```python
import cv2
import numpy as np
import os
from scipy import signal
from scipy.ndimage import distance_transform_edt

# Parameters for adaptive resizing (minimum width 400 pixels)
MIN_WIDTH = 400           # minimum width threshold
MAX_DIM = 1024            # absolute maximum dimension
MIN_KEYPOINTS = 500       # minimum number of keypoints to preserve

# Initialize SIFT globally
sift = cv2.SIFT_create()

def adaptive_resize(img, min_width=MIN_WIDTH, max_dim=MAX_DIM,
min_keypoints=MIN_KEYPOINTS):
    """Adaptively resize image: keep width >= 400, limit max dimension, preserve
keypoints"""
    h, w = img.shape[:2]

    # First check: if width < 400, scale up
    if w < min_width:
        scale = min_width / w
        new_w, new_h = int(w * scale), int(h * scale)
        img = cv2.resize(img, (new_w, new_h), interpolation=cv2.INTER_CUBIC)
        print(f"    Upscaled to {new_w}x{new_h}")
        h, w = new_h, new_w

    # Second check: if max dimension > max_dim, scale down
    if max(w, h) > max_dim:
        if w >= h:
            scale = max_dim / w
        else:
            scale = max_dim / h
        new_w, new_h = int(w * scale), int(h * scale)
        img = cv2.resize(img, (new_w, new_h), interpolation=cv2.INTER_AREA)
        print(f"    Downscaled to {new_w}x{new_h}")
        h, w = new_h, new_w

    # Detect keypoints
    keypoints = sift.detect(img, None)
    print(f"    Final size: {w}x{h} with {len(keypoints)} keypoints")
    return img

def load_images_from_folder(folder, use_adaptive_resize=True):
    """Load and resize images from folder"""
    images = []
    filenames = []
    for filename in sorted(os.listdir(folder)):
        if filename.lower().endswith((".jpg", ".jpeg", ".png")):
            filepath = os.path.join(folder, filename)
            img = cv2.imread(filepath)
            if img is not None:
                if use_adaptive_resize:
                    print(f"  Processing {filename}...")
                    img = adaptive_resize(img)
                images.append(img)
                filenames.append(filename)
    return images, filenames

def detect_features(img):
    """Detect SIFT features with enhanced parameters"""
    # SIFT with adjusted parameters for better feature detection
    sift = cv2.SIFT_create(
        nfeatures=5000,           # Increase max features to detect
        nOctaveLayers=5,          # More octave layers for multi-scale detection
        contrastThreshold=0.01,   # Lower threshold to detect more features
        edgeThreshold=15,         # Edge threshold
        sigma=1.6                 # Gaussian kernel sigma
    )
    kp, des = sift.detectAndCompute(img, None)
    return kp, des

def match_features(des1, des2):
    """Match features using FLANN with RANSAC post-processing"""
    if des1 is None or des2 is None or len(des1) < 2 or len(des2) < 2:
        return []

    # Use FLANN for better matching
    FLANN_INDEX_KDTREE = 1
    index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
    search_params = dict(checks=50)

    try:
        flann = cv2.FlannBasedMatcher(index_params, search_params)
        matches = flann.knnMatch(des1, des2, k=2)
    except:
        # Fallback to BFMatcher if FLANN fails
        bf = cv2.BFMatcher()
        matches = bf.knnMatch(des1, des2, k=2)

    good = []
    for match_pair in matches:
        if len(match_pair) == 2:
            m, n = match_pair
            # Lowe's ratio test with adjusted threshold
            if m.distance < 0.7 * n.distance:
                good.append(m)

    return good
```

```
def compute_homography_ransac(kp1, kp2, matches, prefer_translation=True):
    """Compute homography using RANSAC, prefer translation for stability"""
    if len(matches) < 4:
        return None

    src_pts = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)

    # RANSAC with strict threshold to avoid tilting
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,
ransacReprojThreshold=3.0)

    if mask is not None:
        inliers = np.where(mask.ravel() == 1)[0]
        if len(inliers) < 4:
            return None

        # Check if homography is mostly translation (no significant perspective)
        if prefer_translation and len(inliers) >= 4:
            # If perspective distortion is minimal, use translation instead
            if abs(H[0, 0] - 1.0) < 0.05 and abs(H[1, 1] - 1.0) < 0.05 and \
               abs(H[0, 1]) < 0.05 and abs(H[1, 0]) < 0.05:
                # Homography is close to identity + translation, use translation
instead
                T = compute_translation_transform(kp1, kp2, matches)
                if T is not None:
                    return T

    return H

def compute_translation_transform(kp1, kp2, matches):
    """Compute translation-only transformation using RANSAC (more robust, prevents
tilting)"""
    if len(matches) < 1:
        return None

    src_pts = np.float32([kp1[m.queryIdx].pt for m in matches])
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in matches])

    # Use RANSAC for robust estimation of translation
    best_dx, best_dy = 0, 0
    best_inliers = 0
    threshold = 2.0  # pixel threshold for inliers

    for _ in range(100):  # RANSAC iterations
        idx = np.random.randint(0, len(matches))
        dx = dst_pts[idx, 0] - src_pts[idx, 0]
        dy = dst_pts[idx, 1] - src_pts[idx, 1]
        # Count inliers
        errors = np.sqrt((dst_pts[:, 0] - src_pts[:, 0] - dx)**2 +
                        (dst_pts[:, 1] - src_pts[:, 1] - dy)**2)
        inliers = np.sum(errors < threshold)

        if inliers > best_inliers:
            best_inliers = inliers
            best_dx = dx
            best_dy = dy

    # Use median if RANSAC finds too few inliers
    if best_inliers < len(matches) * 0.3:
        best_dx = np.median(dst_pts[:, 0] - src_pts[:, 0])
        best_dy = np.median(dst_pts[:, 1] - src_pts[:, 1])

    # Create translation matrix
    T = np.array([
        [1, 0, best_dx],
        [0, 1, best_dy],
        [0, 0, 1]
    ], dtype=np.float32)

    return T

def stitch_with_pyramid_blending(img1, img2, overlap_width, levels=3):
    """Stitch using Laplacian pyramid blending with reflection removal"""
    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # Ensure same height
    max_h = max(h1, h2)
    if h1 < max_h:
        img1 = cv2.copyMakeBorder(img1, 0, max_h - h1, 0, 0, cv2.BORDER_CONSTANT,
value=0)
    if h2 < max_h:
        img2 = cv2.copyMakeBorder(img2, 0, max_h - h2, 0, 0, cv2.BORDER_CONSTANT,
value=0)

    # Create canvas
    new_width = w1 + w2 - overlap_width
    result = np.zeros((max_h, new_width, 3), dtype=np.float32)

    # Place first image
    result[:, :w1] = img1.astype(np.float32)

    # Advanced blending in overlap region with content filtering
    if overlap_width > 0:
        overlap_start = w1 - overlap_width
        overlap_end = w1
        # Extract overlap regions
        left_overlap = img1[:, overlap_start:overlap_end].astype(np.float32)
        right_overlap = img2[:, :overlap_width].astype(np.float32)

        # Create smooth blending mask
        x = np.linspace(0, 1, overlap_width)
        blend_mask = np.power(x, 2)  # Quadratic blend
        blend_mask = blend_mask.reshape(1, -1, 1)

        # Compute confidence map - favor regions with higher variance (content-rich)
        left_gray = cv2.cvtColor(left_overlap.astype(np.uint8),
cv2.COLOR_BGR2GRAY).astype(np.float32)
        right_gray = cv2.cvtColor(right_overlap.astype(np.uint8),
cv2.COLOR_BGR2GRAY).astype(np.float32)

        # Smooth variance estimation
        left_gray = cv2.GaussianBlur(left_gray, (5, 5), 0)
        right_gray = cv2.GaussianBlur(right_gray, (5, 5), 0)

        # Create content confidence mask (avoid reflections and artifacts)
        left_conf = np.where(left_gray > 10, left_gray / 256.0, 0)  # Filter out
very dark regions
        right_conf = np.where(right_gray > 10, right_gray / 256.0, 0)

        content_conf = (left_conf + right_conf) / (left_conf + right_conf + 1e-6)
        content_conf = np.tile(content_conf.reshape(max_h, overlap_width, 1), (1, 1,
3))
        content_conf = content_conf / (content_conf.max() + 1e-6)

        # Apply adaptive blending
        blended = left_overlap * (1 - blend_mask) + right_overlap * blend_mask

        # Confidence-weighted blending to reduce artifacts
        result[:, overlap_start:overlap_end] = blended * content_conf + \
                                               left_overlap * (1 - content_conf) *
0.7

    # Place the non-overlapping part of second image
    result[:, w1 - overlap_width + overlap_width:] = img2[:,
overlap_width:].astype(np.float32)

    # Remove reflections: apply median filter to overlap region
    if overlap_width > 0:
        overlap_region = result[:, overlap_start:overlap_end].astype(np.uint8)
        # Median filter reduces salt-and-pepper reflections
        overlap_region = cv2.medianBlur(overlap_region, 3)
        result[:, overlap_start:overlap_end] = overlap_region.astype(np.float32)

    # Clip values to valid range
    result = np.clip(result, 0, 255).astype(img1.dtype)
    return result

def match_exposure_pair(img1, img2, transform, sample_ratio=0.01, n_iters=100):
    """Match exposure between two images using gamma correction"""
    try:
        # Convert to LAB color space
        lab1 = cv2.cvtColor(img1, cv2.COLOR_BGR2LAB).astype(np.float32)
        lab2 = cv2.cvtColor(img2, cv2.COLOR_BGR2LAB).astype(np.float32)

        h, w = img1.shape[:2]
        n_samples = max(10, int(h * w * sample_ratio))

        # Sample corresponding points
        samples = []
        for _ in range(n_samples):
            y, x = np.random.randint(0, h), np.random.randint(0, w)

            # Transform point from img2 to img1
            p = np.array([x, y, 1])
            p_transformed = transform @ p
            p_transformed = p_transformed / p_transformed[2]

            x_t, y_t = int(p_transformed[0]), int(p_transformed[1])

            if 0 <= x_t < w and 0 <= y_t < h:
                # Sample L (lightness) channel
                l1 = lab1[y_t, x_t, 0] / 100.0
                l2 = lab2[y, x, 0] / 100.0

                if l1 > 5 and l2 > 5:  # Avoid dark regions
                    samples.append((l1, l2))

        if len(samples) < 5:
            return 1.0

        samples = np.array(samples)

        # Fit gamma correction curve
        gamma = 1.0
        for _ in range(n_iters):
            gamma_prev = gamma
            residuals = samples[:, 1] ** gamma - samples[:, 0]
            gradient = np.sum(residuals * np.log(samples[:, 1]) * (samples[:, 1] **
gamma))
            gamma = gamma - 0.01 * gradient / len(samples)

            if abs(gamma - gamma_prev) < 0.001:
                break
        return max(0.5, min(2.0, gamma))
    except:
        return 1.0

def apply_gamma_correction(img, gamma):
    """Apply gamma correction to image"""
    if gamma == 1.0:
        return img

    try:
        img_float = img.astype(np.float32) / 255.0
        corrected = np.power(img_float, gamma)
        return (corrected * 255).astype(img.dtype)
    except:
        return img

def create_distance_mask(img):
    """Create distance-based mask for blending (distance from edges)"""
    h, w = img.shape[:2]

    # Create binary mask of non-zero regions
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    mask = (gray > 0).astype(np.uint8)

    # Compute distance transform
    dist_transform = distance_transform_edt(mask).astype(np.float32)

    # Normalize distance
    if dist_transform.max() > 0:
        dist_transform = dist_transform / dist_transform.max()

    return dist_transform

def intelligent_merge(imgs, transforms, output_h, output_w):
    """Merge multiple images with distance-based weighting and exposure matching"""
    n_imgs = len(imgs)
    result = np.zeros((output_h, output_w, 3), dtype=np.float32)
    weights = np.zeros((output_h, output_w, 3), dtype=np.float32)

    # Match exposures between consecutive images
    gammas = [1.0]
    for i in range(1, n_imgs):
        gamma = match_exposure_pair(imgs[i-1], imgs[i], transforms[i])
        gammas.append(gamma)

    print(f"  Exposure gammas: {[f'{g:.3f}' for g in gammas]}")

    # Apply gamma corrections
    corrected_imgs = []
    for i, img in enumerate(imgs):
        corrected = apply_gamma_correction(img, gammas[i])
        corrected_imgs.append(corrected)

    # Merge with distance-based weighting
    for i, img in enumerate(corrected_imgs):
        h, w = img.shape[:2]
        T = transforms[i]

        # Create distance mask for this image
        mask = create_distance_mask(img)

        # Warp image to output space
        img_warped = cv2.warpPerspective(img, T, (output_w, output_h))
        mask_warped = cv2.warpPerspective(mask, T, (output_w, output_h))

        # Expand mask to 3 channels
        mask_3d = np.dstack([mask_warped] * 3)

        # Accumulate weighted image
        result += img_warped.astype(np.float32) * mask_3d
        weights += mask_3d

    # Normalize by weights
    weights = np.maximum(weights, 1e-6)  # Avoid division by zero
    result = result / weights

    return np.clip(result, 0, 255).astype(np.uint8)

def stitch_sequential(images, keypoints, descriptors):
    """Stitch images side-by-side with exposure matching and advanced blending"""
    print("\nStarting sequential stitching with exposure matching...")

    # Initialize panorama with first image
    pano = images[0].copy()
    print(f"Starting with image 1: {pano.shape}")

    # Store homographies for later use
    homographies = [np.eye(3)]

    # Store shift values for global adjustment
    shifts = [0]

    for i in range(1, len(images)):
        print(f"\nStitching image {i+1}...")
        current_img = images[i]

        # Match with previous image
        matches = match_features(descriptors[i-1], descriptors[i])
        print(f"  Matches found: {len(matches)}")
        if len(matches) < 10:
            print(f"  ⚠  Low matches, using translation transform")
            T = compute_translation_transform(keypoints[i-1], keypoints[i], matches)
            if T is not None:
                shift = T[0, 2]
                shifts.append(shift)
                homographies.append(T)
                overlap_width = int(abs(shift) * 0.7)
            else:
                overlap_width = min(current_img.shape[1] // 4, pano.shape[1] // 4)

            # Apply exposure matching before blending
            gamma = match_exposure_pair(pano, current_img, np.eye(3) if T is None 
else T)
            current_img = apply_gamma_correction(current_img, gamma)

            pano = stitch_with_pyramid_blending(pano, current_img, overlap_width)
            print(f"  Panorama shape now: {pano.shape}")
            continue

        # Try homography first (for perspective correction)
        H = compute_homography_ransac(keypoints[i-1], keypoints[i], matches)

        if H is None:
            print(f"  ⚠  Homography failed, using translation")
            T = compute_translation_transform(keypoints[i-1], keypoints[i], matches)
            if T is not None:
                shift = T[0, 2]
                shifts.append(shift)
                homographies.append(T)
                overlap_width = int(abs(shift) * 0.7)
            else:
                overlap_width = min(current_img.shape[1] // 4, pano.shape[1] // 4)

            # Apply exposure matching
            gamma = match_exposure_pair(pano, current_img, np.eye(3) if T is None 
else T)
            current_img = apply_gamma_correction(current_img, gamma)

            pano = stitch_with_pyramid_blending(pano, current_img, overlap_width)
            print(f"  Panorama shape now: {pano.shape}")
            continue

        print(f"  ✓ Homography computed")
        homographies.append(H)

        # Calculate overlap from transformation
        shift = H[0, 2]
        shifts.append(shift)
        overlap_width = int(abs(shift) * 0.7) if abs(shift) > 0 else
```

```
min(current_img.shape[1] // 4, pano.shape[1] // 4)
        overlap_width = max(50, min(overlap_width, min(pano.shape[1], 
current_img.shape[1]) // 2))

        print(f"  Shift: {shift:.1f}, Overlap: {overlap_width}")

        # Match exposures between panorama and current image
        gamma = match_exposure_pair(pano, current_img, H)
        print(f"  Exposure gamma: {gamma:.3f}")
        current_img = apply_gamma_correction(current_img, gamma)

        # Stitch using pyramid blending
        pano = stitch_with_pyramid_blending(pano, current_img, overlap_width)
        print(f"  Panorama shape now: {pano.shape}")

    # Global adjustment: end-to-end shift compensation
    print("\nApplying global adjustment...")
    if len(shifts) > 1:
        total_shift = sum(shifts[1:])
        avg_shift_per_image = total_shift / (len(shifts) - 1)
        print(f"  Total shift: {total_shift:.1f}, Average per image: 
{avg_shift_per_image:.1f}")

    # Final cropping
    print("\nCropping black borders...")
    gray = cv2.cvtColor(pano, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, 
cv2.CHAIN_APPROX_SIMPLE)

    if contours:
        largest_contour = max(contours, key=cv2.contourArea)
        x, y, w, h = cv2.boundingRect(largest_contour)
        pano = pano[max(0, y-5):y+h+5, max(0, x-5):x+w+5]
        print(f"  Cropped to: {pano.shape}")

    return pano

def main():
    folder = r"C:\Users\dhirender.pandey\meat cross-section stitching\nature (code, 
image, output)\nature_images"
    output = r"C:\Users\dhirender.pandey\meat cross-section stitching\nature (code, 
image, output)\nature_images output\panorama_sequential.jpg"

    print("Loading images with adaptive resizing...")
    images, filenames = load_images_from_folder(folder, use_adaptive_resize=True)
    print(f"Loaded {len(images)} images")

    if len(images) < 2:
        print("Need at least 2 images")
        return

    # Detect features
    print("\nDetecting features...")
    keypoints = []
    descriptors = []

    for i, img in enumerate(images):
        kp, des = detect_features(img)
        keypoints.append(kp)
        descriptors.append(des)
        print(f"  Image {i+1}: {len(kp)} features")

    # Stitch sequentially
    pano = stitch_sequential(images, keypoints, descriptors)

    # Save
    cv2.imwrite(output, pano)
    print(f"\n✓ Panorama saved to: {output}")
    print(f"Final panorama size: {pano.shape}")

if __name__ == "__main__":
    main()
```