Home Project 2

Convolution and Filtering

CS180 • Fall 2025

Back to Home ›

1A – Convolution Filters

This project studies linear filtering and multi-scale image representations through a sequence of controlled experiments. In Part 1, I implement 2-D convolution from first principles, estimate spatial derivatives via finite differences, and replace naive differencing with derivative-of-Gaussian (DoG) filters to suppress noise. In Part 2, I treat filtering as a tool for manipulating frequency content: unsharp masking increases the energy of high spatial frequencies to "sharpen" imagery; hybrid images juxtapose low-frequency content from one image with high-frequency content from another; and Gaussian/Laplacian stacks enable multi-resolution blending that hides seams across scales. All results are generated by my own code; third-party routines are used only as references for verification and timing.

Dataset: Box Filter **k = 3** From scratch

SciPy

Box Filter — From scratch (k=3)

I implemented convolution in `proj2/conv.py` in two forms. The pedagogical baseline, `conv2d_four_loops`, explicitly flips the kernel and accumulates the product over a zero-padded neighborhood using four Python loops (two for spatial coordinates, two for kernel indices). The optimized variant, `conv2d_two_loops`, maintains identical semantics but reduces interpreter overhead by vectorizing the inner multiply–accumulate across complete row (or column) slices, leaving only two Python loops over output coordinates. For validation and runtime comparison I use `conv2d_scipy`, a thin wrapper around `scipy.signal.convolve2d` with *same* output size and *fill* (zero) boundary conditions. The box filter used in this section is created by `proj2/filters.box_filter(k)`, which returns a constant kernel whose entries sum to one, preserving DC gain. Visual and numerical comparisons confirm that both custom implementations match SciPy up to floating-point tolerance, while the two-loop version is substantially faster than four loops for moderate kernel sizes.

1B – Finite Difference Operator

To estimate spatial derivatives, I convolve a grayscale image with the forward-difference stencils `Dx = [-1, 1]` and `Dy = Dx`^{T} using `conv2d_scipy`. The partials `Ix` and `Iy` are then combined into a gradient magnitude image `|∇I| = sqrt(Ix² + Iy²)` (implemented in `proj2/edges.py`). Binarized “edge maps” are obtained by thresholding `|∇I|` at a fraction of its maximum. This experiment highlights the high-pass nature of naive differencing: while edges are revealed, amplification of sensor noise and texture is pronounced, motivating the smoothing strategy in the next subsection.

Ix Iy |∇I| Edges **t = 10**

Finite difference — edges (t=10)

1C – Derivative of Gaussian (DoG)

Smooth → Diff DoG (∂G ⊗ I) Ix Iy |∇I|

1C — DoG, |∇I|

I replace raw differences with derivatives of a Gaussian. A normalized Gaussian kernel G_σ is constructed by `proj2/filters.gaussian_kernel(ksize, σ)`, using `ksize ≈ 6σ + 1` to capture sufficient support. Two pipelines are compared: (i) smoothing the image with G_σ and then applying `Dx`/`Dy`; and (ii) forming the DoG filters `∂G/∂x`, `∂G/∂y` (by convolving `G` with the difference stencils) and applying a single convolution per axis. Both methods yield near-identical `Ix`, `Iy`, and `|∇I|` when σ and support are matched, but the DoG route is computationally attractive and conceptually cleaner. For the bells-and-whistles visualization, I compute orientations `θ = atan2(Iy, Ix)` and map them to hue in HSV, with value proportional to `|∇I|`; this exposes coherent edge directions across the scene.

2A – Unsharp Masking / Sharpening

Dataset: Taj ◉ Source ○ Blurred (low-pass) ○ High-pass ○ Sharpened

2A result

2A — Taj • Source

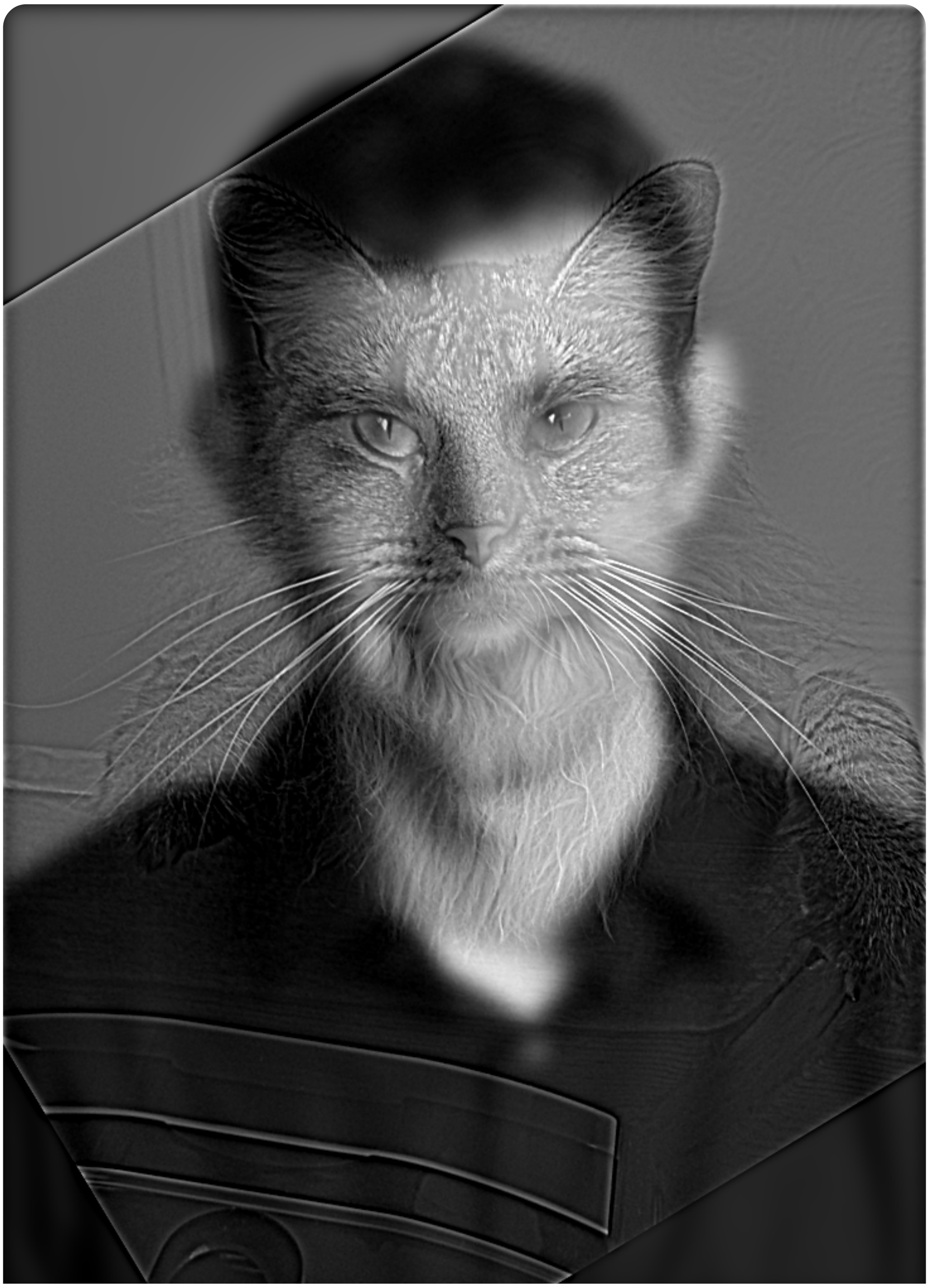
Unsharp masking is implemented in `proj2/sharpen.py`. A Gaussian blur $blur = G_\sigma \otimes I$ is subtracted from the input to isolate high frequencies `high = I − blur`; the sharpened output is `I′ = clip(I + α·high, 0, 1)`. Parameters σ and α govern the spatial scale and strength of the enhancement: increasing σ shifts the emphasis to broader features, whereas large α may introduce halos near strong edges. I also demonstrate reversal by first blurring a sharp image (setting $\alpha = -1$ to visualize the low-pass) and then attempting to re-sharpen it; the comparison clarifies that sharpening restores local contrast but cannot recreate frequencies that have been eliminated by the blur.

2B – Hybrid Images

Set: Example (A+B) ○ A (low-freq carrier) ○ B (high-freq detail) ○ Low-pass(A)

○ High-pass(B) ◉ Hybrid ○ FFT(A) ○ FFT(B) ○ FFT(Low) ○ FFT(High)

○ FFT(Hybrid)



2B — Example • Hybrid

The hybrid construction in `proj2/hybrid.py` forms $H = LP_{\sigma L}(A) + HP_{\sigma H}(B)$, where $HP(B) = B - LP_{\sigma H}(B)$. The function `hybrid_image(A,B, low_ksize, σL, high_ksize, σH)` returns the low-pass of `A`, the high-pass of `B`, their clipped sum, and a log-magnitude FFT for analysis (`log_fft`). Images are pre-aligned and resized via `proj2/io_utils.match_size` so that semantic structures are co-located. Successful hybrids rely on three controls: accurate geometric alignment (to avoid double features), a sufficiently large `σL` that removes mid-frequencies from the carrier, and a moderate `σH` that preserves crisp detail without injecting noise. I tune cutoffs by inspecting the FFTs and by checking whether the interpretation of `H` flips with viewing distance as intended.

2C – Gaussian & Laplacian Stacks

Image: Apple ◉ Gaussian Stack ○ Laplacian Stack

2C — Gaussian stack • Apple

In stacks.py, `gaussian_stack(img, L, k, σ)` generates a stack of `L` images at full resolution by repeated blurring with the same σ; `laplacian_stack` stores differences between adjacent Gaussian levels, with the coarsest Gaussian retained as the residual. I visualize stacks using `proj2/viz.save_stack_grid`. Unlike pyramids, stacks avoid down-sampling and simplify subsequent per-pixel operations (e.g., stack-wise masking) at the expense of memory. The Laplacian representation is a band-pass decomposition of the image and is a natural basis for frequency-aware blending.

2D – Multiresolution Blending

Mask: Vertical ◉ Result (Oraple) ○ Mask Stack ○ Laplacian Blended Stack

2D — Result (Oraple) • Vertical mask

The blender in `proj2/blend.py` implements the Burt–Adelson recipe using stacks. Given two color images `A`, `B` and a mask `M∈[0,1]`, I build Laplacian stacks `LA`, `LB` and a Gaussian stack of the mask `GM`. At each level `i`, I compute L_{blend}`[i] = GM[i]·LA[i] + (1−GM[i])·LB[i]`, then collapse the blended Laplacian stack to reconstruct the output. A hard vertical step mask reproduces the classic "oraple" result; an irregular, feathered mask demonstrates that spatially varying, scale-appropriate smoothing of the seam yields perceptually seamless composites even across textured regions.

JS: 2B viewer