hw3

September 18, 2025

```
[25]: #IMPORTS
import numpy as np
import imageio.v2 as imageio
from pathlib import Path
from scipy import ndimage
from skimage.measure import label, regionprops
```

Problem 1: Generate a 3-level Gaussian pyramid (original image is level-0) and the corresponding Laplacian pyramid of an image (select one from the web, make it grayscale). Use the formula in the notes to first determine a viable image size (use N=3, and pick NC and NR), and crop the image (if needed) to test the pyramid code. Use a=0.4 for the Gaussian mask – use separable masks! Write/use functions for properly reducing and expanding an image. Write your own interpolation function - do not use Matlab/Python in-built interpolation functions (e.g., interp2). Lastly, perform a reconstruction of the original (cropped) image using the Laplacian pyramid. [8 pts]

```
[ ]: def make_gaussian_1d(a=0.4):
    return np.array([0.25 - 0.5*a, 0.25, a, 0.25, 0.25 - 0.5*a], dtype=np.
 ↪float64)

def pad_reflect_1d(x, r):
    # symmetric reflect including edge pixel:
    # [a b c d], r=2 -> left=[b,a], right=[d,c]
    left  = x[:r][::-1]
    right = x[:-r-1:-1]
    return np.concatenate([left, x, right], axis=0)

def conv1d_along_axis(img, k, axis):
    # Convolve along rows (axis=1) or cols (axis=0) with kernel k (odd length)
    r = len(k)//2
    if axis == 1:
        out = np.zeros_like(img, dtype=np.float64)
        for i in range(img.shape[0]):
            row = pad_reflect_1d(img[i, :], r)
            acc = np.zeros(img.shape[1], dtype=np.float64)
            for t, kv in enumerate(k):
                acc += kv * row[t:t+img.shape[1]]
            out[i, :] = acc
        return out
    else:
        # apply by transposing work to row case
        return conv1d_along_axis(img.T, k, axis=1).T

def gauss_blur_sep(im, w):
    return conv1d_along_axis(conv1d_along_axis(im, w, axis=1), w, axis=0)

def reduce(im, w):
    blurred = gauss_blur_sep(im, w)
    return blurred[::2, ::2]

def upsample_zeros(im, target_shape):
    H, W = target_shape
    out = np.zeros((H, W), dtype=np.float64)
    out[::2, ::2] = im
    return out

def expand(im_small, w, target_shape):
    # zero-insert then low-pass with *scaled* kernel so DC is preserved after␣
 ↪upsampling
    up = upsample_zeros(im_small, target_shape)
    w2 = 2.0 * w  # scale factor per 1D pass for correct interpolation energy
    return gauss_blur_sep(up, w2)

def gaussian_pyramid(im0, levels=3, a=0.4):
    w = make_gaussian_1d(a)
    G = [im0]
    for _ in range(levels):
        G.append(reduce(G[-1], w))
    return G, w

def laplacian_pyramid(G, w):
    L = []
    for k in range(len(G)-1):
        Ek = expand(G[k+1], w, G[k].shape)
        L.append(G[k] - Ek)
    return L

def reconstruct_from_laplacian(L, Gtop, w):
    R = Gtop.copy()
    for k in reversed(range(len(L))):
        R = L[k] + expand(R, w, L[k].shape)
    return R

# ---- helpers for cropping to valid size (N=3) ----
def crop_to_valid_size(im, N=3):
    H, W = im.shape
    f = 2**N
    Hc = (H - 1) // f * f + 1
    Wc = (W - 1) // f * f + 1
    return im[:Hc, :Wc]

def rgb2gray_ntsc(img_uint8):
    # Expect HxWx3 uint8; return float64 in [0,1] using class formula
    img = img_uint8.astype(np.float64) / 255.0
    return 0.299*img[...,0] + 0.587*img[...,1] + 0.114*img[...,2]  # Y channel

def rescale_for_display(im):
    # For Laplacians: shift/scale to [0,1] for viewing
    m, M = np.min(im), np.max(im)
    return np.zeros_like(im) if M == m else (im - m) / (M - m)

def run_q1(image_path, N=3, a=0.4, save_prefix="q1_"):
    # 1) load + gray
    raw = imageio.imread(image_path)
    if raw.ndim == 3:
        g = rgb2gray_ntsc(raw)
    else:
        g = raw.astype(np.float64) / 255.0

    # 2) crop to valid size R=MR*2^N+1, C=MC*2^N+1
    g = crop_to_valid_size(g, N)

    # 3) pyramids
    G, w = gaussian_pyramid(g, levels=N, a=a)
    L = laplacian_pyramid(G, w)

    # 4) reconstruction + error
    recon = reconstruct_from_laplacian(L, G[-1], w)
    err   = recon - g
    mse   = np.mean(err**2)
    print("Q1: image shape:", g.shape, "levels:", N, "a:", a)
    print("Q1: reconstruction MSE =", mse)

    # 5) (optional) save PNGs to visually confirm
    try:
        imageio.imwrite(f"{save_prefix}G0.png", (np.clip(G[0],0,1)*255).
↪astype(np.uint8))
        for i in range(1, len(G)):
            imageio.imwrite(f"{save_prefix}G{i}.png", (np.clip(G[i],0,1)*255).
↪astype(np.uint8))
        for i, Li in enumerate(L):
            Vi = rescale_for_display(Li)
            imageio.imwrite(f"{save_prefix}L{i}.png", (np.clip(Vi,0,1)*255).
↪astype(np.uint8))
        imageio.imwrite(f"{save_prefix}recon.png", (np.clip(recon,0,1)*255).
↪astype(np.uint8))
        Vi = rescale_for_display(err)
        imageio.imwrite(f"{save_prefix}error.png", (np.clip(Vi,0,1)*255).
↪astype(np.uint8))
    except Exception as e:
        print("Skipping image saves:", e)

    # 6) return for notebook inspection if needed
    return G, L, recon, err, mse

# Example usage (pseudo):
# im = load_gray_float_image(...)   # range [0,1]
# im = crop_to_valid_size(im, N=3)
# G, w = gaussian_pyramid(im, levels=3, a=0.4)
# L = laplacian_pyramid(G, w)
# recon = reconstruct_from_laplacian(L, G[-1], w)
# print("MSE:", np.mean((recon - im)**2))
G, L, recon, err, mse = run_q1("walk.bmp", N=3, a=0.4, save_prefix="q1_")
w = make_gaussian_1d(0.4); print("w:", w)
print("Cropped shape:", G[0].shape)  # insi
```

```
Q1: image shape: (233, 313) levels: 3 a: 0.4
Q1: reconstruction MSE = 1.808445915913163e-35
w: [0.05 0.25 0.4  0.25 0.05]
Cropped shape: (233, 313)
```

Problem 2: Using the grayscale images (walk.bmp, bg000.bmp) provided on the WWW site, perform background subtraction 1 (abs diff) to extract the object. Make sure your image is of type double! Experiment with thresholds and discuss. [2 pts]

[9]:
```
def to_float_gray(img):
    # make sure image is double in [0,1]; if color slips in, convert to gray␣
↪(NTSC)
    if img.ndim == 3:
        img = 0.299*img[...,0] + 0.587*img[...,1] + 0.114*img[...,2]  # Y␣
↪channel
    if img.dtype != np.float64:
        img = img.astype(np.float64)
    if img.max() > 1.0:
        img /= 255.0
    return img

def load_pair(obj_path="walk.bmp", bg_path="bg000.bmp"):
    I = to_float_gray(imageio.imread(obj_path))
    R = to_float_gray(imageio.imread(bg_path))
    # ensure same size (they should be)
    H = min(I.shape[0], R.shape[0])
    W = min(I.shape[1], R.shape[1])
    return I[:H,:W], R[:H,:W]

def abs_diff_mask(I, R, T):
    D = np.abs(I - R)
    return (D > T).astype(np.uint8), D

def otsu_threshold01(D, nbins=256):
    # Otsu on [0,1] difference image
    hist, edges = np.histogram(D.ravel(), bins=nbins, range=(0.0,1.0))
    hist = hist.astype(np.float64)
    p = hist / hist.sum()
    w_cum = np.cumsum(p)
    mu_cum = np.cumsum(p * (edges[:-1] + edges[1:]) / 2.0)  # bin centers
    mu_t = mu_cum[-1]
    # between-class variance for each split
    num = (mu_t * w_cum - mu_cum)**2
    den = w_cum * (1.0 - w_cum)
    sigma_b2 = np.where(den > 0, num/den, 0.0)
    k = np.argmax(sigma_b2)
    # threshold at the boundary after bin k
    return edges[k+1]

def run_q2(obj_path="walk.bmp", bg_path="bg000.bmp", save_prefix="q2_"):
    I, R = load_pair(obj_path, bg_path)
    B_10, D = abs_diff_mask(I, R, 0.10)   # try a few manual T's
    B_15, _ = abs_diff_mask(I, R, 0.15)
    B_20, _ = abs_diff_mask(I, R, 0.20)
    T_otsu = otsu_threshold01(D)
    B_otsu, _ = abs_diff_mask(I, R, T_otsu)

    print(f"Q2 shapes: I={I.shape}, R={R.shape}  dtype={I.dtype}")
    print("Manual thresholds: T=0.10 / 0.15 / 0.20")
    print("Otsu threshold on |I-R| :", T_otsu)
    for name, B in [("T=0.10", B_10), ("T=0.15", B_15), ("T=0.20", B_20),␣
↪(f"Otsu={T_otsu:.4f}", B_otsu)]:
        fg = int(B.sum())
        print(f"{name:>10}  foreground pixels = {fg}  ({fg / B.size:.3%})")

    # save helpful images for the report
    try:
        imageio.imwrite(f"{save_prefix}diff.png", (np.clip(D,0,1)*255).
↪astype(np.uint8))
        imageio.imwrite(f"{save_prefix}mask_T010.png", (B_10*255).astype(np.
↪uint8))
```

```
        imageio.imwrite(f"{save_prefix}mask_T015.png", (B_15*255).astype(np.
↪uint8))
        imageio.imwrite(f"{save_prefix}mask_T020.png", (B_20*255).astype(np.
↪uint8))
        imageio.imwrite(f"{save_prefix}mask_otsu.png", (B_otsu*255).astype(np.
↪uint8))
    except Exception as e:
        print("Save skipped:", e)

    return D, {"T=0.10":B_10, "T=0.15":B_15, "T=0.20":B_20, "T_otsu":B_otsu},␣
↪T_otsu
run_q2()
```

```
Q2 shapes: I=(240, 320), R=(240, 320)  dtype=float64
Manual thresholds: T=0.10 / 0.15 / 0.20
Otsu threshold on |I-R| : 0.28515625
    T=0.10  foreground pixels = 2885  (3.757%)
    T=0.15  foreground pixels = 1937  (2.522%)
    T=0.20  foreground pixels = 1671  (2.176%)
Otsu=0.2852  foreground pixels = 1441  (1.876%)
```

```
[9]: (array([[0.        , 0.        , 0.        , …, 0.        , 0.        ,
         0.        ],
        [0.00392157, 0.00392157, 0.01568627, …, 0.04313725, 0.00784314,
         0.07058824],
        [0.01176471, 0.00784314, 0.02745098, …, 0.08627451, 0.01960784,
         0.14117647],
        …,
        [0.00784314, 0.05490196, 0.05098039, …, 0.        , 0.00392157,
         0.01176471],
        [0.01960784, 0.08627451, 0.01960784, …, 0.01176471, 0.01176471,
         0.00784314],
        [0.01960784, 0.08627451, 0.01960784, …, 0.01176471, 0.01176471,
         0.00784314]]),
 {'T=0.10': array([[0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 1],
         …,
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0]], dtype=uint8),
  'T=0.15': array([[0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         …,
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0]], dtype=uint8),
  'T=0.20': array([[0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         …,
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0]], dtype=uint8),
  'T_otsu': array([[0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         …,
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0],
         [0, 0, 0, …, 0, 0, 0]], dtype=uint8)},
 np.float64(0.28515625))
```

Discussion: We did abs-diff exactly like the slides: D= I−R then threshold; Otsu picked T 0.285 (~73/255) for a tighter mask (~1.88% fg), while manual T=0.10/0.15/0.20 gave 3.76% / 2.52% / 2.18%, respectively. That tracks: the diff histogram is dominated by background near zero, so Otsu lands higher (more conservative), whereas lower T scoops up noise/shadows—the slides say to set T "above the noise level."

Problem 3: Using the grayscale images (walk.bmp, bg[000-029].bmp) provided on the WWW site, perform background subtraction 2 using statistical distances. Experiment with thresholds and discuss. [5 pts]

[19]:
```python
def run_q3(obj_path="walk.bmp", bg_dir=".", save_prefix="q3_",
           thresholds=(2.5, 3.0), sigma_floor=1e-3):

    def to_float_gray(img):
        if img.ndim == 3:
            img = 0.299*img[...,0] + 0.587*img[...,1] + 0.114*img[...,2]
        img = img.astype(np.float64)
        if img.max() > 1: img /= 255.0
        return img

    def load_bg_cube(bg_dir=".", fmt="bg%03d.bmp", n=30):
        stack = []
        for i in range(n):
            p = Path(bg_dir) / (fmt % i)
            stack.append(to_float_gray(imageio.imread(p)))
        return np.stack(stack, axis=0)  # (N,H,W)

    # 1) load
    I  = to_float_gray(imageio.imread(obj_path))
    Im = load_bg_cube(bg_dir)  # (30,H,W)
    H = min(I.shape[0], Im.shape[1]); W = min(I.shape[1], Im.shape[2])
    I  = I[:H,:W]; Im = Im[:, :H, :W]

    # 2) background model
    mu = Im.mean(axis=0)
    sigma_raw = Im.std(axis=0)
    sigma = np.maximum(sigma_raw, sigma_floor)  # <- raised floor (default 1e-3)

    # quick stats (nice for your write-up)
    print("sigma (raw):   min={:.6f} median={:.6f} max={:.6f}"
          .format(sigma_raw.min(), np.median(sigma_raw), sigma_raw.max()))
    print("fraction at floor =", np.mean(sigma == sigma_floor))

    # 3) z-score
    D = np.abs(I - mu) / sigma

    # 4) thresholds from discussion (defaults: 2.5, 3.0)
    masks = {}
    print(f"Q3 shapes: I={I.shape}, bg-cube={Im.shape}, dtype={I.dtype}")
    for T in thresholds:
        B = (D > T).astype(np.uint8)
        masks[f"T={T:.1f}"] = B
        fg = int(B.sum())
        print(f"T={T:.1f}  foreground pixels = {fg}  ({fg/B.size:.3%})")

    # 5) save visualizations (fix: cast to uint8)
    try:
        z = D / max(1e-8, np.percentile(D, 99))
        imageio.imwrite(f"{save_prefix}zscore.png", (np.clip(z,0,1)*255).
 ↪astype(np.uint8))
        for name,B in masks.items():
            tag = name.replace('=','').replace('.','')
            imageio.imwrite(f"{save_prefix}mask_{tag}.png", (B*255).astype(np.
 ↪uint8))
        # also save your "chosen" mask (cleaner per discussion)
        best_tag = f"T={thresholds[-1]:.1f}".replace('=','').replace('.','')
        imageio.imwrite(f"{save_prefix}mask_best.png",␣
 ↪(masks[f'T={thresholds[-1]:.1f}']*255).astype(np.uint8))
    except Exception as e:
        print("Save skipped:", e)

    return D, masks, mu, sigma
D, masks, mu, sigma = run_q3("walk.bmp", ".", thresholds=(2.5,3.0),␣
 ↪sigma_floor=1e-3)
```

```
sigma (raw):   min=0.000000 median=0.008947 max=0.186374
fraction at floor = 0.008125
Q3 shapes: I=(240, 320), bg-cube=(30, 240, 320), dtype=float64
```

```
T=2.5  foreground pixels = 9389  (12.225%)
T=3.0  foreground pixels = 7074  (9.211%)
```

Discussion: We modeled per-pixel , from the 30 backgrounds and used the z-score D= I− / ; thresholds T=2.5 and T=3.0 flagged 12.23% and 9.21% of pixels, respectively. That checks out: (median 0.00895) normalizes local flicker/noise, and with only ~0.81% of pixels at the 1e-3 floor, the higher T keeps mostly true foreground—so we'll roll with T 3.0 and tidy remaining speckle in Q4.

Problem 4: Dilate your best binary image resulting from problem 3 using: from scipy import ndimage d_bsIm = ndimage.binary_dilation(im, structure=np.ones((3,3)))

[23]:
```
def run_q4(best_mask, save_prefix="q4_"):
    """
    best_mask: binary mask from Q3 (e.g., masks['T=3.0']), dtype uint8 or bool
    """
    B = (best_mask > 0)  # ensure boolean
    se = np.ones((3,3), dtype=bool)       # 3x3 structuring element (square)
    d_bsIm = ndimage.binary_dilation(B, structure=se)  # HW-instructed op

    # quick stats for your write-up
    before = int(B.sum()); after = int(d_bsIm.sum())
    print(f"Dilation (3x3): foreground pixels {before} -> {after} ␣
 ↪(Δ={after-before:+d})")

    # save result
    try:
        imageio.imwrite(f"{save_prefix}dilated.png", (d_bsIm.astype(np.
 ↪uint8)*255))
    except Exception as e:
        print("Save skipped:", e)

    return d_bsIm

run_q4(masks['T=3.0'])
```

```
Dilation (3x3): foreground pixels 7074 -> 16233  (Δ=+9159)
```

[23]:
```
array([[False, False, False, …, False,  True,  True],
       [False, False, False, …, False,  True,  True],
       [False, False, False, …, False,  True,  True],
       …,
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False]])
```

Discussion: We applied binary dilation with a 3×3 ones structuring element to the Q3 mask. As expected, dilation grew the foreground—7074 → 16233 pixels (Δ=+9159)—thickening object regions and bridging small gaps; see q4_dilated.png.

Problem 5: Next perform a connected components algorithm, and keep only the largest region in L (save/display as an image). [1 pt]

```
[ ]: def run_q5(dilated_mask, save_prefix="q5_"):
         # 1) label components (8-connected)
         labels = label(dilated_mask.astype(bool), connectivity=2)
         props  = regionprops(labels)

         if len(props) == 0:
             print("No regions found.")
             only_big = np.zeros_like(dilated_mask, dtype=np.uint8)
             return only_big, labels, None

         # 2) pick largest by area (slides: area = # of 1-pixels)
         #    and make a binary image with only that region
         largest = max(props, key=lambda r: r.area)  # area in pixels
         big_id  = largest.label
         only_big = (labels == big_id).astype(np.uint8)

         # 3) stats + save
         print(f"Q5: regions={len(props)} | largest area={largest.area} px")
         imageio.imwrite(f"{save_prefix}largest.png", only_big*255)

         # optional: save a bbox overlay for your report
         try:
             minr, minc, maxr, maxc = largest.bbox  # bounding box corners
             box = only_big.copy()
             box[minr:maxr, [minc, maxc-1]] = 1
             box[[minr, maxr-1], minc:maxc] = 1
             imageio.imwrite(f"{save_prefix}largest_bbox.png", box*255)
         except Exception as e:
             print("BBox save skipped:", e)

         return only_big, labels, largest

     # Use the dilated mask from Q4 (we chose T=3.0 in Q3)
     dil = run_q4(masks["T=3.0"])         # or load q4_dilated.png > 0
     big, labels, largest = run_q5(dil, save_prefix="q5_")
```

```
Dilation (3x3): foreground pixels 7074 -> 16233  (Δ=+9159)
Q5: regions=411 | largest area=9336.0 px
```

Discussion: We labeled the dilated mask with 8-connected components and kept only the largest region. Out of 411 regions, the largest had 9,336 px, which cleanly isolates the main foreground object while discarding small fragments. Area is just the count of foreground pixels, so this matches the slides' "keep-largest-blob" heuristic; see q5_largest.png (and the optional bbox overlay).