

Phase 5: The Return Trip (Reconstruction)

Modules Covered:

- core/fft_engine.py (Inverse Transform)
- core/color_processing.py (Post-processing & Merging)
- gui/utils.py (Display Conversion)

Goal: Once the High-Boost mask has been applied in the frequency domain, the data exists as abstract complex numbers. This phase explains how we convert those numbers back into visible pixels (Spatial Domain) and render them on the user's screen.

1. The Inverse Transform (fft_engine.py)

After multiplying the image spectrum F by the mask H_b , we get the filtered spectrum G . To see the image, we must perform the Inverse FFT.

A. Inverse Shifting

Before calling the inverse FFT function, we must reverse the "Shift" operation we did in Phase 3. The standard IFFT algorithm expects the DC component at (0,0), not in the center.

```
def ifft_shift(G_shifted):  
    # Move DC component back from Center -> Top-Left Corner  
    return np.fft.ifftshift(G_shifted)
```

B. Computing the IFFT

Now we compute the Inverse Discrete Fourier Transform.

```
def compute_ifft(G):  
    # 1. Compute the complex inverse transform  
    img_back_complex = np.fft.ifft2(G)  
  
    # 2. Discard the Imaginary Part  
    # In theory, if our filter was symmetric, the imaginary part  
    # should be 0.  
    # In practice, tiny floating-point errors (e.g., 1e-15)  
    # might exist.  
    # We explicitly take the Real component to get a viewable  
    # image.  
    return np.real(img_back_complex)
```

Critical Note: This step is where "Hermitian Symmetry" matters. If our mask wasn't symmetric (as ensured in Phase 4), `img_back_complex` would have large imaginary components, and taking `np.real()` would result in a distorted image.

2. Post-Processing (color_processing.py)

At this stage, the pixel values are floating-point numbers. They might be negative (undershoot at edges) or greater than 255 (overshoot due to boosting). We need to fix this.

A. Clipping and Casting

We cannot display a pixel with value -20.5 or 300.0 .

```
def _clip_and_cast(arr, dtype, vmin, vmax):
    # 1. Clip values to the valid range (e.g., 0 to 255)
    # Any value < 0 becomes 0. Any value > 255 becomes 255.
    out = np.clip(arr, vmin, vmax)

    # 2. Cast back to original data type (usually uint8)
    # This rounds floats like 120.6 to integers like 121.
    return out.astype(dtype)
```

B. Merging Channels

If the input was a color image, we now have three separate 2D arrays (Red, Green, Blue) that have been processed individually. We must stack them back into a single 3D image.

```
# 'chans_out' is a list: [Red_Processed, Green_Processed,
                        Blue_Processed]

# Stack them along the "depth" axis (axis 2)
# Result shape: (Height, Width, 3)
stacked = np.stack(chans_out, axis=2)

return stacked
```

3. Rendering to Screen (gui/utils.py)

The final step is to show this NumPy array in the Tkinter window. Tkinter cannot display NumPy arrays directly; it needs a specific image object.

```
def np_to_tkimage(arr):
    # 1. Convert NumPy Array -> Pillow Image
    # 'arr' is the HxWx3 uint8 array from the previous step
    img = Image.fromarray(arr)

    # 2. Convert Pillow Image -> Tkinter PhotoImage
    # This format is compatible with the Canvas widget
    return ImageTk.PhotoImage(img)
```

Why Pillow? Pillow (PIL) acts as the translator. It understands raw matrix data (NumPy) and converts it into a bitmap format that the operating system's window manager understands.