Project 2 – Fun with Filters and Frequencies

CS180: Intro to Computer Vision & Computational Photography

by *Mansoor Mamnoon*

1.1 Convolutions from Scratch
1.2 Finite Difference Operator
1.3 Derivative of Gaussian (DoG)
Part 2 — Fun with Frequencies
2.1 Image “Sharpening”
2.2 Hybrid Images
2.3 Gauss & Laplace Stacks
2.4 Multires Blending (Oraple + Irregular)

Parameter Glossary — what each symbol/field means

These are the exact parameters used across Parts 1–2.

- **`size`** : kernel width/height in pixels (e.g., 9 → 9×9).
- **`σ (sigma)`**: Gaussian standard deviation; larger σ → stronger blur / lower cutoff.
- **`α (alpha)`**: unsharp “amount”; scales the high frequencies added back in 2.1.
- **`low_size, low_sigma`** : Gaussian used to create the low-pass image in hybrids (2.2).
- **`high_size, high_sigma`** : Gaussian used to blur before subtracting to make the high-pass in hybrids (2.2).
- **`levels`** : number of stack levels in Gaussian/Laplacian stacks for blending (2.3–2.4).
- **`mask`** : per-pixel weights in [0,1] used to mix images; blurred across levels to hide seams.
- **`ramp`** : width of the cosine transition in soft masks (when applicable).
- **`mode="same"`** : convolution mode that keeps output size equal to input (1.1–1.3).
- **`boundary="fill"`** : zero padding outside image bounds during convolution (1.1–1.3, 2.1).
- **`D_x=[1,0,-1], D_y=D_x^T`** : finite-difference derivative filters (1.2).
- **`t`** : threshold on normalized gradient magnitude for binary edges (1.2–1.3).

Part 1.1: Convolutions from Scratch

I implemented 2D convolution two ways with numpy (4-loop and 2-loop). I use zero padding so output size matches input.

input

selfie used for box / Dx / Dy tests. (box size=9; Dx=[1,0,-1]; Dy=Dx^T)

what i implement

- zero padding with fill 0 (same spatial size)
- 4-loop convolution (explicit multiplications)
- 2-loop convolution (vectorized window dot-product)
- 9×9 box, $D_x = [1, 0, -1]$, $D_y = [1, 0, -1]^T$

how i compute “difference” vs scipy

i compare my outputs with `scipy.signal.convolve2d` using the same settings (`mode="same"`, `boundary="fill"`, `fillvalue=0`) and a flipped kernel on the scipy call so both are true convolution (not correlation). i report:

- *max abs diff*: $\max_{y,x} |A(y,x) - B(y,x)|$
- (and i save the numbers to `out/q1_1/*.txt` from my script)

snippet (from my script):

```
boxs = convolve2d(img, B9[::-1, ::-1], mode="same", boundary="fill", fill
      dxs  = convolve2d(img, Dx[::-1, ::-1], mode="same", boundary="fill"
      dys  = convolve2d(img, Dy[::-1, ::-1], mode="same", boundary="fill"

      diff = np.max(np.abs(my_result - scipy_result))  # number i report
```

values around 1e−7 mean they're the same up to float rounding. larger values usually come from boundary or kernel flip mismatches.

code (4 loops + padding)

```
def conv_4_loops(img, k):  # zero-pad;
          hi, wi = img.shape
          hk, wk = k.shape
          ph, pw = hk//2, wk//2  # zero padding
          padded = np.pad(img, ((ph,ph),(pw,pw)), mode='constant', constant_val
          out = np.zeros_like(img)
          for y in range(hi):
              for x in range(wi):
                  acc = 0.0
                  for j in range(hk):
                      for i in range(wk):
                          acc = acc + k[j,i] * padded[y+j, x+i]
                  out[y,x] = acc
          return out
```

what this block does

- slides the flipped kernel over a zero-padded image (true convolution)
- keeps output height/width identical to input

4-loop 9×9 box on selfie. (size=9, mode="same", boundary="fill")

diff vs SciPy	3.57627869e-07 (max abs)
runtime	slowest (baseline)
boundary	zero fill

code (2 loops + padding)

```
def conv_2_loops(img, k):  # window multiply-sum
          hi, wi = img.shape
          hk, wk = k.shape
          ph, pw = hk//2, wk//2
          padded = np.pad(img, ((ph,ph),(pw,pw)), mode='constant', constant_val
          out = np.zeros_like(img)
          for y in range(hi):
              row = padded[y:y+hk, :]
              for x in range(wi):
                  win = row[:, x:x+wk]
                  out[y,x] = np.sum(win * k)  # dot-product on the window
```

```
        return out
```

what this block does

- same math as 4-loop, but uses a vectorized multiply–sum per pixel
- cleaner and much faster

2-loop 9×9 box on selfie. (size=9, mode="same", boundary="fill")

diff vs SciPy	3.57627869e-07 (max abs)
diff vs 4-loop	9.53674316e-07 (max abs)
runtime	much faster than 4-loop

finite differences

```
D_x = np.array([[1., 0., -1.]], dtype=np.float32)
      D_y = D_x.T
```

```
gx_2 = conv_2_loops(img, D_x)
gy_2 = conv_2_loops(img, D_y)
gx_4 = conv_4_loops(img, D_x)
gy_4 = conv_4_loops(img, D_y)
```

why some diffs are larger here

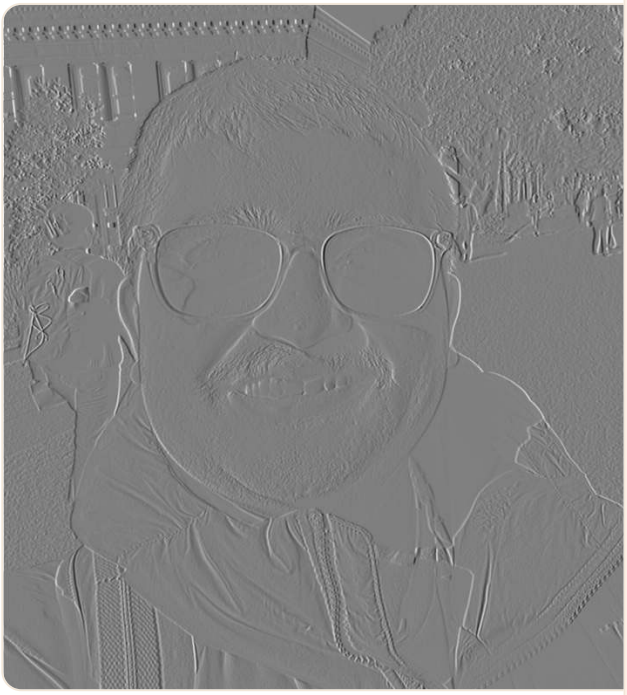
with zero padding, a sharp 0↔1 edge can yield responses near 2 for $[1, 0, -1]$. if kernel orientation or boundary conventions don't match exactly, you'll see diffs around ~2 even when the interior matches. i match both flip and boundary when i compare.

Dx (2-loop) vs SciPy	1.99215686e+00
Dx (4-loop) vs Dx (2-loop)	0.00000000e+00
Dx (4-loop) vs SciPy	1.99215686e+00
Dy (2-loop) vs SciPy	1.78039217e+00
Dy (4-loop) vs SciPy	1.78039217e+00

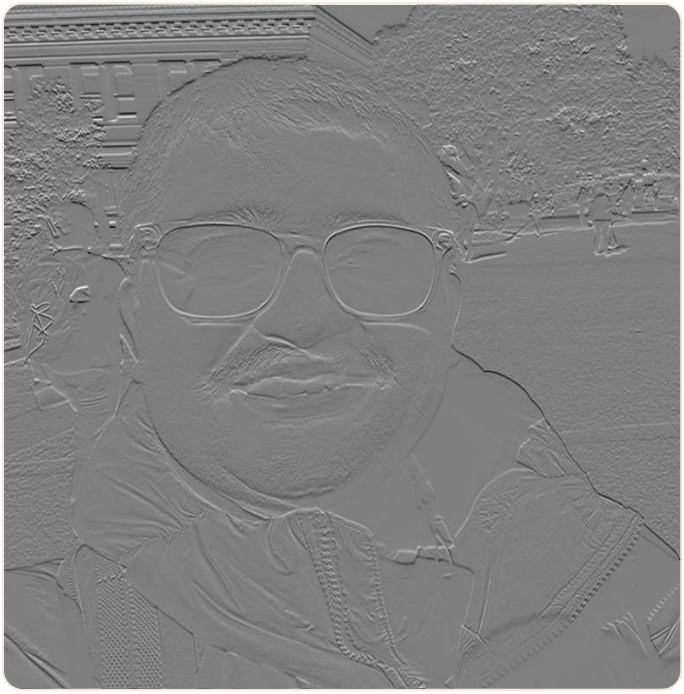
these match my `out/q1_1/*.txt` logs. (filters: Dx=[1,0,-1], Dy=Dx^T; mode="same", boundary="fill")

results

box (2-loop). (size=9, mode="same")



Dx (2-loop). (kernel=[1,0,-1], mode="same")



Dy (2-loop). (kernel=[1,0,-1]ᵀ, mode="same")

how i compare to scipy

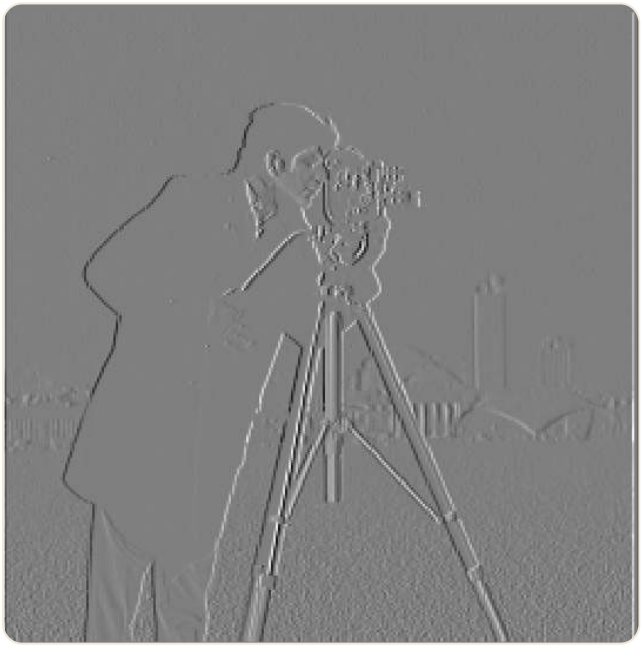
same padding (zero), same size (`same`). i report max-abs difference for equivalence. for timing i run each method multiple times and compare medians. 2-loop beats 4-loop by a lot (scipy's compiled code is still fastest).

> **takeaway:** writing both versions made convolution "click." moving from 4 loops to a 2-loop window dot-product gives a big speedup with the same output. matching kernel flip and boundary is key when comparing to scipy.

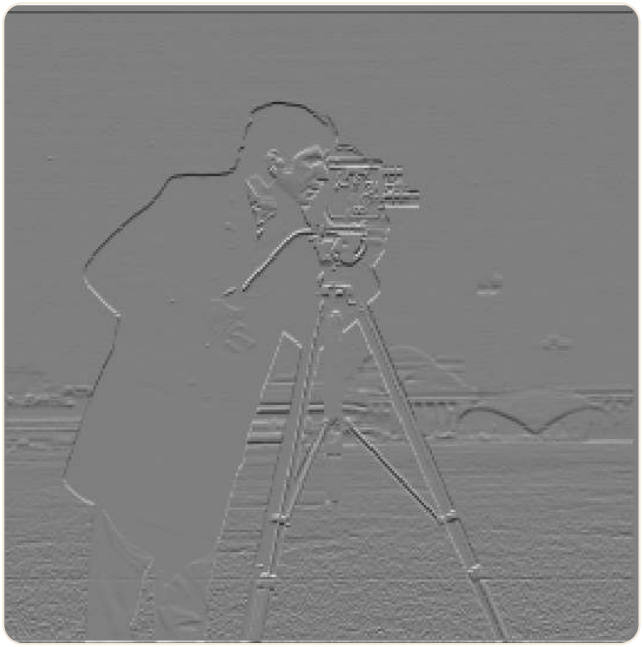
Part 1.2: Finite Difference Operator

I start with the cameraman image and take simple finite differences in x and y: $D_x = [1, 0, -1]$ and $D_y = D_x^\top$. These show the signed rate of change horizontally and vertically. From there I combine them into a gradient magnitude image and finally decide on a binary edge map by thresholding, trading off noise against completeness of real edges.

Signed partial derivatives



∂I/∂x highlights vertical edges (left↔right contrast). (kernel=Dx=[1,0,-1])



∂I/∂y highlights horizontal edges (top↔bottom contrast). (kernel=Dy=Dx^T)

How I compute them

I convolve the grayscale image with D_x and D_y using `scipy.signal.convolve2d` (`mode="same"`, `boundary="fill"`, `fillvalue=0`) so the output size matches the input. This matches my conventions in 1.1 (true convolution with zero padding).

Gradient magnitude

$\|\nabla I\| = \sqrt{(g_x^2 + g_y^2)}$. (built from Dx/Dy responses)

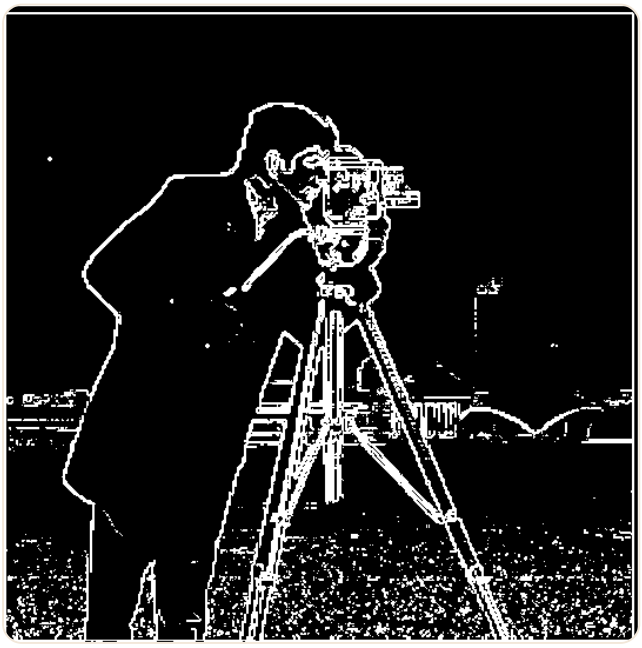
Why magnitude?

Combining g_x and g_y removes sign and strengthens real boundaries even when they are slanted. It also tones down checkerboard artifacts that can show up in the separate partials.

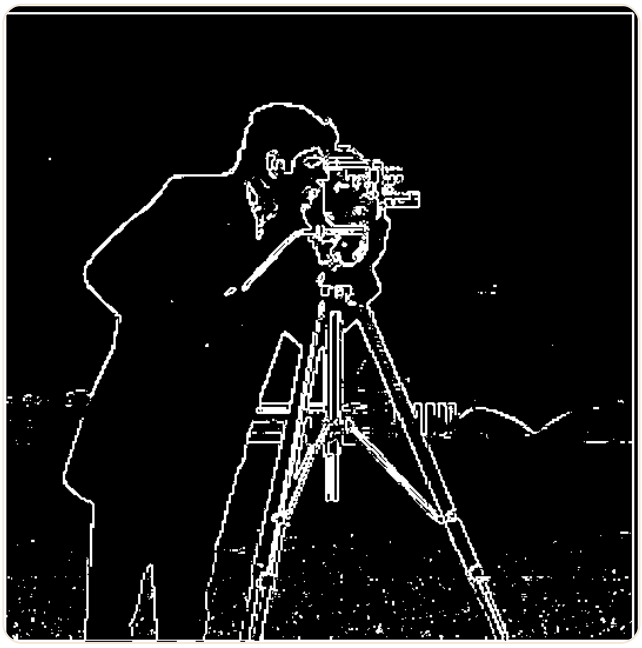
What is t?

I first normalize the gradient magnitude to $[0, 1]$ for display. Then I produce a binary edge map by keeping pixels whose strength is at least t: $\text{edges}(y, x) = \mathbf{1}\{\|\nabla I(y, x)\| \geq t\}$. I swept $t \in \{0.10, 0.15, 0.20, 0.25, 0.30, 0.35\}$ and saved each result for visual comparison.

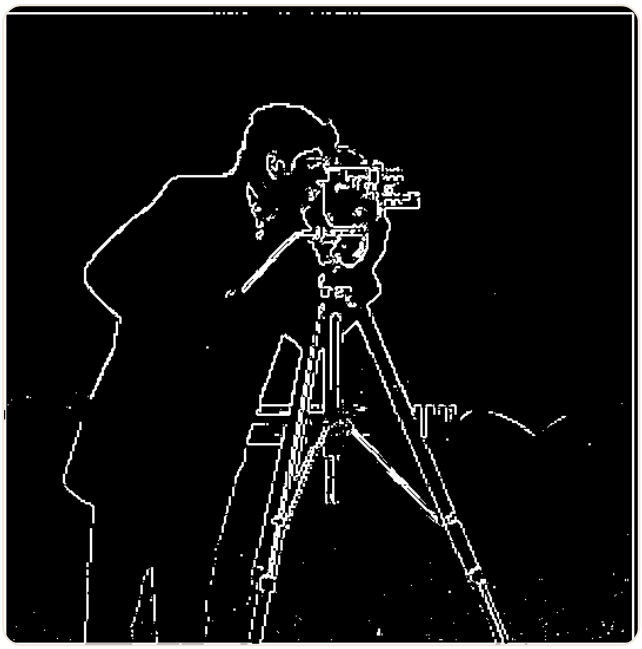
Threshold sweep (for comparison)



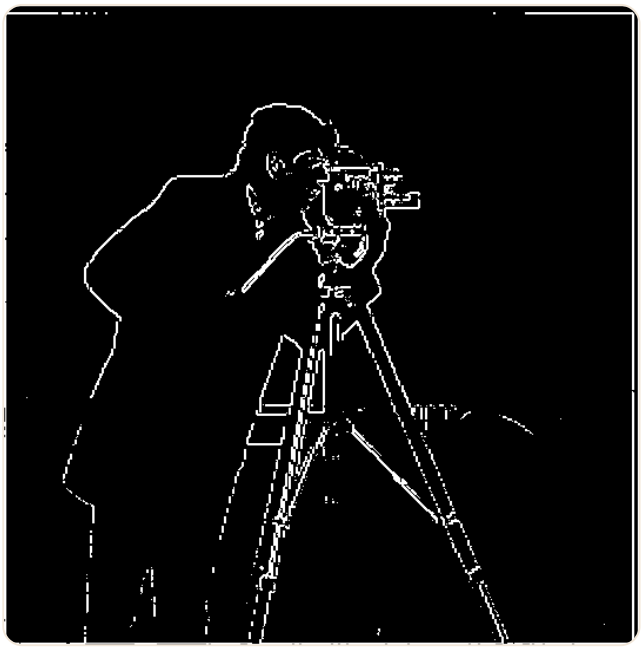
t = 0.10 (threshold on normalized ‖∇I‖)



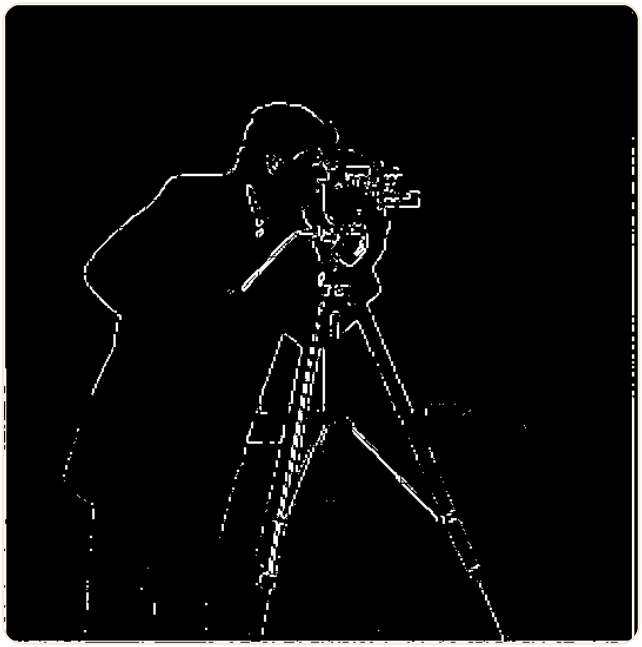
t = 0.15 (threshold on normalized ‖∇I‖)



t = 0.20 (threshold on normalized ‖∇I‖)



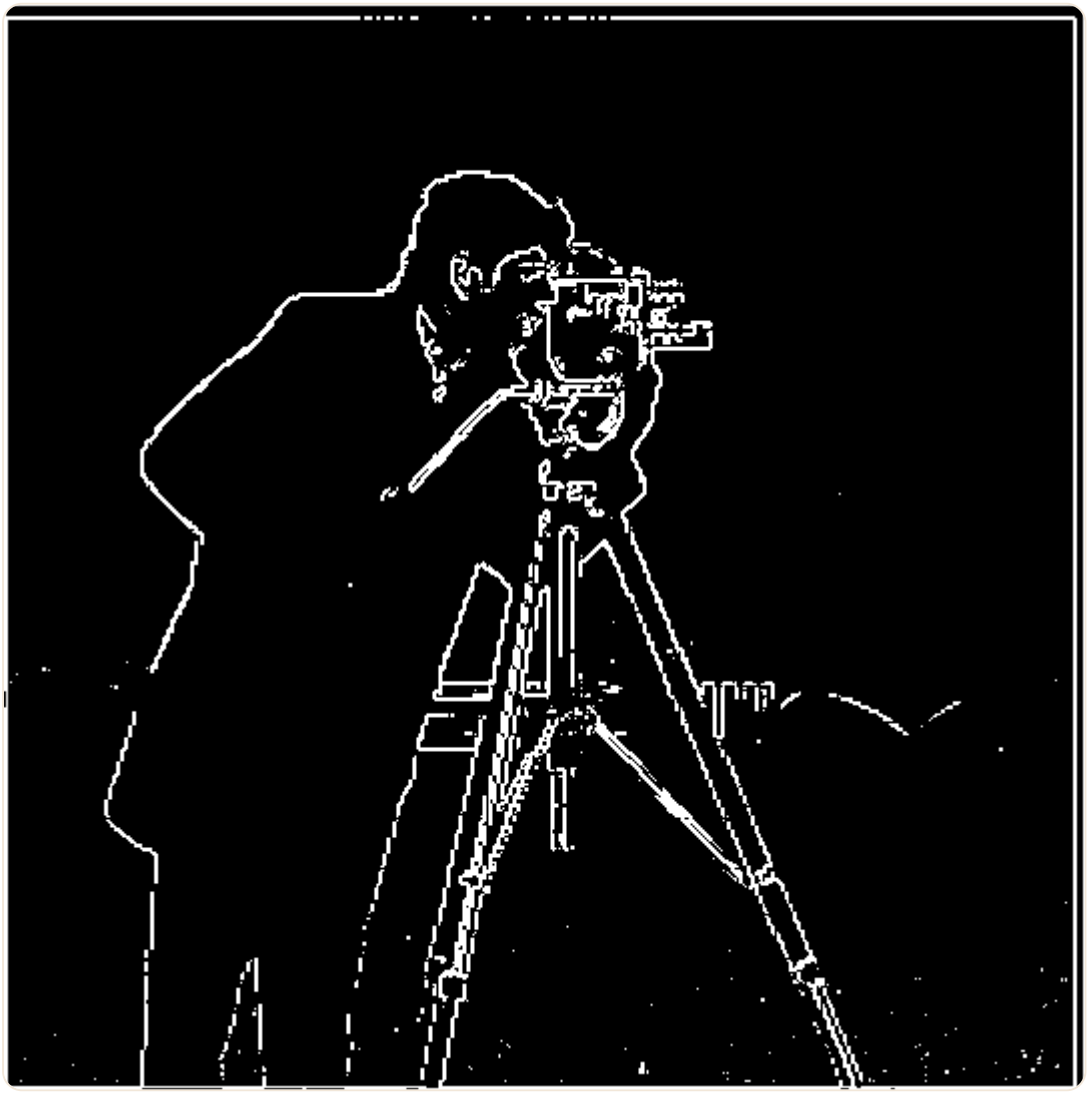
t = 0.25 (threshold on normalized ‖∇I‖)



t = 0.30 (threshold on normalized ‖∇I‖)

t = 0.35 (threshold on normalized ‖∇I‖)

Final edge map



Chosen threshold: **t = 0.20** (on normalized ‖∇I‖)

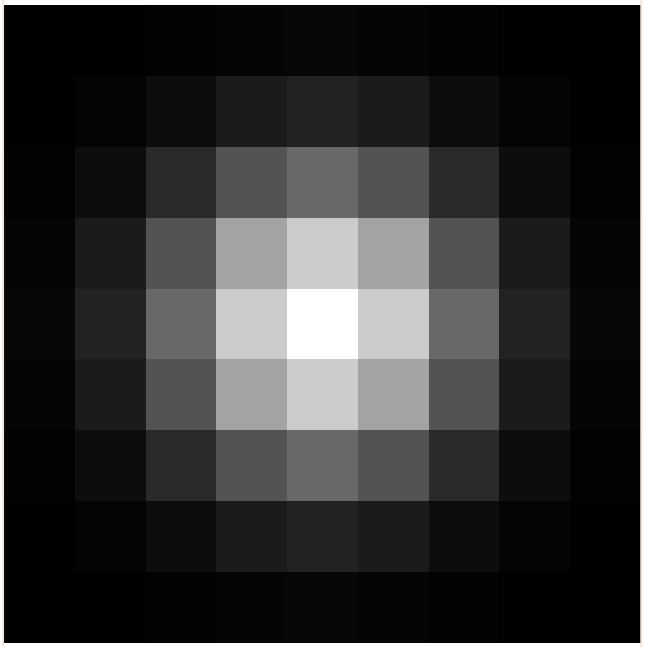
Why I chose t = 0.20

I compared tripod legs, the camera outline, the coat edge, and background speckle across the sweep. At t=0.10 the background speckle is heavy and small textures dominate. t=0.15 is better but still brings in grain. At **t=0.20** the main structure remains clean and continuous while the background noise drops a lot. 0.25 and above start erasing thin tripod edges and details around the hands. So 0.20 is the best tradeoff for keeping real contours without clutter.

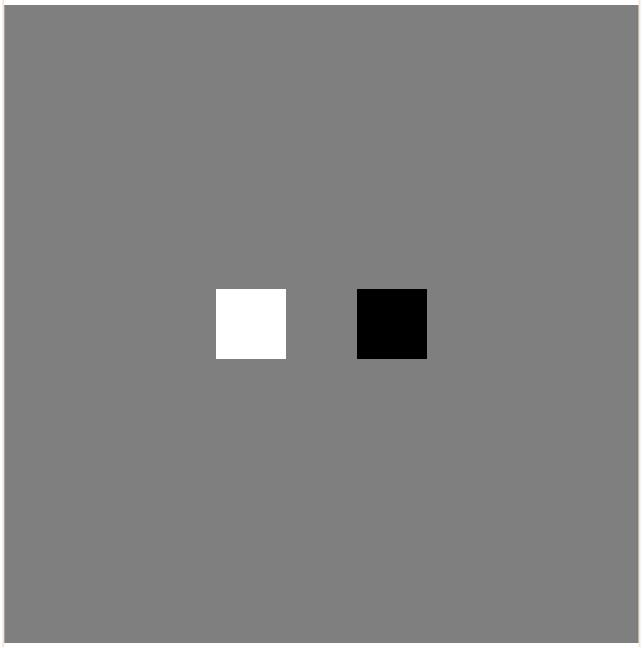
Part 1.3: Derivative of Gaussian (DoG) Filter

The plain difference operator from 1.2 is noisy. I first smooth with a Gaussian G built from `cv2.getGaussianKernel` by taking an outer product to get a 2D kernel. Then I repeat 1.2 on the blurred image. After that I do the same thing in one pass using DoG filters: $k_x = G * D_x$ and $k_y = G * D_y$. I compare the two results.

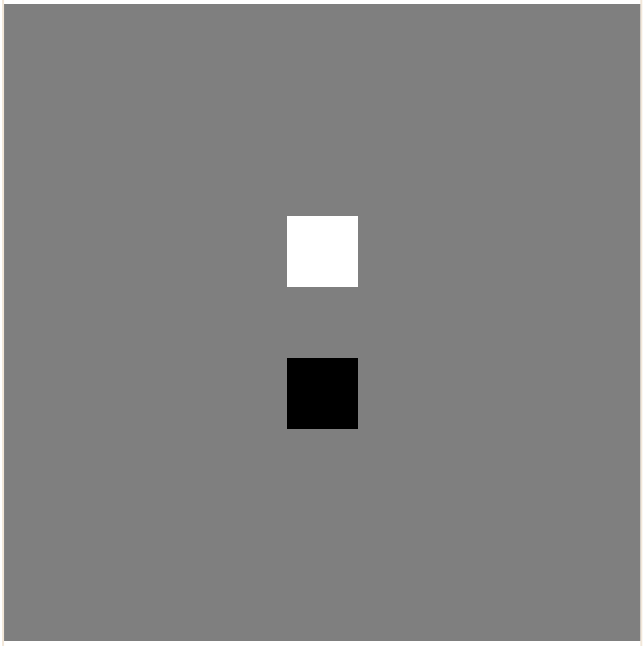
Base filters (visualized)



Gaussian G. (size=9, σ=1.5)

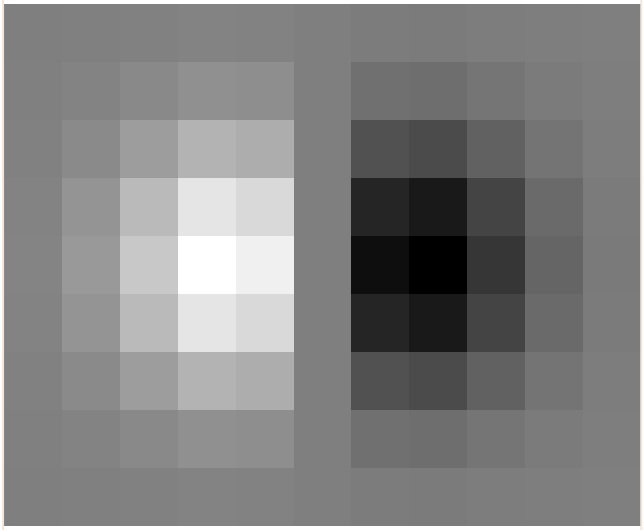


Finite difference $D_x = [1, 0, -1]$. (padded to 9×9 for visualization)

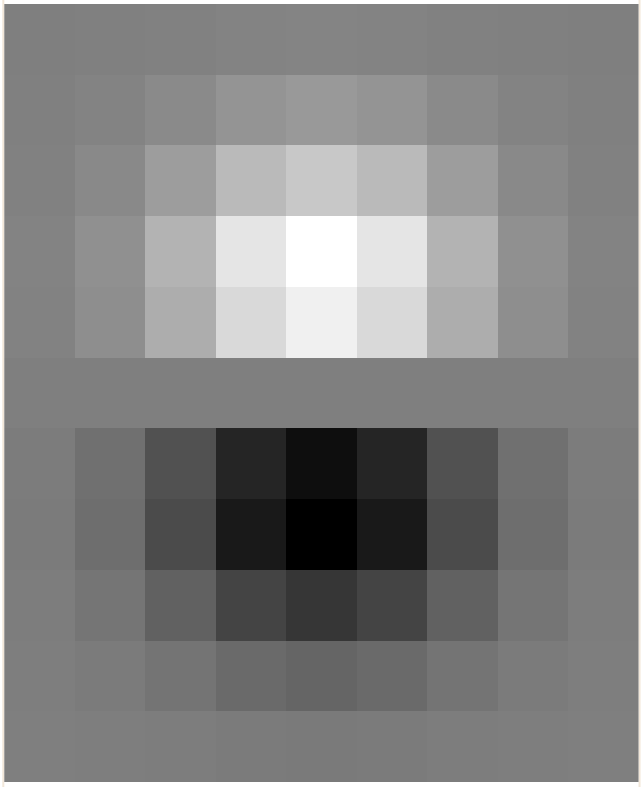


Finite difference $D_y = D_x^\top$. (padded to 9×9 for visualization)

DoG filters (visualized)



$k_x = G * D_x$. (size=9, σ=1.5)



$k_y = G * D_y$. (size=9, σ=1.5)

Blur → gradients

Cameraman after Gaussian blur. (size=9, σ=1.5)

Gradient magnitude of the blurred image. (built from blur with size=9, σ=1.5)

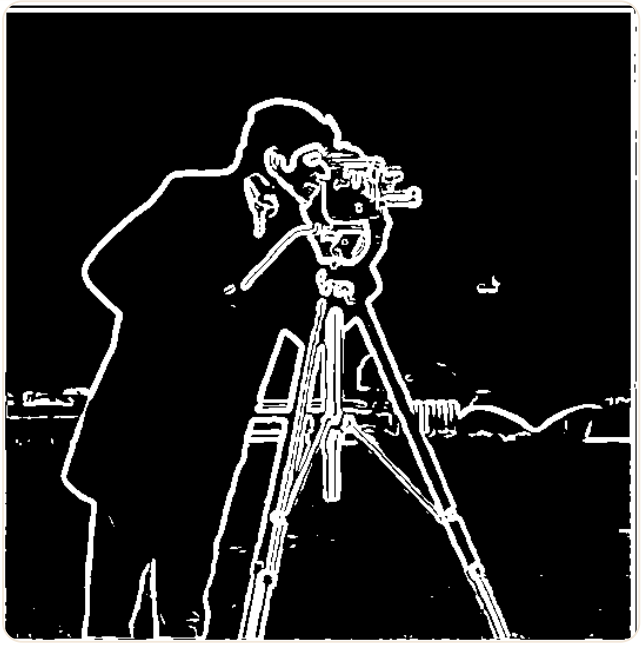
DoG in one pass

Gradient magnitude via DoG. (kx,ky from size=9, σ=1.5)

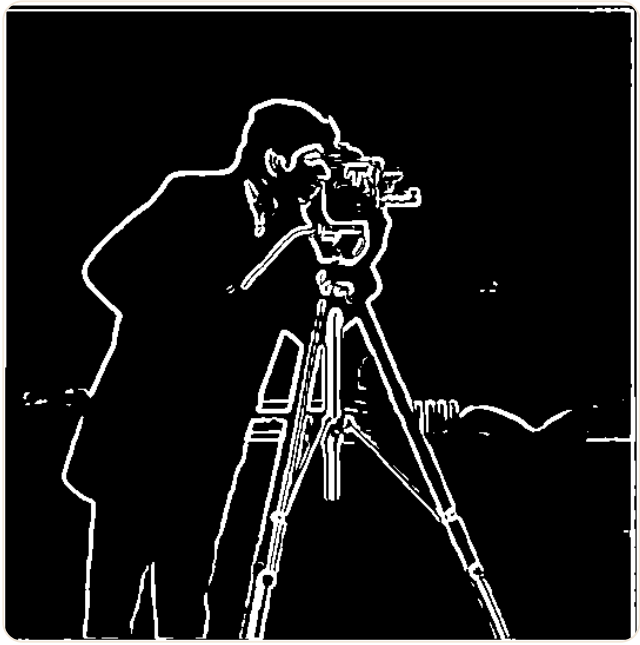
What changes compared to 1.2?

The blur reduces high-frequency noise before differencing. Edges look cleaner and less speckled. The DoG version matches the blur→diff result, since convolving with G and then with D_x is equivalent to convolving once with $G * D_x$ (and similarly for y).

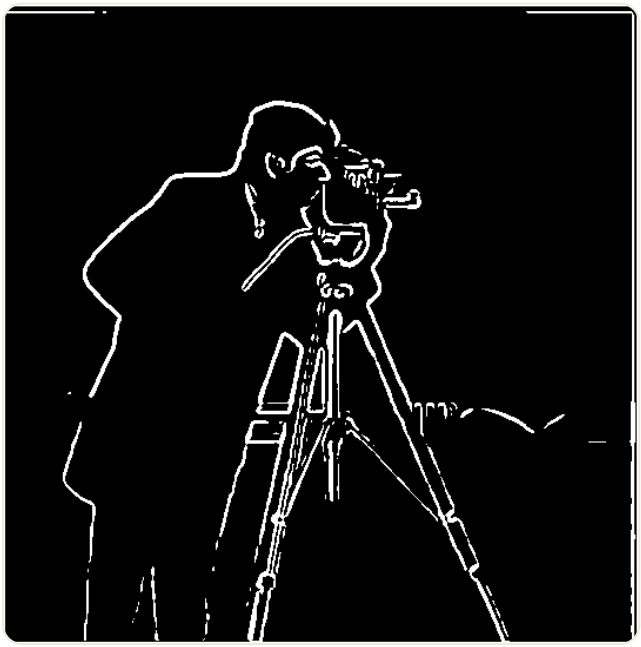
Threshold sweep (blur→diff)



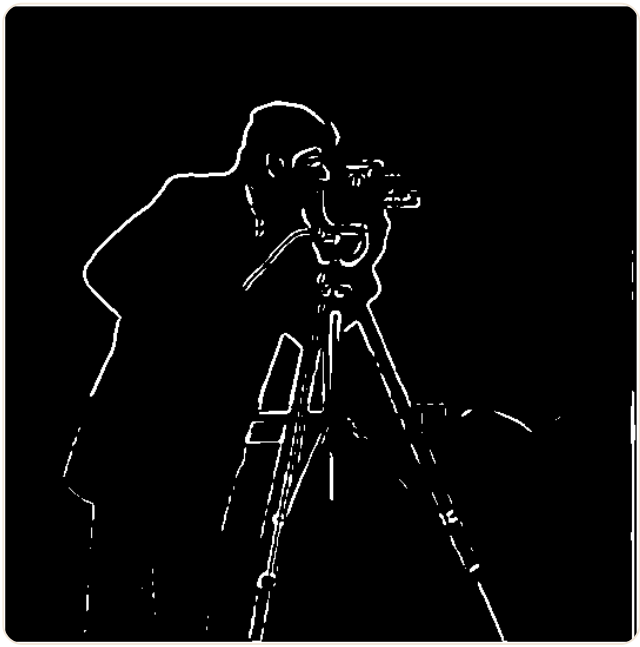
t = 0.10 (size=9, σ=1.5)



t = 0.15 (size=9, σ=1.5)

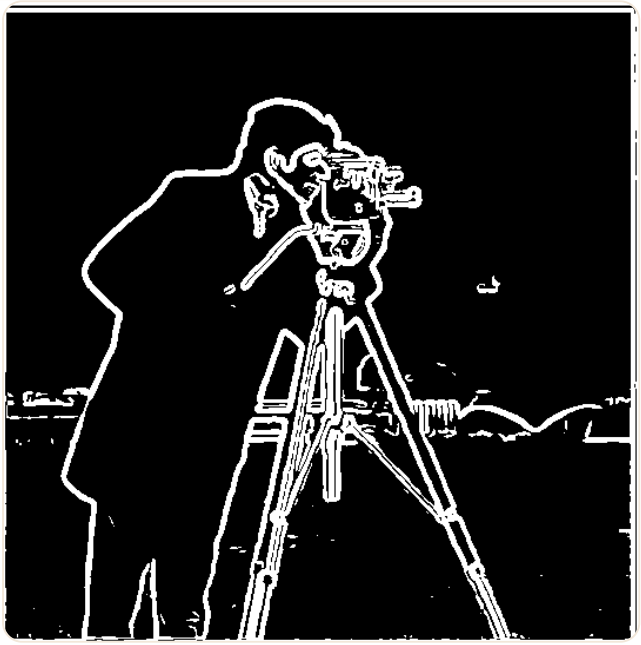


t = 0.20 (size=9, σ=1.5)

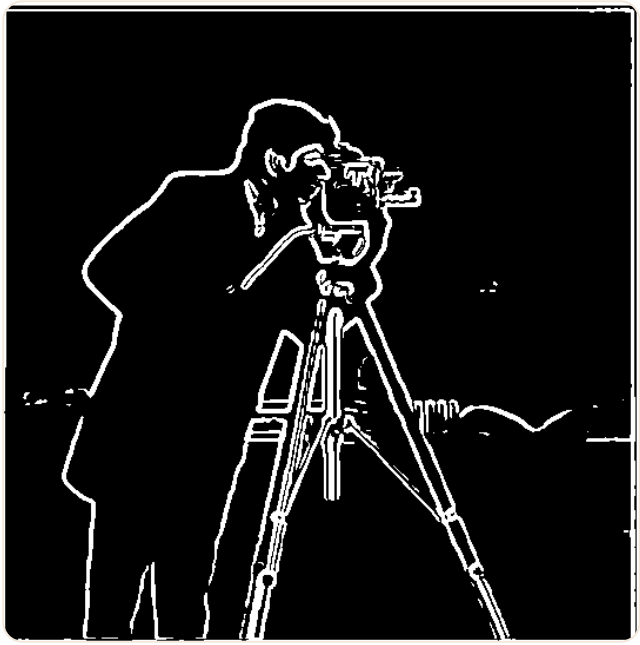


t = 0.25 (size=9, σ=1.5)

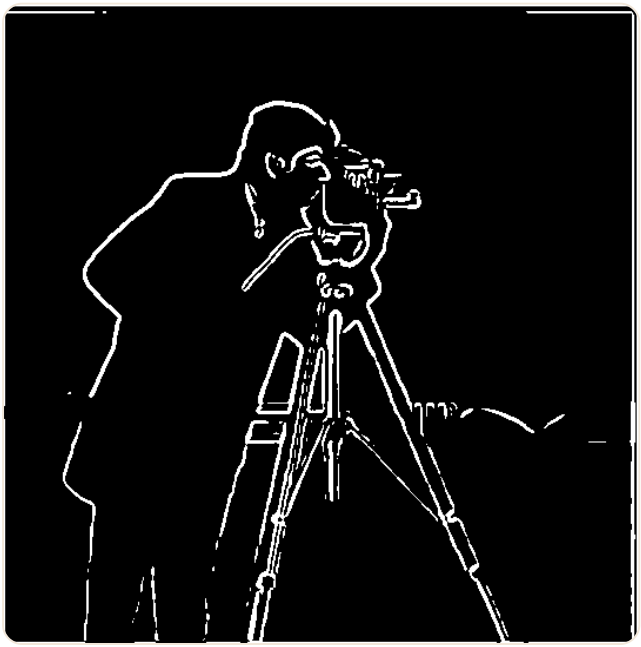
Threshold sweep (DoG once)



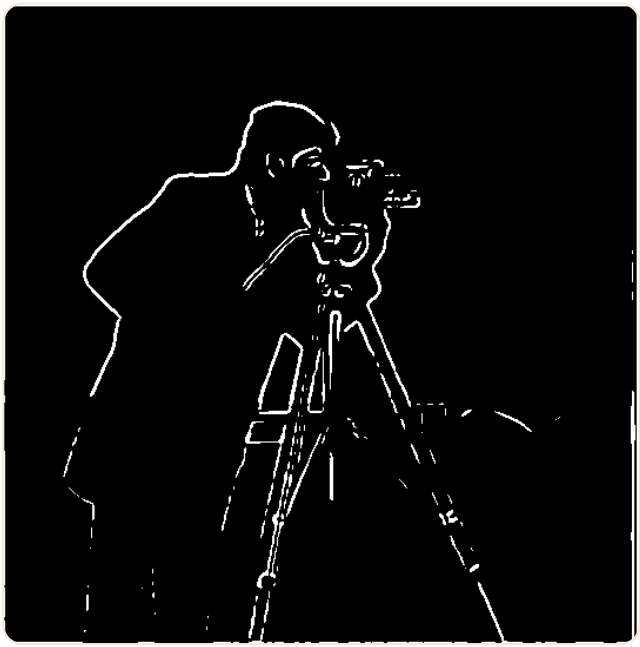
t = 0.10 (DoG size=9, σ=1.5)



t = 0.15 (DoG size=9, σ=1.5)

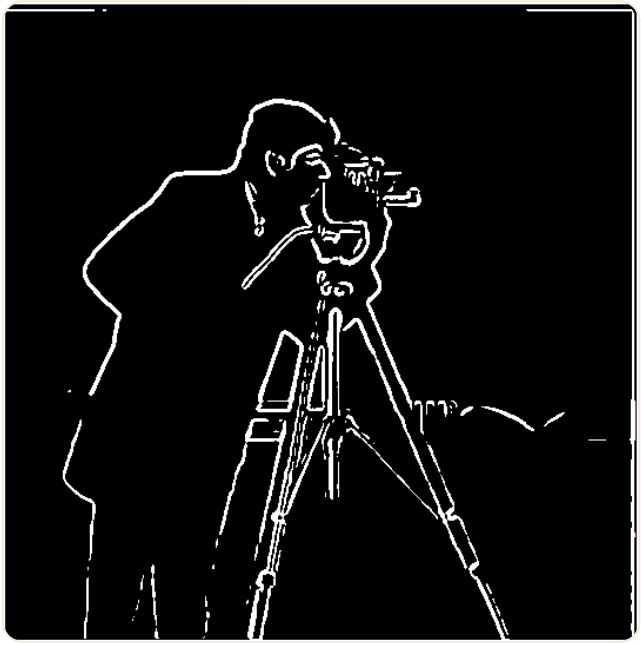


t = 0.20 (DoG size=9, σ=1.5)

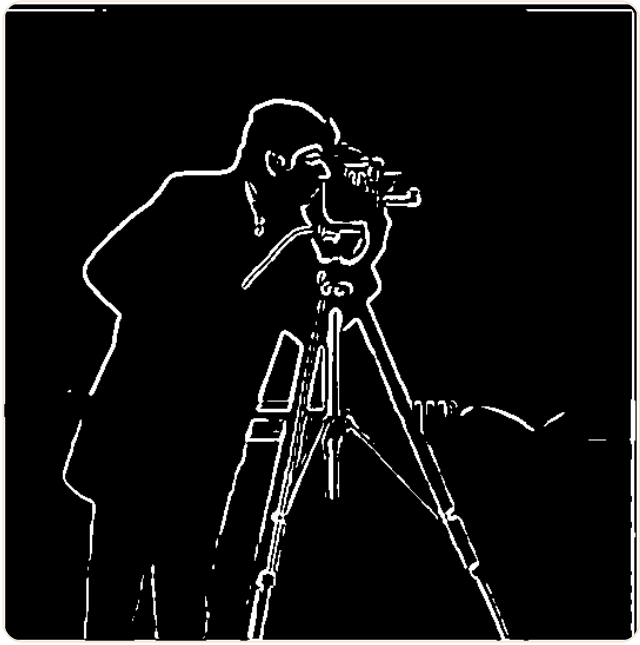


t = 0.25 (DoG size=9, σ=1.5)

Side-by-side at t = 0.20



Blur → diff, t = 0.20. (blur size=9, σ=1.5)



DoG once, t = 0.20. (DoG size=9, σ=1.5)

Do they match?

Interior agreement

On the interior (I trim a small border), the max absolute differences are tiny: $|g_x| \approx 3.34 \times 10^{-7}$, $|g_y| \approx 3.84 \times 10^{-7}$, and $|\|\nabla I\|| \approx 3.92 \times 10^{-7}$.

Why trimming fixes it

With `mode="same"` and zero padding, the blur spreads mass outside the image near the boundary. The order of operations interacts with that padding. DoG and blur→diff treat the first few pixels differently along the border, which gives noticeable edge differences. Once I crop away a small margin equal to the kernel half-size, both pipelines see the same neighborhood and they match up to float rounding.

> Without trimming the border, max diffs are around 3.67×10^{-1} because those boundary conventions do not align. After trimming, the two methods are effectively the same.

Part 2: Fun with Frequencies

Part 2.1: Image “Sharpening”

Unsharp masking uses a Gaussian blur G to split an image into low and high frequencies. Low = $I * G$. High = $I - (I * G)$. To sharpen, I add a multiple of the highs back: $I_{\text{sharp}} = I + \alpha\,(I - I * G)$. This is equivalent to a single convolution with kernel $K = (1+\alpha)\,\delta - \alpha\,G$, where δ is an impulse at the center and α sets the strength.

Taj Mahal: blur → high → sharpen

Input (for unsharp with G size=9, σ=1.5)

Gaussian blur (low frequencies). (size=9, σ=1.5)

High-pass (signed visualization). (derived from size=9, σ=1.5)

$\alpha = 0.5$ (unsharp size=9, σ=1.5)

$\alpha = 1.5$ (unsharp size=9, σ=1.5)

$\alpha = 3.0$ (unsharp size=9, σ=1.5)

$\alpha = 5.0$ (unsharp size=9, σ=1.5)

Single-convolution K with $\alpha = 2.0$. (G size=9, σ=1.5)

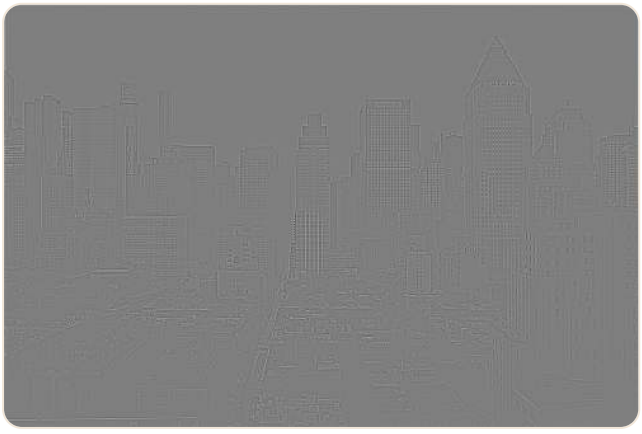
Observation

Moderate α (about 1–2) improves clarity while staying clean. Large α increases contrast across edges and creates halos.

Cityscape: second example

Input (for unsharp with G size=9, σ=1.5)

Gaussian blur (low frequencies). (size=9, σ=1.5)



High-pass (signed). (from size=9, σ=1.5)

$\alpha = 1.0$ (unsharp size=9, σ=1.5)

$\alpha = 3.0$ (unsharp size=9, σ=1.5)

$\alpha = 5.0$ (unsharp size=9, σ=1.5)

Evaluation on waves + forest (sharp → stronger blur → resharpen)

I blurred the sharp image with a stronger G so the softening is obvious, then resharpened with a wide range of α.

Original (high-res, sharp).

Blurred by G. (blur size=31, σ=4.5)

Resharpened, $\alpha = 2.0$. (unsharp size=9, σ=1.5)

$\alpha = 3.0$. (unsharp size=9, σ=1.5)

$\alpha = 5.0$. (unsharp size=9, σ=1.5)

$\alpha = 10$. (unsharp size=9, σ=1.5)

$\alpha = 30$. (unsharp size=9, σ=1.5)

$\alpha = 60$. (unsharp size=9, σ=1.5)

$\alpha = 120$. (unsharp size=9, σ=1.5)

What this shows

- Sharpening a blurred image improves perceived detail, but it does not recreate the lost fine frequencies.
- There is a sweet spot: mid α (about 2–3 here) looks closest to the original without obvious halos.
- Very large α overshoots and produces ringing, especially along the shoreline and tree silhouettes.

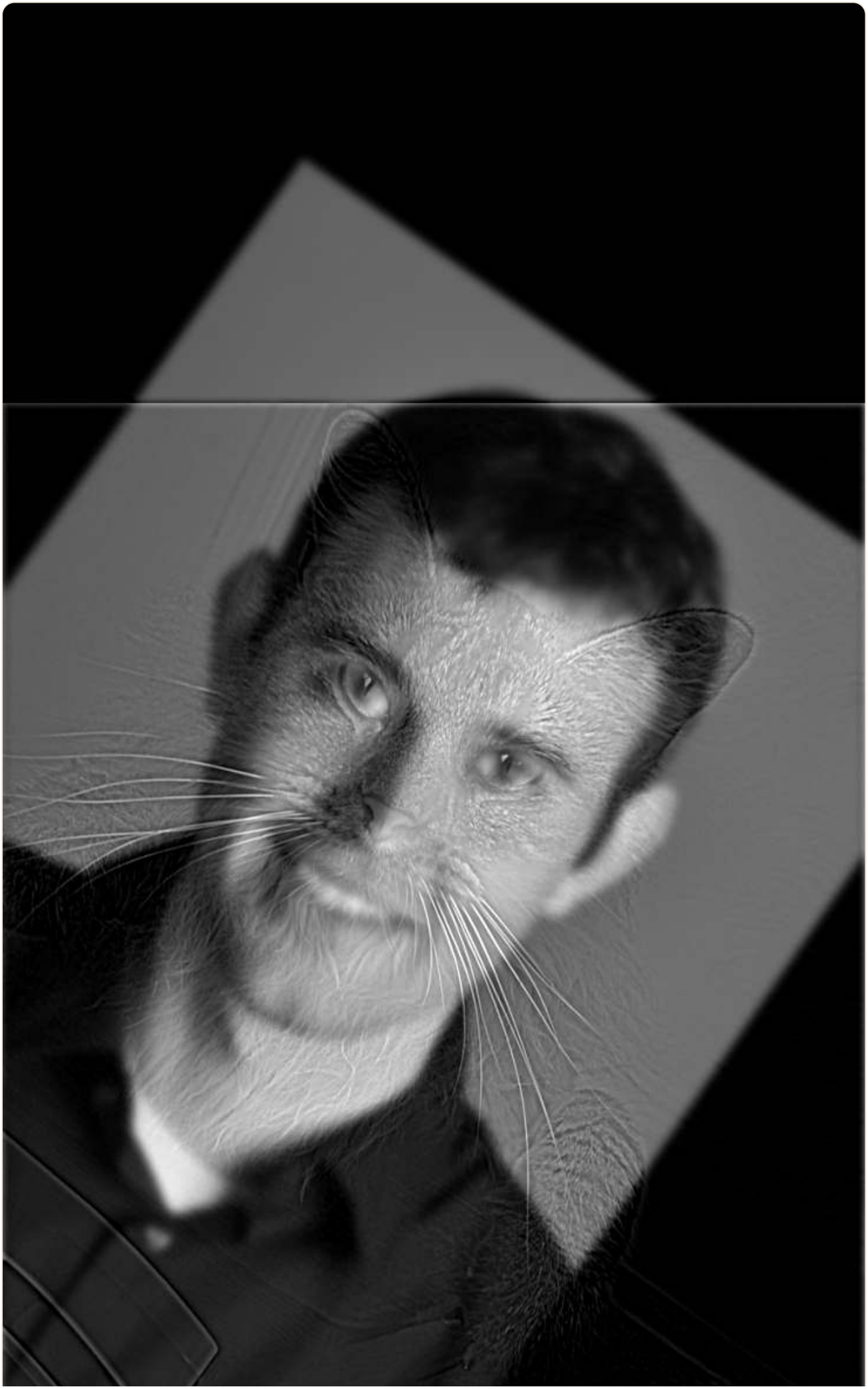
I saved MSE logs for each α in the evaluation folder. The lowest errors fall in the same mid range, which matches the visuals.

Summary: I implemented unsharp masking and explained it in terms of blur and high frequencies. I showed low, high, and sharpened versions for the Taj image and for a second image (cityscape). I also varied α to show how the amount changes the look. In the evaluation, I blurred a sharp photo and tried to get back the sharpness by adding highs. Mid α looks most natural; very large α brings artifacts.

Part 2.2: Hybrid Images

teaser

Look at the image on the right from very close, then from far away. Up close you see the high-frequency subject. From far away the low-frequency subject dominates.



Hybrid image changes with viewing distance. (low: size=21, σ=4; high: size=13, σ=3)

overview

Hybrid images follow the idea from Oliva, Torralba, and Schyns (SIGGRAPH 2006). High frequencies dominate perception up close, while low frequencies survive at distance. I low-pass one image with a Gaussian G to get $I_{\text{low}} = I * G$, high-pass the other with $I_{\text{high}} = J - (J * G)$, then add them: $H = I_{\text{low}} + I_{\text{high}}$. Alignment matters because it controls how features group; I aligned pairs before filtering.

full process: person + old man (with FFTs)

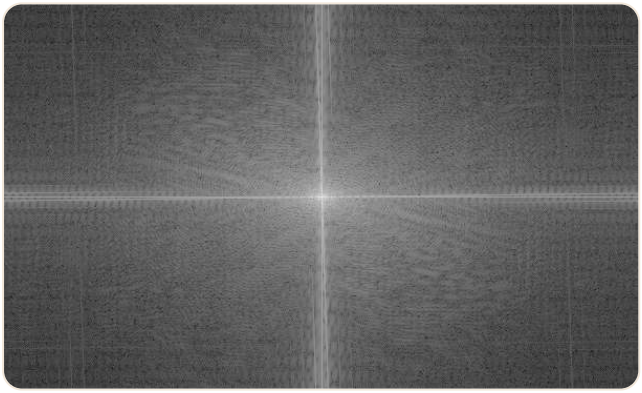
Inputs and alignment

Left image (low). (low_size=21, low_sigma=4)

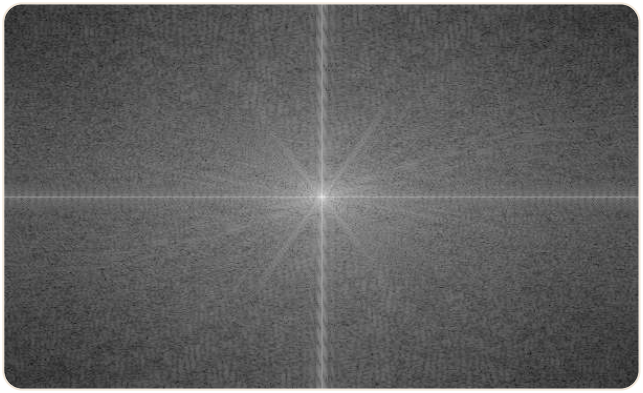
Right image (high). (high_size=13, high_sigma=2)

Final hybrid. (low 21/4 + high 13/2)

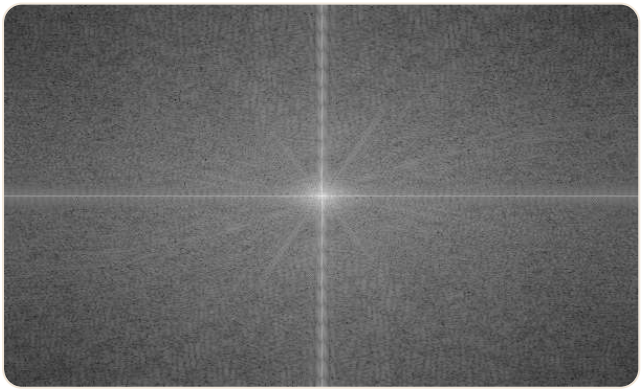
Frequency analysis (log magnitude FFT)



FFT of left (original). (pre-hybrid)



FFT of right (original). (pre-hybrid)



FFT of hybrid. (low 21/4 + high 13/2)

Filtered pieces

Low-pass of left. (size=21, σ=4)

High-pass of right. (size=13, σ=2)

Sum of low and high. (low 21/4 + high 13/2)

Cutoff choices

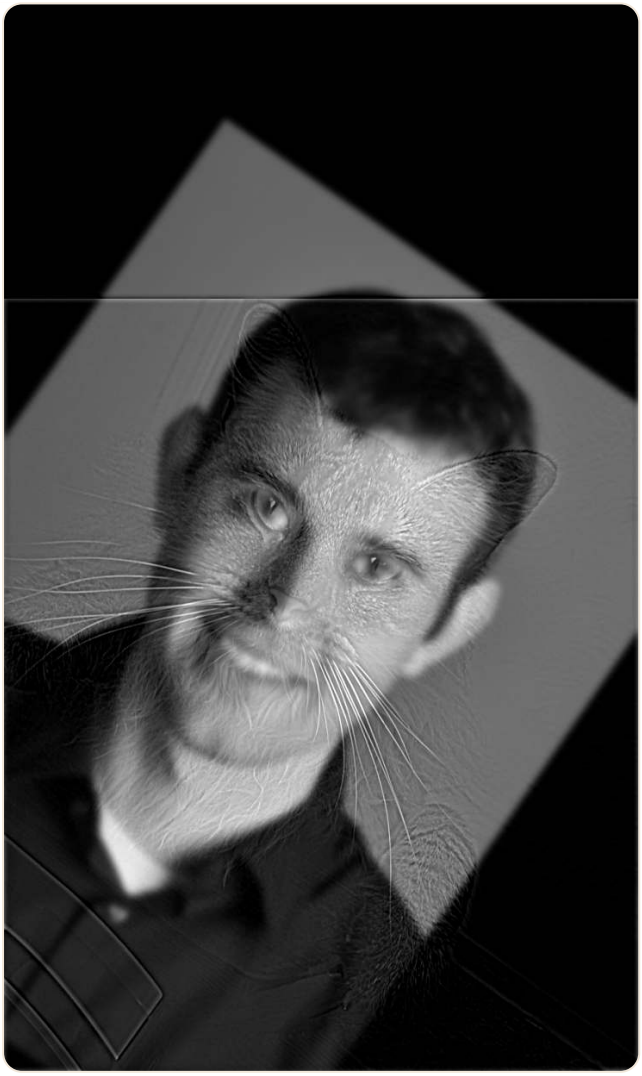
I set the low-pass size/sigma to balance shape and detail in the base face, and set the high-pass size/sigma to keep crisp features without making halos. The FFTs show energy concentrated at

the center for the lows and spread out for the highs, and the hybrid mixes both patterns.

Derek + Nutmeg

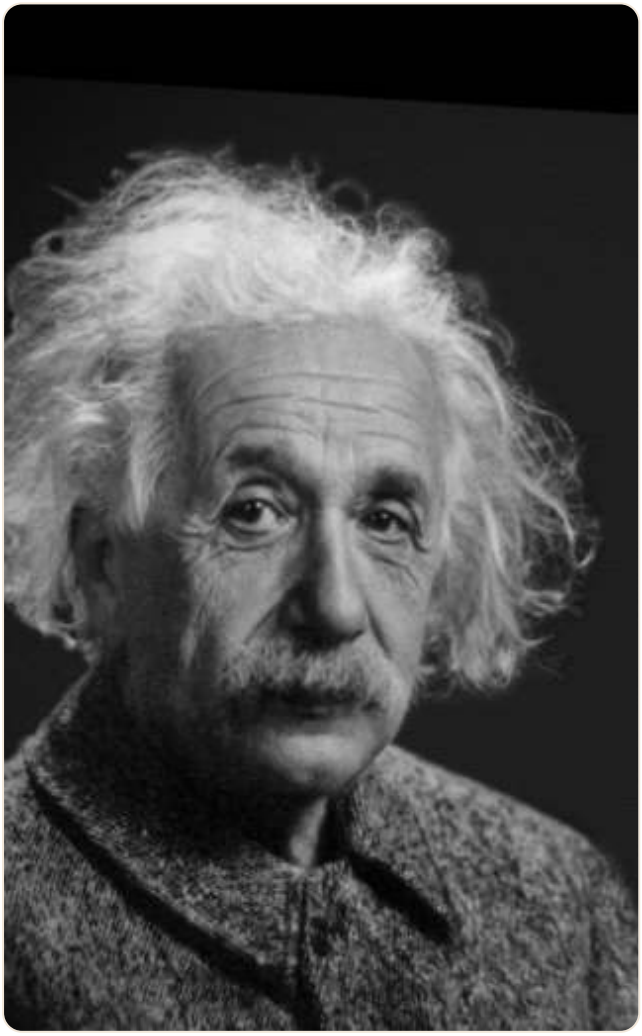
Left input (low). (low_size=21, low_sigma=4)

Right input (high). (high_size=13, high_sigma=3)



Hybrid. (low 21/4 + high 13/3)

Einstein + Marilyn



Left input (low). (low_size=21, low_sigma=4)

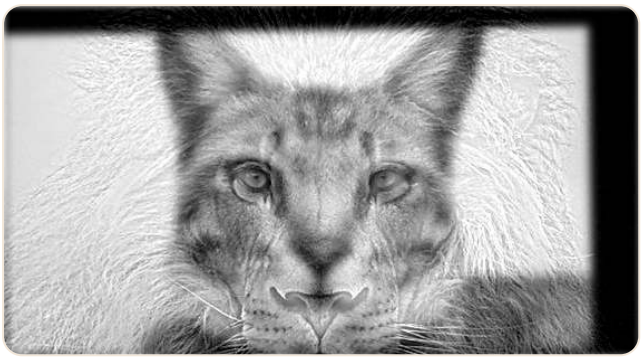
Right input (high). (high_size=17, high_sigma=2)

Hybrid. (low 21/4 + high 17/2)

Cat + Lion

Left input (low). (low_size=21, low_sigma=4)

Right input (high). (high_size=13, high_sigma=3)



Hybrid. (low 21/4 + high 13/3)

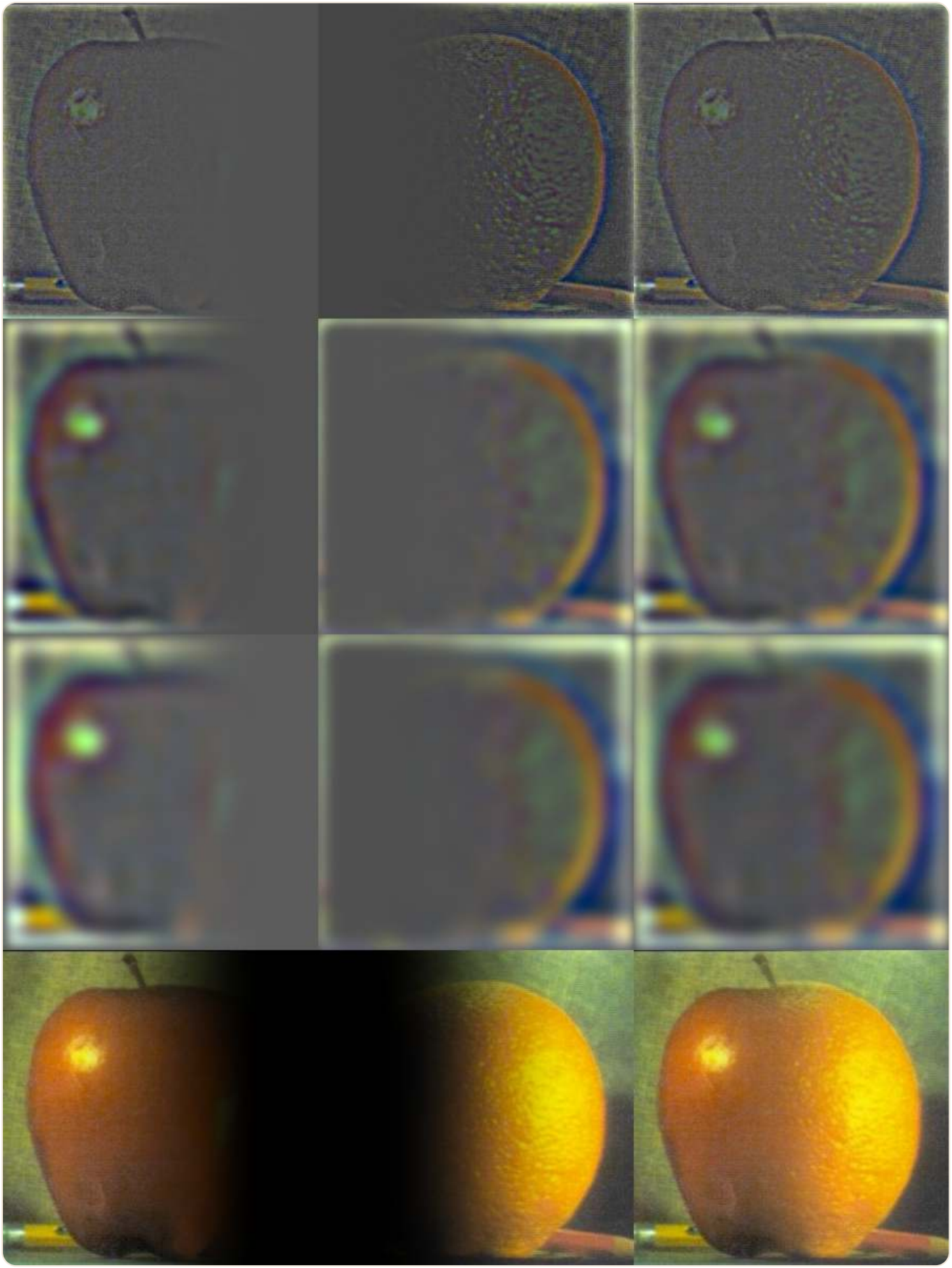
> **Takeaways:** Up close, the high-frequency face drives perception; from far away, the low-frequency face wins. Good alignment and careful cutoffs matter. If the high cutoff is too small, the hybrid looks muddy. If it is too large, halos and ghosting appear.

Part 2.3: Gaussian and Laplacian Stacks

overview

I implemented stacks with no downsampling to prep for multires blending. A Gaussian stack keeps blurring the image at the same size per level. A Laplacian stack stores band-pass info by taking differences between Gaussian levels plus a top residual.

Here I use **levels = 7**, **size = 31**, **σ = 5.0**. Apple and Orange are size matched before stacking so pixels line up.



Target layout for the Oraple visualization. (levels=7, size=31, σ=5.0)

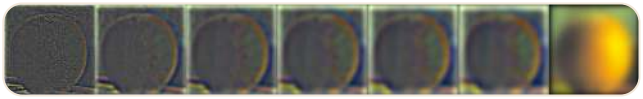
Gaussian & Laplacian stacks

Each row is one level. Laplacian images are normalized for display so the bands are visible.

Apple — Gaussian stack. (levels=7, size=31, σ=5.0). Built by repeated blur at fixed size.

Apple — Laplacian stack. (levels=7). Each band is the difference between two Gaussian levels.

Orange — Gaussian stack. (levels=7, size=31, σ=5.0).



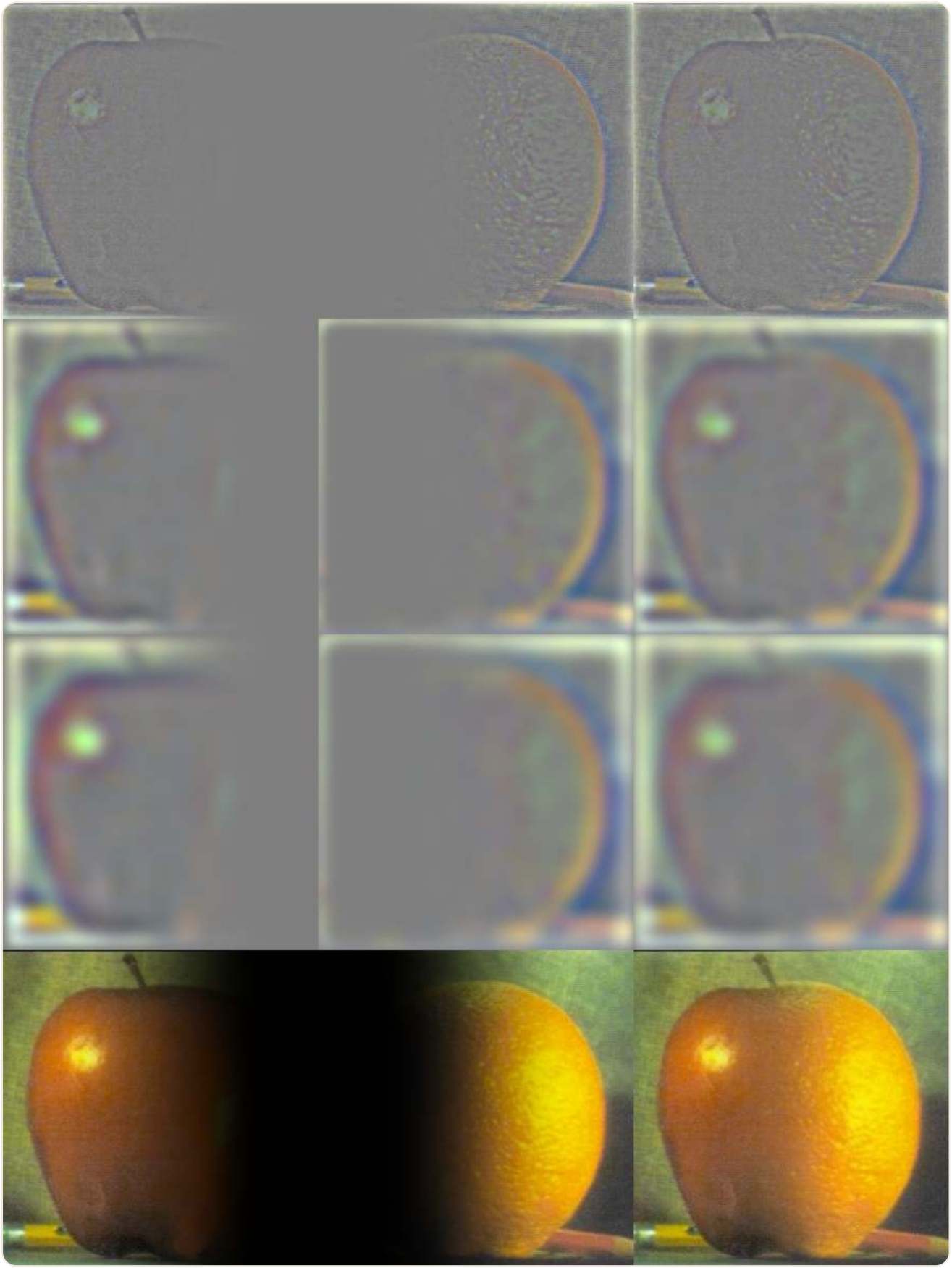
Orange — Laplacian stack. (levels=7). Bands highlight edges and textures at different scales.

Recreating Szeliski Fig. 3.42 (a)–(l)

First I show the full 4×3 grid. Then I list each cell with a short note on what is happening and how I made it. Blend equation per level i:

$$L_{\text{out}}^{(i)} = G_M^{(i)} \cdot L_A^{(i)} + \left(1 - G_M^{(i)}\right) \cdot L_B^{(i)}$$

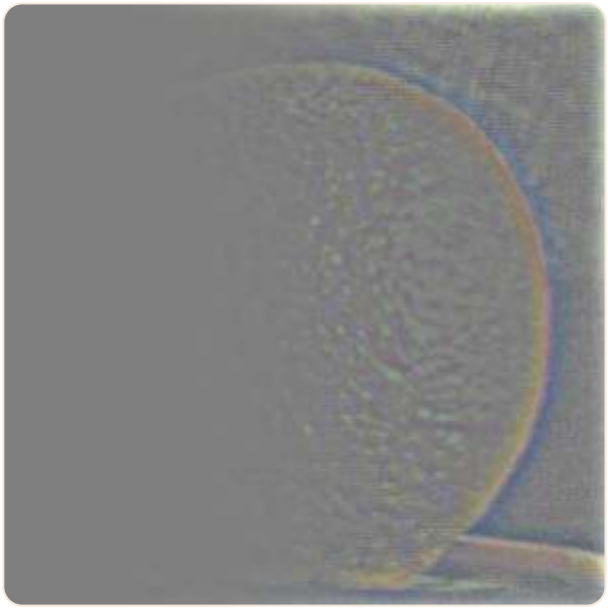
The final image is the sum across levels.



Full grid (a–l). (levels=7, size=31, σ=5.0; vertical cosine mask with its own Gaussian stack)

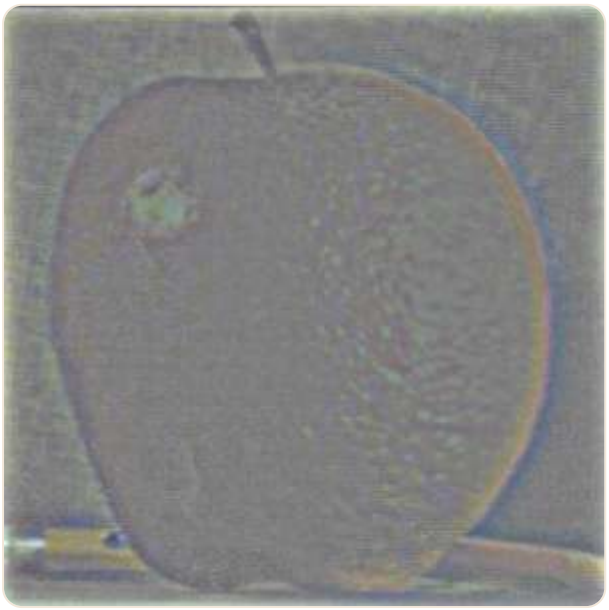
a — Apple band × mask (level 0). Params: levels=7, size=31, σ=5.0.

I take the level-0 Laplacian band from Apple and multiply by the blurred mask $G_M^{(0)}$. This keeps Apple's finest details on the left side.



b — Orange band × (1 − mask) (level 0). Same params.

I take the level-0 Laplacian band from Orange and weight it by $1 - G_M^{(0)}$. This keeps Orange's finest details on the right side.



c — Sum of a and b (level 0 composite).

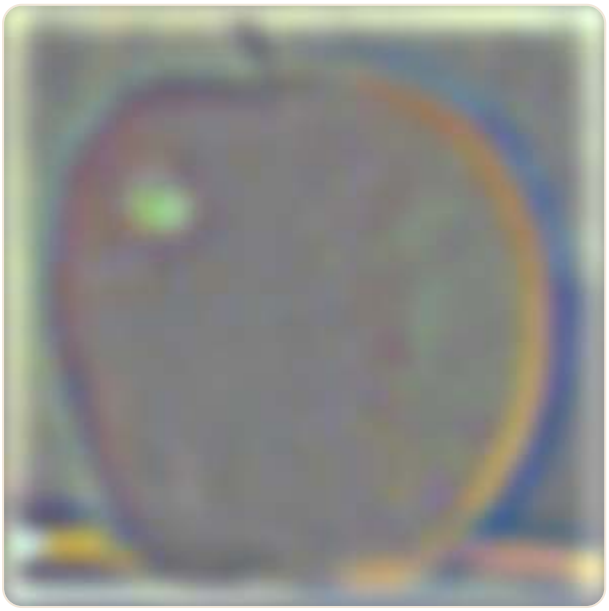
I add the two weighted bands to form the finest scale of the blend. This is the first piece of L_{out}.

d — Apple band × mask (level 2).

Same step at a coarser band. The mask is also blurred at level 2, so the transition spreads wider at this scale.

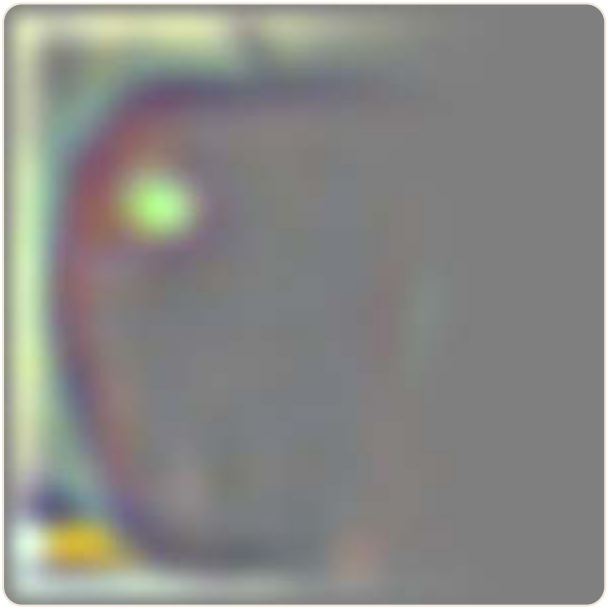
e — Orange band × (1 − mask) (level 2).

The Orange band is weighted by $1 - G_M^{(2)}$. This balances mid-scale edges from both sides.



f — Sum of d and e (level 2 composite).

I add the two level-2 pieces. This builds the medium frequencies of the final blend.



g — Apple band × mask (level 4).

Coarser band from Apple times $G_M^{(4)}$. The mask is very smooth here so the transition is very soft.

h — Orange band × (1 − mask) (level 4).

Coarser band from Orange times $1 - G_M^{(4)}$. This carries broad shading and color changes.

i — Sum of g and h (level 4 composite).

I add the two to finish the large-scale band piece for the blend.

j — Apple × mask (original scale).

I apply the base mask to the original Apple. This shows what the left half contributes before the per-level mixing.

k — Orange × (1 – mask) (original scale).

I apply the inverse mask to the original Orange. This shows the right half contribution.

l — Final multiband blend (Oraple). Params: levels=7, size=31, σ=5.0; vertical cosine mask.

I sum all blended Laplacian bands from fine to coarse and add the top residual. The seam is hidden because each scale transitions over the blurred mask at that scale.

> **How it was created:** I build Gaussian stacks G_A, G_B for Apple and Orange and Laplacian stacks L_A, L_B from those. I make a vertical cosine mask M and build its Gaussian stack G_M. For every level I blend bands with $G_M^{(i)}$ as the weight, then I sum all blended bands and clip to [0,1].

Final blend

Final Oraple blend. (levels=7, size=31, σ=5.0; vertical cosine mask). Built by summing blended Laplacian bands across scales.

> **Notes:** The blurred mask allows wide transitions at coarse scales and tight transitions at fine scales, which hides the seam.

Part 2.4: Multiresolution Blending (the Oraple and others)

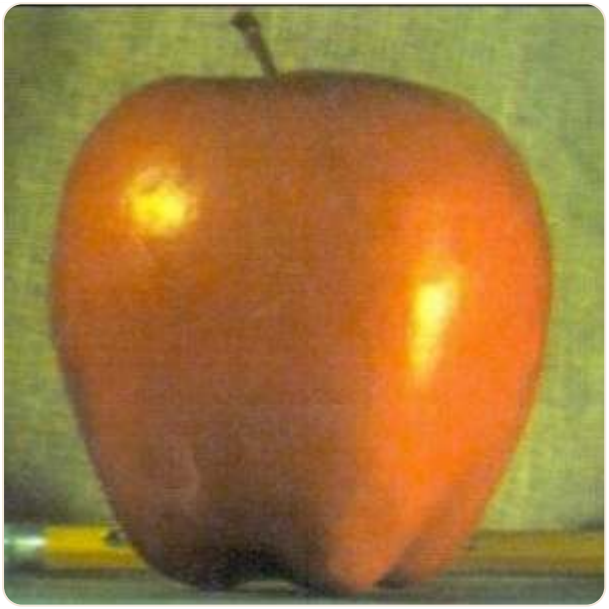
To blend images smoothly I used Gaussian and Laplacian stacks for both inputs and for a soft mask. At each level i I combined the Laplacians with the blurred mask:

$$L_{\text{out}}^{(i)} = G_M^{(i)} \cdot L_A^{(i)} + \left(1 - G_M^{(i)}\right) \cdot L_B^{(i)}$$

Adding these bands together gives the final image. The blurred mask lets low frequencies blend broadly and high frequencies blend narrowly, which hides seams.

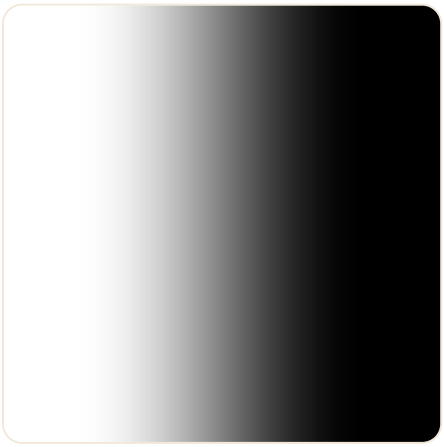
Oraple: Apple + Orange

I chose apple and orange because they are the classic example from the paper. The round shapes and strong color contrast make the seam obvious unless blending is done correctly.



Before: Apple (original input).

Before: Orange (original input).



Cosine vertical mask. Params: levels=7, size=31, σ=5.0.

Masked A (Apple × mask).

Masked B (Orange × (1−mask)).

Final Oraple blend.

City + Snowy Forest

Why these pairs: The city with the snowy forest shows a built space emerging from nature. I wanted to show a city springing out of the forest floor, with the snowy ground fading naturally into the city.

Before: City skyline (original input).

Before: Snowy forest (original input).

Circular soft mask. Params: levels=7, size=31, σ=5.0.

Masked A (City side).

Masked B (Forest side).

Final irregular-mask blend.

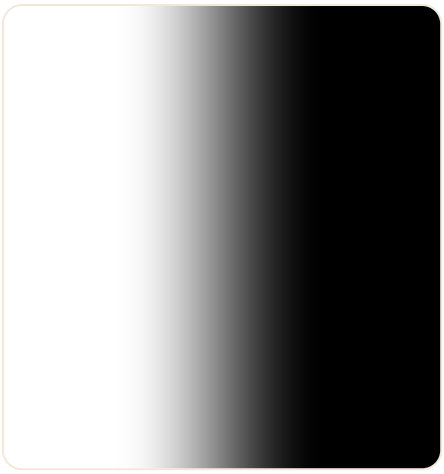
Laplacian outputs per level (like Fig. 10).

Before/After portraits

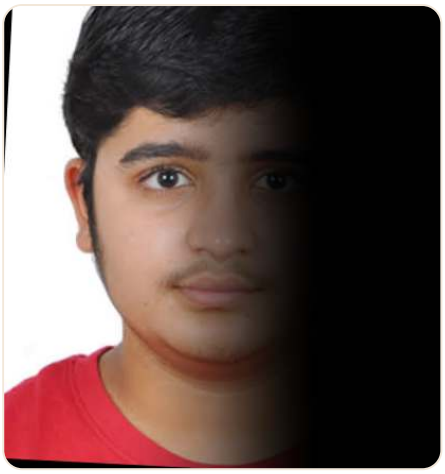
Why these pairs: The portraits show a personal before/after. One photo is from high school six years ago and the other is a recent college portrait. They were aligned using my eye positions so the face parts line up. A vertical cosine mask lets the two halves merge smoothly across the face.

Before: High school portrait (original input).

After: College portrait (original input).



Vertical cosine mask. Params: levels=7, size=31, σ=5.0.



Masked A (Before side).

Masked B (After side).

Final blended portrait.

Laplacian outputs per level (like Fig. 10).

Reflection: What I Learned

The most important thing I learned was how much control you can get once you build everything yourself. Writing my own Gaussian and Laplacian stacks instead of calling a library forced me to understand how each level carries different information, and why blending works when the mask is blurred at multiple scales. When I made the Oraple I saw how a seam that looked terrible at the pixel level almost disappeared once the stacks were combined. The before/after portrait also stood out because I had to decide which features of mine I had to align on to make the merge as seemless as possible. made the math feel real. This project taught me that the details I used to skip over, like the exact sigma or ramp width, actually decide whether an image looks natural or broken: This has led me to be more vigilant towards images I see online!