Image Processing Studio

Desktop UI and Compression Algorithms

(Code Walkthrough and Explanation)

Student Project Tkinter + Flask + OpenCV

November 27, 2025

This document walks through the UI flow, processing pipeline, and compression demos implemented with OpenCV-backed functions.

Quick Manual

- Introduction
- Desktop UI Layer
- Detailed Function Reference
- Compression Algorithms
- End-to-End Flow

Contents

1 Introduction

This document explains the desktop and backend parts of the **Image Processing Studio** project and the core image processing / compression algorithms behind it.

The explanation is structured into three main layers:

- **UI Layer (Tkinter + Matplotlib)**: Files `controls.py` and `preview.py`.
- **Processing Layer**: Functions from the `processing` package: `basic_ops.py`, `filters.py`, `geometry.py`, `interp.py`, `io_utils.py`, `compress.py`.
- **Web API Layer (Flask)**: File `server.py` (optional web frontend).

The goal is to describe what each important function does, how the UI connects to the processing code, and the intuition behind the algorithms.

2 Desktop UI Layer (Tkinter)

The Tkinter-based desktop interface lives in `src/ui/controls.py` and `src/ui/preview.py`.

2.1 ImageApp Frame (controls.py)

Class Definition and Initialization

```python
class ImageApp(ttk.Frame):
    def __init__(self, root):
        super().__init__(root, padding=10)
```

- `ImageApp` inherits from `ttk.Frame` and acts as the main container for the whole desktop UI.
- It is attached directly to the Tk root window (`root`) and uses padding around its contents.

Styling and Theme

```python
root.configure(bg="#111827")
self.style = ttk.Style()
self.style.theme_use("clam")
self.style.configure("TFrame", background="#111827")
self.style.configure("TLabel", background="#111827",
                     foreground="#e5e7eb")
self.style.configure("TButton", background="#1f2937",
                     foreground="#e5e7eb", padding=6)
self.style.map("TButton",
              background=[("active", "#2563eb")])
```

- The root window background is set to a dark color (#111827).
- A `ttk.Style` object is created and the `clam` theme is used as a base.
- Default styles are customized:
  - `TFrame` uses a dark background.
  - `TLabel` uses a dark background and light text.
  - `TButton` uses a dark background, light text, and padding.
  - The button background changes to blue (#2563EB) when active (hover/press).

State Variables and Layout Root

```
self.original_img = None
self.processed_img = None
self.binary_threshold = None

self.grid(column=0, row=0, sticky="nsew")
root.rowconfigure(0, weight=1)
root.columnconfigure(0, weight=1)

self._build_layout()
```

- `original_img`: stores the image loaded from disk (NumPy array).
- `processed_img`: stores the latest processed result (grayscale, filtered, etc.).
- `binary_threshold`: stores the threshold used for grayscale-to-binary conversion.
- The `ImageApp` frame is placed at grid cell (0,0) of the root window and is allowed to expand in all directions (`sticky="nsew"`).
- `rowconfigure` and `columnconfigure` on the root let the frame grow when the window is resized.

The last call `_build_layout()` constructs all child widgets (buttons, entries, frames, etc.).

2.2 Layout Construction (_build_layout)

```
def _build_layout(self):
    header = ttk.Label(self, text="Image Processing Studio",
                       font=("Segoe UI", 18, "bold"))
    header.grid(column=0, row=0, sticky="w", pady=(0, 6))
```

A title label is created at the top of the `ImageApp` frame.

Main Container: Controls vs Preview

```
container = ttk.Frame(self)
container.grid(column=0, row=1, sticky="nsew")
container.columnconfigure(1, weight=1)
container.rowconfigure(0, weight=1)

controls = ttk.Frame(container, padding=8)
controls.grid(column=0, row=0, sticky="ns")

preview_frame = ttk.Frame(container)
preview_frame.grid(column=1, row=0, sticky="nsew")
```

- `container` holds two sub-frames:
  - `controls`: left column, vertical stack of sections.
  - `preview_frame`: right column, where images and histograms are drawn.
- The right column is configured with a positive weight so it grows when the window is resized.

Image Preview and Info Label

```
self.preview = ImagePreview(preview_frame)
self.info_label = ttk.Label(preview_frame, text="No image loaded",
                            anchor="center")
self.info_label.pack(fill="x", pady=4)
```

- `ImagePreview` (from `preview.py`) embeds a Matplotlib figure in Tkinter, with left and right axes.
- `info_label` displays basic information about the currently loaded image (size, channels, data type).

2.3 Control Sections (Labelled Boxes)

The UI is organized into labelled groups using a helper function:

```
def _labeled_box(self, parent, title: str):
    box = ttk.LabelFrame(parent, text=title, padding=6)
    box.configure(style="TFrame")
    box.grid(column=0, row=parent.grid_size()[1],
             sticky="ew", pady=6)
    return box
```

- `ttk.LabelFrame` provides a border and title (e.g. “Image I/O”, “Affine Transforms”).

- `parent.grid_size()[1]` returns the current number of rows of the parent, so each new box is placed below the previous one automatically.

Each section is then built using this helper.

Image I/O

```
file_box = self._labeled_box(controls, "Image I/O")
ttk.Button(file_box, text="Load Image",
          command=self.load_image).grid(...)
self.image_path_var = tk.StringVar(value="No file chosen")
ttk.Label(file_box, textvariable=self.image_path_var,
         wraplength=180).grid(...)
```

- "Load Image" button triggers `self.load_image`.
- `image_path_var` holds the displayed file name.

Conversion (RGB → Grayscale/Binary)

```
conv_box = self._labeled_box(controls, "Convert")
ttk.Button(conv_box, text="To Grayscale",
          command=self.to_grayscale).grid(...)
ttk.Button(conv_box, text="To Binary (avg threshold)",
          command=self.to_binary).grid(...)
```

- **To Grayscale**: converts the original RGB image to grayscale.
- **To Binary**: converts grayscale to binary using the average intensity as threshold.

Affine Transforms (Translate, Scale, Rotate, Shear)

```
affine_box = self._labeled_box(controls, "Affine Transforms")

self.tx_var = tk.DoubleVar(value=30)
self.ty_var = tk.DoubleVar(value=30)
ttk.Label(affine_box, text="tx, ty").grid(...)
ttk.Entry(affine_box, textvariable=self.tx_var, width=6).grid(...)
ttk.Entry(affine_box, textvariable=self.ty_var, width=6).grid(...)
ttk.Button(affine_box, text="Translate",
          command=self.translate).grid(...)
```

- `tx_var`, `ty_var`: translation along x and y.
- "Translate" button calls `self.translate`, which applies `geometry.translate`.

Scaling:

```
self.sx_var = tk.DoubleVar(value=1.2)
self.sy_var = tk.DoubleVar(value=1.2)
ttk.Label(affine_box, text="sx, sy").grid(...)
ttk.Entry(affine_box, textvariable=self.sx_var, width=6).grid(...)
ttk.Entry(affine_box, textvariable=self.sy_var, width=6).grid(...)
ttk.Button(affine_box, text="Scale",
          command=self.scale).grid(...)
```

Rotation:

```
self.angle_var = tk.DoubleVar(value=20)
ttk.Label(affine_box, text="Angle").grid(...)
ttk.Entry(affine_box, textvariable=self.angle_var, width=6).grid(...)
ttk.Button(affine_box, text="Rotate",
          command=self.rotate).grid(...)
```

Shear:

```
self.shx_var = tk.DoubleVar(value=0.2)
self.shy_var = tk.DoubleVar(value=0.2)
ttk.Label(affine_box, text="shx, shy").grid(...)
ttk.Entry(affine_box, textvariable=self.shx_var, width=6).grid(...)
ttk.Entry(affine_box, textvariable=self.shy_var, width=6).grid(...)
ttk.Button(affine_box, text="Shear X",
          command=self.shear_x).grid(...)
ttk.Button(affine_box, text="Shear Y",
          command=self.shear_y).grid(...)
```

Each button calls a corresponding method that uses the geometry utilities.

Interpolation / Resizing

```
interp_box = self._labeled_box(controls, "Interpolation")
self.new_w = tk.IntVar(value=320)
self.new_h = tk.IntVar(value=240)
ttk.Label(interp_box, text="W, H").grid(...)
ttk.Entry(interp_box, textvariable=self.new_w, width=6).grid(...)
ttk.Entry(interp_box, textvariable=self.new_h, width=6).grid(...)

ttk.Button(interp_box, text="Nearest",
          command=lambda: self.resize("nearest")).grid(...)
ttk.Button(interp_box, text="Bilinear",
          command=lambda: self.resize("bilinear")).grid(...)
ttk.Button(interp_box, text="Bicubic",
          command=lambda: self.resize("bicubic")).grid(...)
```

- Width and height are controlled by `new_w` and `new_h`.
- The three buttons call `self.resize` with different interpolation methods.

Crop

```
crop_box = self._labeled_box(controls, "Crop")
self.crop_x = tk.IntVar(value=10)
self.crop_y = tk.IntVar(value=10)
self.crop_w = tk.IntVar(value=200)
self.crop_h = tk.IntVar(value=200)

for idx, (lbl, var) in enumerate([("x", self.crop_x),
                                 ("y", self.crop_y),
                                 ("w", self.crop_w),
                                 ("h", self.crop_h)]):
    ttk.Label(crop_box, text=lbl).grid(...)
    ttk.Entry(crop_box, textvariable=var, width=6).grid(...)

ttk.Button(crop_box, text="Crop",
          command=self.crop).grid(...)
```

- A small loop generates labels and entries for `x`, `y`, `w`, `h`.
- The "Crop" button calls `self.crop`, which uses `basic_ops.crop`.

Histogram

```
hist_box = self._labeled_box(controls, "Histogram")
ttk.Button(hist_box, text="Compute Histogram",
          command=self.show_histogram).grid(...)
ttk.Button(hist_box, text="Equalize",
          command=self.equalize_histogram).grid(...)
self.hist_eval_label = ttk.Label(hist_box, text="",
                              wraplength=180)
self.hist_eval_label.grid(...)
```

- "Compute Histogram" computes and displays the histogram of the current image.
- "Equalize" applies histogram equalization on the grayscale image.
- `hist_eval_label` displays a textual quality assessment from `histogram_goodness`.

Filtering

```
filter_box = self._labeled_box(controls, "Filtering")
ttk.Button(filter_box, text="Gaussian 19x19 (\\sigma=3)",
          command=self.gaussian).grid(...)
ttk.Button(filter_box, text="Median 7x7",
          command=self.median).grid(...)
ttk.Button(filter_box, text="Laplacian (2nd deriv.)",
          command=self.laplacian).grid(...)
ttk.Button(filter_box, text="Sobel",
          command=self.sobel).grid(...)
ttk.Button(filter_box, text="Gradient (1st deriv.)",
          command=self.gradient).grid(...)
```

Each button applies a different spatial filter using functions from `filters.py`.

Compression Section

```
comp_box = self._labeled_box(controls, "Compression")
ttk.Button(comp_box, text="Huffman",
          command=lambda: self.compress_action("huffman")).grid(...)
ttk.Button(comp_box, text="Golomb-Rice",
          command=lambda: self.compress_action("golomb")).grid(...)
ttk.Button(comp_box, text="Arithmetic",
          command=lambda: self.compress_action("arithmetic")).grid(...)
ttk.Button(comp_box, text="LZW",
          command=lambda: self.compress_action("lzw")).grid(...)
ttk.Button(comp_box, text="Run-Length",
          command=lambda: self.compress_action("rle")).grid(...)
ttk.Button(comp_box, text="Symbol-based",
          command=lambda: self.compress_action("symbol")).grid(...)
ttk.Button(comp_box, text="Bit-plane",
          command=lambda: self.compress_action("bitplane")).grid(...)
ttk.Button(comp_box, text="Block DCT",
          command=lambda: self.compress_action("dct")).grid(...)
ttk.Button(comp_box, text="Predictive",
          command=lambda: self.compress_action("predictive")).grid(...)
ttk.Button(comp_box, text="Wavelet (Haar)",
          command=lambda: self.compress_action("wavelet")).grid(...)
```

Each button triggers `compress_action(mode)`, integrating the UI with the different compression methods in `compress.py`.

Status Bar

```
self.status_var = tk.StringVar(value="Ready")
ttk.Label(self, textvariable=self.status_var).grid(
    column=0, row=2, sticky="ew", pady=(6, 0)
)
```

A small status bar at the bottom of the window displays messages such as “Loaded image.”, “Applied Gaussian filter.”, and compression ratios.

2.4 Core Helper Methods

_require_image

```
def _require_image(self):
    if self.original_img is None:
        messagebox.showwarning("No image",
                               "Please load an image first.")
        return False
    return True
```

Any action that needs an image calls this helper first. If no image is loaded, a warning is shown and the action is aborted.

_update_info

```
def _update_info(self):
    if self.original_img is None:
        self.info_label.config(text="No image loaded")
        return
    info = io_utils.info(self.original_img)
    text = (f"{info['width']}x{info['height']} | "
            f"channels: {info['channels']} | "
            f"dtype: {info['dtype']}")
    self.info_label.config(text=text)
```

This function updates `info_label` with basic image properties from `io_utils.info`.

_set_processed

```
def _set_processed(self, img, title="Processed"):
    self.processed_img = img
    self.preview.show_processed(img, title)
```

Stores the processed image and displays it on the right axis of the preview.

2.5 Major Actions

load_image

```
def load_image(self):
    path = filedialog.askopenfilename(
        filetypes=[("Images",
                    "*.png *.jpg *.jpeg *.bmp *.tif *.tiff")]
    )
    if not path:
        return
    try:
        self.original_img = io_utils.load_image(path)
    except Exception as exc:
        messagebox.showerror("Error", str(exc))
        return
    self.processed_img = None
    self.preview.show_original(self.original_img)
    self.preview.show_processed(None)
    self._update_info()
    self.image_path_var.set(path.split("/")[-1])
    self.status_var.set("Loaded image.")
```

- Shows an "Open File" dialog and restricts file types to images.
- Loads the image as RGB float with `io_utils.load_image`.
- Resets processed image and updates both preview and info label.

`to_grayscale` and `to_binary`

```
def to_grayscale(self):
    if not self._require_image():
        return
    gray = basic_ops.rgb_to_grayscale(self.original_img)
    self._set_processed(gray, "Grayscale")
    self.status_var.set("Converted to grayscale.")
```

- Always converts from the original RGB image for a clean grayscale.

```
def to_binary(self):
    if not self._require_image():
        return
    gray = basic_ops.rgb_to_grayscale(self.original_img)
    binary, threshold = basic_ops.grayscale_to_binary(gray)
    self.binary_threshold = threshold
    self._set_processed(binary, f"Binary (t={threshold:.1f})")
    self.status_var.set("Binary conversion complete.")
```

- Threshold is computed as the average intensity of the grayscale image.
- Threshold is stored in `binary_threshold` and displayed in the title.

Geometric Transformations

All geometric operations follow the same pattern: use the latest processed image if available, otherwise use the original.

```python
def translate(self):
    if not self._require_image():
        return
    img = self.processed_img if self.processed_img is not None \
        else self.original_img
    result = geometry.translate(img,
                                self.tx_var.get(),
                                self.ty_var.get())
    self.preview.show_original(self.original_img)
    self._set_processed(result, "Translated")
    self.status_var.set("Applied translation.")
```

- Uses `geometry.translate`, which builds an affine matrix and calls `cv2.warpAffine`.
- Original is refreshed on the left, translation result on the right.

The other functions, `scale`, `rotate`, `shear_x`, and `shear_y`, work similarly with their respective geometry helpers.

Resizing and Cropping

```python
def resize(self, method: str):
    if not self._require_image():
        return
    img = self.processed_img if self.processed_img is not None \
        else self.original_img
    result = interp.resize(img,
                           self.new_w.get(),
                           self.new_h.get(),
                           method=method)
    self._set_processed(result, f"Resize ({method})")
    self.status_var.set(f"Resized using {method}.")
```

- Delegates to `interp.resize`, which uses `cv2.resize` with the chosen interpolation method.

```python
def crop(self):
    if not self._require_image():
        return
    img = self.processed_img if self.processed_img is not None \
        else self.original_img
    result = basic_ops.crop(
        img,
        self.crop_x.get(),
        self.crop_y.get(),
        self.crop_w.get(),
        self.crop_h.get()
    )
    self._set_processed(result, "Cropped")
    self.status_var.set("Cropped region.")
```

Histogram Operations

```python
def show_histogram(self):
    if not self._require_image():
        return
    gray = basic_ops.rgb_to_grayscale(
        self.processed_img if self.processed_img is not None
        else self.original_img
    )
    hist = basic_ops.histogram(gray)
    self.preview.show_histogram(hist)
    self.hist_eval_label.config(
        text=basic_ops.histogram_goodness(hist)
    )
    self.status_var.set("Histogram computed.")
```

- Converts the current image to grayscale.
- Computes a histogram (256 bins) using `basic_ops.histogram`.
- Displays the histogram using `ImagePreview.show_histogram`.
- Displays a textual assessment from `histogram_goodness`.

```python
def equalize_histogram(self):
    if not self._require_image():
        return
    gray = basic_ops.rgb_to_grayscale(
        self.processed_img if self.processed_img is not None
        else self.original_img
    )
    eq = basic_ops.histogram_equalization(gray)
    self._set_processed(eq, "Equalized")
    self.status_var.set("Histogram equalization applied.")
```

Filtering

Examples:

```python
def gaussian(self):
    if not self._require_image():
        return
    img = self.processed_img if self.processed_img is not None \
          else self.original_img
    result = filters.gaussian_blur(img, 19, 3.0)
    self._set_processed(result, "Gaussian Blur")
    self.status_var.set("Applied Gaussian filter.")
```

```python
def laplacian(self):
    if not self._require_image():
        return
    gray = basic_ops.rgb_to_grayscale(
        self.processed_img if self.processed_img is not None
        else self.original_img
    )
    result = filters.laplacian_filter(gray)
    self._set_processed(result, "Laplacian")
    self.status_var.set("Applied Laplacian filter.")
```

- The Gaussian and median filters work directly on the image (RGB or grayscale).
- The Laplacian, Sobel and gradient filters operate on grayscale for edge detection / first derivative magnitude.

2.6 Compression Integration (`compress_action`)

```python
def compress_action(self, mode: str):
    if not self._require_image():
        return
    img = self.processed_img if self.processed_img is not None \
          else self.original_img
    gray = basic_ops.rgb_to_grayscale(img)
```

All compression modes start from a grayscale image, then branch by `mode`.

Huffman Example

```python
if mode == "huffman":
    data = compress.huffman_compress(gray)
    decoded = compress.huffman_decompress(
        data["bitstring"], data["tree"], gray.shape
    )
    self._set_processed(decoded,
                        f"Huffman (r={data['ratio']:.2f})")
    self.status_var.set(f"Huffman ratio {data['ratio']:.2f}")
```

- `huffman_compress` returns the code tree, code map, bitstring and compression ratio.
- `huffman_decompress` reconstructs the image from the bitstring and Huffman tree.
- The reconstructed image is shown, and the compression ratio appears both in the title and status bar.

Other Modes

Similarly for:

- **Golomb-Rice**: `golomb_rice_encode` / `golomb_rice_decode`.
- **LZW**: `lzw_encode` / `lzw_decode`.
- **Run-Length**: `rle_encode` / `rle_decode`.
- **Symbol-based**: `symbol_based_encode` / `symbol_based_decode`.
- **Bit-plane**: uses `bit_planes`, then visualizes the eight bit-planes as a grid.
- **Block DCT**: uses `dct_compress` and displays the reconstructed image and number of kept coefficients.
- **Predictive Coding**: `predictive_encode` / `predictive_decode`.
- **Wavelet (Haar)**: applies Haar wavelet transform and its inverse for reconstruction.

Bit-plane Visualization Helper

```
def _stack_bitplanes(self, planes):
    # Combine bit-planes into a square grid for visualization.
    rows = []
    for i in range(0, 8, 2):
        left = np.hstack((planes[i], planes[i + 1]))
        rows.append(left)
    return np.vstack(rows)
```

- The eight bit-planes are arranged in a 4x2 grid:
  - Rows: (0,1), (2,3), (4,5), (6,7).
  - Each row is built using horizontal stacking; then all rows are stacked vertically.

2.7 ImagePreview Class (preview.py)

Figure and Axes Setup

```
class ImagePreview:
    def __init__(self, parent):
        self.figure = Figure(figsize=(6.5, 4), dpi=100)
        self.ax_original = self.figure.add_subplot(121)
        self.ax_processed = self.figure.add_subplot(122)
        self.ax_original.axis("off")
        self.ax_processed.axis("off")
```

- A Matplotlib `Figure` is created with two subplots side by side:
  - `ax_original`: left, for the original image.
  - `ax_processed`: right, for processed result or histogram.
- Axis decorations (ticks, spines) are turned off to show pure images.

Embedding in Tkinter

```
self.canvas = FigureCanvasTkAgg(self.figure, master=parent)
self.canvas_widget = self.canvas.get_tk_widget()
self.canvas_widget.pack(fill="both", expand=True)
self.canvas.draw()
```

- `FigureCanvasTkAgg` embeds the Matplotlib figure as a Tkinter widget.
- The canvas widget is packed so it fills the entire parent frame.
- A first draw call renders an initial empty figure.

Image Preparation Helper

```
def _prep_image(img: np.ndarray):
    if img is None:
        return None
    if img.ndim == 2:
        return np.clip(img, 0, 255)
    return np.clip(img, 0, 255).astype(np.float32) / 255.0
```

- If the image is grayscale (2D), it simply clips intensities to the valid range.
- If it is color (H x W x 3), it clips and scales to [0,1], which is the format expected by Matplotlib for RGB images.

Showing the Original Image

```python
def show_original(self, img: np.ndarray, title="Original"):
    self.ax_original.clear()
    self.ax_original.axis("off")
    img_prep = _prep_image(img)
    if img_prep is not None:
        if img_prep.ndim == 2:
            self.ax_original.imshow(img_prep, cmap="gray")
        else:
            self.ax_original.imshow(img_prep)
    self.ax_original.set_title(title)
    self.canvas.draw_idle()
```

- Clears the left axis and hides axes again.
- Prepares the image; if it exists:
  - Uses grayscale colormap for 2D images.
  - Uses default RGB for color images.
- Sets an axis title and schedules a redraw.

Showing the Processed Image

```python
def show_processed(self, img: np.ndarray, title="Processed"):
    self.ax_processed.clear()
    self.ax_processed.axis("off")
    img_prep = _prep_image(img)
    if img_prep is not None:
        if img_prep.ndim == 2:
            self.ax_processed.imshow(img_prep, cmap="gray")
        else:
            self.ax_processed.imshow(img_prep)
    self.ax_processed.set_title(title)
    self.canvas.draw_idle()
```

The same logic as `show_original`, but operates on the right axis.

Showing a Histogram

```python
def show_histogram(self, hist: np.ndarray):
    self.ax_processed.clear()
    self.ax_processed.bar(range(256), hist, color="#7f8c8d")
    self.ax_processed.set_title("Histogram")
    self.ax_processed.set_xlim(0, 255)
    self.canvas.draw_idle()
```

- Clears the right axis.
- Plots a bar chart with 256 bars for intensities 0–255.
- Fixes the x-axis range to [0, 255].

3 Compression and Processing Algorithms (High-Level)

This section summarizes the main compression and coding techniques provided in `compress.py`, which are invoked from the UI.

3.1 Huffman Coding

- **Goal**: assign variable-length binary codes to each pixel value such that:
  - Frequent symbols get short codes.
  - Rare symbols get longer codes.
  - The code is prefix-free (no code is prefix of another), so decoding is unambiguous.
- **Implementation steps**:
  1. Flatten the grayscale image to a list of pixel values.
  2. Compute symbol frequencies with a `Counter`.
  3. Build a min-heap of leaf nodes (frequency, symbol).
  4. Repeatedly pop the two smallest nodes and merge them into a parent node.
  5. The final tree root represents the Huffman tree.
  6. Traverse the tree:
     - Left edge adds a "0" to the current code.
     - Right edge adds a "1".
     - Reaching a leaf assigns the accumulated bitstring to that symbol.
  7. Encode each pixel by replacing it with its bitstring code.
- The project estimates compression ratio by comparing:
  - `original_bits` = number of pixels × 8.
  - `compressed_bits` = length of the Huffman bitstring.

3.2 Golomb-Rice Coding

- Designed for non-negative integers where small values are more probable.
- Rice coding chooses a divisor $m = 2^k$ and encodes a value:
  - $q = \lfloor \text{val}/m \rfloor$ (quotient).

  - $r = \text{val} \bmod m$ (remainder).

- Code = unary for q (a string of q ones followed by a zero), then fixed k-bit binary for r.
- The project applies this to grayscale intensities and estimates the bit cost from the concatenated bitstring.

3.3 Arithmetic Coding (Entropy-Based Estimate)

- True arithmetic coding encodes an entire sequence into a sub-interval of $[0, 1)$, achieving near-optimal compression close to Shannon entropy.
- In the project, the implementation:
  - Computes symbol frequencies and probabilities.
  - Calculates Shannon entropy:

$$H = -\sum_i p_i \log_2 p_i$$

  - Estimates an ideal number of bits as $H \times N$ where N is the number of pixels.
  - This is used to report a theoretical compression ratio, while reconstruction simply returns the original grayscale image.

3.4 LZW (Lempel–Ziv–Welch)

- **Dictionary-based** lossless compression.
- Starts with a dictionary of all single-byte sequences (0–255).
- While scanning the data:
  - Maintain the longest current sequence that exists in the dictionary.
  - When adding another symbol would break it, output the code of the current sequence, and add the extended sequence to the dictionary.
- Decoding reconstructs the same dictionary on the fly by interpreting each code as a sequence and adding new combinations of previous sequences and first characters.

3.5 Run-Length Encoding (RLE)

- Ideal for sequences with many repeated values.
- Stores data as $(\text{value}, \text{count})$ pairs.
- In the implementation:
  - The flattened image is scanned, and consecutive runs of identical pixel values are grouped.

  - Bit cost is estimated assuming 8 bits for the value and 16 bits for the count.

3.6 Symbol-Based Fixed-Length Coding

- Counts symbol frequencies and sorts symbols from most to least frequent.
- Assigns each symbol an index (0, 1, 2, . . .) and encodes indices using a fixed number of bits:

$$\text{bits\_needed} = \lceil \log_2(\#\text{symbols}) \rceil$$

- This is simpler than Huffman (fixed length instead of variable length) and generally less efficient, but easier to implement.

3.7 Bit-Plane Coding

- Decomposes 8-bit grayscale images into 8 binary images, each corresponding to a bit position.
- For each pixel:
  - The least significant bit forms plane 0.
  - The most significant bit forms plane 7.
- Each bit-plane can be visualized independently.
- The original image can be reconstructed by summing contributions from all planes.

3.8 Block DCT Coding

- Very similar in spirit to JPEG:
  - The image is divided into 8x8 blocks.
  - Each block is shifted by subtracting 128 to center values around zero.
  - A 2D Discrete Cosine Transform (DCT) is applied to each block.
  - The magnitude of coefficients is analyzed, and only a certain ratio of largest coefficients is kept.
  - Remaining coefficients are zeroed out and the block is inverse-transformed.
- Because energy in natural images is concentrated in low frequencies, keeping only the largest coefficients can still preserve most visual information while reducing storage requirements.

3.9 Predictive Coding (DPCM)

- Instead of storing raw pixel values, store the difference between a pixel and a prediction, often its left neighbor.

- If adjacent pixels are similar, these differences (residuals) are small, and thus more compressible.
- Decoding reconstructs each pixel by adding the stored residual to the same predictor used at encoding.

3.10 Haar Wavelet Transform (1 Level)

- The image is processed in 2x2 blocks.
- For each block of four pixels (a, b, c, d):
  - An approximation (low-frequency average).
  - Horizontal, vertical, and diagonal detail coefficients.
- The transform can be inverted exactly using the inverse formulas.
- Wavelets are useful for multi-resolution representation and form the basis of schemes like JPEG 2000.

4 Detailed Function Reference

This section goes *function by function* through the processing and backend code. For each function we summarize:

- What it does.
- Its inputs and outputs.
- Important implementation details.

4.1 `basic_ops.py` (Basic Operations)

`rgb_to_grayscale(img)`

```python
def rgb_to_grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY).astype(np.float32)
```

- **Input**: `img` is an RGB image as a NumPy array of shape $(H, W, 3)$.
- **Operation**:
  - Uses OpenCV's `cvtColor` to convert from RGB to a single-channel grayscale image.
  - `cv2.COLOR_RGB2GRAY` uses a weighted sum of R,G,B channels to approximate luminance.
- **Output**: a `float32` array of shape (H, W).

grayscale_to_binary(grayscale_image)

```python
def grayscale_to_binary(grayscale_image):
    """Convert grayscale to binary using average intensity threshold."""
    threshold = float(cv2.mean(grayscale_image)[0])
    _, binary = cv2.threshold(grayscale_image.astype(np.float32),
                              threshold, 255, cv2.THRESH_BINARY)
    return binary, threshold
```

- **Input**: `grayscale_image` as (H, W).
- **Step 1**: `cv2.mean` computes the mean intensity over all pixels.
- **Step 2**: `cv2.threshold` binarizes:
  - Pixels $\geq$ threshold $\rightarrow$ 255 (white).
  - Pixels $<$ threshold $\rightarrow$ 0 (black).
- **Output**: the binary image and the threshold used.

crop(img, x, y, w, h)

```python
def crop(img, x, y, w, h):
    return img[y:y+h, x:x+w]
```

- **Input**:
  - `img`: 2D or 3D image.
  - `x,y`: top-left corner of the crop.
  - `w,h`: width and height of the crop.
- Uses NumPy slicing: rows first (`y:y+h`), columns second (`x:x+w`).
- **Output**: cropped sub-image.

histogram(gray)

```python
def histogram(gray):
    """Compute histogram for grayscale image."""
    hist = cv2.calcHist([gray.astype(np.uint8)],
                        [0], None, [256], [0, 256]).flatten()
    return hist.astype(np.int32)
```

- **Input**: grayscale image, any numeric type.
- **Operation**:
  - Converts to `uint8`.

  - `cv2.calcHist` counts how many pixels fall into 256 intensity bins (0–255).
- **Output**: 1D array of length 256, with integer counts.

`histogram_goodness(hist)`

```python
def histogram_goodness(hist):
    """Assess histogram spread."""
    total = np.sum(hist)
    if total == 0:
        return "Empty histogram."
    nonzero_bins = np.count_nonzero(hist)
    spread_ratio = nonzero_bins / 256.0
    if spread_ratio > 0.7:
        return "Histogram is well-distributed; good contrast."
    if spread_ratio > 0.4:
        return "Histogram is moderately spread; contrast is acceptable."
    return "Histogram is concentrated; consider equalization to improve
        contrast."
```

- Counts how many bins are non-empty and divides by 256 to get a ratio.
- Simple heuristic:
  - > 0.7: good contrast.
  - > 0.4: acceptable.
  - Else: intensities are concentrated; contrast is weak.

`histogram_equalization(gray)`

```python
def histogram_equalization(gray):
    """Apply histogram equalization via OpenCV."""
    gray_uint8 = np.clip(gray, 0, 255).astype(np.uint8)
    return cv2.equalizeHist(gray_uint8).astype(np.float32)
```

- Clips values to [0,255] and converts to `uint8`.
- Calls `cv2.equalizeHist`:
  - Spreads out intensity values to use more of the available range.
  - Enhances global contrast in many images.
- Returns a `float32` image.

4.2 `filters.py` (Filtering Operations)

`convolve(img, kernel)`

```python
def convolve(img, kernel):
    """Convolution via OpenCV with replicate padding."""
    return cv2.filter2D(img, ddepth=cv2.CV_32F,
                        kernel=kernel,
                        borderType=cv2.BORDER_REPLICATE)
```

- Generic 2D convolution.
- `BORDER_REPLICATE`: pads the image border by repeating edge pixels to avoid dark borders.
- `CV_32F`: output is 32-bit float for good precision.

`gaussian_blur(img, size=19, sigma=3.0)`

```python
def gaussian_blur(img, size: int = 19, sigma: float = 3.0):
    return cv2.GaussianBlur(img, (size, size),
                            sigmaX=sigma, sigmaY=sigma,
                            borderType=cv2.BORDER_REPLICATE)
```

- Applies Gaussian smoothing with a kernel of size `size` x `size`.
- `sigma` controls how strong the blur is.

`median_filter(img, size=7)`

```python
def median_filter(img, size: int = 7):
    ksize = size if size % 2 == 1 else size + 1
    return cv2.medianBlur(img, ksize)
```

- Ensures kernel size is odd (required by OpenCV).
- Median filter is very good at removing salt-and-pepper noise.

`laplacian_filter(img)`

```python
def laplacian_filter(img):
    return cv2.Laplacian(img, ddepth=cv2.CV_32F,
                         ksize=3,
                         borderType=cv2.BORDER_REPLICATE)
```

- Uses the Laplacian operator (second derivative) to highlight edges and rapid intensity changes.

sobel_filter(img)

```python
def sobel_filter(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0,
                   ksize=3,
                   borderType=cv2.BORDER_REPLICATE)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1,
                   ksize=3,
                   borderType=cv2.BORDER_REPLICATE)
    return cv2.magnitude(gx, gy)
```

- Computes horizontal gradient (`gx`) and vertical gradient (`gy`) using Sobel filters.
- `cv2.magnitude` returns $\sqrt{g_x^2 + g_y^2}$, a gradient magnitude image.

gradient_first_derivative(img)

```python
def gradient_first_derivative(img):
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0,
                   ksize=1,
                   borderType=cv2.BORDER_REPLICATE)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1,
                   ksize=1,
                   borderType=cv2.BORDER_REPLICATE)
    return cv2.magnitude(gx, gy)
```

- Same idea as Sobel, but with `ksize=1`, effectively a simpler first-derivative operator.

4.3 `geometry.py` (Geometric Transforms)

Constants and Interpolation Map

```python
MAX_OUTPUT_PIXELS = 50_000_000

_INTERP = {
    "nearest": cv2.INTER_NEAREST,
    "bilinear": cv2.INTER_LINEAR,
    "bicubic": cv2.INTER_CUBIC,
}
```

- `MAX_OUTPUT_PIXELS`: safety limit to avoid creating extremely large images.
- `_INTERP`: maps human-friendly names to OpenCV interpolation flags.

apply_affine(img, matrix, output_shape=None, method="bilinear")

```python
def apply_affine(img, matrix, output_shape=None,
                 method: str = "bilinear"):
    """Affine warp via cv2.warpAffine with replicate padding."""
    h, w = img.shape[:2]
    out_h, out_w = output_shape if output_shape is not None else (h, w)
    flags = _INTERP.get(method.lower(), cv2.INTER_LINEAR)
    return cv2.warpAffine(img.astype(np.float32),
                          matrix.astype(np.float32),
                          (out_w, out_h),
                          flags=flags,
                          borderMode=cv2.BORDER_REPLICATE)
```

- Generic helper: warps an image by any 2x3 affine matrix.
- `output_shape` lets you specify the new height and width; if omitted, uses original size.
- Interpolation is chosen via `method` string.

translate(img, tx, ty)

```python
def translate(img, tx: float, ty: float):
    mat = np.array([[1, 0, tx],
                    [0, 1, ty]],
                   dtype=np.float32)
    return apply_affine(img, mat, method="bilinear")
```

- Builds an affine matrix that shifts all pixels by `(tx, ty)`.
- Positive `tx` moves right, positive `ty` moves down.

scale(img, sx, sy)

```python
def scale(img, sx: float, sy: float):
    new_w = max(1, int(round(img.shape[1] * sx)))
    new_h = max(1, int(round(img.shape[0] * sy)))
    pixels = new_w * new_h
    if pixels > MAX_OUTPUT_PIXELS:
        factor = (MAX_OUTPUT_PIXELS / pixels) ** 0.5
        new_w = max(1, int(round(img.shape[1] * sx * factor)))
        new_h = max(1, int(round(img.shape[0] * sy * factor)))
    return cv2.resize(img.astype(np.float32),
                      (new_w, new_h),
                      interpolation=cv2.INTER_LINEAR)
```

- Computes the intended new size from scale factors.

- If the number of pixels would exceed `MAX_OUTPUT_PIXELS`, it computes a shrink factor to respect the limit.
- Uses bilinear interpolation for resizing.

`rotate(img, angle_deg)`

```
def rotate(img, angle_deg: float):
    h, w = img.shape[:2]
    center = (w / 2.0, h / 2.0)
    mat = cv2.getRotationMatrix2D(center, angle_deg, 1.0)
    return cv2.warpAffine(img.astype(np.float32),
                          mat, (w, h),
                          flags=cv2.INTER_LINEAR,
                          borderMode=cv2.BORDER_REPLICATE)
```

- Rotates around the image center by `angle_deg` degrees.
- Keeps the same output size (w, h); rotated content may be cropped.

`shear_x(img, shx)` and `shear_y(img, shy)`

```
def shear_x(img, shx: float):
    mat = np.array([[1, shx, 0],
                    [0, 1, 0]],
                   dtype=np.float32)
    out_w = int(img.shape[1] + abs(shx) * img.shape[0])
    return apply_affine(img, mat,
                        output_shape=(img.shape[0], out_w),
                        method="bilinear")

def shear_y(img, shy: float):
    mat = np.array([[1, 0, 0],
                    [shy, 1, 0]],
                   dtype=np.float32)
    out_h = int(img.shape[0] + abs(shy) * img.shape[1])
    return apply_affine(img, mat,
                        output_shape=(out_h, img.shape[1]),
                        method="bilinear")
```

- Horizontal shear (`shear_x`) slides rows horizontally proportional to their vertical position.
- Vertical shear (`shear_y`) slides columns vertically proportional to their horizontal position.
- Output dimensions are expanded to fit the sheared content.

4.4 interp.py (Interpolation / Resampling)

sample_image(img, xs, ys, method="bilinear")

```
def sample_image(img, xs, ys, method: str = "bilinear"):
    """Remap with OpenCV interpolation."""
    interp = {"nearest": cv2.INTER_NEAREST,
              "bilinear": cv2.INTER_LINEAR,
              "bicubic": cv2.INTER_CUBIC}.get(method.lower(),
                                               cv2.INTER_LINEAR)
    return cv2.remap(img.astype(np.float32),
                     xs.astype(np.float32),
                     ys.astype(np.float32),
                     interpolation=interp,
                     borderMode=cv2.BORDER_REPLICATE)
```

- `xs` and `ys` are coordinate grids specifying where each output pixel samples from the input.
- `cv2.remap` is a general warping function using arbitrary coordinate maps.

resize(img, new_width, new_height, method="nearest")

```
def resize(img, new_width: int, new_height: int,
           method: str = "nearest"):
    """Resize via cv2.resize with chosen interpolation."""
    interp = {"nearest": cv2.INTER_NEAREST,
              "bilinear": cv2.INTER_LINEAR,
              "bicubic": cv2.INTER_CUBIC}.get(method.lower(),
                                               cv2.INTER_NEAREST)
    w = max(1, int(new_width))
    h = max(1, int(new_height))
    return cv2.resize(img.astype(np.float32),
                      (w, h),
                      interpolation=interp)
```

- Clamps width/height to at least 1 pixel.
- Chooses interpolation based on `method` string.

4.5 io_utils.py (Image I/O Helpers)

load_image(path)

```
def load_image(path: str):
    """Load image from disk as RGB float array."""
    img_bgr = cv2.imread(path, cv2.IMREAD_COLOR)
    if img_bgr is None:
        raise ValueError(f"Unable to read image at {path}")
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
    return img_rgb.astype(np.float32)
```

- Reads an image from disk using OpenCV (BGR order).
- Converts to RGB for consistency with Matplotlib and UI.
- Throws a `ValueError` if reading fails.

`ensure_uint8(img)`

```
def ensure_uint8(img):
    """Clip and convert to uint8 for display."""
    clipped = np.clip(img, 0, 255)
    return clipped.astype(np.uint8)
```

- Ensures an image can be safely encoded/saved as an 8-bit image.

`info(img)`

```
def info(img):
    """Return basic image info."""
    h, w = img.shape[:2]
    channels = 1 if img.ndim == 2 else img.shape[2]
    dtype = str(img.dtype)
    return {"width": w, "height": h,
            "channels": channels, "dtype": dtype}
```

- Extracts basic metadata: width, height, number of channels, and data type.

4.6 `compress.py` (Compression Algorithms)

Utility: `_to_gray_uint8(img)`

```
def _to_gray_uint8(img):
    if img.ndim == 3:
        img = np.mean(img, axis=2)
        return np.clip(img, 0, 255).astype(np.uint8)
    return np.clip(img, 0, 255).astype(np.uint8)
```

- If `img` is RGB, takes the average of channels.
- Clips values to [0,255] and converts to `uint8`.

Utility: `_compression_ratio`

```python
def _compression_ratio(original_bits: int,
                       compressed_bits: int) -> float:
    if compressed_bits == 0:
        return 0.0
    return original_bits / compressed_bits
```

- Simple helper to avoid division by zero.

Huffman: `_HuffNode` and Tree Building

```python
class _HuffNode:
    def __init__(self, freq, symbol=None,
                 left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq
```

- Node structure for Huffman tree (frequency + symbol or children).
- `__lt__` allows nodes to be compared inside a `heapq`.

```python
def _build_huffman_tree(data):
    flat = data.ravel().tolist()
    freq = Counter(flat)
    heap = []
    for sym, f in freq.items():
        heapq.heappush(heap, (f, _HuffNode(f, sym)))
    while len(heap) > 1:
        f1, n1 = heapq.heappop(heap)
        f2, n2 = heapq.heappop(heap)
        merged = _HuffNode(f1 + f2, None, n1, n2)
        heapq.heappush(heap, (merged.freq, merged))
    return heap[0][1]
```

- Builds a Huffman tree by always merging the two least frequent nodes.

`_generate_codes(node, prefix="", codes=None)`

```python
def _generate_codes(node: _HuffNode, prefix="", codes=None):
    if codes is None:
        codes = {}
    if node.symbol is not None:
        codes[node.symbol] = prefix or "0"
        return codes
    _generate_codes(node.left, prefix + "0", codes)
    _generate_codes(node.right, prefix + "1", codes)
    return codes
```

- Recursively traverses the tree:
  - Left child adds “0”, right child adds “1”.
  - Leaves store final codes.

huffman_compress(img)

```
def huffman_compress(img):
    gray = _to_gray_uint8(img)
    data = gray.ravel().tolist()
    tree = _build_huffman_tree(gray)
    codes = _generate_codes(tree)
    bitstring = "".join(codes[val] for val in data)
    original_bits = len(data) * 8
    compressed_bits = len(bitstring)
    return {
        "bitstring": bitstring,
        "codes": codes,
        "tree": tree,
        "ratio": _compression_ratio(original_bits,
                                    compressed_bits),
        "original_bits": original_bits,
        "compressed_bits": compressed_bits,
    }
```

- Produces:
  - `bitstring`: concatenation of symbol codes.
  - `codes`: mapping from symbol to bitstring.
  - `tree`: the root node for decoding.
  - Ratio and bit counts.

huffman_decompress(bitstring, tree, shape)

```
def huffman_decompress(bitstring: str,
                    tree: _HuffNode, shape):
    decoded_vals = []
    node = tree
    for bit in bitstring:
        node = node.left if bit == "0" else node.right
        if node.symbol is not None:
            decoded_vals.append(node.symbol)
            node = tree
    arr = np.array(decoded_vals, dtype=np.uint8)
    if arr.size < shape[0] * shape[1]:
        pad = shape[0] * shape[1] - arr.size
        arr = np.pad(arr, (0, pad), mode="edge")
    return arr.reshape(shape)
```

- Walks down the tree bit by bit.
- When a leaf is reached, appends its symbol and resets to root.
- Pads if needed to match original pixel count.

Golomb-Rice: `golomb_rice_encode` / `golomb_rice_decode`

```
def golomb_rice_encode(img, k: int = 2):
    gray = _to_gray_uint8(img)
    m = 1 << k
    bits = []
    for val in gray.ravel():
        q = val // m
        r = val % m
        bits.append("1" * q + "0")
        bits.append(format(r, f"0{k}b"))
    bitstring = "".join(bits)
    original_bits = gray.size * 8
    return {
        "bitstring": bitstring,
        "k": k,
        "ratio": _compression_ratio(original_bits,
                                    len(bitstring)),
        "original_bits": original_bits,
        "compressed_bits": len(bitstring),
    }
```

```
def golomb_rice_decode(bitstring: str, shape, k: int = 2):
    m = 1 << k
    values = []
    idx = 0
    while idx < len(bitstring) and \
         len(values) < shape[0] * shape[1]:
        q = 0
        while idx < len(bitstring) and \
             bitstring[idx] == "1":
            q += 1
            idx += 1
        idx += 1 # skip zero
        if idx + k > len(bitstring):
            break
        r = int(bitstring[idx:idx + k], 2)
        idx += k
        values.append(q * m + r)
    arr = np.array(values, dtype=np.uint8)
    if arr.size < shape[0] * shape[1]:
        arr = np.pad(arr,
                     (0, shape[0] * shape[1] - arr.size),
                     mode="edge")
    return arr.reshape(shape)
```

- Encodes each pixel as unary quotient followed by fixed-length remainder.
- Decoder reverses the process by reading unary part then remainder bits.

Arithmetic Coding (Entropy Estimate)

```
def arithmetic_encode(img):
    gray = _to_gray_uint8(img)
    data = gray.ravel().tolist()
    freq = Counter(data)
    total = sum(freq.values())
    import math
    entropy = -sum((f / total) * math.log2(f / total)
                   for f in freq.values() if f > 0)
    original_bits = len(data) * 8
    compressed_bits = max(1, int(entropy * len(data)))
    ratio = _compression_ratio(original_bits,
                               compressed_bits)
    original_bits = len(data) * 8
    return {
        "ratio": ratio,
        "original_bits": original_bits,
        "compressed_bits": compressed_bits,
        "shape": gray.shape,
        "reconstructed": gray,
    }
```

```
def arithmetic_decode(code, meta):
    # Identity reconstruction for demo
    return meta["reconstructed"]
```

- Does not implement full arithmetic coding; instead estimates ideal compression using entropy.
- Returns the original grayscale image as “reconstructed”.

LZW: `lzw_encode` / `lzw_decode`

```
def lzw_encode(img):
    gray = _to_gray_uint8(img)
    data = gray.ravel().tolist()
    dict_size = 256
    dictionary = {bytes([i]): i for i in range(dict_size)}
    w = bytes([data[0]])
    codes = []
    for k in data[1:]:
        wk = w + bytes([k])
        if wk in dictionary:
            w = wk
        else:
            codes.append(dictionary[w])
            dictionary[wk] = dict_size
            dict_size += 1
            w = bytes([k])
    codes.append(dictionary[w])
    original_bits = len(data) * 8
    compressed_bits = len(codes) * \
        math.ceil(math.log2(dict_size + 1))
    return {
        "codes": codes,
        "dict_size": dict_size,
        "ratio": _compression_ratio(original_bits,
                                    compressed_bits),
        "original_bits": original_bits,
        "compressed_bits": compressed_bits,
        "shape": gray.shape,
    }
```

```
def lzw_decode(codes, shape):
    dict_size = 256
    dictionary = {i: bytes([i]) for i in range(dict_size)}
    w = bytes([codes[0]])
    result = bytearray(w)
    for k in codes[1:]:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + bytes([w[0]])
        else:
            entry = bytes([0])
        result.extend(entry)
        dictionary[dict_size] = w + bytes([entry[0]])
        dict_size += 1
        w = entry
    arr = np.frombuffer(result, dtype=np.uint8)
    return arr[:shape[0] * shape[1]].reshape(shape)
```

- Classic LZW with a dynamically built dictionary of byte sequences.

Run-Length Encoding: *`rle_encode`* */* *`rle_decode`*

```
def rle_encode(img):
    gray = _to_gray_uint8(img)
    data = gray.ravel()
    pairs = []
    count = 1
    for i in range(1, len(data)):
        if data[i] == data[i - 1]:
            count += 1
        else:
            pairs.append((int(data[i - 1]), count))
            count = 1
    pairs.append((int(data[-1]), count))
    bit_estimate = len(pairs) * (8 + 16)
    ratio = _compression_ratio(len(data) * 8, bit_estimate)
    return {"pairs": pairs, "ratio": ratio,
            "original_bits": len(data) * 8,
            "compressed_bits": bit_estimate,
            "shape": gray.shape}
```

```
def rle_decode(pairs, shape):
    values = []
    for val, count in pairs:
        values.extend([val] * count)
    arr = np.array(values, dtype=np.uint8)
    return arr[:shape[0] * shape[1]].reshape(shape)
```

Symbol-Based Coding: `symbol_based_encode` / `symbol_based_decode`

```
def symbol_based_encode(img):
    gray = _to_gray_uint8(img)
    freq = Counter(gray.ravel().tolist())
    sorted_symbols = [s for s, _ in freq.most_common()]
    bits_needed = max(1, math.ceil(
        math.log2(len(sorted_symbols) or 1)))
    codes = {sym: format(idx, f"0{bits_needed}b")
             for idx, sym in enumerate(sorted_symbols)}
    bitstring = "".join(codes[int(v)] for v in gray.ravel())
    compressed_bits = len(bitstring)
    original_bits = gray.size * 8
    return {
        "codes": codes,
        "bitstring": bitstring,
        "bits_len": bits_needed,
        "ratio": _compression_ratio(original_bits,
                                    compressed_bits),
        "original_bits": original_bits,
        "compressed_bits": compressed_bits,
        "shape": gray.shape,
    }
```

```
def symbol_based_decode(bitstring: str, codes: dict,
                        shape):
    reverse = {v: k for k, v in codes.items()}
    bits_len = max(len(k) for k in reverse.keys())
    values = []
    for i in range(0, len(bitstring), bits_len):
        chunk = bitstring[i:i + bits_len]
        if chunk in reverse:
            values.append(reverse[chunk])
    arr = np.array(values, dtype=np.uint8)
    target = shape[0] * shape[1]
    if arr.size < target:
        arr = np.pad(arr, (0, target - arr.size),
                     mode="edge")
    if arr.size > target:
        arr = arr[:target]
    return arr.reshape(shape)
```

Bit-Planes: `bit_planes(img)`

```python
def bit_planes(img):
    gray = _to_gray_uint8(img)
    planes = []
    for i in range(8):
        plane = ((gray >> i) & 1) * 255
        planes.append(plane.astype(np.float32))
    reconstructed = sum(((planes[i] > 0).astype(np.uint8)
                        << i) for i in range(8)).astype(np.float32)
    return planes, reconstructed
```

- Extracts each bit as a separate plane (0 or 255).
- Reconstructs by shifting planes back and summing.

DCT Utility: `_dct_matrix(n=8)` *and Global Matrices*

```python
def _dct_matrix(n=8):
    C = np.zeros((n, n))
    for k in range(n):
        for i in range(n):
            alpha = math.sqrt(1 / n) if k == 0 \
                    else math.sqrt(2 / n)
            C[k, i] = alpha * math.cos(
                (math.pi * (2 * i + 1) * k) / (2 * n)
            )
    return C

DCT_MATRIX = _dct_matrix()
DCT_MATRIX_T = DCT_MATRIX.T
```

- Builds the 1D DCT transform matrix for 8-point DCT.
- Precomputes matrix and its transpose to be reused if needed.

`dct_compress(img, keep_ratio=0.5)`

```python
def dct_compress(img, keep_ratio: float = 0.5):
    gray = _to_gray_uint8(img)
    h, w = gray.shape
    h8, w8 = h - (h % 8), w - (w % 8)
    gray = gray[:h8, :w8]
    reconstructed = np.zeros_like(gray, dtype=np.float32)
    total_coeffs = kept = 0
    threshold = None
    for y in range(0, h8, 8):
        for x in range(0, w8, 8):
           block = gray[y:y+8, x:x+8].astype(np.float32) - 128
           dct = cv2.dct(block)
           flat = np.abs(dct).ravel()
           total_coeffs += flat.size
           k = max(1, int(flat.size * keep_ratio))
           threshold = np.partition(flat, -k)[-k]
           mask = (np.abs(dct) >= threshold).astype(np.float32)
           kept += int(np.count_nonzero(mask))
           dct_filtered = dct * mask
           block_rec = cv2.idct(dct_filtered) + 128
           reconstructed[y:y+8, x:x+8] = block_rec
    compressed_bits = kept * 16
    original_bits = gray.size * 8
    return {
        "image": reconstructed,
        "ratio": _compression_ratio(original_bits,
                                 compressed_bits),
        "kept_coefficients": kept,
        "total_coefficients": total_coeffs,
        "threshold": threshold,
    }
```

- For each 8x8 block, keeps only the largest `keep_ratio` of coefficients (by magnitude).
- Approximates bit cost by assuming 16 bits per kept coefficient.

Predictive Coding: `predictive_encode` / `predictive_decode`

```
def predictive_encode(img):
    gray = _to_gray_uint8(img)
    residual = np.zeros_like(gray, dtype=np.int16)
    predictor = np.zeros_like(gray, dtype=np.uint8)
    for y in range(gray.shape[0]):
        for x in range(gray.shape[1]):
            pred = gray[y, x - 1] if x > 0 else 0
            predictor[y, x] = pred
            residual[y, x] = int(gray[y, x]) - int(pred)
    compressed_bits = residual.size * 4 # assume entropy coding halves size
    ratio = _compression_ratio(gray.size * 8, compressed_bits)
    return {"residual": residual,
            "predictor": predictor,
            "ratio": ratio}
```

```python
def predictive_decode(residual):
    h, w = residual.shape
    reconstructed = np.zeros((h, w), dtype=np.int16)
    for y in range(h):
        for x in range(w):
            pred = reconstructed[y, x - 1] if x > 0 else 0
            reconstructed[y, x] = pred + residual[y, x]
    return np.clip(reconstructed, 0, 255).astype(np.uint8)
```

Haar Wavelet: *`haar_wavelet_transform`* / *`haar_wavelet_inverse`*

```python
def haar_wavelet_transform(img):
    gray = _to_gray_uint8(img).astype(np.float32)
    h, w = gray.shape
    h2, w2 = (h + 1) // 2, (w + 1) // 2
    approx = np.zeros((h2, w2), dtype=np.float32)
    horiz = np.zeros_like(approx)
    vert = np.zeros_like(approx)
    diag = np.zeros_like(approx)
    for y in range(0, h, 2):
        for x in range(0, w, 2):
            a = gray[y, x]
            b = gray[y, x + 1] if x + 1 < w else a
            c = gray[y + 1, x] if y + 1 < h else a
            d = gray[y + 1, x + 1] \
                if (y + 1 < h and x + 1 < w) else a
            idx_y = y // 2
            idx_x = x // 2
            approx[idx_y, idx_x] = (a + b + c + d) / 4
            horiz[idx_y, idx_x] = (a + c - b - d) / 4
            vert[idx_y, idx_x] = (a + b - c - d) / 4
            diag[idx_y, idx_x] = (a - b - c + d) / 4
    return approx, horiz, vert, diag, (h, w)
```

```python
def haar_wavelet_inverse(approx, horiz, vert, diag,
                         original_shape=None):
    h2, w2 = approx.shape
    h, w = h2 * 2, w2 * 2
    reconstructed = np.zeros((h, w), dtype=np.float32)
    for y in range(h2):
        for x in range(w2):
            a = approx[y, x] + horiz[y, x] + vert[y, x] + diag[y, x]
            b = approx[y, x] - horiz[y, x] + vert[y, x] - diag[y, x]
            c = approx[y, x] + horiz[y, x] - vert[y, x] - diag[y, x]
            d = approx[y, x] - horiz[y, x] - vert[y, x] + diag[y, x]
            reconstructed[2 * y, 2 * x] = a
            reconstructed[2 * y, 2 * x + 1] = b
            reconstructed[2 * y + 1, 2 * x] = c
            reconstructed[2 * y + 1, 2 * x + 1] = d
    reconstructed = np.clip(reconstructed, 0, 255)
    if original_shape:
        oh, ow = original_shape
        return reconstructed[:oh, :ow]
    return reconstructed
```

4.7 server.py (Flask Web API)

Flask App Setup

```
COMPRESSION_MAX_PIXELS = 200_000

app = Flask(__name__, static_folder="static",
            static_url_path="/static")
```

- `COMPRESSION_MAX_PIXELS` limits image size used for compression demos to keep responses fast.
- `static_folder` and `static_url_path` tell Flask where to serve `index.html` and assets from.

_decode_image(data_url, max_dim=1600)

```
def _decode_image(data_url: str, max_dim: int = 1600) \
        -> tuple[np.ndarray, dict]:
    """Decode base64 data URL to RGB float32 numpy array,
    with optional downscale for performance."""
    if "," in data_url:
        data_url = data_url.split(",", 1)[1]
    img_bytes = base64.b64decode(data_url)
    buf = np.frombuffer(img_bytes, np.uint8)
    bgr = cv2.imdecode(buf, cv2.IMREAD_COLOR)
    if bgr is None:
        raise ValueError("Failed to decode image.")
    h, w = bgr.shape[:2]
    meta = {"downsized": False,
            "original_size": (w, h)}
    if max(h, w) > max_dim:
        ratio = max_dim / max(h, w)
        new_w, new_h = int(w * ratio), int(h * ratio)
        bgr = cv2.resize(bgr, (new_w, new_h),
                         interpolation=cv2.INTER_AREA)
        meta["downsized"] = True
        meta["new_size"] = (new_w, new_h)
    rgb = cv2.cvtColor(bgr, cv2.COLOR_BGR2RGB)
    return rgb.astype(np.float32), meta
```

- Strips the `"data:image/...;base64,"` prefix.
- Base64-decodes and uses `cv2.imdecode` to get a BGR image.
- Optionally downsizes to at most `max_dim` pixels in the larger dimension.
- Returns the RGB float32 array and a metadata dict with resizing info.

`_encode_image(img)`

```
def _encode_image(img: np.ndarray) -> str:
    """Encode numpy image to base64 PNG data URL."""
    if img.ndim == 2:
        img_to_save = cv2.cvtColor(io_utils.ensure_uint8(img),
                                   cv2.COLOR_GRAY2RGB)
    else:
        img_to_save = cv2.cvtColor(io_utils.ensure_uint8(img),
                                   cv2.COLOR_RGB2BGR)
    success, buffer = cv2.imencode(".png", img_to_save)
    if not success:
        raise ValueError("Failed to encode image.")
    b64 = base64.b64encode(buffer).decode("utf-8")
    return f"data:image/png;base64,{b64}"
```

- Ensures grayscale images are converted to 3-channel RGB before saving.
- Encodes to PNG, then base64, returning a full data URL.

`_process_request(img, action, params, decode_meta)`

```
def _process_request(img: np.ndarray, action: str,
                     params: dict, decode_meta: dict
                     ) -> Tuple[np.ndarray, dict]:
    """Apply requested operation and return (image, extra_info)."""
    extra = {}
    if decode_meta.get("downsized"):
        extra["downsized_from"] = decode_meta["original_size"]
        extra["processed_size"] = decode_meta["new_size"]
    gray = basic_ops.rgb_to_grayscale(img)
    act = action.lower()
    ...
```

- Central dispatcher for all actions from the web UI.
- Creates an `extra` dictionary to send metadata back (histograms, ratios, etc.).
- For each `action` string, it calls the appropriate function in `basic_ops`, `filters`, `geometry`, `interp`, or `compress`.
- Returns:
  - The processed image.
  - The `extra` information dictionary.

Compression actions use a guard:

```
if act in {"huffman", "golomb", "arithmetic", "lzw",
           "rle", "symbol", "bitplane", "dct",
           "predictive", "wavelet"}:
    if gray.size > COMPRESSION_MAX_PIXELS:
        ...
        img = cv2.resize(img, (new_w, new_h),
                         interpolation=cv2.INTER_AREA)
        gray = basic_ops.rgb_to_grayscale(img)
        extra["compression_downscaled_from"] = (w, h)
        extra["compression_size"] = (new_w, new_h)
```

- Ensures compression algorithms never receive a huge image (for performance reasons).

Routes: `index` and `/api/process`

```
@app.route("/")
def index():
    return send_from_directory(app.static_folder,
                               "index.html")
```

- Serves the web app's front-end HTML.

```
@app.route("/api/process", methods=["POST"])
def process_image():
    try:
        payload = request.get_json(force=True)
        img_data = payload.get("image")
        action = payload.get("action")
        params = payload.get("params", {})
        if img_data is None or action is None:
            return jsonify({"error": "Missing image or action."}), 400

        img, decode_meta = _decode_image(img_data)
        result_img, extra = _process_request(img, action,
                                             params, decode_meta)
        encoded = _encode_image(result_img)
        info = io_utils.info(result_img)
        return jsonify({"image": encoded,
                        "info": info,
                        "extra": extra})
    except ValueError as exc:
        return jsonify({"error": str(exc)}), 400
    except Exception as exc: # noqa: BLE001
        return jsonify({"error": str(exc)}), 500
```

- Reads JSON payload from client (image data URL + action + params).
- Decodes the image, processes it, encodes result back to data URL.
- Returns image, basic info, and extra metadata as JSON.
- Uses proper HTTP status codes for errors (400 for bad input, 500 for unexpected errors).

4.8 `app.py` (Tkinter Entry Point)

`main()`

```
import tkinter as tk

from ui.controls import ImageApp

def main():
    root = tk.Tk()
    root.title("Image Processing Studio")
    root.geometry("1100x720")
    root.configure(bg="#111827")
    ImageApp(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

- Creates the root Tk window with a title, fixed geometry, and dark background.
- Instantiates `ImageApp`, which builds all controls inside the root.

- Starts the Tk event loop via `root.mainloop()`, keeping the application running.

5 End-to-End Flow

1. **Load**: Desktop uses a file dialog (`controls.load_image` + `io_utils.load_image`); web API decodes base64 in `server._decode_image`.
2. **Dispatch**: User action maps to a processing function (`_process_request` for web; button handlers in `controls.py` for desktop).
3. **Process**: cv2-backed operations run in `basic_ops`, `filters`, `geometry`, `interp`; compression uses `compress`.
4. **Return**: Desktop shows via `ImagePreview`; web re-encodes PNG data URL and returns JSON with extras (histograms, ratios, scale guards).
5. **Display**: UI updates status/info panels; web frontend renders the returned data URL and metadata.

6 Conclusion

This document describes how the Tkinter-based **Image Processing Studio** UI, the processing modules, and the Flask web API orchestrate a wide variety of image processing and compression algorithms implemented in the `processing` package. The application combines:

- A structured UI with labelled sections and clear actions.
- A Matplotlib-based preview system to visualize original and processed images, as well as histograms.
- A collection of geometric transforms, filters, histogram operations, and compression methods, each accessible via single-click buttons or web API calls.

The result is an educational and interactive environment for experimenting with image processing concepts and compression techniques, both on the desktop and through a browser.