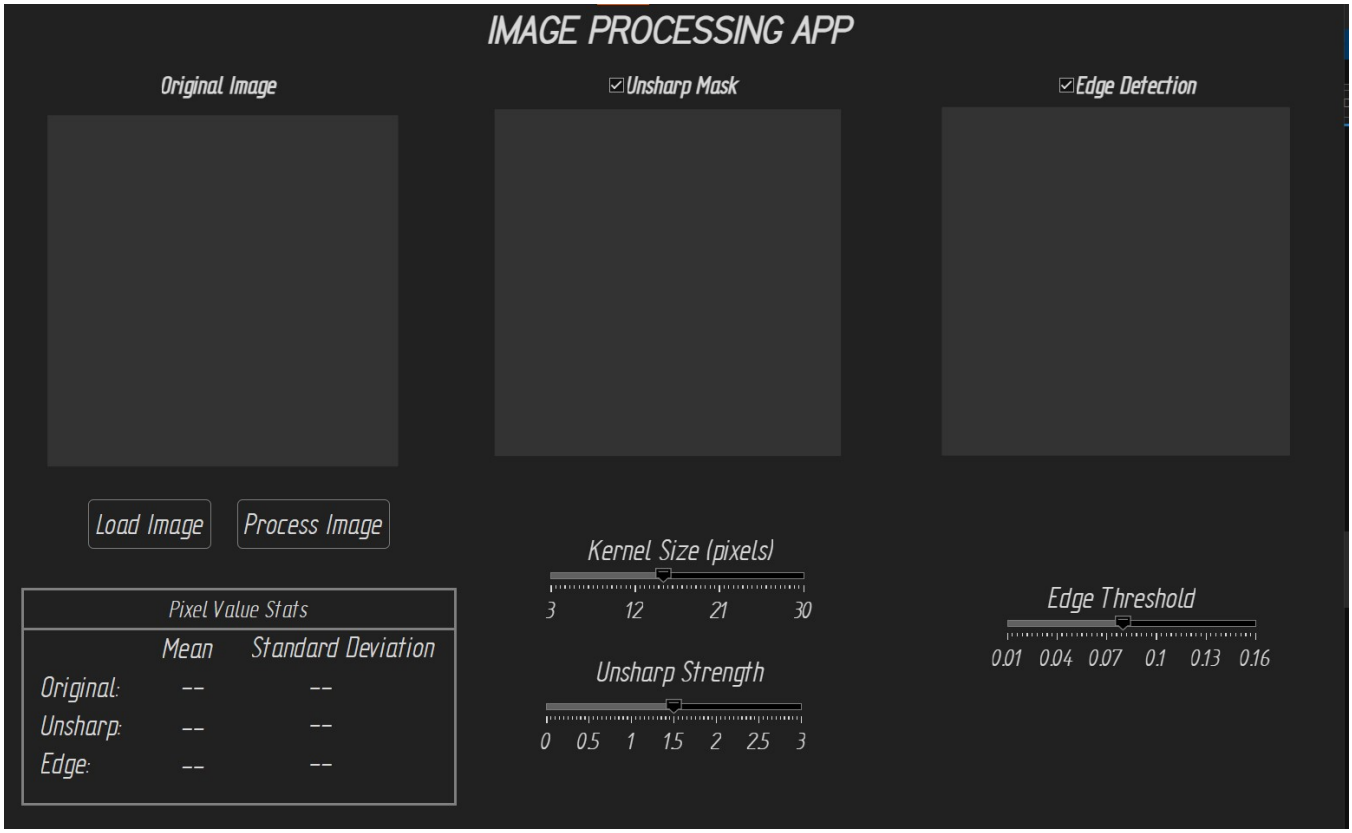
Technical Brief: Image Processing Application

Andrei Ayson

1. Overview

The Image Processing App lets users load an image and apply two filters: unsharp masking for sharpening and Prewitt edge detection. The app is built in MATLAB App Designer and includes checkboxes and sliders to turn filters on or off and adjust kernel size, alpha, and edge threshold. The app provides direct control over filter parameters and displays the processed image based on those settings.

Figure 1. Screenshot of the UI.



2. Algorithms Implemented

2.1 Grayscale Conversion

Images loaded with `imread` are typically stored as 8-bit integer arrays, so the app first converts them to `double` values in the range $[0, 1]$ for processing. If the image contains three color channels, the app computes a grayscale intensity image using the luminance model

$$I_{\text{gray}}(i, j) = 0.299\, R(i, j) + 0.587\, G(i, j) + 0.114\, B(i, j).$$

This operation is implemented in the helper function `rgbToGrayLuminance.m`, which extracts the red, green, and blue channels from the input image, applies the weighted sum above, and returns a single-channel grayscale image. If the input is already grayscale, the function simply returns it unchanged.

2.2 Unsharp Masking

Unsharp masking in the app is implemented in the function `unsharpFilter.m`. The method begins by forming a blurred version of the input image using a normalized box filter of odd size k. For a selected kernel size, the filter

$$h = \frac{1}{k^2}\,\mathbf{1}_{k\times k}$$

is applied with two–dimensional convolution to produce the blurred image

$$B(i,j) = (I * h)(i,j).$$

The sharpened output is then computed using

$$J(i,j) = I(i,j) + \alpha\big(I(i,j) - B(i,j)\big),$$

where $\alpha > 0$ is the sharpening strength chosen by the user. In the implementation, the blur is applied per channel for RGB inputs using `conv2`, and the final image is clipped to the range $[0,1]$.

2.3 Prewitt Edge Detection

Edges are detected by approximating horizontal and vertical image derivatives. For this project the Prewitt operators were used, which approximate partial derivatives with finite-difference stencils that include a small amount of smoothing. The kernels are

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \qquad K_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

Given a grayscale image I_{gray}, the horizontal and vertical derivatives are computed using two-dimensional convolution:

$$I_x = I_{\text{gray}} * K_x, \qquad I_y = I_{\text{gray}} * K_y.$$

These are combined into a gradient magnitude image,

$$G(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}\,.$$

Before thresholding, the gradient magnitude G is normalized to the range $[0,1]$, so the slider value acts as a direct threshold. A binary edge map is then obtained by

$$E = G > T$$

In the app, this procedure is implemented in the function `prewittEdges.m`, which applies the Prewitt kernels with `conv2`, computes the gradient magnitude, normalizes it, and applies the user-selected threshold.

3. Key Design Decisions

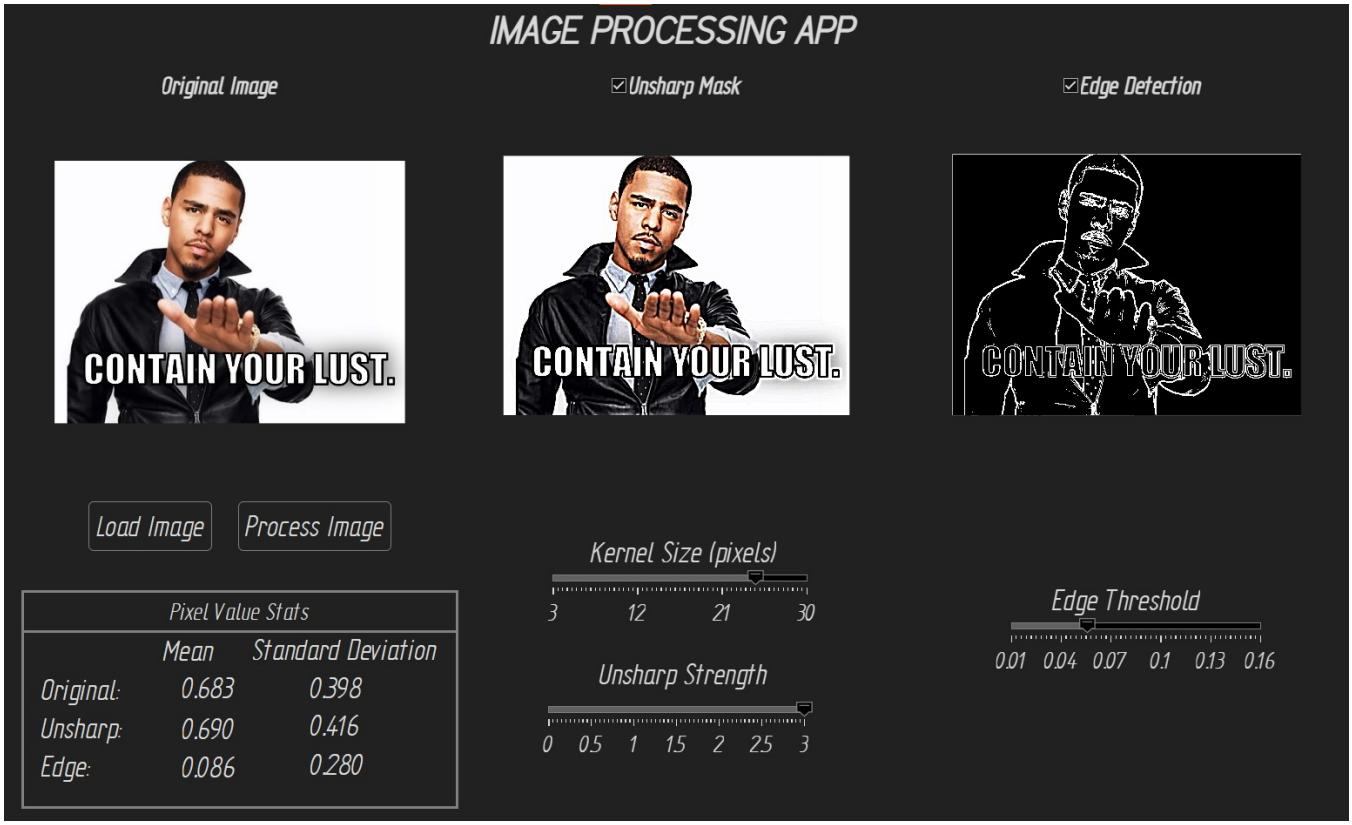
- **Separation of UI and algorithm code.** All computational functions were placed in individual files inside the `/src` folder. The App Designer callbacks simply pass user-selected parameters to these functions.

- **User adjustability.** Sliders were included for kernel size k, unsharp strength α, and threshold T so users can observe how each parameter affects the output.
- **Pixel value statistics.** Mean and standard deviation of the processed image are computed after each filtering step and displayed in the interface, giving users a simple way to quantify changes in intensity and contrast beyond the visual result.
- **Visual organization.** Three output axes were used to clearly display the original image, the sharpened image, and the edge map side by side.
- **Gray placeholders.** Empty axes were filled with neutral gray patches and labels to enhance the UI's visual clarity before images are loaded.

4. Test Results

Various images were used to check that the filters behaved as expected. Increasing α made the sharpening effect more noticeable, and larger kernel sizes k produced a smoother blur before sharpening. For edge detection, lowering the threshold T showed more edges, while higher values kept only the strongest outlines.

Figure 2. Example results showing (left) original image, (middle) unsharp mask output, and (right) Prewitt edge map.



AI Usage

ChatGPT 5.1, was used as a support tool during the development of this project, mainly to help with debugging, clarifying MATLAB syntax, and refining the structure of the processing functions. The design of the interface, layout decisions, parameter choices, and overall workflow were created without the use of AI, while AI assistance was used to iterate on code details such as convolution setup, slider handling, and separating functions into the `/src` folder. AI was also helpful for rewriting explanations, checking that mathematical expressions matched the implementation, and producing LaTeX for the technical brief.

To get accurate results, prompts were written with detailed context, including screenshots of code, error messages, and the relevant textbook excerpts. This made the AI responses more reliable and easier to adapt. Future students can benefit from a similar approach by prompting with straightforward "braindump" questioning. Prompts such as "this output looks wrong, can you help me figure out why?", "here is the kernel from the book, am I applying it correctly?", or "explain what this part of the equation means in code" tend to produce the most useful responses.

Sources

https://www.ultralytics.com/blog/edge-detection-in-image-processing-explained

https://www.shutterbug.com/content/sharpening-basics-here%E2%80%99s-how-unsharp-masking-makes-your-photos-look-extra-crispy

MATLAB Book - Dillon Allen