3rd Year Computational Physics : Jessica Murphy

02/04/2024

Assignment 8

This python notebook investigates Image Analysis through an introduction to Fourier Transforms. The notebook assists the learning and understanding of how *Imaging Systems* work and how linear image systems can be described by their response to a poit source, or *Point Spread Function (PSF)*. The tasks involve investigating;

- Computing Fourier Analsyis
- Applying a Gaussian Filter
- Attempting Inverse Foruier Transform

Essentially it involves taking the Fourier Transform of objects and images in the *Spatial Domain*, to obtain Fourier transforms in the *Spatial Frequency* domain.

Background Information: The ideal PSF would itself be infinitely narrow; i.e., the response to a delta function (a point source) would itself be a delta function (another point). However, the PSF is always broadened because of various effects, the most important of which are diffraction at the system aperture and optical aberrations in the system.

If the PSF of an imaging system is known, $P(x)$, then the output image $I(x)$ corresponding to any object $O(x)$ can be obtained by a Convolution Integral in spatial domain:

$$I(x) = \int O(x')P(x-x')dx$$

or

$$I(x) = O(x) \circledast P(x)$$

where the $\circledast$ denotes *Convolution*. For simplicity this is written in one-dimension only here. The effect of convolution is to blur the image of the object. In principle, we can directly measure the PSF; e.g., by obtaining an image of a point source. Using knowledge of the PSF, we can *Deconvolve* or *Deblur* any image. Because of the complicated operation of convolution in the Space Domain, this is usually done in the Frequency (Fourier) Domain as the process turns into simple multiplication, i.e.

$$FTI(f) = FTO(f) \cdot FTP(f)$$

where FTI is the Fourier transform of the image, FTO is the Fourier transofrm of the object, and FTP is the Fourier transform of the PSF.

Therefore, by knowing the PSF, *Deconvolution* is done by dividing the FT of the image by the FT of the PSF and inverse transforming;

$$O(x) = FT^{-1}\left[\frac{FTI(f)}{FTP(f)}\right]$$

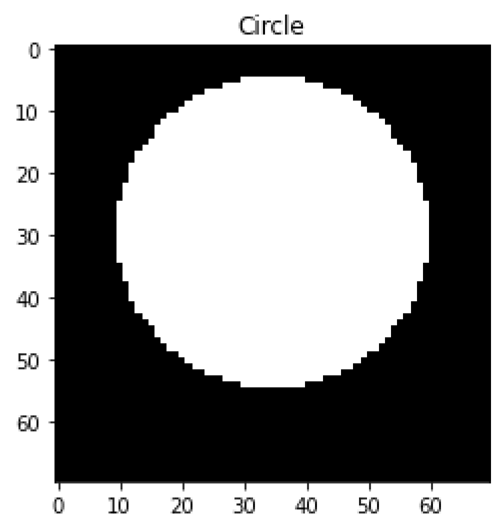
Task One:

The objective of task one is to write a python program that outputs a simple shape. It introduces image processing and analsyis techniques using the python library PIL (Pyhton Imaging Library).

1. Import necessary classes from the PIL library for image creation and drawing, and the matplotlib.pyplot module for displaying images.
2. Set up the width, height, and bounding box for the shape to be drawn.
3. Create a new image in 32-bit signed integer mode with a specified size.
4. Initiate a drawing context for the image.
5. Draw a pie slice using the PIL library.
6. The matplotlib library is used to display the image with a title. The show() method from PIL is commented out in favor of matplotlib's imshow() for a more detailed and controlled visualization.

In [1]:
```python
from PIL import Image, ImageDraw
import matplotlib.pyplot as plt

# Define the width and height of the image
w, h = 70, 70
# Define the shape's bounding box
shape = [(20, 40), (50, 20)]

# Create a new image with 'I' mode for 32-bit signed integer pixels and the specified size
im = Image.new('I', (w, h))
# Create a drawing context for the image
img = ImageDraw.Draw(im)

# The circle method is not directly available, you might want to use ellipse for drawing a circle-like shape
# Draw an ellipse/circle on the image. The parameters for ellipse are two points to define the bounding box
```

```
img.pieslice(shape, start=3, end=5, fill='blue', outline='white', width=50)
# img.arc(shape, start=5, end=50, fill='blue', width=100)

# Display the image
#im.show()
plt.imshow(im)
plt.title('Circle')
plt.show()
```



A simple white circle is drawn

Task Two:

The objective of task two is to perform a Fourier analysis on one of the two images drawn. The image picked was the one that has a simple pattern, like the one above.

Background Information: A Gaussian Filter is used in reducing noise in the image, and also the details of the image. The Gaussian function based on the size of σ (standard deviation) is defined as:

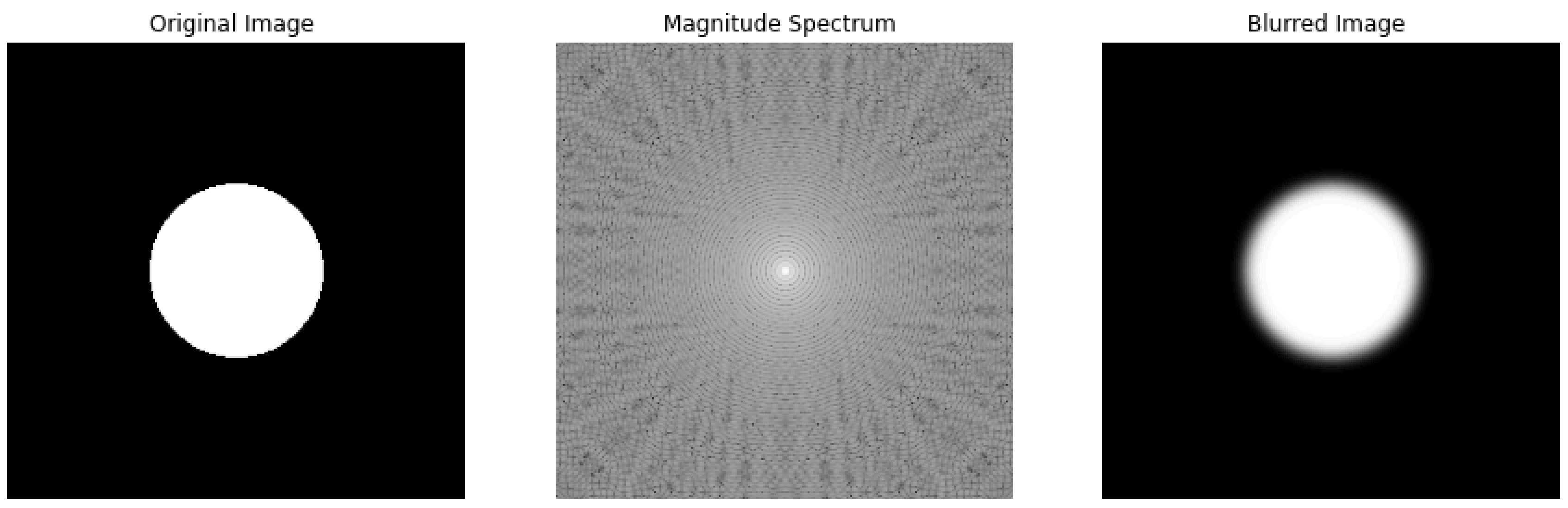
$$G_\sigma = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

1. Firstly in a cell, import the numpy package, and from this import the functions that focus on the fourier transform, a function to shift the zero-frequency component to the centre of the spectrum. These are fft2, fftshift, and ifft2 respectively.
2. Then define a function to create a Gaussian filter to apply in the frequency domain. This is a common operation in image processing for blurring or reducing detail. Then assign a variable to apply the Gaussian filter.
3. Now in a new cell, create a solid colour image and an image with a simple pattern
4. Now using the image witha simple pattern only (image 2), convert the image to a numpy array for image processing
5. Perform a 2D Fourier transform to shift the zero-frequency component to the centre.
6. Apply a Gaussian filter to blur the image in the frequency domain.
7. Take the inverse Fourier transform to convert the image back to the spatial domain.
8. Convert the numpy array back to PIL Image for visualisation
9. Use matplotlib to visualise the original image, the magnitude spectrum, and the blurred image.

In [2]:
```
# task 2 perform a fourier analysis on the images
## from PIL import Image, ImageDraw
import numpy as np
from numpy.fft import fft2, fftshift, ifft2

# Function to create a Gaussian filter
def make_gaussian_filter(size, sigma=10):
    ax = np.arange(-size // 2 + 1., size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = (1/(2 * np.pi * sigma**2)) * np.exp(-(xx**2 + yy**2) / (2. * sigma**2))
    return kernel / np.sum(kernel)

# Apply Gaussian filter
gaussian_filter = make_gaussian_filter(256, sigma=10)
```

In [3]:
```
# Define the first image with a solid colour
img1 = Image.new('L', (256, 256), 'black')
draw1 = ImageDraw.Draw(img1)
draw1.rectangle([80, 80, 176, 176], fill='white')

# Define the second image with a simple pattern (circle in the center)
img2 = Image.new('L', (256, 256), 'black')
draw2 = ImageDraw.Draw(img2)
draw2.ellipse([80, 80, 176, 176], fill='white')

# Convert the second image to a numpy array for processing
image_array2 = np.array(img2)

# Perform the 2D Fourier transform and shift
f_transform2 = fft2(image_array2)
f_shift2 = fftshift(f_transform2)
```

```python
# Apply Gaussian filter
f_shift_filtered2 = f_shift2 * gaussian_filter

# Inverse Fourier transform to get the image back in spatial domain
f_ishift2 = np.fft.ifftshift(f_shift_filtered2)
img_back2 = np.fft.ifft2(f_ishift2)
img_back2 = np.abs(img_back2)

# Visualization with matplotlib
plt.figure(figsize=(15, 5))

# Original image visualization
plt.subplot(1, 3, 1)
plt.imshow(image_array2, cmap='gray')
plt.title('Original Image ')
plt.axis('off')

# Magnitude spectrum of the Fourier transformed image
magnitude_spectrum2 = 20 * np.log(np.abs(f_shift2))
plt.subplot(1, 3, 2)
plt.imshow(magnitude_spectrum2, cmap='gray')
plt.title('Magnitude Spectrum ')
plt.axis('off')

# Blurred image visualization
plt.subplot(1, 3, 3)
plt.imshow(img_back2, cmap='gray')
plt.title('Blurred Image ')
plt.axis('off')

plt.show()
```



A simple white circle is created, a Gaussian filter is applied to the image to show its magnitude sopectrum, and then the inverse transform of the image is taken to displayits blurred image.

Task Three:

Task three delves into formatting the code so that the Fourier analsyis is contained in a function that can be applied to any image that is being analysed. Essentially it is written in a way to encapsulate the process of applying Foruier analysis to an image, applying a Gaussian filter, and then taking an inverse fourier transfrom to return a blurred image. By defining a function you can easily apply the same process to different images without rewriting the code each time.

1. Define the function apply_fourier_analysis
   - it takes a PIL image and a sigma value for the Gaussian filter as inputs, * it returns a numpy array representing the blurred image after applying Fourier analysis.
2. The make_gaussian_filter function is used within the main function to create the Gaussian filter used for blurring.

In [4]:
```python
# task 3 - formatting fourier code - i.e. defining a function
def apply_fourier_analysis(image, sigma=10):
    """
    Applies Fourier analysis to an image, applies a Gaussian filter, and returns the blurred image.

    Parameters:
    - image: A PIL.Image object.
    - sigma: Standard deviation for the Gaussian filter.

    Returns:
    - A numpy array representing the blurred image.
    """
    # Convert the image to a numpy array
    image_array = np.array(image)

    # Perform the 2D Fourier transform and shift the zero-frequency component to the center
    f_transform = fft2(image_array)
    f_shift = fftshift(f_transform)

    # Apply Gaussian filter in the frequency domain
    gaussian_filter = make_gaussian_filter(image_array.shape[0], sigma=sigma)
    f_shift_filtered = f_shift * gaussian_filter

    # Inverse Fourier transform to get the image back in spatial domain
    f_ishift = ifftshift(f_shift_filtered)
    img_back = ifft2(f_ishift)
    img_back = np.abs(img_back)

    return img_back
```

Task Four:

The aim of task four is to display the original image of a cat (from cat.jpg) alongside its magnitude spectrum after applying a Fourier transform.

It should be noted that the image was converted to greyscale to simplify the process, as it ensures that we are working with a single channel of pixel intensities.

It should be expected that the magnitude spectrum reveals the frequency components of the image, with brighter areas indicating higher magnitudes.

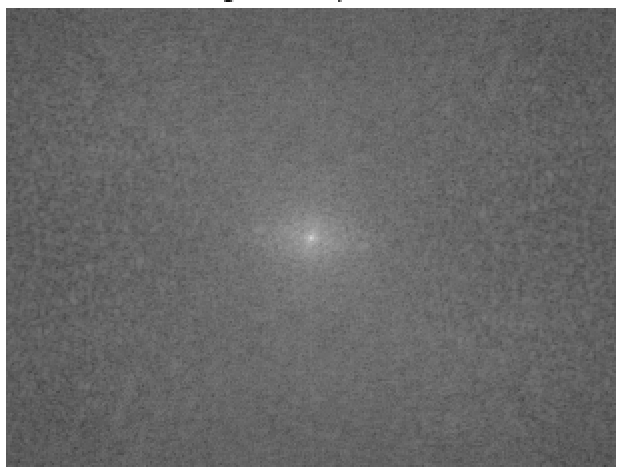
1. Load an image
    - use PIL to do this,
    - display it in its original and greyscale form using matplotlib.pyplot.
2. Utilising numpy for array manipulation and numpy.fft for Fourier transform operations, perform a 2D Discrete Fourier Transform (DFT) on the greyscale image to obtain the frequency representation.
3. Shift the zero-frequency component to the centre.
4. Apply logarithmic compression to enhance visibilty.
5. Display both the original image, greyscale image, and magnitude spectrum using matplotlib

In [5]:
```
# task 4
## from PIL import Image
from IPython.display import display

# Load the image
image_path = 'cat.jpg'
image1 = Image.open(image_path)
image = Image.open(image_path).convert('L')  # Convert to grayscale

# Display the original image
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image1)
plt.title('Original Image')
plt.axis('off')

# Display the original image
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.title('Greyscale Image')
plt.axis('off')

# Compute the 2D Discrete Fourier Transform
f_transform = fft2(np.array(image))

# Shift the zero-frequency component to the center
f_shift = fftshift(f_transform)

# Take the log of the spectrum to compress the range of values
magnitude_spectrum = 20 * np.log(np.abs(f_shift))

# Display the Magnitude Spectrum image
plt.subplot(1, 2, 2)
plt.imshow(magnitude_spectrum, cmap='gray')
plt.title('Magnitude Spectrum')
plt.axis('off')

plt.show()
```

Original Image

Greyscale Image

Magnitude Spectrum



The output is deemed satisfactory as a resulting image is shown that resembles the one required in the manual.

Task Five:

The challenge of task five is to reconstruct and display the original image of 'mag_spec.jpg' given only its magnitude spectrum. Comments are made throughout each attempt to describe what is being shown.

It should be noted that reconstructing an original image from its magnitude spectrum alone, without phase information, is generally not possible because the phase contains crucial information about how the pixel values are arranged spatially. The magnitude spectrum gives us the amount of each frequency present in the image, but the phase tells us how these frequencies are oriented and combined to form the original image. Without phase information, any reconstruction will be at best a rough approximation, often resulting in a significantly different image from the original.

Attempt One:

Firstly since there is no phase information given, a unifrom phase (phase=0) is assumed for all frequencies, combining the magnitude with this phase to form a complex spectrum. An inverse Fourier Transfrom is then attempted. An approximation of the reconstructed image is then displayed.

1. Load the image in and convert the magnitude spectrum to a numpy array as before.
2. Exponentiate the array to get back to the magnitude since we are dealing with a log magnitude spectrum.
3. Assume a uniform phase of zero.
4. Combine magnitude and phase to form a complex spectrum.
5. Shift the zero-frequency component back to the original position.
6. Perform the inverse Fourier transform.

In [6]:
```
# task 5
# attempt 1
from numpy.fft import ifft2, ifftshift

# Load the magnitude spectrum image
mag_spec_path = 'mag_spec.jpg'
mag_spec_image = Image.open(mag_spec_path).convert('L')  # Convert to grayscale

# Convert the magnitude spectrum image to a numpy array
mag_spec_array = np.array(mag_spec_image)

# Since we're working with a log magnitude spectrum, we first exponentiate to get back to the magnitude
magnitude = np.exp(mag_spec_array / 20)

# Assume a uniform phase of zero
phase = np.zeros(magnitude.shape)

# Combine magnitude and phase to form a complex spectrum
complex_spectrum = magnitude * np.exp(1j * phase)

# Shift the zero-frequency component back to the original position
complex_spectrum_shifted = ifftshift(complex_spectrum)

# Perform the inverse Fourier transform
reconstructed_image = ifft2(complex_spectrum_shifted)
reconstructed_image_abs = np.abs(reconstructed_image)

# # Inverse Fourier transform to get the image back in spatial domain
# f_ishift3 = np.fft.ifftshift(magnitude)
# img_back3 = np.fft.ifft2(f_ishift3)
# img_back3 = np.abs(img_back3)

# Display the "reconstructed" image
plt.figure()
plt.imshow(magnitude, cmap='gray')
plt.title('Attempt One of Reconstructed Image')
plt.axis('off')
plt.show()
```

Attempt One of Reconstructed Image

Given the limitations, the result is as expected. A little white dot is shown in the middle of the picture, but basically a non-discernable picture is shown, as a result of the loss of phase information, which is critical for accurately reconstructing the spatial arrangement of pixels.

In [7]:
```
# # Convert the image to a numpy array for processing
# # task 5
# from numpy.fft import ifft2, ifftshift

# # Load the magnitude spectrum image
# mag_spec_path = 'mag_spec.jpg'
# mag_spec_image = Image.open(mag_spec_path).convert('L')  # Convert to grayscale

# # Convert the magnitude spectrum image to a numpy array
# mag_spec_array = np.array(mag_spec_image)

# # Inverse Fourier transform to get the image back in spatial domain
# f_ishift3 = np.fft.ifftshift(mag_spec_array)
# img_back3 = np.fft.ifft2(f_ishift3)
# img_back3 = np.abs(img_back3)

# # Visualization with matplotlib
# plt.figure(figsize=(15, 5))

# # Display the "reconstructed" image
# plt.figure()
# plt.imshow(img_back3, cmap='gray')
# plt.title('Reconstructed Image (Approximation)')
# plt.axis('off')
# plt.show()
```

Because the problem is the lack of phase information, it was pondered if there was a way to iterate through phase information until a suitable image is created. After research on this topic, there appears to be some potential algorithms that could be tried. However, this idea proved to be practically challenging as seen below. This is because the space of possible phase configurations is vast and the relationship between phase information and spatial image quality is complex and non-linear. Even slight changes to the phase can result in significant alterations to the reconstructed image.

Alas, a **phase retrieval** approach was attempted anyway, i.e. an attempt to iterate over possible phase configurations and use an optimisation approach to search for a phase pattern that yields a reconstructed image closest to some desired criteria. A phase retrieval algorithm is ventured below, however these algprithms usually rely on additional constraints on prior knowledge about the image.

There is practical challenges to these methods.

1. There is a hige search space. For an image size of NxN, there are N^2 phase values that could take any value between 0 and 2π. This makes the search space effectively infinite.
2. There is a lack of a clear objective. Without a clear, quantifiable measure of what makes an image "suitable," it's difficult to guide the search process. Additional information of the image can help define this measure.
3. Computational complexity is also a factor. Iterating through these possible phase values and performing an inverse Fourier transform for each is computationally intensive.

In regards to these challenges, there is some possible approaches.

- Gradient Descent and Optimization Algorithms - uses optimisation techniques to minimise the difference between the observed magnitude spectrum and the magnitude spectrum of an image with guessed phase information.
- Hybrid Input-Output (HIO) and Error Reduction (ER) Algorithms - these work by iteratively applying constraints in both the spatial and frequency domains. This algorithm is attempted below.
- Deep Learning - resent approaches have seen models predict phase information from magnitude spectra or to learn a mapping directly from magnitude spectra to spatial images.

For the purposes of this assignment, the HIO and ER algorithm was attempted. However after even more research, there appears to be extensive examination into enhanced models of this. For example the Hybrid Input-Output with Randomized Overrelaxation (HIO+OR) followed by Error Reduction (ER) algorithm, aka ((HIO+OR)+ER), is a method that is useful for reconstructing phase infprmation from the magnitude of a Fourier transform where direct phase measurement is difficult or impossible, which is applicable in this scenario.

The HIO algorithm is a well-established iterative method for phase retrieval that alternates between the spatial and Fourier domains, applying constraints in each domain to gradually refine the estimate of the phase. The inclusion of overrelaxation (OR) techniques aims to improve the convergence of the HIO algorithm by adjusting the update step size based on the progress of the solution. This can help escape local minima and potentially lead to a faster and more accurate reconstruction. After applying HIO+OR for a set number of iterations to get close to a potential solution, the process is often followed by Error Reduction (ER) iterations. ER is a simpler and more stable algorithm that can fine-tune the solution by further reducing discrepancies between the measured and calculated magnitudes, without risking divergence or introducing artifacts that might occur with prolonged use of HIO+OR.

A link to this research is the following:

https://opg.optica.org/oe/fulltext.cfm?uri=oe-20-15-17093&id=239803

Attempt Two:

So, a simplified conceptual outline in python code is written to illsutrate how this problem could be approached.

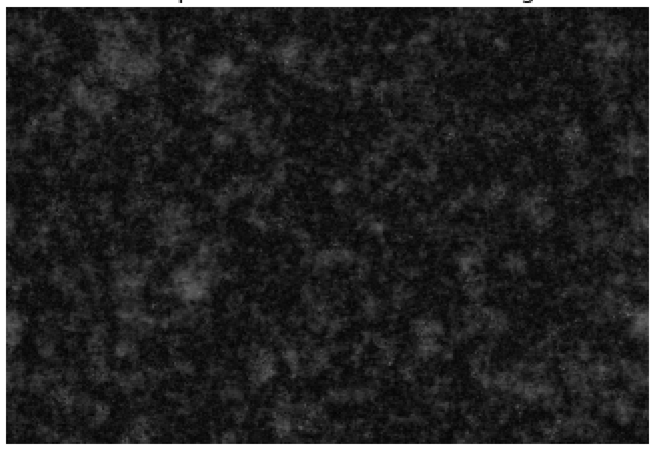
1. Define a function to embed the (HIO+OR)+ER algorithm.
2. Create a for loop to go through the set number of iterations.
    - Write an if statement to apply a non-negativity constraint.
    - Write an else statement for the HIO step with Overrelaxation.
3. Return the final image.
4. In a new cell use matplotlib to display the image.

In [8]:
```
# phase retrieval methods
# (HIO+OR)+ER-algorithm # is based on randomized overrelaxation
```

In [9]:
```
# Function to define the (HIO+OR)+ER-algorithm as a phase retrieval method
# The beta parameter controls the overrelaxation in the HIO step, which is crucial for the algorithm's performance
# The er_every_n parameter determines how frequently to perform
#an Error Reduction step instead of HIO, to stabilise the solution
# This is more for demonstration purposes and how to go about solving a problem like this,
#actual implementation should include a support constraint

def hio_er(magnitude, iterations=1000, beta=0.9, er_every_n=10):
    """
    A simplified version of the HIO+ER algorithm for phase retrieval.

    Parameters:
    - magnitude: The magnitude of the Fourier transform (assumed to be a 2D numpy array).
    - iterations: Total number of iterations to run.
    - beta: Overrelaxation parameter for HIO.
    - er_every_n: Perform Error Reduction (ER) after every n HIO iterations.

    Returns:
    - The reconstructed image (as a 2D numpy array).
    """
    # Initialize random phase
    phase = np.exp(2j * np.pi * np.random.rand(*magnitude.shape))

    # Combine magnitude with random phase to start
    complex_estimate = magnitude * phase

    # Initialize support constraint (e.g., image is non-zero within certain region)
    # For simplicity, this example does not apply any specific support constraint

    for i in range(iterations):
        # Inverse Fourier Transform to get back to spatial domain
        image_estimate = np.fft.ifft2(complex_estimate)

        # Apply constraints in the spatial domain
        # For example, here we might apply non-negativity constraint as a placeholder
        # More sophisticated constraints based on prior knowledge could be used
        if i % er_every_n == 0:
            # Error Reduction step
            image_estimate[image_estimate < 0] = 0
        else:
            # Hybrid Input-Output step with Overrelaxation
            updated_phase = np.angle(np.fft.fft2(image_estimate))
            complex_estimate = magnitude * np.exp(1j * updated_phase)
            # Apply HIO formula with beta parameter for pixels violating the constraints
            # This would require defining a 'support' mask indicating where the object is expected to be
            # image_estimate[outside_support] = previous_estimate[outside_support] - beta * image_estimate[outside_support]

        # Update complex estimate for next iteration
        complex_estimate = np.fft.fft2(image_estimate)

# Final image reconstruction
    final_image = np.abs(np.fft.ifft2(complex_estimate))
    return final_image
```

In [10]:
```
# plt.imshow(final_image, cmap='gray')

# Call the function and store the result in final_image
final_image = hio_er(magnitude)

# ADD TITLES & AXES

# Now you can display it
plt.imshow(final_image, cmap='gray')
plt.title('Attempt Two of Reconstructed Image')
plt.axis('off')
plt.show()
```

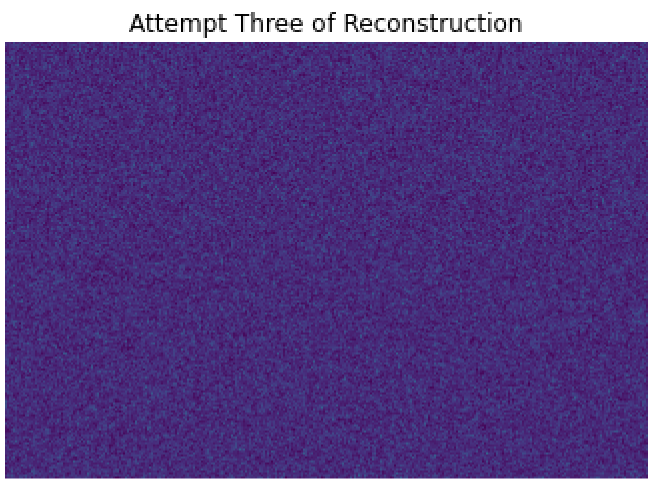
Attempt Two of Reconstructed Image



There is slightly more being shown here than in the last, but still a non-discernable picture is shown.

In [11]:
```
# The algorithm is rewritten to sample different different values for the parameters;
#namely the iterations, overrelaxation(beta) and error reduction(er_every_n)

def hio_er2(mag_spec_array, iterations=900, beta=0.8, er_every_n=6):
    """
    A simplified version of the HIO+ER algorithm for phase retrieval.

    Parameters:
    - magnitude: The magnitude of the Fourier transform (assumed to be a 2D numpy array).
    - iterations: Total number of iterations to run.
    - beta: Overrelaxation parameter for HIO.
    - er_every_n: Perform Error Reduction (ER) after every n HIO iterations.

    Returns:
    - The reconstructed image (as a 2D numpy array).
    """
    # Initialize random phase
    phase = np.exp(2j * np.pi * np.random.rand(*mag_spec_array.shape))

    # Combine magnitude with random phase to start
    complex_estimate = mag_spec_array * phase

    # Initialize support constraint (e.g., image is non-zero within certain region)
    # For simplicity, this example does not apply any specific support constraint
    # support = ...

    for i in range(iterations):
        # Inverse Fourier Transform to get back to spatial domain
        image_estimate = np.fft.ifft2(complex_estimate)

        # Apply constraints in the spatial domain
        # For example, here we might apply non-negativity constraint as a placeholder
        # More sophisticated constraints based on prior knowledge could be used
        if i % er_every_n == 0:
            # Error Reduction step
            image_estimate[image_estimate < 0] = 0
        else:
            # Hybrid Input-Output step with Overrelaxation
            updated_phase = np.angle(np.fft.fft2(image_estimate))
            complex_estimate = mag_spec_array * np.exp(1j * updated_phase)
            # Apply HIO formula with beta parameter for pixels violating the constraints
            # This would require defining a 'support' mask indicating where the object is expected to be
            # image_estimate[outside_support] = previous_estimate[outside_support] - beta * image_estimate[outside_support]

        # Update complex estimate for next iteration
        complex_estimate = np.fft.fft2(image_estimate)

# Final image reconstruction
    final_image = np.abs(np.fft.ifft2(complex_estimate))
    return final_image
```

In [12]:
```
# plt.imshow(final_image, cmap='gray')

# Call the function and store the result in final_image
final_image = hio_er2(mag_spec_array)

# Now you can display it
plt.imshow(final_image)
plt.title('Attempt Three of Reconstruction')
plt.axis('off')
plt.show()
```



Alas, a non-discernable picture is shown again.

Discussion

While HIO+OR+ER provides a theoretical framework for tackling the challenge of reconstructing an image from its magnitude spectrum, the practical success of such an endeavor would require a robust implementation of the algorithm, careful tuning, and perhaps most critically, realistic expectations about the fidelity of the reconstructed image to the original.

For further imporvement, a consideration of hybrid approaches or the use of such algorithms along with deep learning techniques could prove useful for further improvement.

What was learned from this assignment is that Gaussian blur in image processing reduces noise by smoothing fine details, corresponding to attenuating high spatial frequencies in the Fourier domain, while preserving low spatial frequencies maintains the image's overall structure but may result in a loss of fine detail. This balance between high and low frequencies is crucial for tasks like reconstructing an image from its magnitude spectrum, where decisions on filtering and emphasis can significantly impact the clarity and recognisability of the resulting image.