Phase 4: The Mathematical Core (Deep Dive)

1 Overview

This document provides a comprehensive, line-by-line analysis of the core mathematical engine driving the Fourier High-Boost Filter. Unlike the high-level architectural view, this document focuses on the algorithmic implementation of the Fourier Transform, the geometric construction of filter masks, and the precise arithmetic of the high-boost operation.

Modules Covered:

- `core/fft_engine.py`: Abstractions for Frequency Domain transformations.
- `core/filters.py`: Mathematical generation of Low-Pass kernels.
- `core/highboost.py`: Logic for high-frequency amplification and symmetry enforcement.

2 Module 1: The FFT Engine (`fft_engine.py`)

This module serves as the interface between our application logic and the underlying NumPy FFT library. It enforces consistency in data types and handling of the DC (Zero Frequency) component.

2.1 Function: `compute_fft(image)`

This function performs the Forward Fourier Transform, converting spatial pixel intensity data into frequency coefficients.

```python
def compute_fft(image: np.ndarray) -> np.ndarray:
    if image.ndim != 2:
        raise ValueError("compute_fft expects a 2D grayscale
            array.")
    return np.fft.fft2(image)
```

Detailed Explanation:

- **Input Validation:** The function explicitly checks if the input is a 2D array. The FFT algorithm operates on 2D planes (x, y). Passing a 3D array (like RGB) here without splitting channels would result in incorrect multidimensional transforms that mix color channels mathematically.
- **The Transform:** It calls `np.fft.fft2`. This computes the 2-Dimensional Discrete Fourier Transform (DFT). The output is a complex-valued array where:
  - The **Real part** represents the cosine components.
  - The **Imaginary part** represents the sine components.
- **Output Structure:** By default, `fft2` places the "DC Component" (the average brightness, frequency $u = 0, v = 0$) at index $[0, 0]$ (top-left corner).

2.2 Function: `compute_ifft(F)`

This function performs the Inverse Fourier Transform to recover the spatial image.

```python
def compute_ifft(F: np.ndarray, imag_tol=1e-9) -> np.ndarray:
    img_back = np.fft.ifft2(F)
    imag_max = float(np.max(np.abs(np.imag(img_back))))
    if imag_max > imag_tol:
        warnings.warn(...)
    return np.real(img_back)
```

Detailed Explanation:

- **Complex Inversion:** The Inverse FFT (`ifft2`) mathematically returns a complex array. Ideally, if our frequency filter was perfectly symmetric, the imaginary parts would all be zero.
- **Numerical Stability Check:** Due to floating-point precision errors (machine epsilon), tiny imaginary values (e.g., 10^{-15}) often persist.
- **Error Handling:** The code calculates `imag_max`. If significant imaginary content remains (above `imag_tol`), it warns the user, as this indicates a broken filter symmetry (a critical bug we solved in `highboost.py`).
- **Real Extraction:** Finally, `np.real()` discards the residual imaginary noise to return a viewable image.

2.3 Functions: `fft_shift` and `ifft_shift`

```python
def fft_shift(F): return np.fft.fftshift(F)
def ifft_shift(Fs): return np.fft.ifftshift(Fs)
```

Why these are crucial:

- **fft_shift:** Swaps the quadrants of the frequency spectrum. It moves the DC component from the top-left $[0, 0]$ to the exact geometric center $[M/2, N/2]$. This allows us to construct filter masks (like circles or Gaussian bells) that are naturally centered.
- **ifft_shift:** Reverses this operation before inversion. The standard `ifft2` algorithm expects the DC component back at $[0, 0]$. Failing to call this would result in a spatial image multiplied by a checkerboard pattern $(-1)^{x+y}$.

2.4 Function: `magnitude_spectrum(F)`

```python
def magnitude_spectrum(F, log=True, eps=1e-8):
    mag = np.abs(F)
    if log:
        return np.log1p(mag + eps)
    return mag
```

Visualization Logic: Raw FFT magnitude values have a massive dynamic range (the DC component can be millions of times larger than high-frequency noise). A linear display would look like a single white dot on black. This function applies a **Logarithmic Transformation** ($\log(1 + |F|)$) to compress the dynamic range, allowing us to visualize the spectral texture.

3 Module 2: The Filter Factory (`filters.py`)

This module is responsible for generating the mathematical "surfaces" used to attenuate specific frequencies.

3.1 Function: `_distance_grid(shape)`

This helper function creates the coordinate system for the frequency domain.

```python
def _distance_grid(shape, center=None):
    M, N = shape
    u0, v0 = (M-1)/2.0, (N-1)/2.0  # Calculate Center
    u = np.arange(M).reshape(M, 1)
    v = np.arange(N).reshape(1, N)
    # Euclidean Distance: D(u,v)
    D = np.hypot(u - u0, v - v0)
    return D
```

Technical Insight:

- We use **Broadcasting** (`reshape`) to create grid matrices efficiently without explicit loops.
- `np.hypot` is used instead of `sqrt(u**2 + v**2)` because it is numerically safer against underflow/overflow for extreme values.
- The distances represent frequency. $D = 0$ is the DC component (Center). Large D represents high frequencies (Edges).

3.2 The Pinhole Logic: `_choose_pinhole_index`

This is a robust engineering fix for the edge case where the user sets Cutoff $D_0 = 0$.

```python
def _choose_pinhole_index(shape, center=None):
    # Selects the single pixel closest to the mathematical
       center
    # ... logic to handle even vs odd dimensions ...
    return (row, col)
```

Why this exists: Mathematical filters like Gaussian involve division by D_0 (e.g., $e^{-D^2/2D_0^2}$). If $D_0 = 0$, the code would crash. Instead of crashing, this function identifies the exact center pixel indices so we can manually set that single pixel to 1.0 (passing only the average brightness) and everything else to 0.

3.3 Filter Kernels

3.3.1 `gaussian_lowpass_mask(shape, D0)`

$$H(u,v) = e^{-\frac{D^2(u,v)}{2D_0^2}} \tag{1}$$

Implementation:

```
    denominator = 2.0 * (float(D0) ** 2)
    arg = -(D**2) / denominator
    mask = np.exp(arg)
```

The Gaussian is the preferred filter for image processing because its Fourier Transform is also a Gaussian. This means a smooth transition in frequency results in a smooth transition in space, eliminating "ringing" artifacts.

3.3.2 `butterworth_lowpass_mask(shape, D0, order)`

$$H(u,v) = \frac{1}{1 + [\frac{D(u,v)}{D_0}]^{2n}} \tag{2}$$

Implementation:

```
    ratio = (D / float(D0)) ** (2 * order)
    mask = 1.0 / (1.0 + ratio)
```

The Butterworth filter offers a compromise.

- **Low Order ($n = 1$):** Soft, like Gaussian.
- **High Order ($n > 5$):** Sharp, approaching Ideal.
- It allows the user to tune the "sharpness" of the cutoff using the `order` parameter.

3.3.3 `radial_lowpass_mask(shape, D0)`

This is the "Ideal" filter.

```
    mask = (D <= float(D0)).astype(float)
```

It acts as a binary gate. While mathematically simple, the sharp discontinuity in the frequency domain corresponds to a Sinc function in the spatial domain, causing severe ringing (ripples) around edges.

4 Module 3: High-Boost Logic (`highboost.py`)

This module implements the specific requirement of the project: The High-Boost Filter formula.

4.1 Function: `compose_highboost_mask(L, r)`

This function takes a Low-Pass mask (L) generated by the previous module and converts it into a High-Boost mask (H_b).

The Mathematical Derivation:

1. We start with the identity (All-Pass): 1.
2. The High-Pass component is the inverse of Low-Pass: $HP = 1 - L$.
3. High-Boost is defined as the original image plus an amplified version of the high frequencies:
$$H_b = 1 + k \cdot HP$$
4. Substituting HP:
$$H_b = 1 + k(1 - L)$$
5. Using the Boost Factor notation r (where A in standard texts usually relates to r via $r = A - 1$ logic, here we use the project-specific formulation):
$$H_b = r + (1 - r) \cdot L$$

Code Implementation:

```python
def compose_highboost_mask(L, r):
    if r <= 1.0:
        raise ValueError("Boost factor r must be > 1")

    # 1. Formula Application
    Hb = r + (1.0 - r) * L

    # 2. HERMITIAN SYMMETRY ENFORCEMENT (Critical)
    Hb = 0.5 * (Hb + Hb[::-1, ::-1])

    return Hb
```

Deep Dive: The Symmetry Enforcement Line The line `Hb = 0.5 * (Hb + Hb[::-1, ::-1])` is perhaps the most critical line for robustness in the entire project.

- **The Problem:** The Low-Pass mask L is generated using floating-point distance calculations. Due to tiny machine precision errors, $L(u, v)$ might not be *exactly* equal to $L(-u, -v)$.

- **The Consequence:** The FFT property states that for a spatial signal to be purely Real, its frequency spectrum must be **Hermitian Symmetric** (conjugate symmetric). If our mask H_b violates this even by 10^{-15}, the Inverse FFT will produce complex numbers (Real + Imaginary).
- **The Solution:** `Hb[::-1, ::-1]` rotates the matrix by 180 degrees. Averaging the mask with its rotated self mathematically forces perfect symmetry about the center. This guarantees that the imaginary components in the final output cancel out to zero.

4.2 Function: `apply_highboost_to_frequency(F, Hb)`

```python
def apply_highboost_to_frequency(F, Hb):
    if F.shape != Hb.shape:
        raise ValueError("Shape mismatch")
    return F * Hb
```

This function performs the filtering operation. Since we are in the frequency domain, filtering is simply **Element-wise Multiplication** (Hadamard Product) of the image spectrum F and the mask H_b. This is computationally much faster ($O(N)$) than performing convolution in the spatial domain ($O(N^2)$).