Panoramic Image Stitching: Complete Conceptual & Code Breakdown

PART 1: CORE CONCEPTS

1. What is Panorama Stitching?

Panorama stitching combines multiple overlapping images into a single wide-angle image. Think of it like:

- You take 5 photos while rotating your camera left to right
- The overlapping areas are detected and aligned
- A single seamless panoramic image is created

Why it works: Each consecutive image overlaps by 30-50% with the next. The algorithm finds these overlaps, calculates how much one image needs to rotate/scale/translate to match the other, and blends them together.

---

2. The Stitching Pipeline (7 Steps)

```
Step 1: FEATURE DETECTION → Find distinctive points (corners, edges)
          ↓
Step 2: FEATURE MATCHING → Match corresponding points between images
          ↓
Step 3: COMPUTE HOMOGRAPHY → Calculate transformation matrix
          ↓
Step 4: SANITY CHECK → Reject weird/unreliable transformations
          ↓
Step 5: CYLINDRICAL WARP → (Optional) Reduce distortion from wide angles
          ↓
Step 6: PERSPECTIVE WARP & COMPOSE → Transform and overlay images
          ↓
Step 7: BLENDING & CROPPING → Smooth seams and remove black borders
          ↓
OUTPUT: Final panorama
```

---

3. Key Computer Vision Concepts

A. Feature Detection (SIFT vs ORB)

What are features?

- Distinctive points in an image (corners, blobs, edges)
- Used to find corresponding points between images
- Like finding a unique landmark that appears in both photos

SIFT (Scale-Invariant Feature Transform):

- **Advantage:** Works at different scales (zoom levels), very robust
- **Disadvantage:** Slower, computationally expensive
- **Best for:** High-quality panoramas with challenging conditions
- **How it works:** Detects corners and edges that look the same whether zoomed in or out

ORB (Oriented FAST and Rotated BRIEF):

- **Advantage:** Fast, lightweight, good for mobile/real-time
- **Disadvantage:** Less robust than SIFT
- **Best for:** Quick panoramas, low-power devices
- **How it works:** Detects fast corners and encodes them as binary patterns

Keypoint: Both return:

- **Keypoints (KPs):** x, y coordinates of detected features
- **Descriptors:** Numerical "fingerprints" describing what each feature looks like (so you can match them)

B. Feature Matching (Finding Corresponding Points)

The Problem: You have 5000 keypoints in Image A and 4000 in Image B. Which ones correspond?

Solution: Mutual Ratio Matching

1. **BFMatcher (Brute Force Matcher):** Compare every descriptor in Image A to every descriptor in Image B
2. **K-Nearest Neighbors:** For each keypoint, find the 2 nearest matches in the other image
3. **Lowe's Ratio Test:**
   - If the distance to the closest match is < 0.75 × distance to second-closest → Keep it
   - This filters out ambiguous matches
4. **Mutual Matching:** Only keep matches that work both ways (A→B and B→A agree)

Result: You get ~50-200 high-confidence matching points (instead of thousands of noisy guesses)

C. Homography (The Transformation Matrix)

What is Homography?
A 3×3 matrix that describes how to transform one image's plane to match another's plane.

```
Homography Matrix H:
┌                  ┐
│ h00  h01  h02  │
│ h10  h11  h12  │
│ h20  h21  h22  │
└                  ┘
```

What it does:

- **Rotation:** h00, h01, h10, h11 (tilt the image)
- **Translation:** h02, h12 (shift left/right, up/down)
- **Perspective/Scale:** h20, h21, h22 (3D perspective effects)
- Together: `new_point = H × old_point` (in homogeneous coordinates)

Why it matters: By finding H, we know exactly how to warp Image B so it aligns with Image A.

Calculating H:

- **Input:** 4+ matching keypoints from both images
- **Method:** RANSAC (Random Sample Consensus)
  - Randomly picks 4 points
  - Calculates H
  - Tests how many other points agree with this H
  - Repeats 1000+ times
  - Returns H with the most "inlier" points (points that agree)
- **Result:** Robust H even with 50% noise/bad matches

USAC/MAGSAC: Modern variants of RANSAC that are even more robust.

D. Perspective Warping

The Challenge: Image B is flat, but after stitching, it needs to wrap around to align with Image A.

Solution: cv2.perspectiveTransform

- Takes the corners of Image B
- Applies the transformation H to them
- This tells us: "Image B's corners will end up HERE in the new coordinate system"
- This defines the output canvas size

Example:

```
Image B's original corners: (0,0), (800,0), (800,600), (0,600)
After transformation H: (50,100), (900,80), (850,680), (10,650)
```

```
→ Output canvas needs to be at least 950×680 to hold all pixels
```

E. Cylindrical Projection (Optional Preprocessing)

The Problem: Wide panoramas suffer from "wedge distortion"

- The image stretches and skews at the edges
- Creates huge black triangular areas
- Looks unnatural

The Solution: Cylindrical Warp
Instead of assuming the camera captures a flat plane, assume it captures a cylindrical surface wrapping around it.

Math:

```
For each pixel (x, y) in the original image:
1. Normalize to angle: angle = (x - center_x) / focal_length
2. Apply cylinder projection:
   x_cylinder = focal_length × tan(angle)
   y_cylinder = y / cos(angle)
3. Map back to new coordinates
```

Effect:

- Reduces extreme stretching at edges
- Creates a more natural-looking panorama
- Focal length parameter controls how much: higher = less distortion

When to use:

- **Wide panoramas (>120°):** Very helpful
- **Narrow panoramas (<60°):** Less necessary

F. Blending (Feather Blend)

The Problem: When overlapping areas are directly composited, you see a harsh seam line.

The Solution: Feather Blending

1. Create a mask for each image showing which pixels are valid (not black)
2. In the overlap region, create a **gradient alpha mask** using distance transform
   - Pixels far from the edge: fully from one image
   - Pixels near the edge: blended (50/50)
3. This creates a smooth fade between images

Result: Invisible seams where images overlap

G. Stitching Order: Sequential vs Center Reference

Sequential Stitching:

```
Image 1 + Image 2 → Panorama A
Panorama A + Image 3 → Panorama B
Panorama B + Image 4 → Final Panorama
```

- **Pro:** Simple logic
- **Con:** Errors accumulate (drift); later images misaligned
- **Use:** When images are in perfect order

Center Reference Stitching:

```
        Image 1
           ↗
```

```
Image 0 ← Image 2 (center)
            ↖
          Image 3

Stitch left images (0,1) to center
Stitch right images (3,4) to center
```

- **Pro:** Errors don't accumulate; more balanced
- **Con:** Requires images in proper order
- **Use:** High-quality panoramas; recommended default

---

4. Safety Mechanisms in the Code

A. Homography Sanity Check

```
Reject H if:
- Perspective coefficients are too large (h20, h21 > 0.01)
  → Means H is creating extreme perspective (impossible without 3D rotation)
- Projected bbox is wildly larger than input (>3× expansion)
  → Means H is creating a huge wedge (unreliable)
```

B. Quality Gates

```
Require:
- Minimum 18 inlier matches (out of usually 100+)
- Inlier ratio > 25% (75% matches are good, not just lucky)
- Otherwise, try with relaxed ratio thresholds
- Final fallback: Use affine transformation (more stable, simpler)
```

C. Canvas Size Cap

```
If output gets larger than 2500×4000:
  - Scale everything down proportionally
  - Prevents multi-GB memory usage
  - Trades resolution for stability
```

PART 2: CODE BREAKDOWN

Section 1: Utility Functions

`resize_to_max_width(img, max_width=800)`

```python
def resize_to_max_width(img, max_width=MAX_INPUT_WIDTH):
    h, w = img.shape[:2]  # Get height, width (ignore channels)
    if w <= max_width:
        return img  # Already small enough
    scale = max_width / float(w)  # Calculate scale factor
    # Resize maintaining aspect ratio
    return cv2.resize(img, (int(w * scale), int(h * scale)),
                      interpolation=cv2.INTER_AREA)
```

Purpose: Reduce image size to speed up feature detection
Why INTER_AREA? Best quality for downsampling

`to_gray(img)`

```python
def to_gray(img):
    # Handle RGBA (remove alpha channel)
    if len(img.shape) == 3 and img.shape[2] == 4:
        img = cv2.cvtColor(img, cv2.COLOR_BGRA2BGR)
    # Convert to grayscale for feature detection
    return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Purpose: Standardize input to grayscale (required for SIFT/ORB)

`crop_black_borders(image, threshold=10)`

```python
def crop_black_borders(image, threshold=10):
    """Remove black areas added by warping."""
    if image is None or image.size == 0:
        return image

    # Find non-black pixels
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    mask = (gray > threshold).astype(np.uint8) * 255
    # 'threshold' = pixels with intensity > 10 are considered valid

    if cv2.countNonZero(mask) == 0:  # All black?
        return image

    # Clean up the mask (remove noise)
    kernel = np.ones((7, 7), np.uint8)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel, iterations=2)
    # MORPH_CLOSE: Fill small holes in the mask

    # Find the bounding box of all non-black content
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if not contours:
        return image

    largest = max(contours, key=cv2.contourArea)
    x, y, w, h = cv2.boundingRect(largest)

    # Add small margin to avoid cutting content
    margin = 3
    x = max(0, x + margin)
    y = max(0, y + margin)
    w = max(1, w - 2 * margin)
    h = max(1, h - 2 * margin)

    return image[y:y+h, x:x+w]  # Crop
```

Purpose: Remove black borders created by perspective warping
Method:

1. Create mask of non-black pixels
2. Fill small gaps (morphological closing)
3. Find bounding box
4. Crop to that box

`feather_blend(base, warped, feather_iters=2)`

```python
def feather_blend(base, warped, feather_iters=2):
    """Blend two overlapping images with smooth seams."""

    # Get masks of valid pixels
    base_gray = cv2.cvtColor(base, cv2.COLOR_BGR2GRAY)
    warped_gray = cv2.cvtColor(warped, cv2.COLOR_BGR2GRAY)

    base_mask = (base_gray > 0).astype(np.uint8)
    warped_mask = (warped_gray > 0).astype(np.uint8)
    # Binary masks: 1 = valid content, 0 = transparent/black

    overlap = (base_mask & warped_mask).astype(np.uint8)

    # If no overlap, just composite
    if cv2.countNonZero(overlap) == 0:
        out = base.copy()
        out[warped_mask == 1] = warped[warped_mask == 1]
        return out
```

```
    # Erode the warped mask to create a feather region
    warped_mask_255 = (warped_mask * 255).astype(np.uint8)
    kernel = np.ones((3, 3), np.uint8)
    for _ in range(max(1, feather_iters)):
        # Shrink the mask by 1 pixel each iteration
        warped_mask_255 = cv2.erode(warped_mask_255, kernel, iterations=1)

    # Distance transform: for each pixel, compute distance to nearest eroded boundary
    dist = cv2.distanceTransform(warped_mask_255, cv2.DIST_L2, 3)
    # This creates a gradient: 0 at edges, max at center

    # Normalize to [0, 1]
    alpha = dist / (dist.max() + 1e-6) if dist.max() > 0 else dist
    alpha *= warped_mask  # Zero out invalid regions

    # Apply alpha blending
    alpha_3 = np.dstack([alpha, alpha, alpha]).astype(np.float16)
    base_f = base.astype(np.float16)
    warped_f = warped.astype(np.float16)
    # Blend: warped×alpha + base×(1-alpha)
    out = (warped_f * alpha_3 + base_f * (1.0 - alpha_3)).astype(np.uint8)

    # Handle non-overlapping regions (just place the image)
    out[(base_mask == 0) & (warped_mask == 1)] = warped[(base_mask == 0) & (warped_mask == 1)]
    out[(base_mask == 1) & (warped_mask == 0)] = base[(base_mask == 1) & (warped_mask == 0)]

    return out
```

Purpose: Smooth blending at image seams
Key Idea: Distance transform creates smooth alpha gradient at edges

---

Section 2: Cylindrical Warping

```
def cylindrical_warp(img, f):
    """
    Project 2D image onto cylindrical surface.
    f = focal length (higher = less distortion, typical 700-1200)
    """
    h, w = img.shape[:2]
    cx, cy = w / 2.0, h / 2.0  # Image center

    # Create coordinate grids for the entire image
    y_i, x_i = np.indices((h, w))

    # Normalize coordinates relative to center
    x = (x_i - cx) / f
    y = (y_i - cy) / f

    # Apply cylindrical projection:
    # x_c = tan(x) creates the cylindrical effect
    # y_c = y / cos(x) adjusts vertical based on horizontal angle
    x_c = np.tan(x)
    y_c = y / np.cos(x)

    # Convert back to pixel coordinates
    map_x = (f * x_c + cx).astype(np.float32)
    map_y = (f * y_c + cy).astype(np.float32)

    # Apply the warp
    warped = cv2.remap(img, map_x, map_y,
                       interpolation=cv2.INTER_LINEAR,
                       borderMode=cv2.BORDER_CONSTANT)
    return warped
```

Purpose: Reduce distortion in wide-angle panoramas
How it works:

- Instead of assuming flat plane, assume cylinder
- `tan(x)` maps linear pixel distance to curved angle

- Result: More natural-looking seams

Section 3: Feature Detection & Matching

```python
def detect_and_describe(img, method="SIFT"):
    """Detect keypoints and compute descriptors."""
    gray = to_gray(img)

    if method == "SIFT":
        detector = cv2.SIFT_create()
    else:
        detector = cv2.ORB_create(5000)  # Find up to 5000 keypoints

    kps, des = detector.detectAndCompute(gray, None)
    # kps = list of keypoint objects with .pt (x,y) attribute
    # des = NxM matrix where each row is a descriptor

    if kps is None or len(kps) == 0 or des is None:
        return None, None

    # Convert keypoints to numpy array of (x, y) coordinates
    pts = np.float32([kp.pt for kp in kps])
    return pts, des
```

Purpose: Find distinctive features in image
Returns:

- `pts` : Nx2 array of (x, y) coordinates
- `des` : NxD array of descriptors (D=128 for SIFT, D=256 for ORB)

```python
def mutual_ratio_matches(des1, des2, method="SIFT", ratio=0.75):
    """Find reliable matching keypoints using Lowe's ratio test."""

    matcher = _bfmatcher(method)

    # Find 2 nearest neighbors for each descriptor in des1
    m12 = matcher.knnMatch(des1, des2, k=2)
    good12 = {}
    for pair in m12:
        if len(pair) == 2:
            m, n = pair  # 1st nearest, 2nd nearest
            # Lowe's ratio test: 1st match must be significantly better than 2nd
            if m.distance < ratio * n.distance:
                good12[(m.queryIdx, m.trainIdx)] = m.distance

    # Reverse direction: des2 to des1
    m21 = matcher.knnMatch(des2, des1, k=2)
    good21 = {}
    for pair in m21:
        if len(pair) == 2:
            m, n = pair
            if m.distance < ratio * n.distance:
                good21[(m.trainIdx, m.queryIdx)] = m.distance

    # Keep only mutual matches (both directions agree)
    mutual = []
    for (q, t), d in good12.items():
        if (q, t) in good21:
            mutual.append((q, t, d))

    # Sort by distance (best matches first)
    mutual.sort(key=lambda x: x[2])
    return mutual
```

Purpose: Find corresponding keypoints between two images
Process:

1. For each keypoint in Image 1, find 2 nearest neighbors in Image 2
2. If 1st nearest is much closer than 2nd → good match
3. Verify match works both ways (mutual consistency)
4. Return sorted list of matching keypoint indices

Section 4: Homography Computation

```python
def find_homography(src_pts, dst_pts, reproj=4.0):
    """Compute homography using RANSAC."""
    if hasattr(cv2, "USAC_MAGSAC"):
        # Modern robust method
        H, mask = cv2.findHomography(src_pts, dst_pts,
                                     method=cv2.USAC_MAGSAC,
                                     ransacReprojThreshold=reproj)
    else:
        # Fallback to classic RANSAC
        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, reproj)
    return H, mask
    # H: 3x3 homography matrix
    # mask: 1x N binary mask (1=inlier, 0=outlier)
```

Purpose: Calculate transformation from matched points
reproj: Reprojection threshold (pixels) - how much error is acceptable

---

```python
def homography_is_sane(H, img2_shape, max_expand=3.0):
    """Reject homographies that create unrealistic transformations."""

    if H is None:
        return False

    # Reject if perspective components are too large
    if abs(H[2, 0]) > 0.01 or abs(H[2, 1]) > 0.01:
        return False  # Extreme perspective → wrong match

    h2, w2 = img2_shape[:2]
    # Get the 4 corners of the image
    corners2 = np.float32([[0, 0], [w2, 0], [w2, h2], [0, h2]]).reshape(-1, 1, 2)

    # Apply homography to corners (where do they end up?)
    wc = cv2.perspectiveTransform(corners2, H)
    xs = wc[:, 0, 0]
    ys = wc[:, 0, 1]

    # Calculate bounding box of warped corners
    w_proj = (xs.max() - xs.min())
    h_proj = (ys.max() - ys.min())

    if w_proj <= 0 or h_proj <= 0:
        return False  # Degenerate

    # Reject if projection creates massive expansion (wedge effect)
    if w_proj > max_expand * w2 or h_proj > max_expand * h2:
        return False  # Creates huge triangle/wedge

    return True
```

Purpose: Reject bad homographies before using them
Checks:

1. Perspective coefficients not too extreme
2. Warped image bbox not unreasonably large

Section 5: Warping & Composition

```python
def warp_and_compose(img1, img2, H):
    """Apply homography warp and blend two images."""

    h1, w1 = img1.shape[:2]
    h2, w2 = img2.shape[:2]

    # Get warped corners of img2
    corners2 = np.float32([[0, 0], [w2, 0], [w2, h2], [0, h2]]).reshape(-1, 1, 2)
    warped_corners2 = cv2.perspectiveTransform(corners2, H)

    # Get original corners of img1
    corners1 = np.float32([[0, 0], [w1, 0], [w1, h1], [0, h1]]).reshape(-1, 1, 2)

    # Combine all corners to find output canvas size
    all_corners = np.concatenate((corners1, warped_corners2), axis=0)

    [xmin, ymin] = np.int32(all_corners.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(all_corners.max(axis=0).ravel() + 0.5)

    # Translation matrix: shift everything to positive coordinates
    translation = [-xmin, -ymin]
    T = np.array([[1, 0, translation[0]],
                  [0, 1, translation[1]],
                  [0, 0, 1]], dtype=np.float64)

    output_size = (xmax - xmin, ymax - ymin)
    ow, oh = output_size

    # Cap canvas size to prevent memory issues
    max_canvas_w = MAX_PANO_WIDTH * 2
    max_canvas_h = 4000
    if ow > max_canvas_w or oh > max_canvas_h:
        scale = min(max_canvas_w / float(ow), max_canvas_h / float(oh))
        output_size = (max(1, int(ow * scale)), max(1, int(oh * scale)))
        S = np.array([[scale, 0, 0],
                      [0, scale, 0],
                      [0, 0, 1]], dtype=np.float64)
        H = S @ H  # Apply scale to transformation
        T = S @ T
        translation = [int(translation[0] * scale), int(translation[1] * scale)]
        img1 = cv2.resize(img1, (int(w1 * scale), int(h1 * scale)), interpolation=cv2.INTER_AREA)
        h1, w1 = img1.shape[:2]

    # Apply the combined transformation to img2
    warped = cv2.warpPerspective(img2, T @ H, output_size)

    # Create canvas and place img1
    canvas = np.zeros_like(warped)
    y0, x0 = translation[1], translation[0]
    y1, x1 = y0 + h1, x0 + w1

    # Clip to canvas boundaries
    y0c, x0c = max(0, y0), max(0, x0)
    y1c, x1c = min(canvas.shape[0], y1), min(canvas.shape[1], x1)
    if y1c > y0c and x1c > x0c:
        canvas[y0c:y1c, x0c:x1c] = img1[0:(y1c - y0c), 0:(x1c - x0c)]

    # Blend overlapping regions
    result = feather_blend(canvas, warped, feather_iters=2)
    return result
```

Purpose: Combine two images using homography
Steps:

1. Calculate output canvas size from warped corners
2. Create translation matrix to shift into positive coordinates
3. Optionally scale down to prevent memory explosion
4. Apply perspective transform to img2
5. Place img1 on canvas
6. Blend seams with feather blending

Section 6: Pair Stitching

```
def stitch_pair(img1, img2, method="SIFT", diagnostics=False):
    """Stitch two images together."""

    diag = {}

    # Step 1: Detect and describe features
    pts1, des1 = detect_and_describe(img1, method=method)
    pts2, des2 = detect_and_describe(img2, method=method)
    if des1 is None or des2 is None or pts1 is None or pts2 is None:
        raise ValueError("Not enough features in one of the images.")

    # Step 2: Try different matching ratios (adaptive)
    ratios = [0.75, 0.85] if method == "SIFT" else [0.70, 0.80]

    for r in ratios:
        # Find mutual matches
        mutual = mutual_ratio_matches(des1, des2, method=method, ratio=r)

        # Require minimum matches
        if len(mutual) < (25 if method == "ORB" else 12):
            continue

        # Use top matches
        mutual_use = mutual[:800]
        src = np.float32([pts2[t] for (q, t, _) in mutual_use])  # img2 points
        dst = np.float32([pts1[q] for (q, t, _) in mutual_use])  # img1 points

        # Compute homography
        H, mask = find_homography(src, dst, reproj=4.0 if method == "SIFT" else 6.0)
        if H is None or mask is None:
            continue

        # Count inliers
        inliers = int(mask.sum())
        inlier_ratio = inliers / float(len(mutual_use))

        # Sanity check
        if not homography_is_sane(H, img2.shape, max_expand=3.0):
            continue

        # Quality gates
        if inliers < 18 or inlier_ratio < 0.25:
            continue

        # Success! Save diagnostics and return
        if diagnostics:
            diag["method"] = method
            diag["ratio_used"] = r
            diag["kps_img1"] = len(pts1)
            diag["kps_img2"] = len(pts2)
            diag["mutual_matches"] = len(mutual_use)
            diag["inliers"] = inliers
            diag["inlier_ratio"] = round(inlier_ratio, 3)
            diag["match_viz_rgb"] = draw_match_viz(
                img1, img2, pts1, pts2, mutual_use,
                inlier_mask=mask.ravel().astype(bool)
            )

        return warp_and_compose(img1, img2, H), diag

    # Fallback: Use affine transformation (more stable, simpler)
    mutual = mutual_ratio_matches(des1, des2, method=method, ratio=0.9)
    if len(mutual) >= 20:
        mutual_use = mutual[:600]
        src = np.float32([pts2[t] for (q, t, _) in mutual_use])
        dst = np.float32([pts1[q] for (q, t, _) in mutual_use])

        A, inl = cv2.estimateAffinePartial2D(src, dst,
                                             method=cv2.RANSAC,
                                             ransacReprojThreshold=6.0)
        if A is not None:
```

```
            # Convert affine to homography (add bottom row [0, 0, 1])
            H_aff = np.array([[A[0, 0], A[0, 1], A[0, 2]],
                              [A[1, 0], A[1, 1], A[1, 2]],
                              [0, 0, 1]], dtype=np.float64)

            if diagnostics:
                diag["used_affine_fallback"] = True
                diag["kps_img1"] = len(pts1)
                diag["kps_img2"] = len(pts2)
                diag["mutual_matches"] = len(mutual_use)
                diag["inliers"] = int(inl.sum()) if inl is not None else 0
                diag["match_viz_rgb"] = draw_match_viz(img1, img2, pts1, pts2, mutual_use, inlier_mask=None)

            return warp_and_compose(img1, img2, H_aff), diag

    raise ValueError("Homography estimation failed (matches unreliable / overlap weak).")
```

Purpose: Stitch two specific images
Strategy:

1. Try homography with strict matching ratio
2. If that fails, try with relaxed ratio
3. If that fails, use affine transformation (simpler, more robust)
4. If that fails, raise error

Section 7: Multi-Image Stitching

```
def stitch_images_sequential(images, method="SIFT", diagnostics=False, progress_callback=None):
    """Stitch multiple images in order."""
    pano = images[0].copy()  # Start with first image
    all_diags = []
    total_steps = len(images) - 1

    for i in range(1, len(images)):
        if progress_callback:
            pct = int((i / total_steps) * 100)
            progress_callback(pct, f"Stitching image {i+1}/{len(images)}...")

        # Stitch accumulated panorama with next image
        pano, diag = stitch_pair(pano, images[i], method=method, diagnostics=diagnostics)
        all_diags.append(diag)

    # Remove black borders
    pano = crop_black_borders(pano, threshold=10)
    return pano, all_diags
```

Purpose: Stitch sequence of images left-to-right
Pro: Simple logic
Con: Errors accumulate

```
def stitch_images_center(images, method="SIFT", diagnostics=False, progress_callback=None):
    """Stitch images around a central reference image."""

    center_idx = len(images) // 2
    pano = images[center_idx].copy()  # Start from middle
    all_diags = []
    total_ops = len(images) - 1
    done = 0

    # Stitch images to the LEFT of center
    for i in range(center_idx - 1, -1, -1):
        done += 1
        if progress_callback:
            pct = int((done / total_ops) * 100)
            progress_callback(pct, f"Stitching image {i+1} (Left)...")

        # Stitch image i (left) with current panorama
```

```
        pano, diag = stitch_pair(images[i], pano, method=method, diagnostics=diagnostics)
        all_diags.append(diag)

    # Stitch images to the RIGHT of center
    for i in range(center_idx + 1, len(images)):
        done += 1
        if progress_callback:
            pct = int((done / total_ops) * 100)
            progress_callback(pct, f"Stitching image {i+1} (Right)...")

        # Stitch current panorama with image i (right)
        pano, diag = stitch_pair(pano, images[i], method=method, diagnostics=diagnostics)
        all_diags.append(diag)

    pano = crop_black_borders(pano, threshold=10)
    return pano, all_diags
```

Purpose: Stitch images starting from center, expanding outward
Pro: Errors distributed symmetrically, better quality
Con: More complex logic

Section 8: Streamlit UI

The Streamlit code creates the web interface with:

- **Sidebar:** Controls (SIFT vs ORB, Sequential vs Center, cylindrical warp toggle)
- **File uploader:** Accept multiple images
- **Preview:** Show uploaded images
- **Stitch button:** Run the panorama stitching
- **Results:** Display panorama and diagnostics

Key settings:

```
method              # "SIFT" (slow, accurate) or "ORB" (fast)
stitch_mode         # "Sequential" or "Center Reference"
use_cyl             # Enable cylindrical warping
f_slider            # Focal length for cylindrical warp (700-1200 recommended)
resize_enabled      # Enable image downsampling
custom_width        # Max input width (trade-off: speed vs quality)
diagnostics         # Show detailed matching info and match visualization
```

SUMMARY TABLE

Component	Purpose	Key Parameter
SIFT	Detect scale-invariant features	Method choice
ORB	Fast feature detection	Method choice
BFMatcher	Compare descriptors	Ratio threshold
Lowe's Ratio Test	Filter bad matches	0.75-0.85
Mutual Matching	Bidirectional consistency	Both directions
Homography	3×3 transformation matrix	RANSAC threshold
Sanity Check	Reject unreliable H	max_expand param
Cylindrical Warp	Reduce distortion	Focal length
Perspective Warp	Apply transformation	Canvas size calc
Feather Blend	Smooth seams	Distance transform
Sequential Stitch	Left-to-right order	Simple but drifts
Center Stitch	Balanced approach	Better quality

TROUBLESHOOTING GUIDE

Issue	Solution
Huge black wedge	Enable cylindrical warp, increase focal length
Ghosting/double images	Insufficient overlap (need 30-50%)
Wrong keypoint matches	Try SIFT instead of ORB, reduce image width
Misaligned images	Check image order, try Center Reference mode
Memory issues	Reduce MAX_INPUT_WIDTH to 600-800
Homography failed	Ensure real overlap, adjust focal length, try lower quality images