Properties and Applications of the 2D Fourier Transform

Ron Michael Acda
2019-03839

Objectives

- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

Objectives

- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

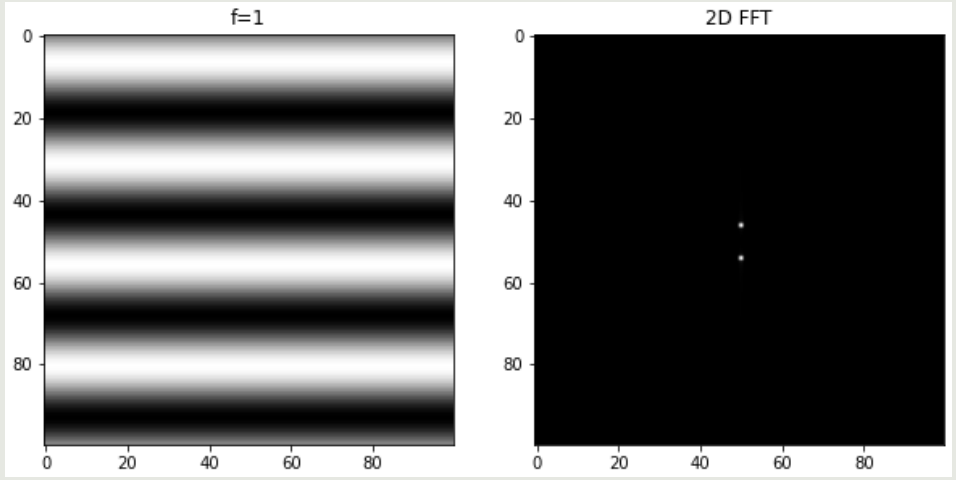
Sinusoids

To construct a 2D sinusoid with a phase angle (how rotated or “tilted” it is) and the frequency (“grating spacing”), the following code is implemented.

```
def corrugated_roof(theta, f, N=200): #Theta is measured with respect to vertical.
    x = np.linspace(-2,2,num = N)
    y = x
    X,Y = np.meshgrid(x,y)
    omega = 2*np.pi*f
    vals = 0.5*np.sin(omega*(X*np.sin(theta) + Y*np.cos(theta)))
    return vals
```

This creates an NxN xy-grid whose z-values are vals, as defined. This function returns a NumPy array, in which we can perform np.fft.fft2 (then performing fftshift to zero-center it).

2D FFT of a Sinusoid

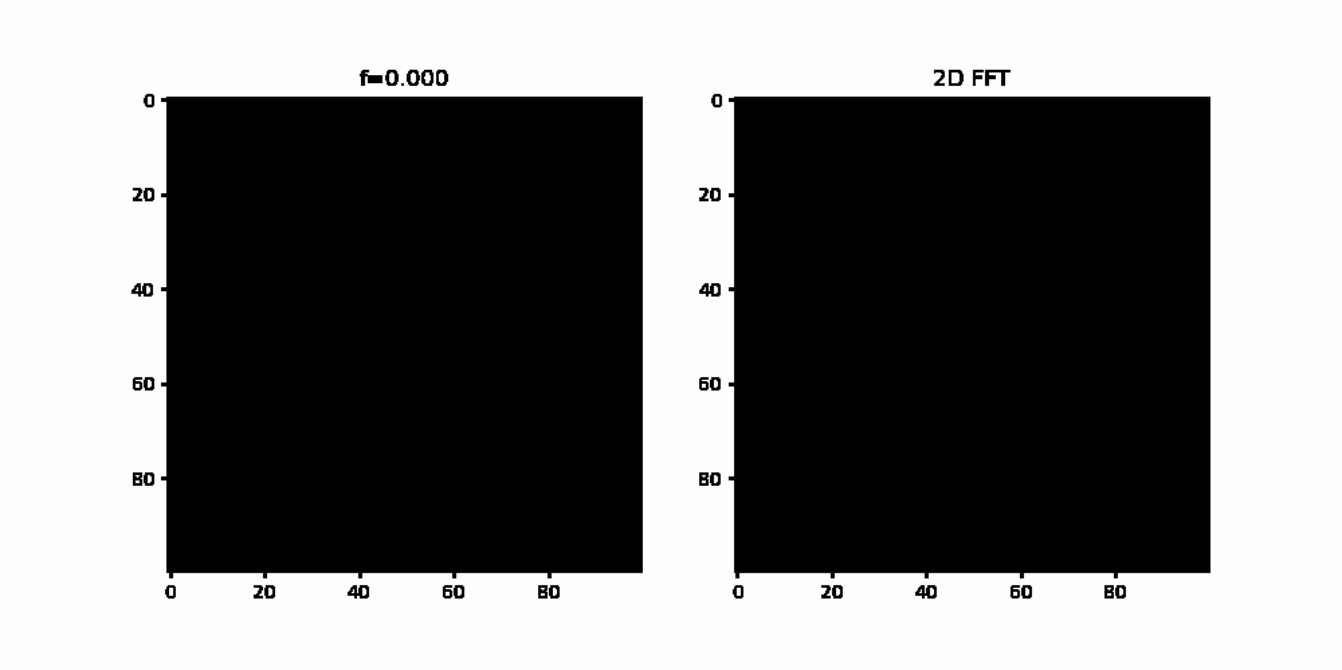


Note that for a sinusoid, the corresponding 2D FFT plot appears as a pair of points. This can be explained mathematically. The original sinusoid, in the complex plane, can be viewed as:

$$e^{i2\pi(ax+by)} = \cos[2\pi(ax+by)] + i\sin[2\pi(ax+by)]$$

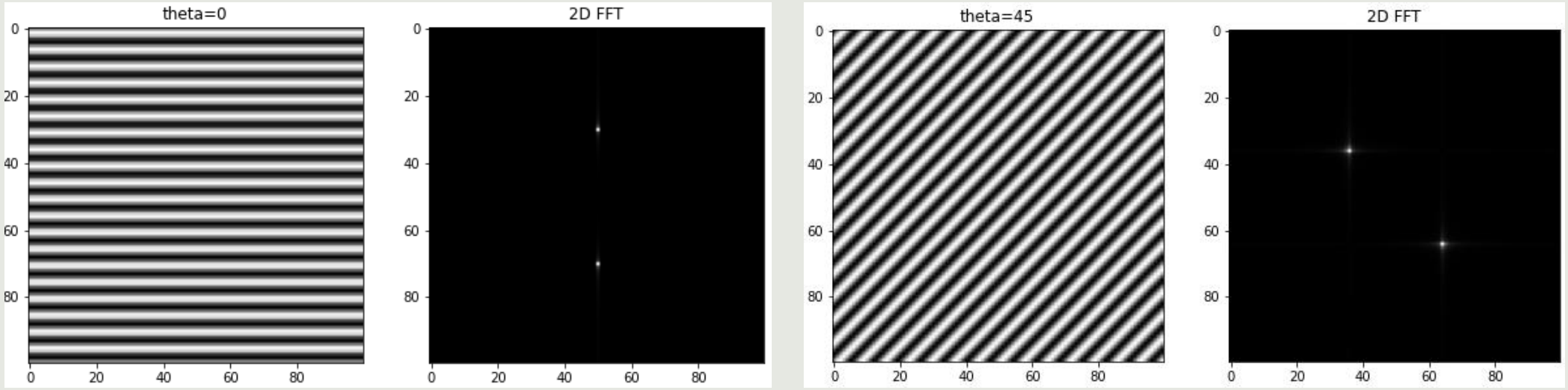
where a and b are the Fourier pairs. If (a,b) is a solution, then so is (-a,-b), hence, two points appear in the 2D FFT. We can connect these two points by a vector in (a,b)-space.

Effect of the Frequency on the 2D FFT



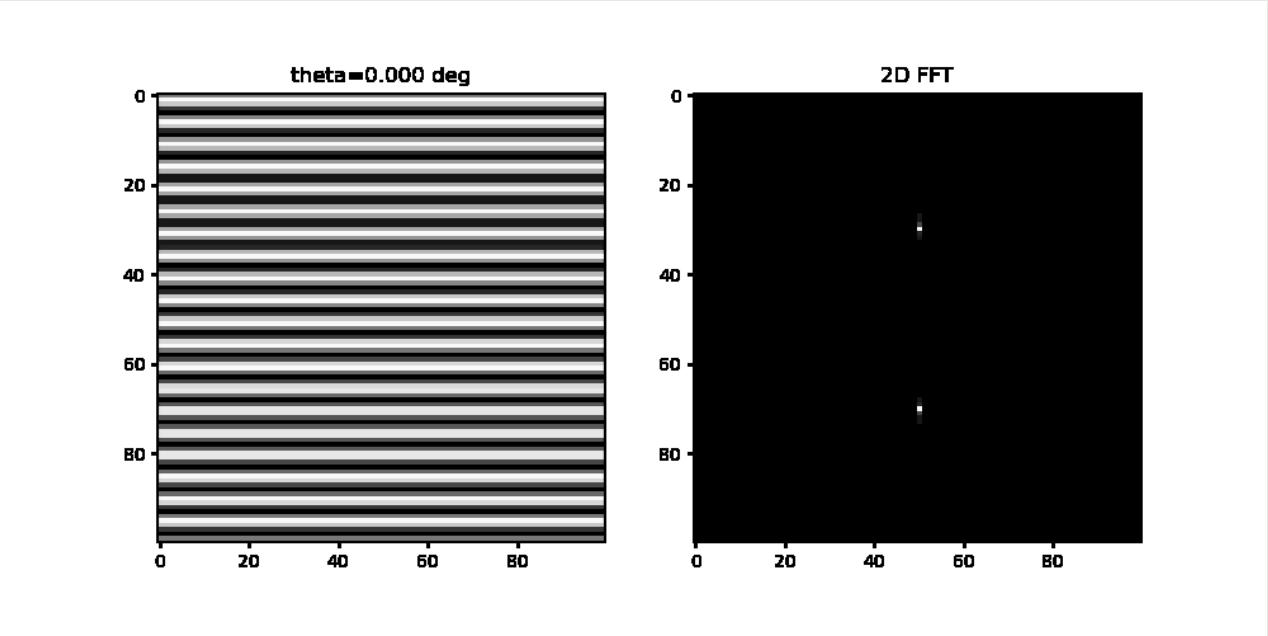
As the frequency is increased, the corresponding Fourier pairs (a,b) ought to increase as well, so their distance to the center of the 2D FFT image increases. The magnitude of the (a,b) vector indicates the frequency of the original sinusoid.

Effect of the Phase Angle on the 2D FFT



Note that this (a,b) vector has the same orientation as the original sinusoid, which is intuitive: the vector connecting these two points in the (a,b)-space has the same direction connecting the maxima (or minima) of the sinusoids.

Effect of the Phase Angle on the 2D FFT

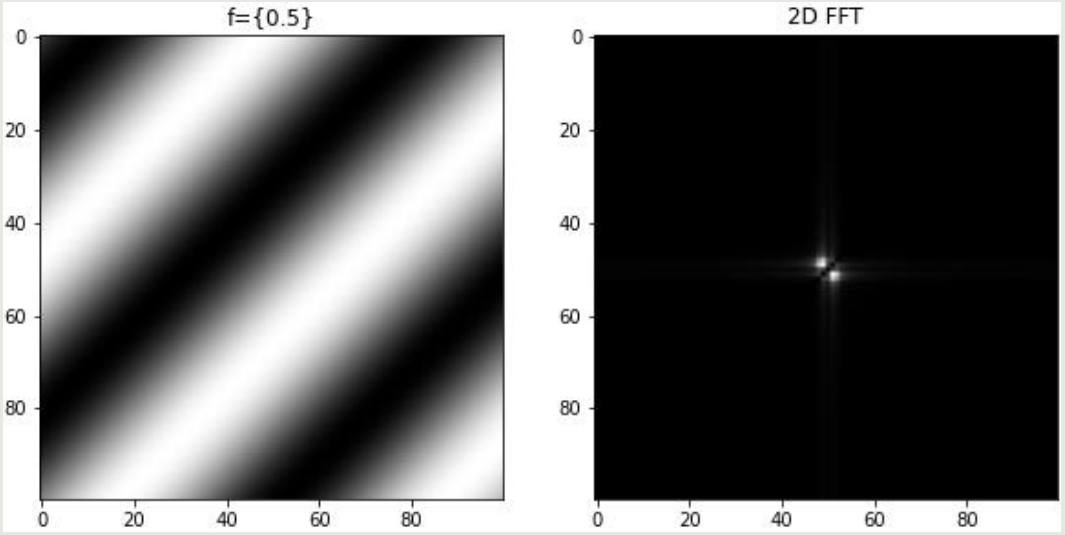


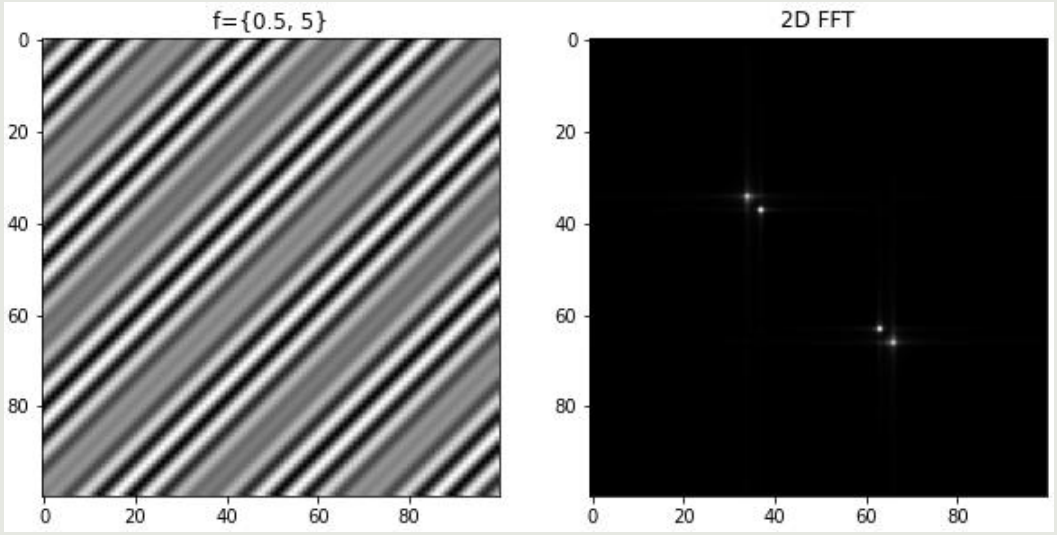
We can even see from the animation that as the phase angle increases, so does the direction of the (a,b) vector in the FFT space i.e., both “rotate” in the same orientation over time.

Objectives

- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

Superimposition of Frequencies





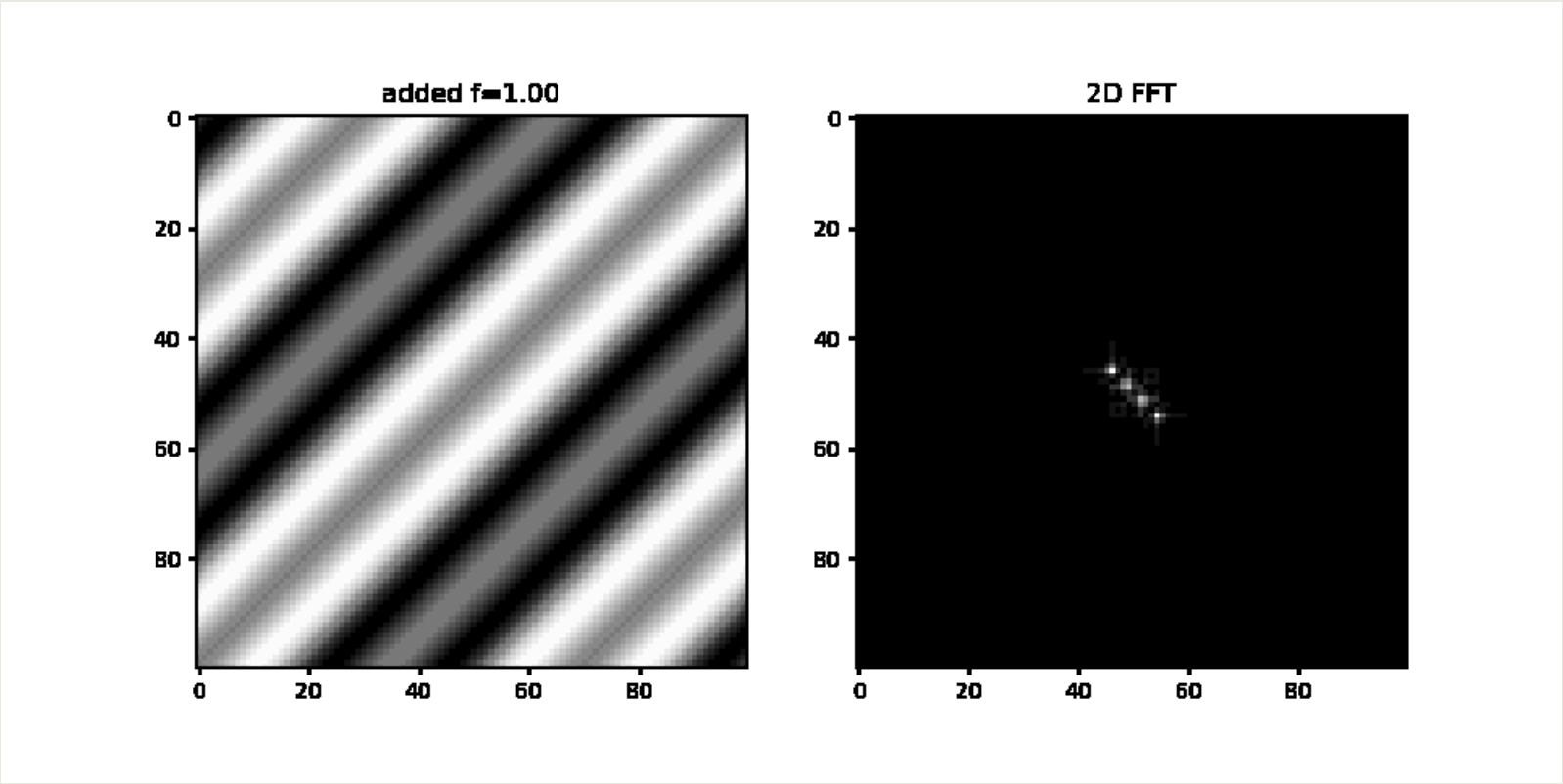
We can even combine sinusoidal waves with varying frequencies by multiplication of sinusoids; i.e,

```
corrugated_roof(np.pi/4, 0.5,100)*corrugated_roof(np.pi/4, 5,100)
```

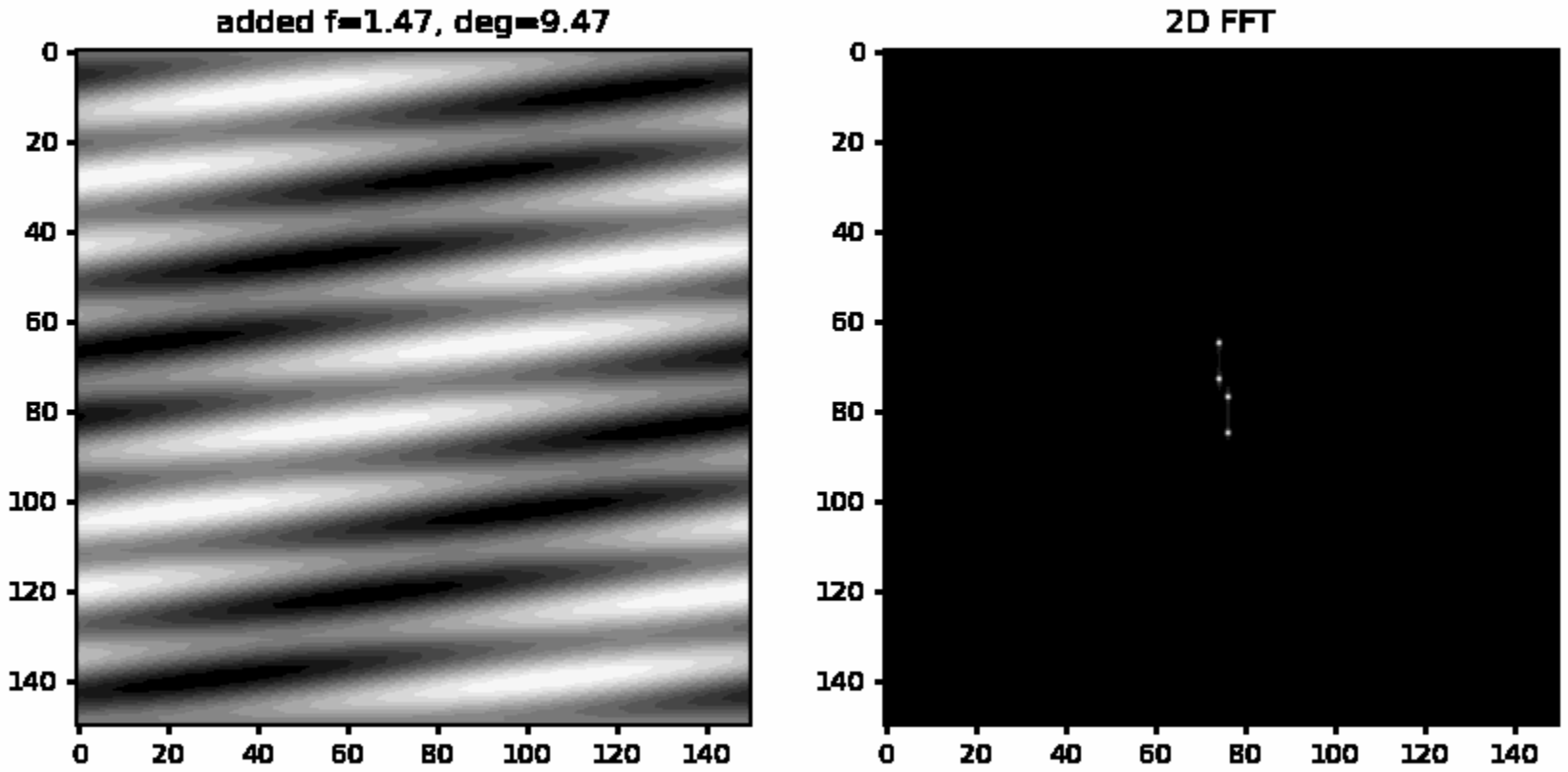
This produces an image with frequencies 0.5 and 5, as shown in the right.

Note that the length of the (a,b) vector isn't generally preserved by this operation.

Superimposition of Frequencies



If the 2D function is a product of n sinusoids with varying frequencies, then the resulting 2D FFT has 2*n peaks.

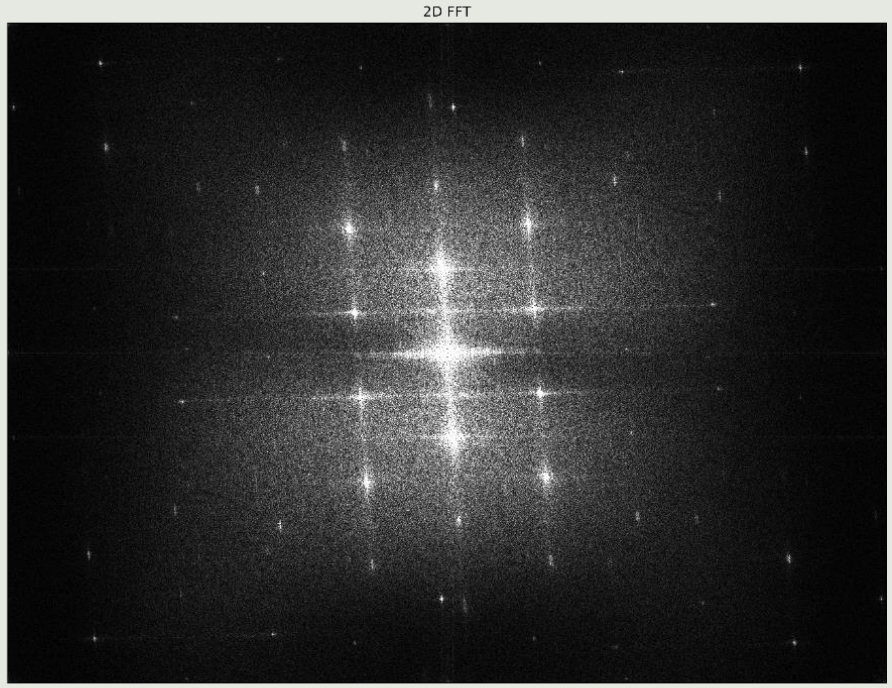


We combine all of these effects into a single, cool animation.

Objectives

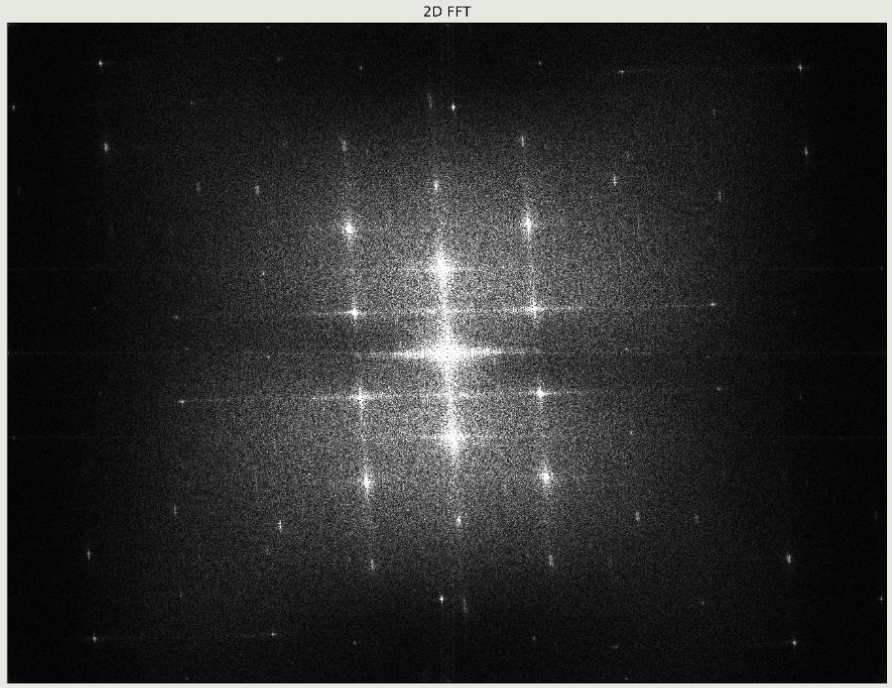
- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

Fourier Filtering



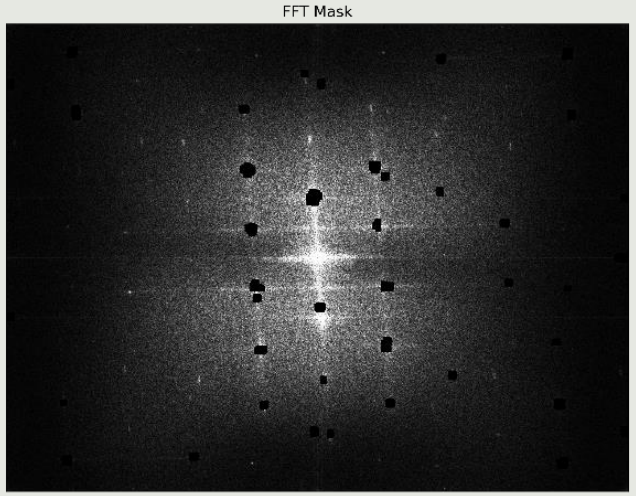
Sometimes, images have unwanted, “high-frequency” artifacts like the canvas weave pattern of the painting above. These artifacts appear as high-frequency, **symmetric** points in the 2D FFT image. They are perhaps more evident when the original 2D FFT is contrast stretched or log-shifted; however, for my purposes, I find *percentile stretching* of the 2D FFT to be clearer (code was recycled from Module 1).

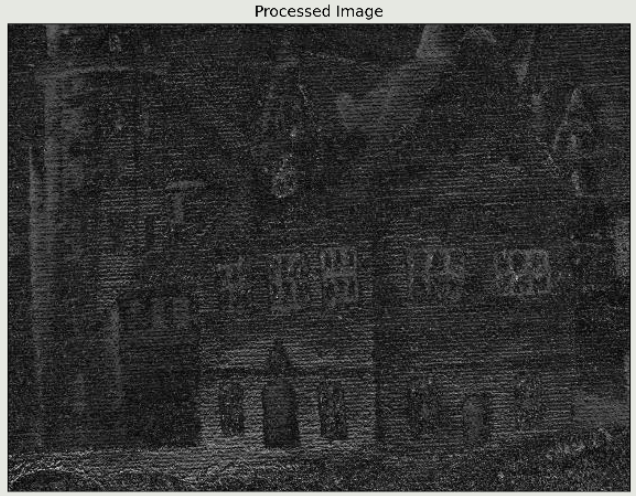
Fourier Filtering

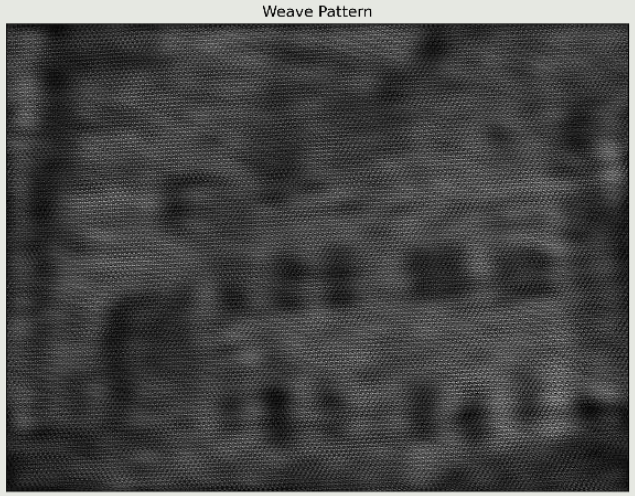


Our goal is to remove these “unwanted” points by creating a mask in the 2D FFT array, and performing inverse Fourier transform on the “masked” 2D FFT array. If we want to retrieve the “weave pattern”, we just invert our mask, easily done by using np.logical_not.

Automatic Masking







One can manually create a mask, but where's the fun in that? Here's my attempt at automation, which obviously has a lot of room for improvement, but what I'm more proud of is how I executed the idea (the journey matters more to me!). Although the image processing could use some improvement, the result weave pattern extraction is somewhat satisfactory.

Automatic Masking Algorithm

The algorithm I implemented works as follows:

Step 1: Perform percentile thresholding on each mxn subarray of the MxN fft2 image (**localized thresholds**).

Step 2: Use **DBSCAN** (density-based spatial clustering of applications with noise) algorithm to detect clusters of points (white specks) in a subarray.

Step 3: For the points in the detected cluster, generate a "**blot**" to increase the size of the mask.

Step 4: Remove all processing at the middle (cropping) to reduce image data loss.

Localized Thresholding

A quite naïve way of automating (although much faster) would be to set a global **threshold** of FFT intensity, and mask those values that exceed that threshold. However, this method runs the risk of “over-filtering” and possibly removing important frequencies.

An alternative that I came up with is to subdivide the original MxN array into smaller mxn arrays. Then, the usual thresholding is performed on each of these arrays. In exchange for less risk of overfiltering, we add another parameter to the algorithm: **subarray size**.

Localized Thresholding

This method depends on two crucial functions.

1. split_array as a function that takes the original MxN array as an input and returns a list of subarrays of user-defines sizes mxn (note that m and n need not necessarily be divisors of M and N, respectively, so the shapes of the subarrays in the list need not be the same).
2. restore_array as a function that takes as an input the list of the subarrays and outputs the original array, basically the “inverse” of split_array.

The idea is to process the subarrays and recombine them by using a variant of the restore_array function.

Localized Thresholding

```
def split_array(arr, M_sub, N_sub):
    # Determine the shape of the original array
    M, N = arr.shape

    # Determine the shape of the subarrays
    num_rows = M // M_sub + int(M % M_sub > 0)
    num_cols = N // N_sub + int(N % N_sub > 0)

    # Create a list to hold the subarrays
    subarrays = [[] for _ in range(num_rows)]

    # Split the original array into subarrays
    for row in range(num_rows):
        for col in range(num_cols):
            subarr = arr[row*M_sub:(row+1)*M_sub, col*N_sub:(col+1)*N_sub]
            subarrays[row].append(subarr)

    return subarrays
```

Localized Thresholding

```
def restore_array(subarrays, M_sub, N_sub):
    # Determine the shape of the original array
    M = len(subarrays) * M_sub
    N = len(subarrays[0]) * N_sub

    # Initialize the restored array
    restored_arr = np.zeros((M, N), dtype=subarrays[0][0].dtype)

    # Restore the subarrays to the original array
    for row, subrow in enumerate(subarrays):
        for col, subarr in enumerate(subrow):
            if subarr.size > 0:
                restored_arr[row*M_sub:row*M_sub+subarr.shape[0], col*N_sub:col*N_sub+subarr.shape[1]] = subarr

    return restored_arr[0:M,0:N]
```

DBSCAN

DBSCAN is a machine learning algorithm which I picked up when I was studying ML for fun back ago. It's a cluster detection algorithm which depends on two important parameters: **eps** and **min_samples**. Here, a cluster can be defined as a neighborhood of points containing at least min_samples, with each point having a distance greater than eps.

DBSCAN

```
def detect_zero_clusters(array, eps, min_samples):
    # Flatten the array into a list of points
    points = [(i, j) for i in range(array.shape[0]) for j in range(array.shape[1]) if array[i, j] == 0]

    if len(points) == 0:
        # No clusters detected, return the original array
        return array

    # Run DBSCAN on the list of points
    clustering = DBSCAN(eps=eps, min_samples=min_samples).fit(points)

    # Get the indices of the points in the cluster
    cluster_indices = [i for i in range(len(points)) if clustering.labels_[i] == 0]

    # Create an empty array to hold the cluster
    cluster_array = np.ones_like(array)

    # Set the values of the cluster points in the array
    for i in cluster_indices:
        cluster_array[points[i]] = 0

    return cluster_array
```

Cropping

The last step is trivial: remove all processing at the center of the image, defined by a rectangle with length **L** and width **W**. This is because most of the data lie in the center of the fftshifted FFT2 image.

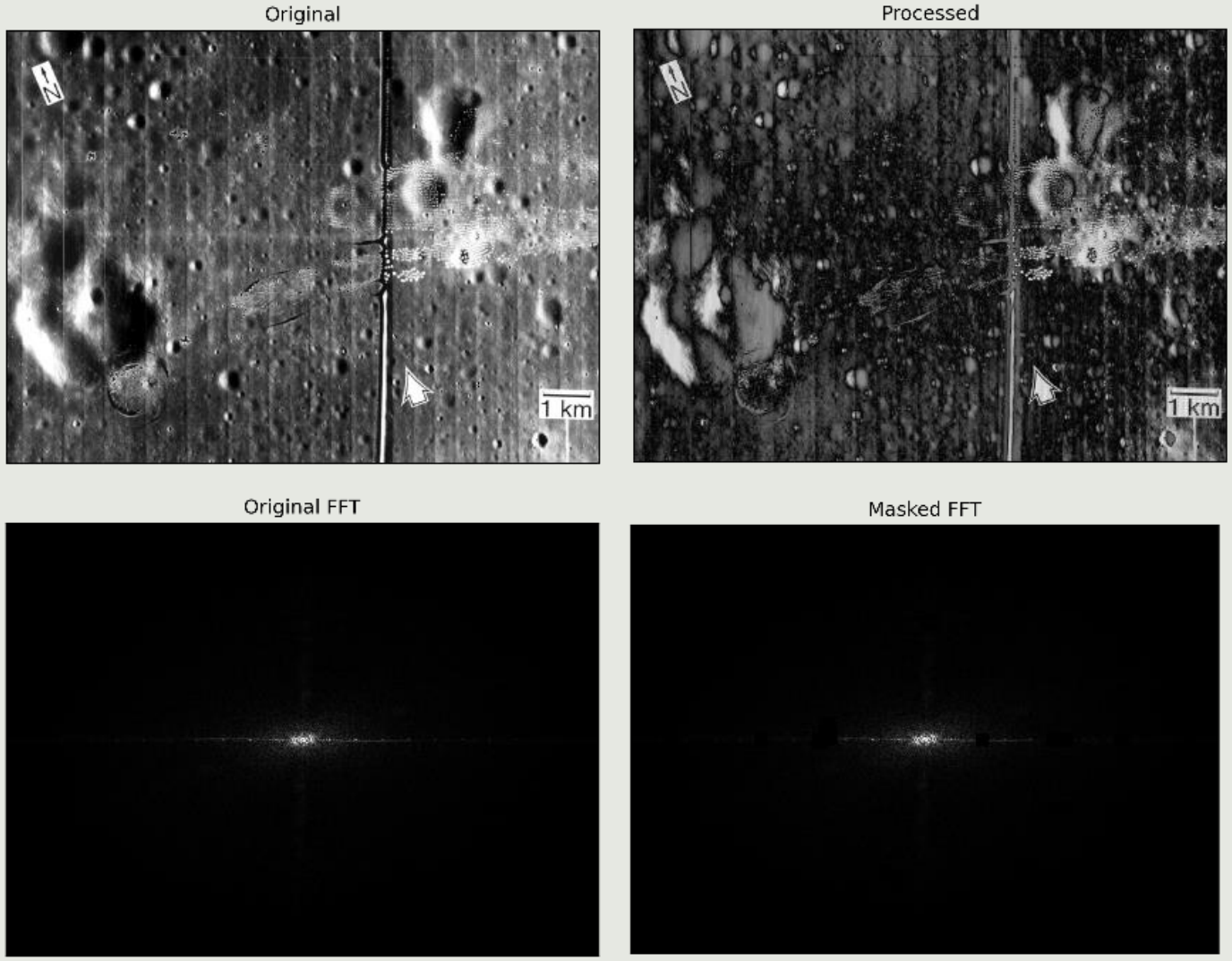
```
def rect_cropper(arr, L=0.08, W=0.2):
    Y,X = np.shape(arr)
    mask = np.zeros((Y,X))
    cen = np.array([1/2, 1/2])
    cen_coord = np.array([cen[0]*Y, cen[1]*X])
    low_y = int(cen_coord[1] - np.round(W*Y/2))
    high_y = int(cen_coord[1] + np.round(W*Y/2))
    left_x = int(cen_coord[0] - np.round(L*X/2))
    right_x = int(cen_coord[0] + np.round(L*X/2))
    mask[left_x : right_x, low_y : high_y] = 1
    return np.array(mask+arr, dtype='bool')
```

The (monster) Algorithm

Combining everything, the algorithm I devised depends on seven parameters! Compare this to one by global thresholding.

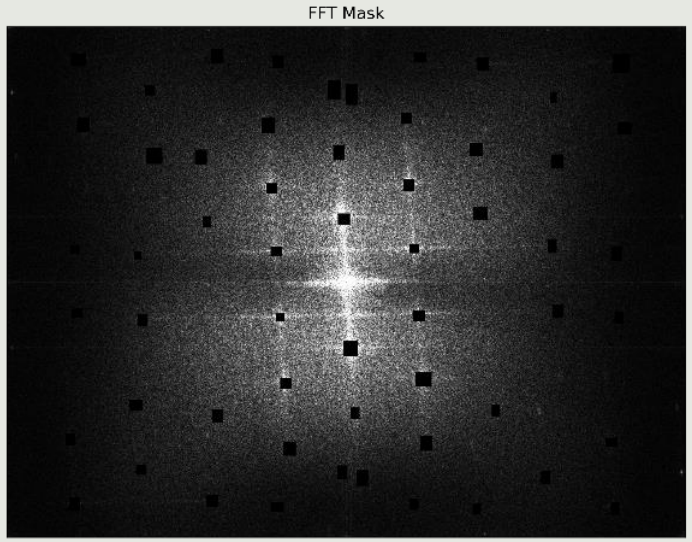
```
def auto_masking(arr, M_sub, N_sub, threshold=98, eps=3, min_samples=8, blot_size=7, L=0.08, W=0.2):
    M, N = arr.shape

    # Determine the shape of the subarrays
    num_rows = M // M_sub + int(M % M_sub > 0)
    num_cols = N // N_sub + int(N % N_sub > 0)

    # Create a list to hold the subarrays
    subarrays = [[] for _ in range(num_rows)]

    # Split the original array into subarrays
    for row in range(num_rows):
        for col in range(num_cols):
            subarr = arr[row*M_sub:(row+1)*M_sub, col*N_sub:(col+1)*N_sub]
            step1= threshold_masker(subarr, threshold)
            step2= detect_zero_clusters(step1, eps=eps, min_samples=min_samples)
            if 0 in step2:
                step3=blotter(step2,size=blot_size)
            else:
                step3=step2
            subarrays[row].append(step3)

    semi_restored1=restore_array(subarrays, M_sub=N_sub, N_sub=N_sub)[0:M,0:N]
    semi_restored2=rect_cropper(semi_restored1,L=L,W=W)
    return semi_restored2
```

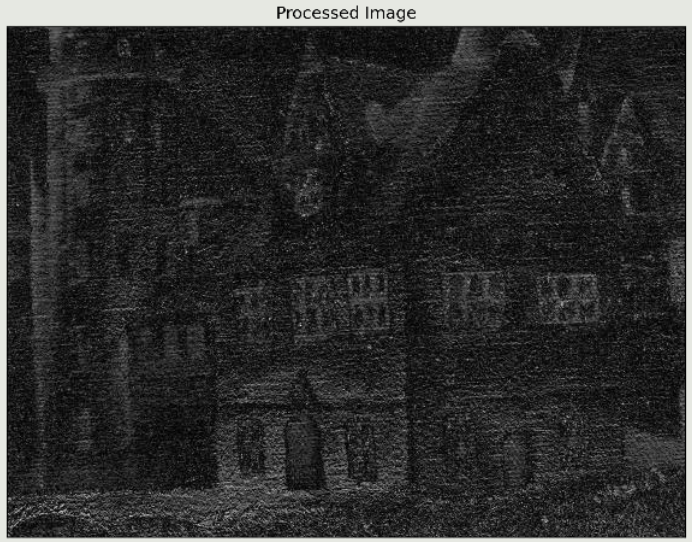
The (monster) Algorithm, applied on line removal

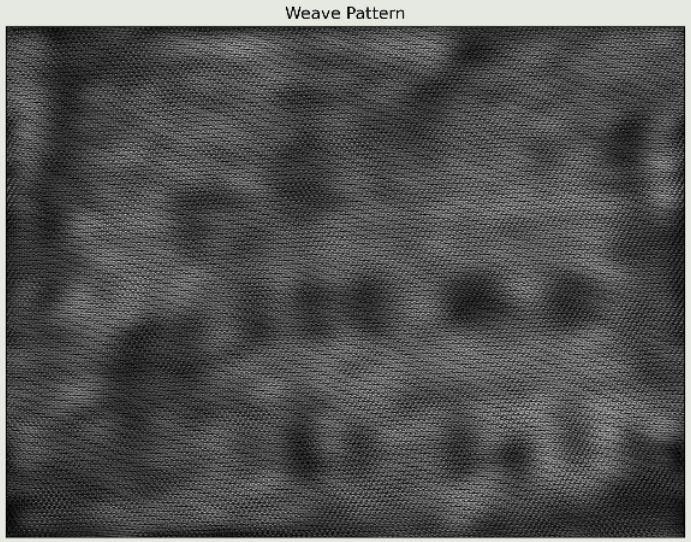


There are rooms for improvement on the algorithm, mostly because my execution is quite inefficient (takes a full second or two to fully run, depending on the parameters). The blotting procedure can also be improved (one idea is to convolve a “+” pattern on the detected clusters). Still, it’s a proof of concept!

Manual Masking

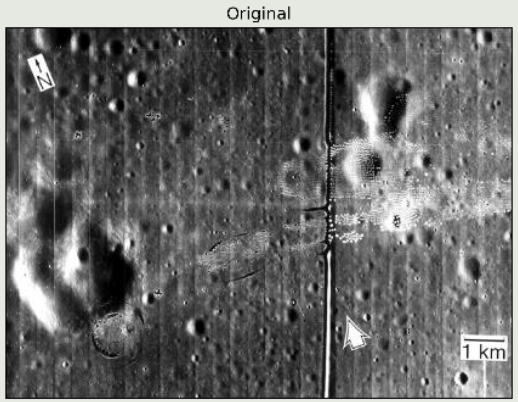


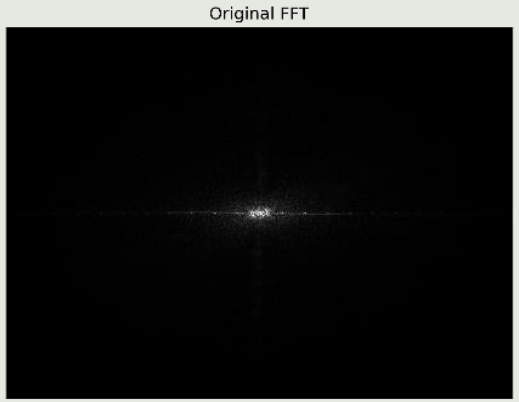


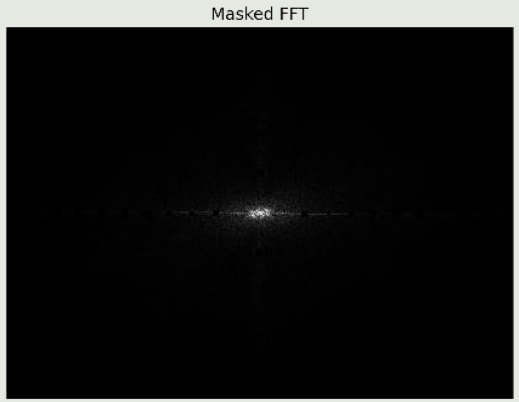


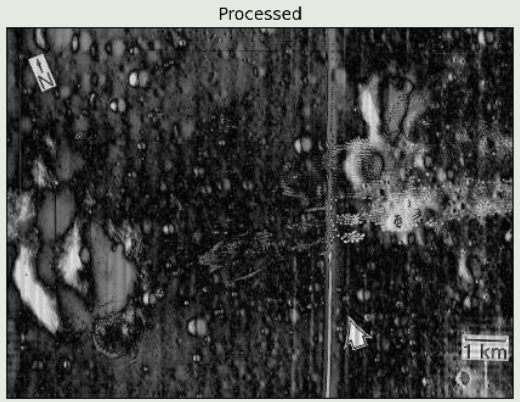
Now the boring (but quick) method. The mask was generated using GIMP. I feel like I could've done better than just haphazardly adding rectangles (could've used "+" patterns instead) but at least I was able to successfully retrieve the pattern. The use of rectangles may have inadvertently caused the loss of data, resulting to a dark blurred image.

Manual Masking



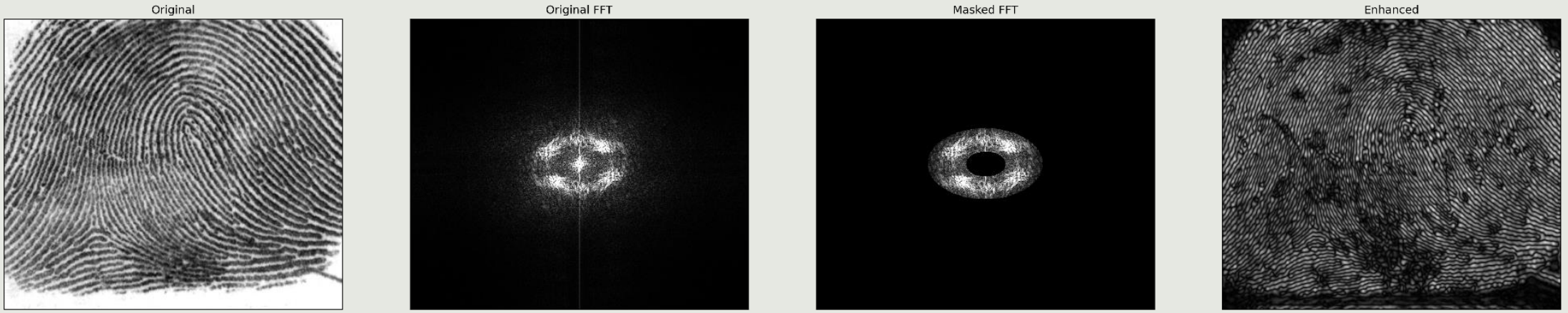






The use of relatively large rectangles may also have inadvertently caused the loss of data, resulting to a darker image, although the “lines” were successfully removed.

Fingerprint Ridge Enhancement



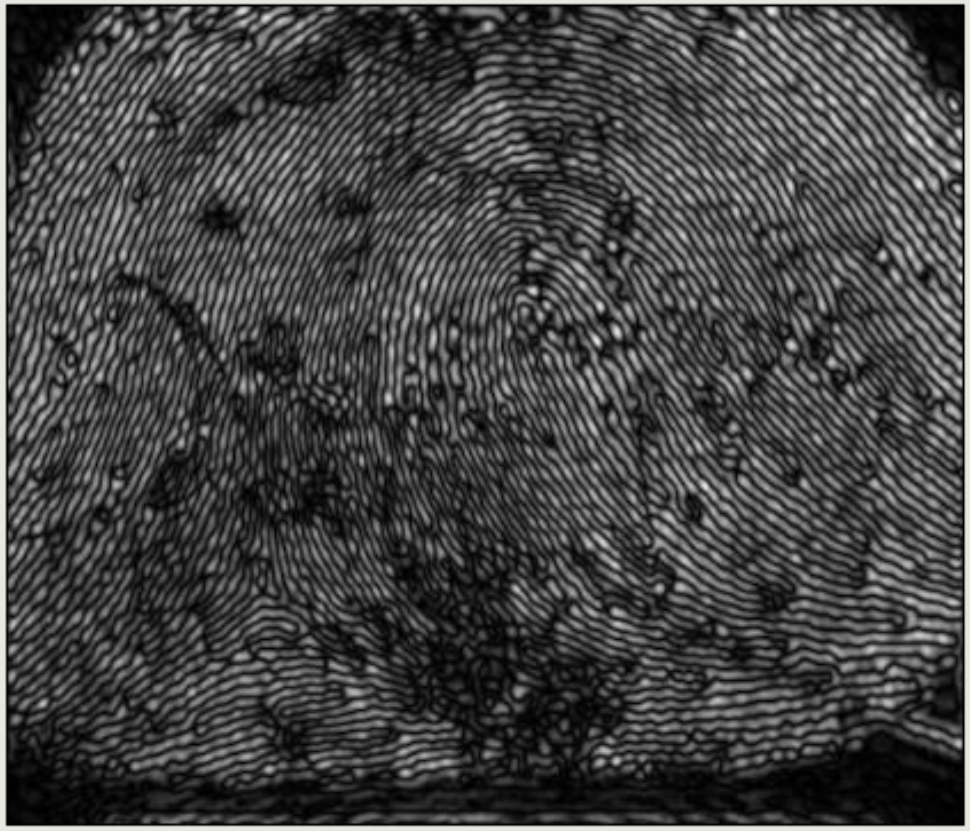
The concept of Fourier filtering and masking is useful in forensics since it can enhance fingerprint images, as shown above. The mask was generated by considering that the “curves” of the fingerprint marks can manifest as “symmetric” bright points. Note that this is now the reverse of what we were doing earlier.

The mask used was also generated in GIMP.

Fingerprint Ridge Enhancement

Original

Enhanced

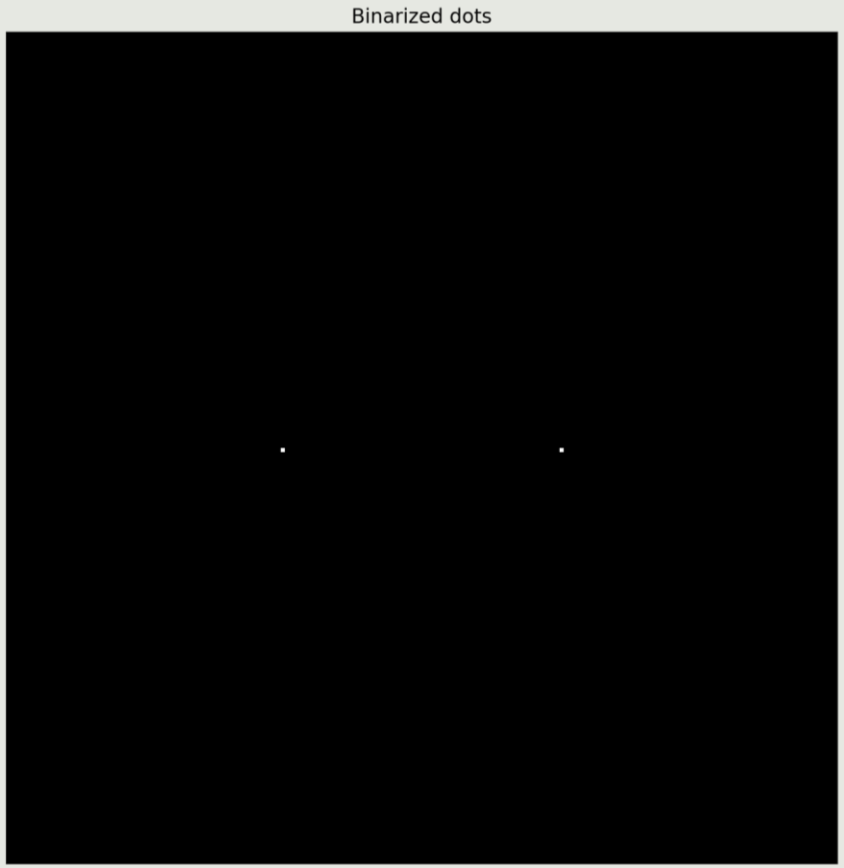


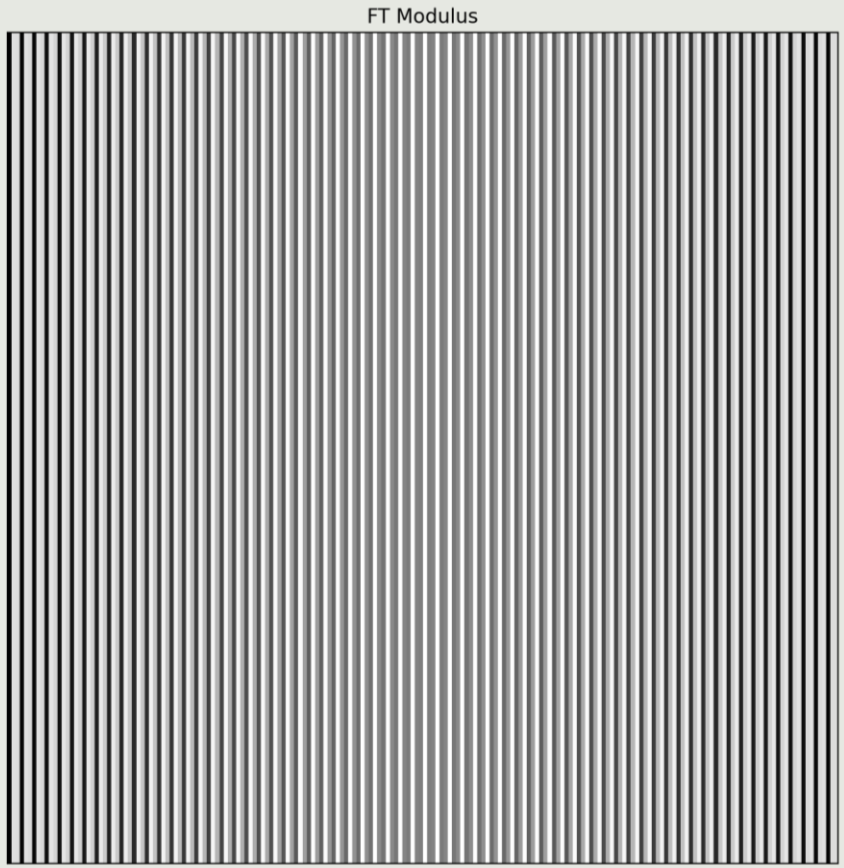
The enhanced image show the ridges more clearly.

Objectives

- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

Binarized Dots





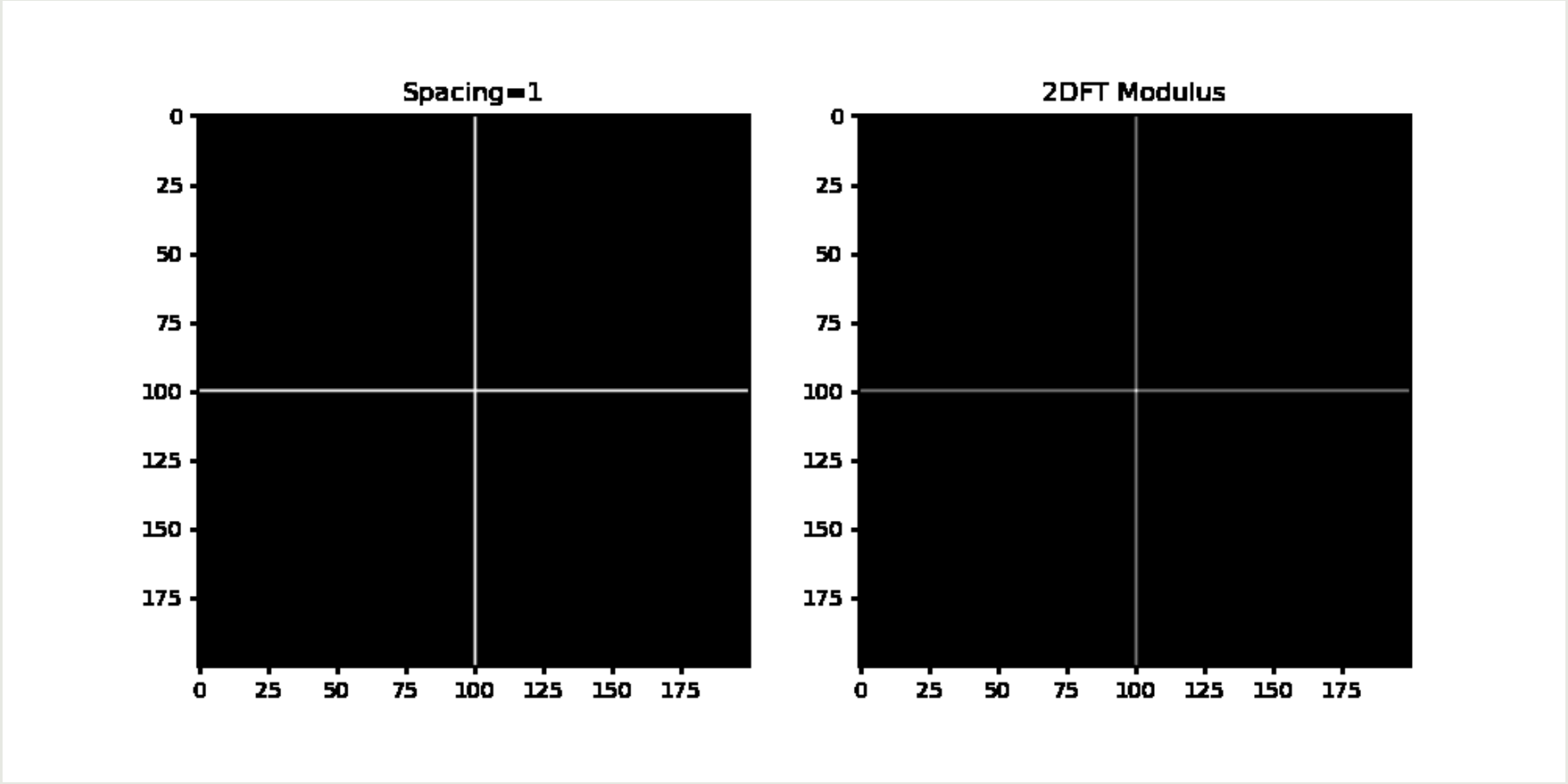
Note that the corresponding FFT of two dots is a single-frequency 2D sinusoid, just like how the FFT of a 2D sinusoid is two dots. This isn’t surprising, as the Fourier transform is a self-invertible operation.

Binarized Dots

```
def dots(number=2, N=200):
    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)/(number+1)) for i in range(number)])
    cen_trans = np.array(cen*N, dtype='int')
    for c in cen_trans:
        vals[c[0], c[1]] = 1
    return vals
```

Here's the code to generate two dots in an NxN array. We locate the coordinates of the dots, and assign a value of 1 by indexing on the vals array. This can be generalized into any number of dots.

Equally-spaced Ones



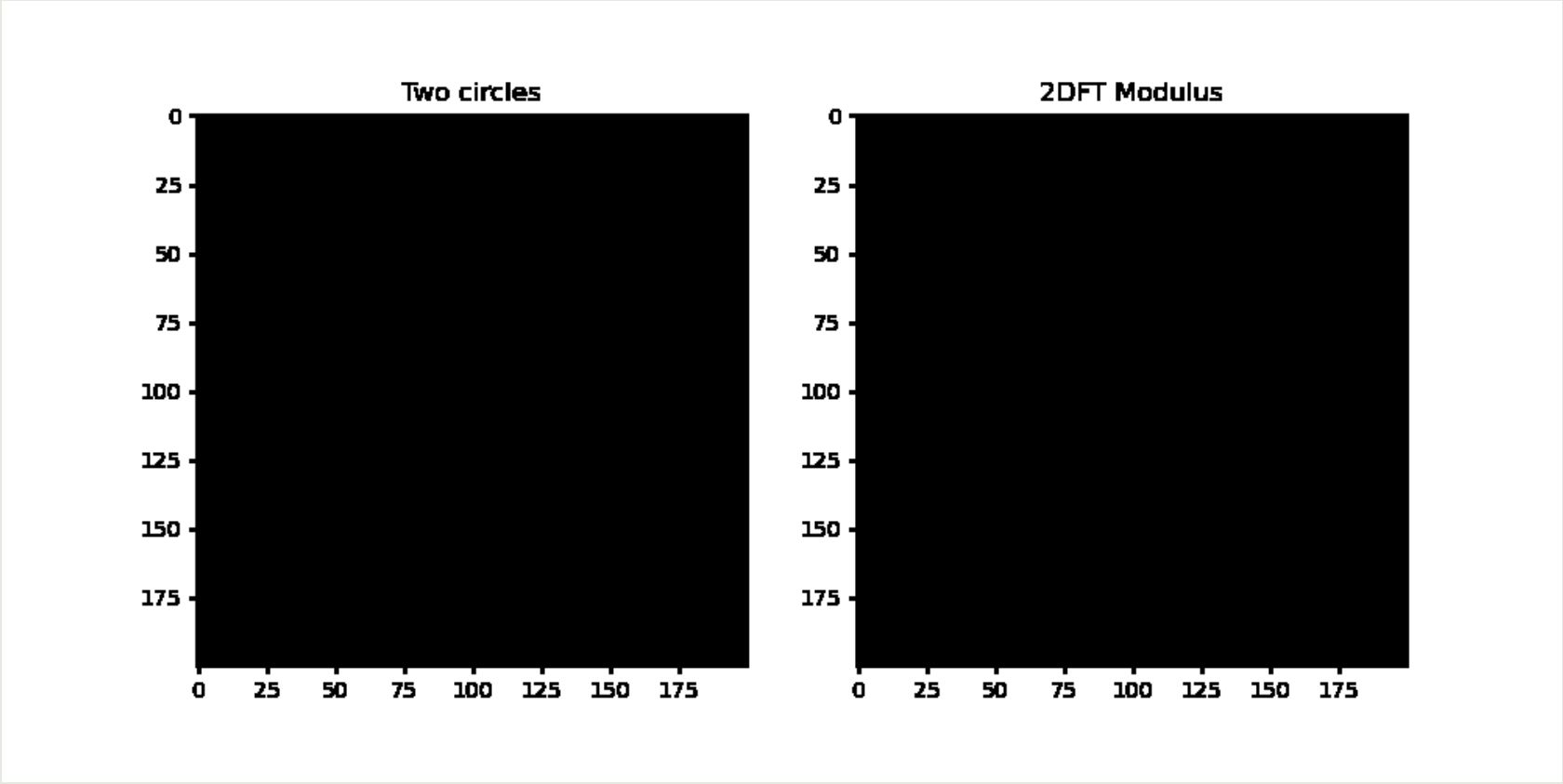
Recall that each pair of point in the 2D FFT corresponds to a corrugated roof (sinusoid) with a particular frequency in the original image. The reverse is also true due to the invertibility of Fourier transform: We observe gratings on the 2D FFT modulus if we have equally spaced dots on the original image.

Equally-spaced Ones

```
def ones_cross(spacing, N=200):
    r= np.arange(0,N, spacing)
    a = np.zeros((N,N))
    a[N//2, r]=1 #x-axis
    a[r, N//2]=1 #y-axis
    return a
```

Here's a simple code to create equally-spaced points on the axes. We first assign a value of 1 on the points in the x-axis, then on the y-axis.

Circles



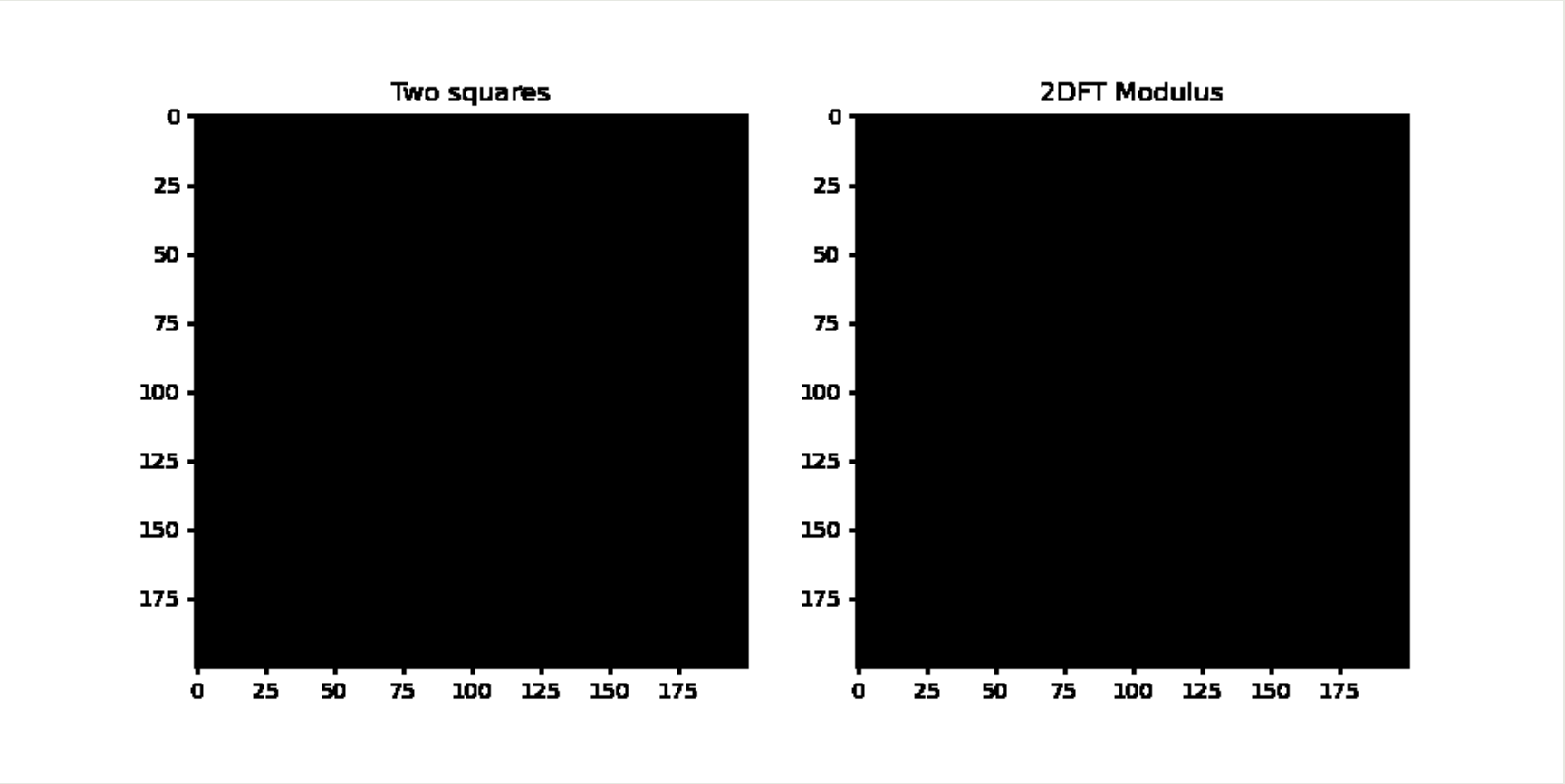
Note that the resulting FFT image of two circles look like a superimposition of an Airy disk and a corrugated roof (2D sinusoid). Moreover, at small radii, the Airy diffraction pattern is more evident, and more of the corrugated roof pattern can be seen (not surprising, as the circle appear more as points). The corrugated roof appears due to the symmetry of the original image. It can also be seen that the “intensity” maximum appears to shrink as the radii 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 (see report on Module 2).

Circles

Here's the code for the generation of n symmetric circles

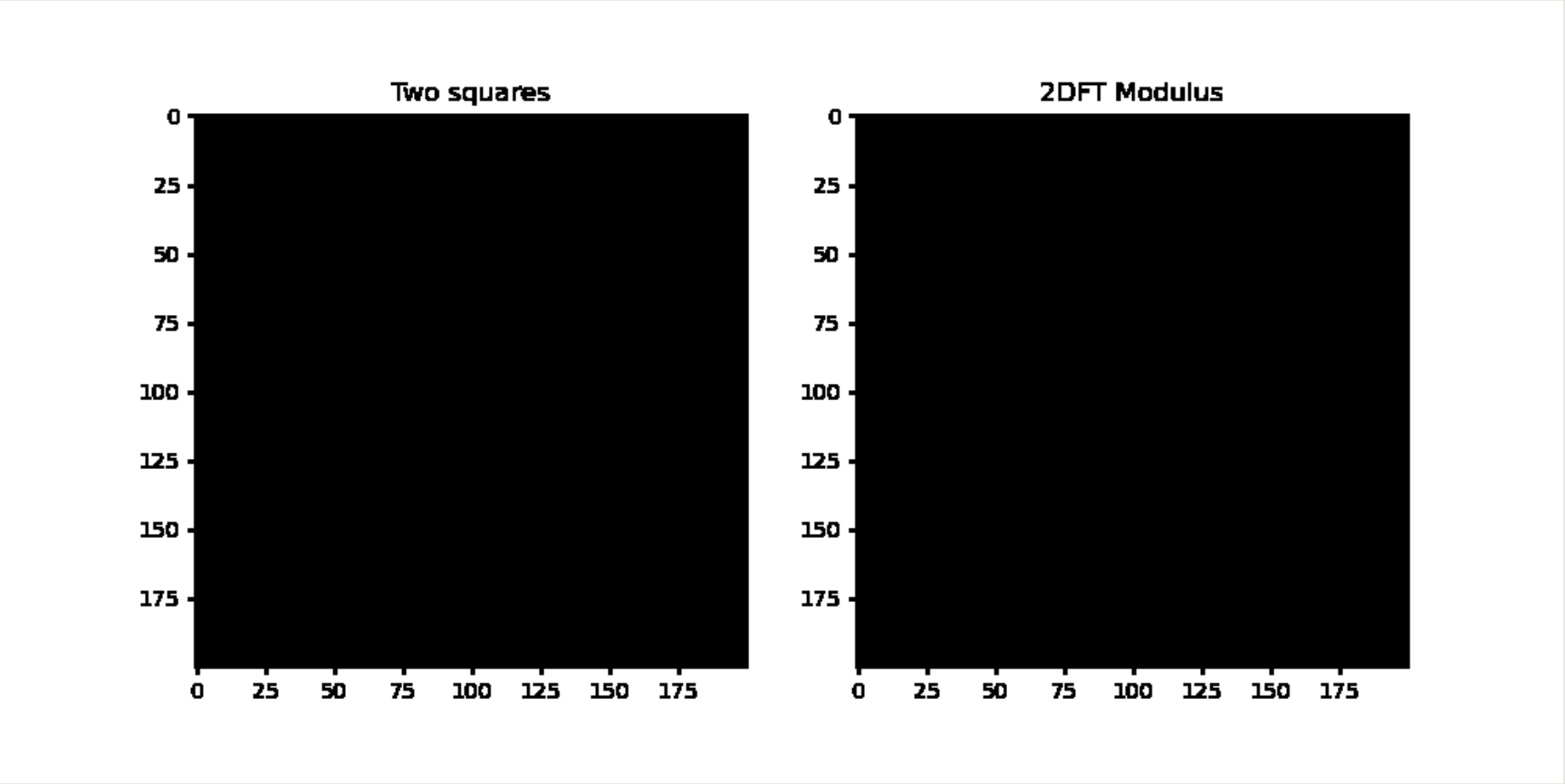
```
def circles(number=2, D=0.1, N=100):
    x = np.linspace(-1,1,num = N)
    y= np.linspace(-1,1,num=N)
    x= y
    radius = D/2
    X,Y = np.meshgrid(x,y)
    cen = []
    if number%2 == 0: #If even number of circles, include the circle at the origin
        for i in range(1,number//2 +1):
            cen.append((0,i/number))
            cen.append((0, -i/number))
    if number%2 == 1: #If odd number circles, do not include circle centered at origin
        for i in range(1,number//2 +1):
            cen.append((0,i/number))
            cen.append((0, -i/number))
        cen.append((0,0))
    A = np.zeros((N,N))
    for c in cen:
        R = np.sqrt((X-c[1])**2 + (Y)**2) #Setup an offcenter circle.
        A[np.where(R<radius)] = 1.0
    return A
```

Squares



The same case can be said for two squares. Since they are symmetric on the x-axis, we expect a sinusoidal grating to appear in the 2D FFT modulus. Moreover, the associated diffraction pattern through a square aperture will also be observed in the FFT image.

Squares

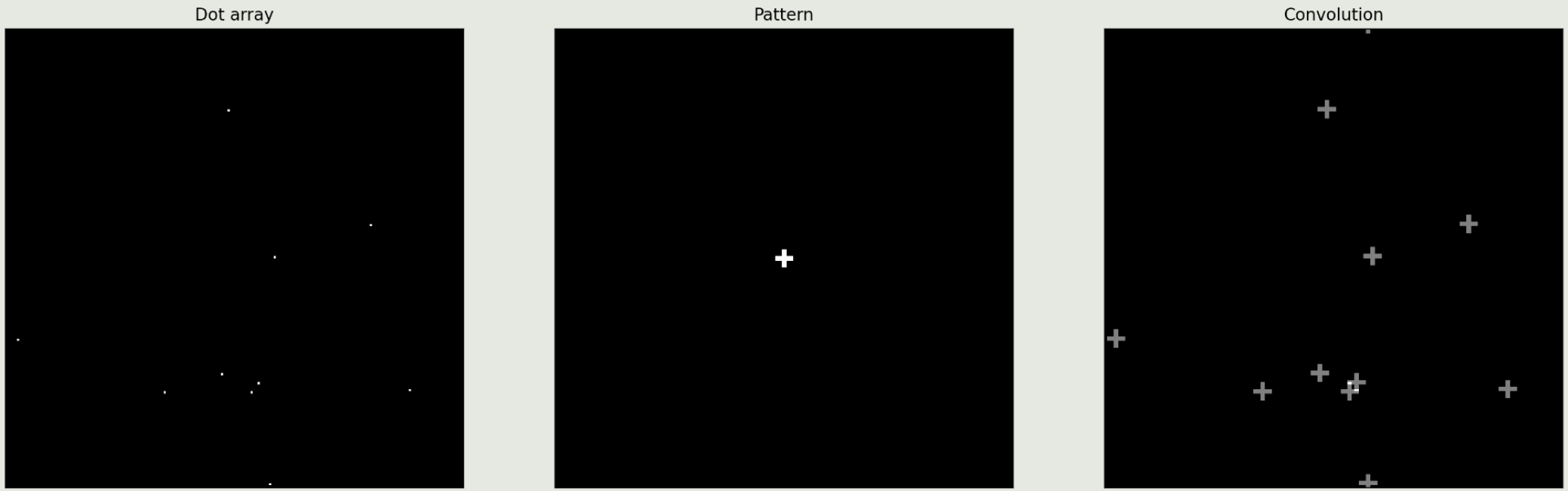


The same case can be said for two squares. Since they are symmetric on the x-axis, we expect a sinusoidal grating to appear in the 2D FFT modulus. Moreover, the associated diffraction pattern through a square aperture will also be observed in the FFT image.

Objectives

- Investigate the effect on the 2D FFT of a 2D sinusoid with varying phase and frequency.
- Investigate the effect on the 2D FFT as multiple sinusoids are superimposed.
- Perform image processing by filtering in the Fourier space.
- Investigate the corresponding FFT of geometric shapes with varying properties.
- Replicate a pattern using image convolution.

Binarized Dots



The convolution of a pattern to a dot array (representing Dirac-delta functions) results in the replication of the pattern at the location of the dots. Indeed, if f(x,y) represents the pattern array, then:

$$\int \delta(x - x_0 - x', y - y_0 - y') f(x', y') dx' dy' = f(x - x_0, y - y_0)$$

"Dirac-Delta" Dots and Cross Pattern

```
def dots_array(n=10, N=200) #Creates a NxN zero array with n random 1s interspersed
    a=np.zeros((N,N))
    for i in range(n):
        a[np.random.randint(N), np.random.randint(N)] = 1
    return a

def cross(width=1, length=9, N=200): #arbitary 9x9 pattern
    vals = np.zeros((N,N))
    min_x, max_x= int(N//2-width), int(N//2+width)
    min_y, max_y = int(N//2-np.floor(length/2)), int(N//2+np.floor(length/2))
    vals[min_x:max_x, min_y:max_y] =1
    vals[min_y:max_y, min_x:max_x] =1
    return vals
```

We emphasize that both the convolving aperture (pattern) and the original array (dots) must have the same size, similar to the correlation and pattern-matching we've done on the previous module.

Binarized Dots

```
dirac_deltas = dots_array(10, N=200)
pattern=cross(N=200)

pattern_fft2d = np.fft.fftshift(np.fft.fft2(pattern))
dots_array_fft2d = np.fft.fftshift(np.fft.fft2(dirac_deltas))

convolved = dots_array_fft2d*pattern_fft2d
deconvolved = np.fft.fftshift(np.fft.ifft2(convolved))
```

The final image is the inverse FFT of FG, where F is the FFT of the dot array and G is the FFT of the pattern or convolving aperture. We restore the correct orientation by performing fftshift on the final image.

Reflection

- Honestly, I think I could've done better on tweaking the (monster) algorithm given more time. Nevertheless, I think that was the highlight of this module for me, since I got to actually apply some machine learning. Also, special thanks to ChatGPT for my buddy: although you were somewhat unreliable in writing error-free programs, you are a gold mine of insights (like raw, unpolished gemstones).
- The mathematics and the resulting geometry of the 2D FFT is basically a rehash of what I established in the previous module, so explaining things were sort of easy.
- I could've done better on the manual masking (use crosses, just like what Nino Ramones did).

Self-score: 110

References and Acknowledgments

- Andreas Muller, Introduction to Machine Learning with Python.
- ChatGPT (no, I didn't just copy :p as I've explained in my reflection)
- Special thanks to Nino Ramones for helping me with the ridge enhancement.
- https://www.lpi.usra.edu/publications/slidesets/apollolanding/ApolloLanding/slide_05.html (Moon picture)
- Dr. Vincent Daria's painting (provided by Dr. Soriano)
- https://www.researchgate.net/figure/Examples-of-different-classes-of-fingerprints-a-right-loop-b-whorl-and-c-arch_fig2_265986190 (fingerprint)