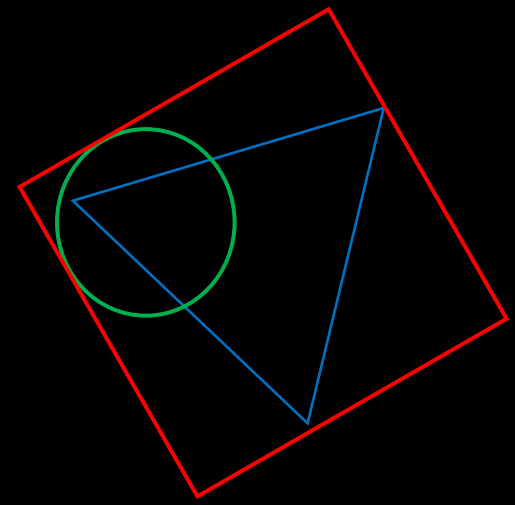


Fraunhofer Diffraction Pattern of Regular Polygons*

```
ngons = [i+3 for i in range(8)]
fig, ax = plt.subplots(len(ngons), 2, figsize=(10, 5*len(slits)))
ax = ax.flatten()
for i in range(len(ngons)):
    my_ngon = polygon_aperture(i+3, r=0.03, N=500)
    fft_ngon= np.fft.fftshift(np.fft.fft2(my_ngon))
    ax[2*i].imshow(my_ngon, cmap='gray')
    ax[2*i].set_title('{n}-gon'.format(n=ngons[i]), color='white')
    ax[2*i].set_xticks([])
    ax[2*i].set_yticks([])
    ax[2*i+1].imshow(abs(fft_ngon), cmap='hot')
    ax[2*i+1].set_title('Diffraction', color='white')
    ax[2*i+1].set_xticks([])
    ax[2*i+1].set_yticks([])
```

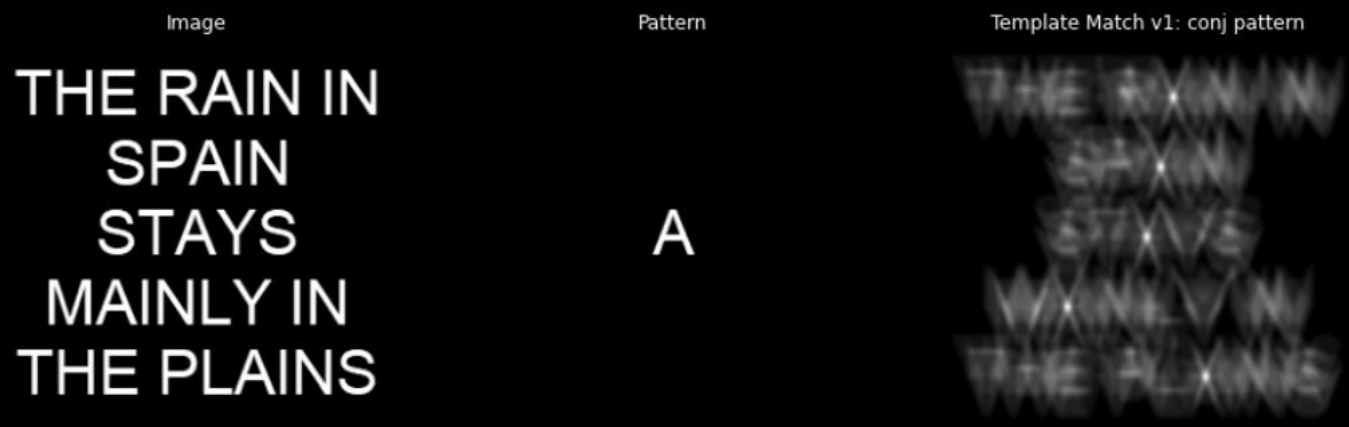
Objectives

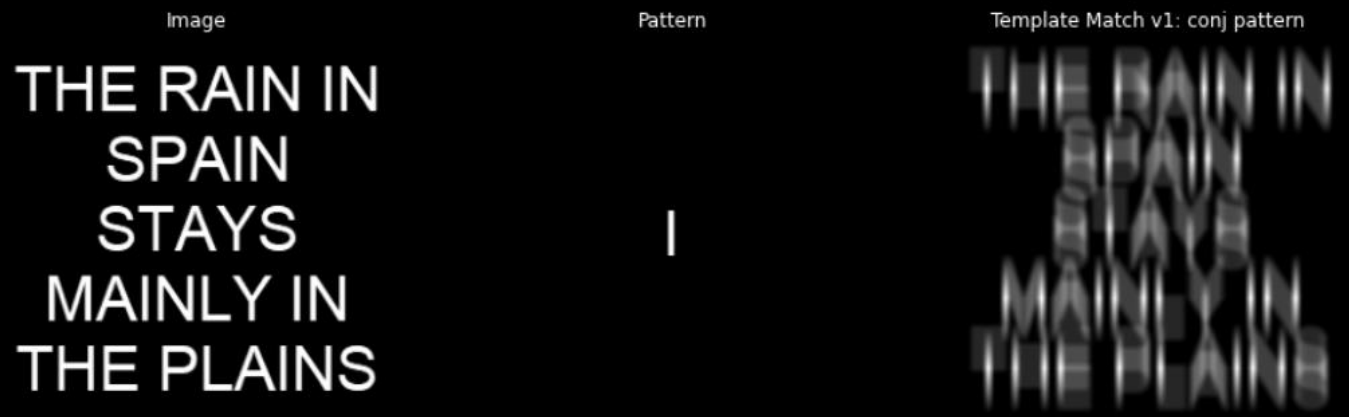
1. To understand the numpy implementation 2D Fast Fourier Transform (2D-FFT) in images.
2. To simulate an imaging system by convolution of an image with an aperture.
3. To perform basic template matching using function correlation.



Template Matching

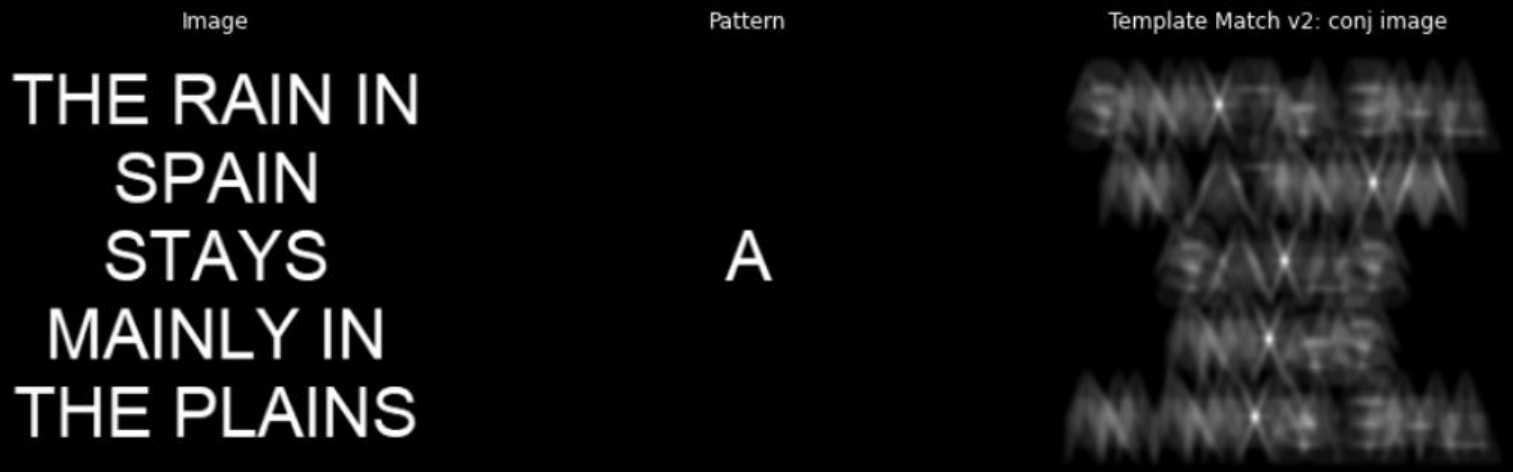
The theory behind template matching involves the use of a correlation function and taking the conjugate of the FFT of either one of the image I or the template/pattern P. I was curious about the effect of this conjugation, so I made 4 versions of this algorithm. Here, I present the result where the FFT of the pattern/template is conjugated.

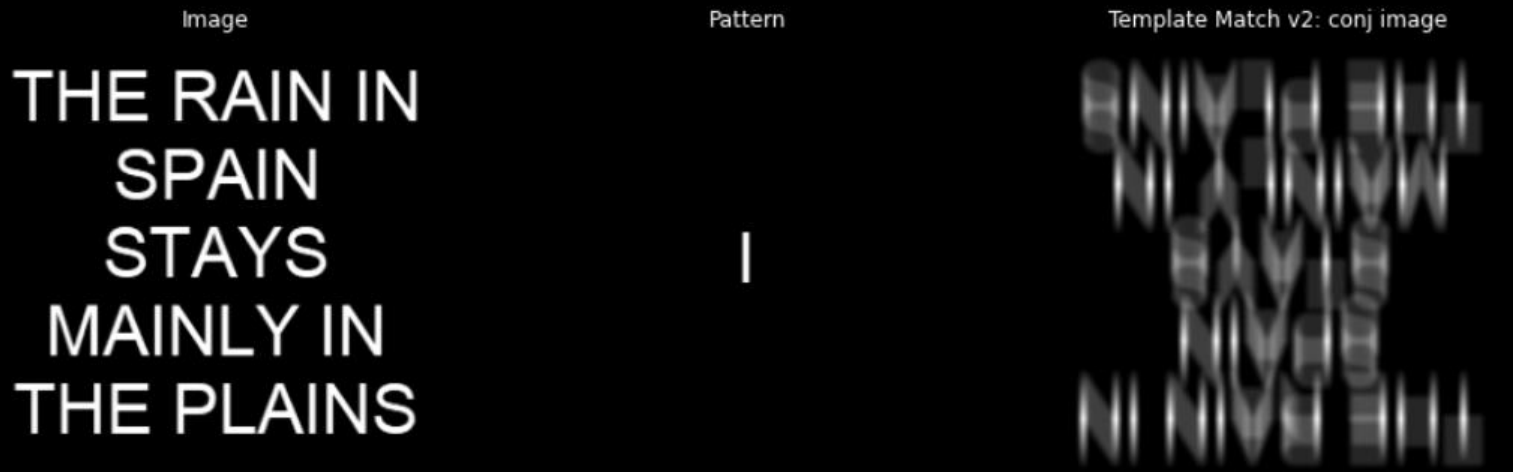




Template Matching

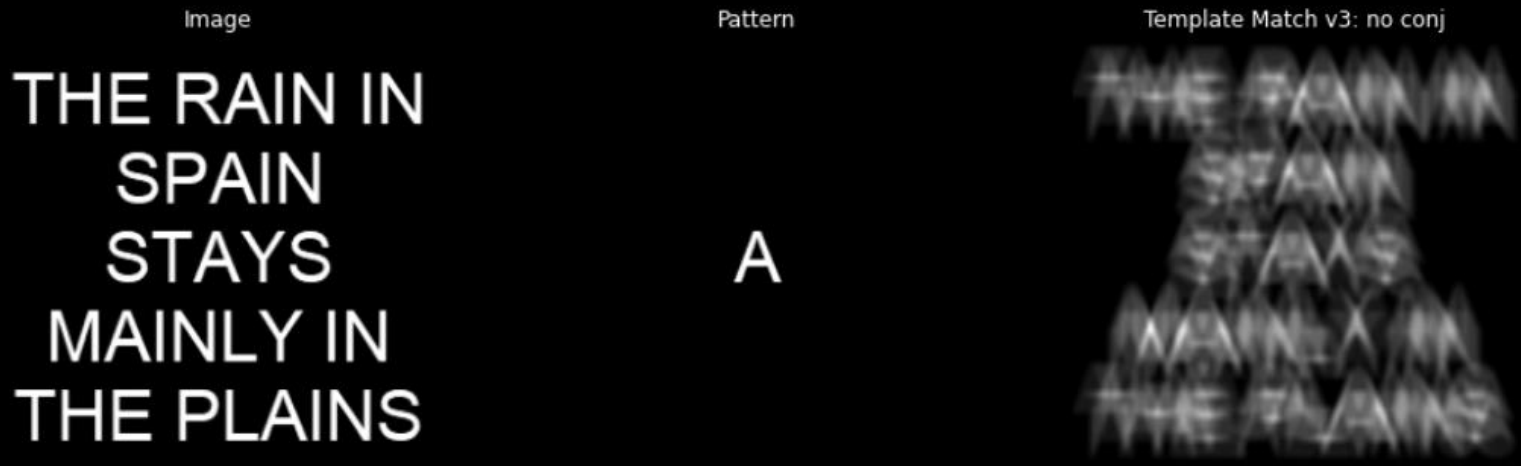
Here is the result when the image FFT is conjugated. Notice that the final result is inverted.

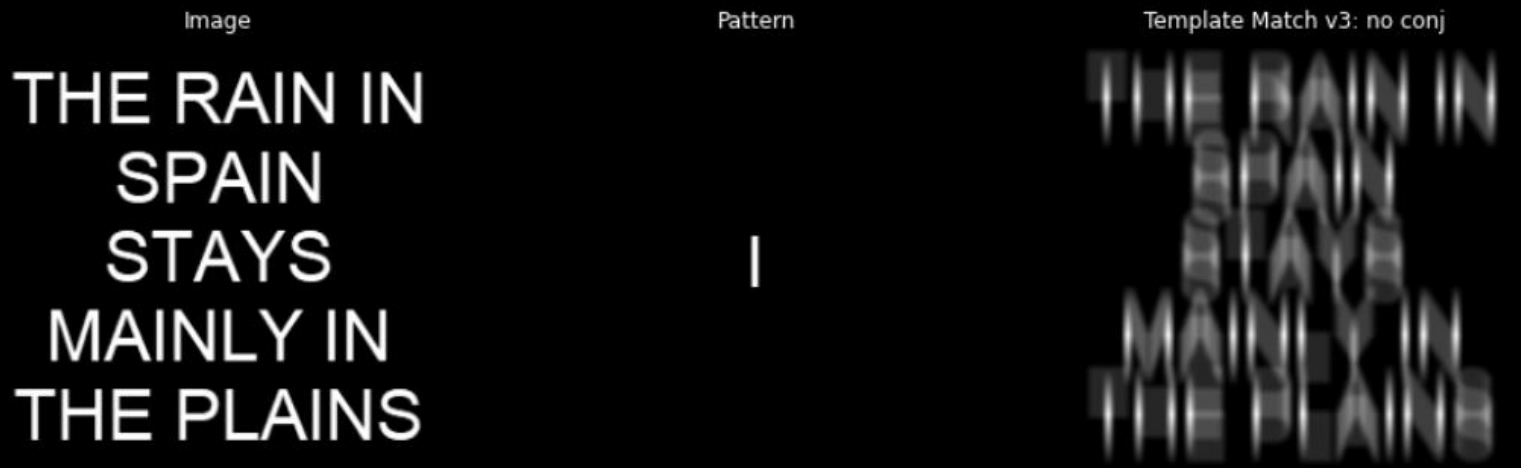




Template Matching

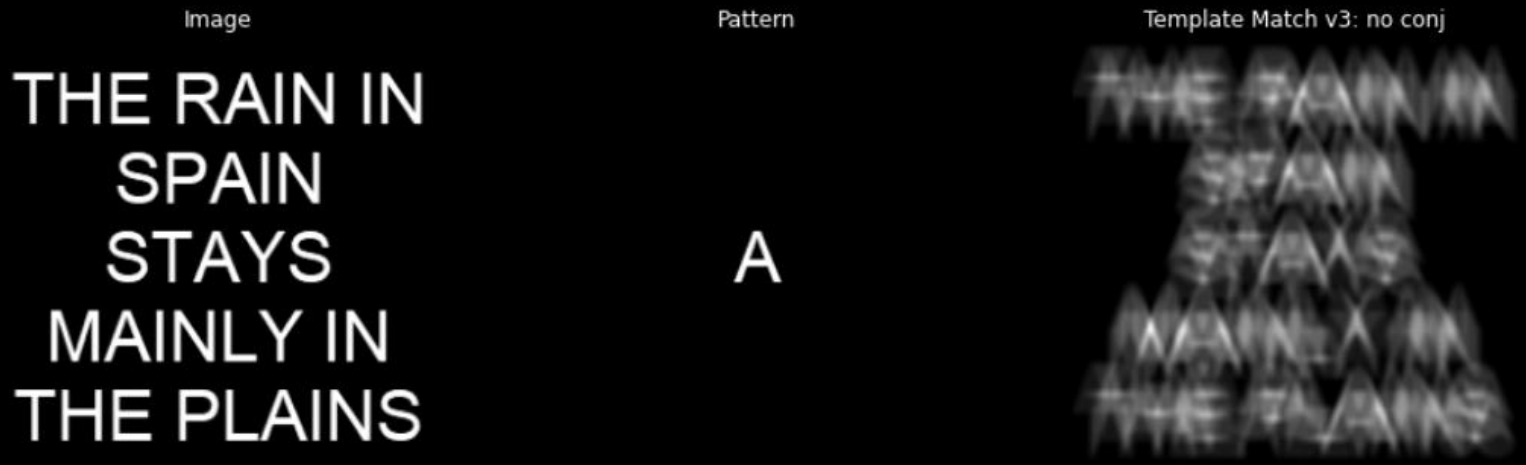
Here is the result when none of the image and pattern FFTs are conjugated. Observe that there are no "bright spots" indicating a template or feature match. The final image is upright.

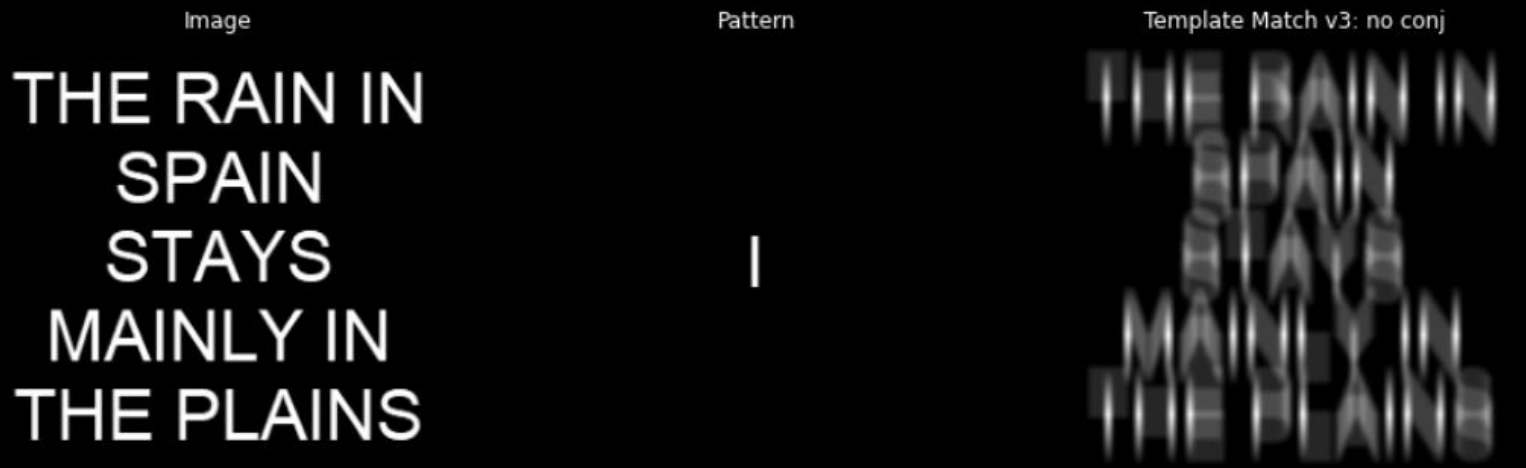




Template Matching

When both the image and pattern FFTs are conjugated, no bright spots are observed and the final image is upright.





Template Matching

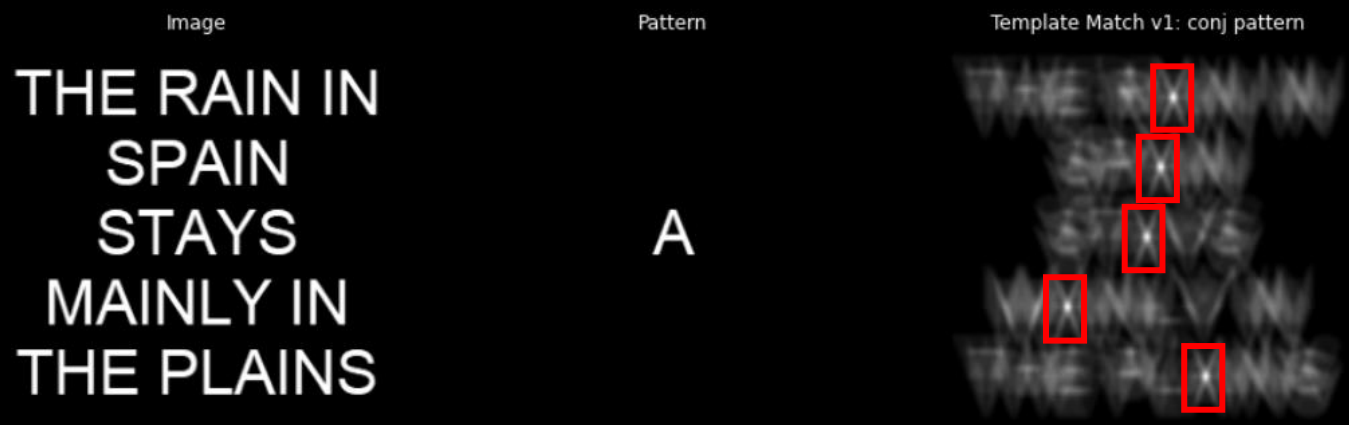
To summarize, conjugation of either one of the image or pattern FFTs (but not both) is important in the appearance of the bright spot on the image locations corresponding to a template match. Conjugating the image FFT will yield a flipped image upon ifft2, and conjugating the pattern FFT will yield an upright image. This can be thought of as flipping the sign of the phase of a complex sinusoid.

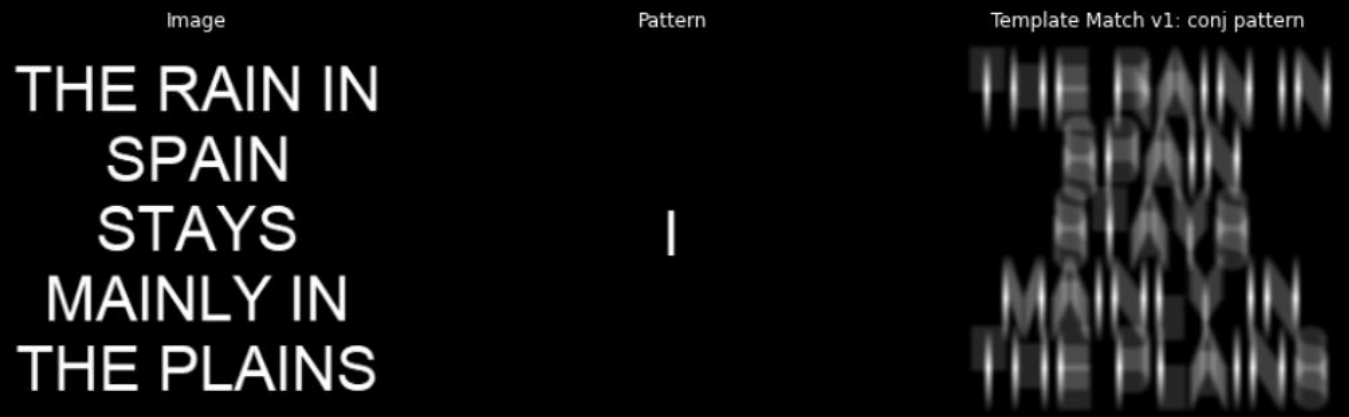
Template Matching

```
def template_match(image, pattern): #template match version where AFFT (pattern) is conjugated.
    AFFT = np.conj(np.fft.fft2(pattern))
    FImg = np.fft.fft2(image)
    P = FImg*AFFT
    smallp = np.fft.fftshift(np.fft.ifft2(P))
    return abs(smallp)

def template_matchv2(image, pattern): #template match version where FImg (image) is conjugated.
    AFFT = np.fft.fft2(pattern)
    FImg = np.conj(np.fft.fft2(image))
    P = FImg*AFFT
    smallp = np.fft.fftshift(np.fft.ifft2(P))
    return abs(smallp)

def template_matchv3(image, pattern): #experimental template match version where none of the FImg and AFFT are
conjugated.
    AFFT = np.fft.fft2(pattern)
    FImg = np.fft.fft2(image)
    P = FImg*AFFT
    smallp = np.fft.fftshift(np.fft.ifft2(P))
    return abs(smallp)

def template_matchv4(image, pattern): #experimental template match version where none of the FImg and AFFT are
conjugated.
    AFFT = np.conj(np.fft.fft2(pattern))
    FImg = np.conj(np.fft.fft2(image))
    P = FImg*AFFT
    smallp = np.fft.fftshift(np.fft.ifft2(P))
    return abs(smallp)
```

Template Matching

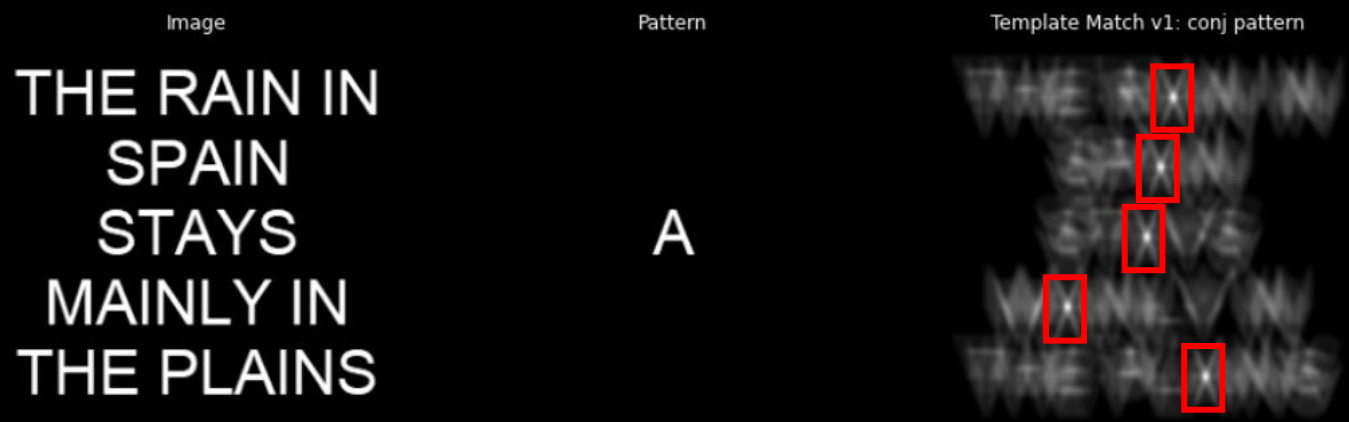
The matched locations are indicated by the presence of bright white spots, which I boxed in red below. Note that for Pattern I, it is difficult to find an exact match. This is because the pattern is essentially an inconspicuous vertical line, a pattern that is abundant in the original image.

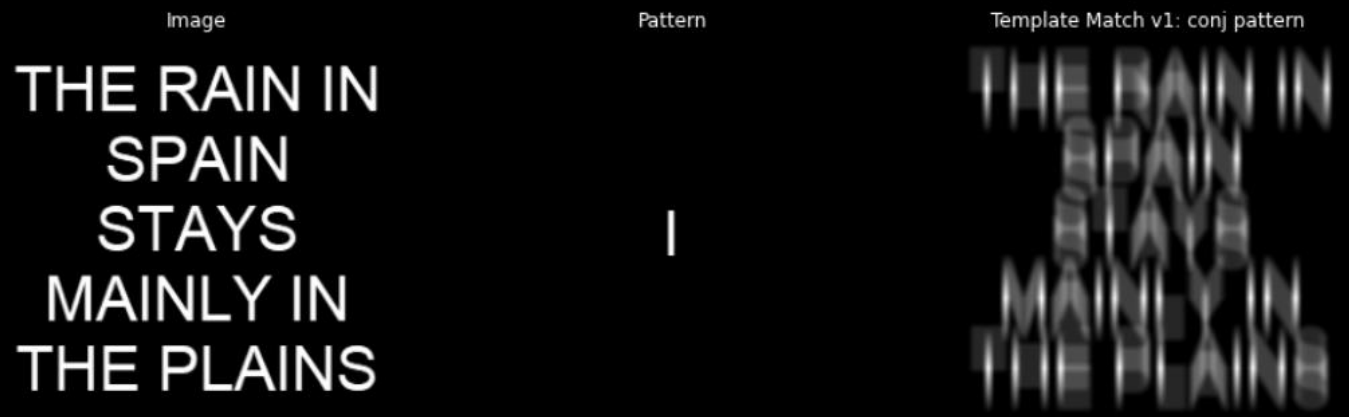




Template Matching

This poses one of the limitations of this algorithm. The algorithm works well if the pattern to be sought "stands out" from the other features present in the image. A pattern like a vertical line "I" will not work well for this image.





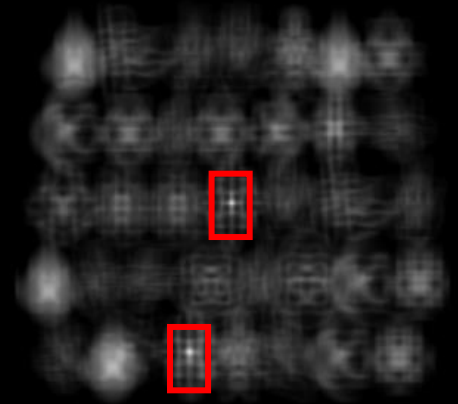
Template Matching

This algorithm is not perfect and will potentially detect features or patterns that are similar but not identical to the desired pattern.

Image

Pattern

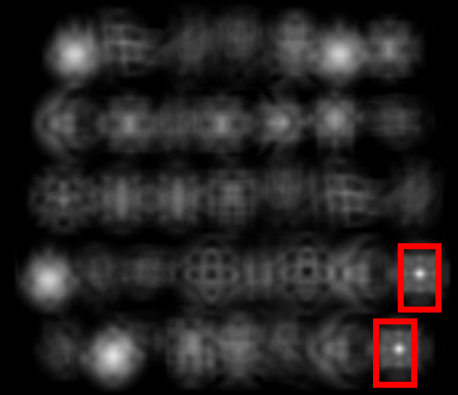
Template Match



Image

Pattern

Template Match



Self-Reflection

In this module, I've personally delved deeper into the intuition and insight behind 2D FFT of images (as someone who enjoys theory quite a lot). The parts I've especially enjoyed in this module are:

1. Establishing the connection between the far-field diffraction of apertures and their corresponding 2D FFT.
2. Thinking about the mathematics behind convolution and correlation. I especially liked how I developed my intuition behind the formula in the conjugation operation in correlation (template matching).
3. Testing out various versions of code to develop an intuition behind the concept feels like conducting a scientific experiment for me.

It took me a while to develop an intuition on the "quadrant flipping" and the reason behind fftshift(), but once I got them, everything else clicked into place.

Self grade: 105/100

Reason: I've experimented with various aperture shapes and thought hard about generalizing and automating the figure generation. I also experimented with various versions of template matching and the effect on the output image. I would've given myself a higher score if I experimented with actual applications like image blurring, and possibly template matching in RGB images.

References

1. (How the 2D FFT Works) https://www.youtube.com/watch?v=v743U7gvLq0
2. https://codereview.stackexchange.com/questions/206704/marking-a-rectangular-region-in-a-numpy-array-as-an-image-mask
3. (Generating a polygon array) https://stackoverflow.com/questions/37117878/generating-a-filled-polygon-inside-a-numpy-array
4. https://thepythoncodingbook.com/2021/08/30/2d-fourier-transform-in-python-and-fourier-synthesis-of-images/
5. Principles and Practice of Physics by Erik Mazur (section about diffraction grating intensity and amplitude)

Special thanks to Julian Maypa and Nino Ramones for answering my questions about the code in template matching