Introduction to the Fast Fourier Transform

The Fast Fourier Transform (FFT) is a computational algorithm used to efficiently compute the Discrete Fourier Transform (DFT) of a signal or image. The DFT converts data from the spatial or time domain into the frequency domain, allowing us to analyze how different frequency components contribute to the overall structure. While the DFT provides extremely valuable information, it is computationally expensive for large datasets, requiring $O(N^2)$ operations.

The FFT overcomes this limitation by reducing the computational complexity to $O(N \log N)$, making frequency-domain analysis practical for real-time and large-scale applications. The key idea behind the FFT is to exploit symmetries and periodicities in the DFT equations, breaking a large DFT into many smaller ones and recombining them efficiently. This improvement makes FFT essential in areas like image processing, communication systems, signal analysis, and scientific computing.

In image processing, the FFT transforms an image into its frequency components, separating low-frequency information (smooth regions, background) from high-frequency information (edges, fine textures). Once in the frequency domain, various filters—such as low-pass, high-pass, or high-boost filters—can be applied with precise control. After filtering, the inverse FFT reconstructs the processed image back into the spatial domain. This combination of mathematical power and computational efficiency makes the FFT a foundational tool in modern signal and image processing.

How the program decides whether an image is colour or grayscale

The project uses a simple, robust rule at runtime to classify input arrays: if the NumPy array produced by the image reader has three dimensions and the third dimension has size three (i.e. shape = (H, W, 3)), the array is treated as an RGB colour image; otherwise (shape = (H, W)) it is treated as a single-band grayscale image. Concretely, the helper function used throughout the codebase is equivalent to `detect_is_color(arr) -> bool: return arr.ndim == 3 and arr.shape[2] == 3`. When an image file contains an alpha channel (RGBA) or a palette (P), the reader converts or splits the channels such that the GUI receives either a standardized RGB array (H,W,3) plus a separate alpha array in the metadata, or a 2-D grayscale array (H,W). The code therefore relies on the array shape and channel count returned by the image I/O layer (Pillow) as the primary discriminator.

Notes and improvements: the simple shape test is reliable in most practical cases, but you may encounter two corner cases: (1) a "colour" file where all three channels are identical (effectively grayscale) will still be classified as colour — this is harmless but may cause redundant per-channel processing; (2) some exotic formats or multi-band rasters (e.g. single-band GeoTIFF tiles) are read as 2-D arrays and are correctly classified as single-band. If you prefer a more conservative heuristic for classifying a colour image you can additionally test channel variance (e.g. check $\max(|R-G|, |G-B|, |B-R|)$ above a small threshold) and treat arrays with near-identical channels as grayscale for processing/display efficiency.

Adding a new image file format and integrating it into the GUI

Most image format support in this project is provided by `Pillow` (PIL) or by optional Pillow plugins (e.g. `pillow-avif-plugin` for AVIF). To add a new format you generally follow three steps:

1. **Enable reading/writing at the I/O layer:** ensure Pillow (or the relevant plugin) can open the format. If an external plugin is required (AVIF, WebP variants, or specialised geotiff readers), install it and add it to your environment requirements (`requirements.txt` and optionally `environment.yml`). Example additions: `pillow-avif-plugin` for AVIF, or use `rasterio` for advanced geospatial raster formats. After installing, verify `io_utils/image_handler.py::read_image(path)` can `Image.open(path)` and produce a valid NumPy array plus metadata.

2. **Accept the extension in the GUI file dialog:** update the file filter list used by `filedialog.askopenfilename(...)` in `gui/callbacks.py` or `gui/interface.py` to include the new extension (for example add `"*.avif"`). This only affects the file-picker convenience — the image reader still determines how to parse the bytes.

3. **Handle any special post-processing or metadata:** if the new format has multi-band semantics (e.g. 8 bands, alpha, or geospatial metadata), extend `read_image` to return extra metadata (e.g. `meta['bands']=n`) and ensure callbacks choose an appropriate preview band. For single-band scientific rasters, use the existing percentile-stretch preview logic to avoid white/blank previews; for multi-band imagery treat the first three bands as RGB or allow the user to pick a band for preview.

Integration checklist (code locations):

- `requirements.txt` / `environment.yml` — add package names for new format support.
- `io_utils/image_handler.py::read_image` — ensure opening, conversion and returned metadata support the format.
- `gui/callbacks.py` — add the file extension to the `askopenfilename filetypes` list and handle any special preview selection (e.g. choose band 1 when file is multi-band).
- `tests/` — add a unit test to `test_io_utils.py` that reads a small example file in the new format and verifies shape, dtype and meta fields.

Adding support for a new format is therefore usually a matter of a few lines across these files (file dialog types, requirements, and any format-specific fallback handling in the reader). For AVIF specifically you typically add the plugin to `requirements.txt` and ensure Pillow is imported after the plugin — this is a one-time install then the rest of the GUI flows remain unchanged because the callbacks operate on the normalized NumPy arrays the I/O layer returns.

Interpretation and practical limits of the cutoff parameter D_0

What D_0 is (units): the cutoff D_0 is a radius in the frequency domain grid and is measured in *pixels in the discrete frequency domain* (i.e. units of frequency-sample coordinates). When the Fourier transform is represented on an $M \times N$ grid with the DC frequency at the centre (after `fftshift`), a frequency coordinate at array index (u, v) corresponds to a spatial frequency whose radial distance from the centre is

$$D(u, v) = \sqrt{(u - M/2)^2 + (v - N/2)^2}.$$

The scalar D_0 is directly comparable to this distance; for example, $D_0 = 50$ means the cutoff radius is 50 frequency-sample pixels.

Upper and lower practical bounds: the minimum meaningful value is $D_0 > 0$. In practice set a small lower bound (e.g. $D_{0,\min} = 1$) to avoid division-by-zero or ill-conditioned masks (especially in Butterworth formulas). The theoretical maximum radius — the largest possible distance from the centre for an $M \times N$ array — is the half-diagonal:

$$D_{\max} = \sqrt{\left(\frac{M}{2}\right)^2 + \left(\frac{N}{2}\right)^2}.$$

A practical UI-friendly upper limit is usually the smaller of the two half-dimensions or the half-diagonal; we recommend clamping GUI sliders to $[1,\ \min(M, N)/2]$ or $[1,\ D_{\max}]$ and supplying helpful labels that show the current image size so users understand the scale.

Consequences of extreme D_0 values:

- **Very small D_0 (e.g. 5 or 10)**: the low-pass mask passes only the very lowest frequency components (broad smooth structure) and rejects most mid/high frequencies. When forming the high-boost mask $H_b = r + (1 - r)L$ this produces a mask that strongly amplifies a very broad band of high frequencies. Practically this yields *strong edge emphasis*, increased visibility of noise, and possibly ringing/artifacts in the output. Spatially, the image will tend to look highly sharpened or noisy with haloing around edges if the mask transition is abrupt (Ideal mask) or if r is large.

- **Very large D_0 (comparable to half the image size)**: the low-pass mask approaches unity across most frequency samples; therefore $L \approx 1$ and $H_b \approx r + (1 - r) \cdot 1 = 1$. In other words the high-boost mask becomes neutral and the output tends back toward the original image (very little sharpening). Thus very large D_0 yields negligible high-frequency enhancement.

- **Out-of-range values:** if $D_0 \leq 0$ or numerically zero, Butterworth formulas can suffer division-by-zero or produce invalid values; if D_0 is set far larger than $D_{\max}$ the mask-building function should clamp it to $D_{\max}$ to remain meaningful. The implementation should therefore *validate and clamp* user-supplied D_0 values before building masks.

Recommended UI and implementation practices:

- Present D_0 in the GUI as a slider or numeric field whose bounds are computed from the current image dimensions (for example min = 1, max = $\min(M, N)/2$ or $D_{\max}$). Display the current image size near the control so users can reason about what a particular pixel radius means.

- In the back end, always clamp the numeric value to $[1, D_{\max}]$ and convert to a safe floating-point epsilon if required to avoid division by zero in formulae.
- Offer a convenient normalized alternative (optional): allow the user to supply D_0 as a fraction $f \in (0, 1]$ of the Nyquist radius, then internally compute $D_0 = f \cdot D_{\max}$. This is often easier for users to reason about when images have different resolutions.