

Part 1 Image Resizing

1.1 nn_interpolate()

```
def nn_interpolate(im: Image, x: float, y: float, c: int) -> float:
```

Concept

“Nearest-neighbor interpolation uses the value of the closest existing pixel. At (2.3, 4.7), we choose the nearest integer location, (2, 5), and return that pixel.”

Code

“Round x and y to the nearest integer coordinates, then call `get_pixel()` with those coordinates and channel c . Return the result.

Be careful with half-integer rounding. Python’s `round()` uses banker’s rounding, while many C implementations behave like `floor(x + 0.5)` for nonnegative coordinates. Match the assignment’s expected convention.”

1.2 nn_resize()

```
def nn_resize(im: Image, w: int, h: int) -> Image:
```

Concept

“Use backward mapping. For every output pixel, ask which floating-point location it corresponds to in the input. This avoids holes that can occur if we push input pixels forward into the output.”

$$x = (i + 0.5) \times \text{xscale} - 0.5$$

$$y = (j + 0.5) \times \text{yscale} - 0.5$$

Code

“Create an output image of size w by h with the same number of channels as the input.

Compute $\text{xscale} = \text{im.w} / w$ and $\text{yscale} = \text{im.h} / h$. Loop over each output channel, row, and column.

Map the output pixel center back to the input with $x = (i + 0.5) \times \text{xscale} - 0.5$ and $y = (j + 0.5) \times \text{yscale} - 0.5$. Call `nn_interpolate()`, then write the value with `set_pixel()`.”

Part 2 Bilinear Interpolation

2.1 bilinear_interpolate()

```
def bilinear_interpolate(im: Image, x: float, y: float, c: int) -> float:
```

Concept

“Bilinear interpolation produces smoother transitions by considering the four pixels surrounding the sampling point. Find the top-left, top-right, bottom-left, and bottom-right samples around (x, y) . Pixels closer to the sample location receive more weight. Interpolate left-to-right across the top and bottom rows, then interpolate between those two results from top to bottom.”

$$q_1 = (1 - dx)v_{00} + dx \cdot v_{10}$$

$$q_2 = (1 - dx)v_{01} + dx \cdot v_{11}$$

$$\text{value} = (1 - dy)q_1 + dy \cdot q_2$$

Code

“Use $\text{floor}(x)$ and $\text{floor}(y)$ to locate the upper-left corner. Let dx and dy be the fractional offsets from that corner.

Read v_{00} , v_{10} , v_{01} , and v_{11} with `get_pixel()`. The helper’s clamping behavior handles boundary samples.

Compute the two horizontal blends q_1 and q_2 , then blend them vertically and return the result.”

2.2 bilinear_resize()

```
def bilinear_resize(im: Image, w: int, h: int) -> Image:
```

Concept

“Use the same backward mapping and pixel-center coordinate formula as nearest-neighbor resizing.

Keeping the mapping identical makes the visual comparison between interpolation methods meaningful.”

Code

“Create the output image, compute x_{scale} and y_{scale} , and loop over channels and output pixels.

Map each output pixel center back into input coordinates. Call `bilinear_interpolate()` and set the output value.”

Part 3 Image Filtering

3.1 l1_normalize()

```
def l1_normalize(im: Image) -> None:
```

Concept

“A three-by-three array of ones sums to nine. Applied directly, it would add nine neighbors and make the image brighter. Dividing each entry by nine turns the sum into an average. Constant regions then remain constant after filtering.”

$$w_i \leftarrow w_i / \sum_j w_j$$

Code

“Sum every value in the image. If the sum is zero, do not divide; return without changing the image.

Otherwise, divide every stored value by the total. This is an in-place function: it modifies `im.data` and does not create a replacement image.”

3.2 make_box_filter()

```
def make_box_filter(w: int) -> Image:
```

Concept

“Every location in the kernel receives equal weight. For a w by w kernel, normalization changes every entry from 1 to $1/w^2$. Larger neighborhoods usually create stronger blur.”

Code

“Create a w by w , one-channel image. Fill every element with 1, call `l1_normalize()`, and return the filter.”

Part 4 Convolution

4.1 convolve_image()

```
def convolve_image(im: Image, filt: Image, preserve: int) -> Image:
```

Concept

“Slide the filter across the image. At each output location, multiply each filter weight by the corresponding input value, sum all products, and store that sum as the output response.”

$$\text{output}(x, y, c) = \sum_j \sum_i \text{image}(x + i, y + j, c) \cdot \text{filter}(i, j)$$

Code

“Assert that the filter has one channel or the same number of channels as the image.

Create an output with im.c channels when preserve is 1; otherwise create a one-channel output.

Loop over each output location and relevant channel. Center the filter at that location, read each neighboring image value, multiply by the matching filter value, and accumulate the sum.

At image boundaries, rely on get_pixel() to clamp coordinates to the nearest valid pixel.

If preserve is 1, write each channel response separately. If preserve is 0, sum channel responses into the single output channel.”

Part 5 Highpass Sharpen and Emboss Filters

5.1 make_highpass_filter()

```
def make_highpass_filter() -> Image:
```

Concept

“The positive center is compared with its four immediate neighbors. In a constant region, positive and negative terms cancel. Where the center differs from its neighbors, the response becomes large in magnitude.”

Code

“Create a three-by-three, one-channel filter and assign the highpass weights exactly. No need to normalize it: the weights intentionally sum to zero.”

5.2 make_sharpen_filter()

```
def make_sharpen_filter() -> Image:
```

Concept

“The sharpen kernel is the identity filter plus the highpass filter. The center weight becomes 5 instead of 4, so constant regions pass through while edges receive extra contrast.”

Code

“Create a three-by-three, one-channel filter and write the sharpen weights. The weights sum to one, which preserves constant brightness before display-range clipping.”

5.3 make_emboss_filter()

```
def make_emboss_filter() -> Image:
```

Concept

“Negative weights occupy the upper-left side and positive weights occupy the lower-right side. Flat regions pass through because the weights sum to one, while directional changes create the relief effect.”

Code

“Create a three-by-three, one-channel filter and assign the directional weights. Keep the orientation consistent with the assignment; reversing the kernel reverses the apparent lighting direction.”

Part 6 Gaussian Filtering

6.1 make_gaussian_filter()

```
def make_gaussian_filter(sigma: float) -> Image:
```

Concept

“Sigma controls the spread. A small sigma concentrates weight near the center and creates mild blur. A large sigma spreads weight across a wider neighborhood and creates stronger blur.

The kernel must be large enough to contain most of the Gaussian mass. This assignment uses the next odd size produced by $((\text{int})(6 \times \sigma)) \mid 1$, which covers roughly three standard deviations on each side and guarantees a center cell.”

$$G(x, y) = [1 / (2\pi\sigma^2)] \cdot \exp[-(x^2 + y^2) / (2\sigma^2)]$$

kernel width = $(\text{int}(6\sigma))$ OR 1

Code

“Compute the kernel width from sigma, then create a square, one-channel filter.

Treat the kernel center as coordinate zero. For each cell, compute its x and y offsets from the center and evaluate the two-dimensional Gaussian.

After every cell is filled, call `l1_normalize()` so the weights sum to one. Return the filter.”

Part 7 Image Arithmetic

7.1 add_image()

```
def add_image(a: Image, b: Image) -> Image:
```

Concept

“The two inputs must have identical width, height, and channel count. Every output element is the sum of the corresponding values in a and b.”

Code

“Assert that the shapes match. Create an output image with that same shape, then loop through all stored values and write `a.data[i] + b.data[i]`. Return the new image instead of modifying either input.”

7.2 sub_image()

```
def sub_image(a: Image, b: Image) -> Image:
```

Concept

“As with addition, both images must have the same dimensions and channels. Each output value is $a - b$ at the matching location.”

detail = original - blurred

Code

“Assert equal shapes, allocate a same-shaped output, subtract corresponding elements, and return the result. Keep the order meaningful: $a - b$ is not the same as $b - a$.”

Part 8 Sobel Edge Detection

8.1 make_gx_filter()

```
def make_gx_filter() -> Image:
```

Concept

“Negative weights sample the left side, positive weights sample the right side, and the middle column is zero. The center row receives twice the weight, which adds a small amount of smoothing in the perpendicular direction.”

Code

“Create a three-by-three, one-channel image and assign the Sobel Gx weights. Do not normalize the filter; its signed response is the quantity we want.”

8.2 make_gy_filter()

```
def make_gy_filter() -> Image:
```

Concept

“The upper row is negative, the lower row is positive, and the middle row is zero. In image coordinates, y usually increases downward, so interpret the sign with that coordinate system in mind.”

Code

“Create a three-by-three, one-channel filter and assign the Gy weights. Keep its orientation consistent with the direction formula used later.”

8.3 feature_normalize()

```
def feature_normalize(im: Image) -> None:
```

Concept

“This is min-max normalization. It preserves ordering within one image but changes the absolute scale. If every value is identical, the denominator is zero, so the assignment defines the normalized image to be all zeros.”

$$x' = (x - \min) / (\max - \min)$$

Code

“Find the global minimum and maximum across all values. If max equals min, fill the image with zero and return.

Otherwise, replace each value x with $(x - \min) / (\max - \min)$. The operation is in place.”

8.4 sobel_image()

```
def sobel_image(im: Image) -> List[Image]:
```

Concept

“At each pixel, treat G_x and G_y as the horizontal and vertical components of a vector. The vector length is the magnitude, and atan2 uses both signed components to recover the correct direction and quadrant.

The direction is the gradient direction, which is perpendicular to the visual edge line itself.”

$$\text{magnitude} = \sqrt{G_x^2 + G_y^2}$$

$$\text{direction} = \text{atan2}(G_y, G_x)$$

Code

“Create two one-channel outputs with the same width and height as the input: one for magnitude and one for direction.

Build the G_x and G_y filters and convolve the input with $\text{preserve} = 0$ so each response is a one-channel feature map.

For every pixel, compute the Euclidean length of the two responses and compute $\text{atan2}(G_y, G_x)$. Store the results and return $[\text{magnitude}, \text{direction}]$.

Do not normalize inside $\text{sobel_image}()$. Keeping raw values makes the function useful for later numerical processing. Normalize only when a caller needs a display representation.”

8.5 `colorize_sobel()`

```
def colorize_sobel(im: Image) -> Image:
```

Concept

“Use HSV color space because hue can represent direction while saturation and value represent edge strength. Different directions become different colors; stronger edges become brighter and more saturated.”

H = normalized direction

S = normalized magnitude

V = normalized magnitude

Code

“Call `sobel_image()` to obtain magnitude and direction. Normalize both images into $[0, 1]$.

Create a three-channel result. Copy normalized direction into the hue channel, and copy normalized magnitude into both saturation and value.

Call `hsv_to_rgb()` in place, then return the RGB result.”