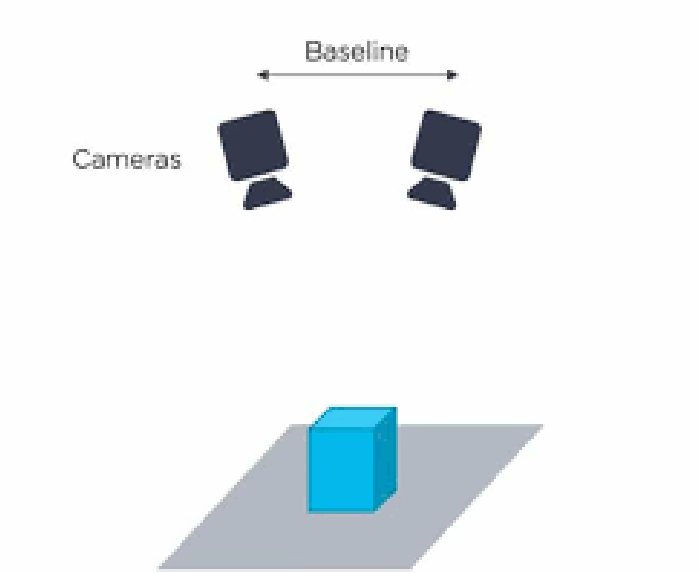
CV Assignment 3

Stereo Vision



Ahmed Samir Said Ahmed	20010107
Youssef Hossam Aboelwafa	20012263
Mohamed Raffeek Mohamed	20011574

Introduction

This report presents the implementation and results of stereo vision algorithms as required in Assignment 3. The assignment involves computing disparity maps between stereo image pairs using two primary approaches:

1. Block Matching with different cost functions and window sizes
2. Dynamic Programming for optimal scanline matching

Implementation Details

1. Block Matching

Block matching is implemented as a window-based approach where for each pixel in the left image, we find the best matching pixel in the right image along the same scanline.

Implementation Steps:

For each pixel position (x, y) in the left image:

For each possible disparity d from 0 to disparity range:

Check if the x-d is out of range

Extract a window of size w × w centered at (x, y)

Extract a window of the same size from the right image at position (x-d, y)

Compute the matching cost using either SAD or SSD

Assign the disparity with the minimum cost to the disparity map at position (x, y)

Update disparity map with best disparity in the range

Return disparity map

Cost Functions:

- **Sum of Absolute Differences (SAD)**:

$$cost = \sum |W_l - W_r|$$

- **Sum of Squared Differences (SSD)**:

$$cost = \sum \|W_l - W_r\|$$

Window Sizes:

As required, the implementation uses window sizes of w = 1, 5, and 9 pixels.

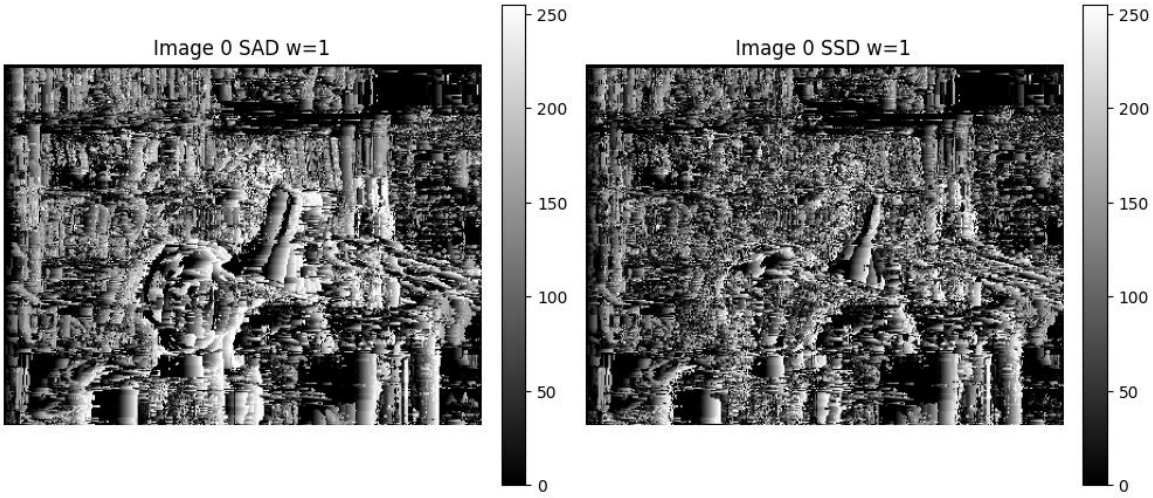
Code:

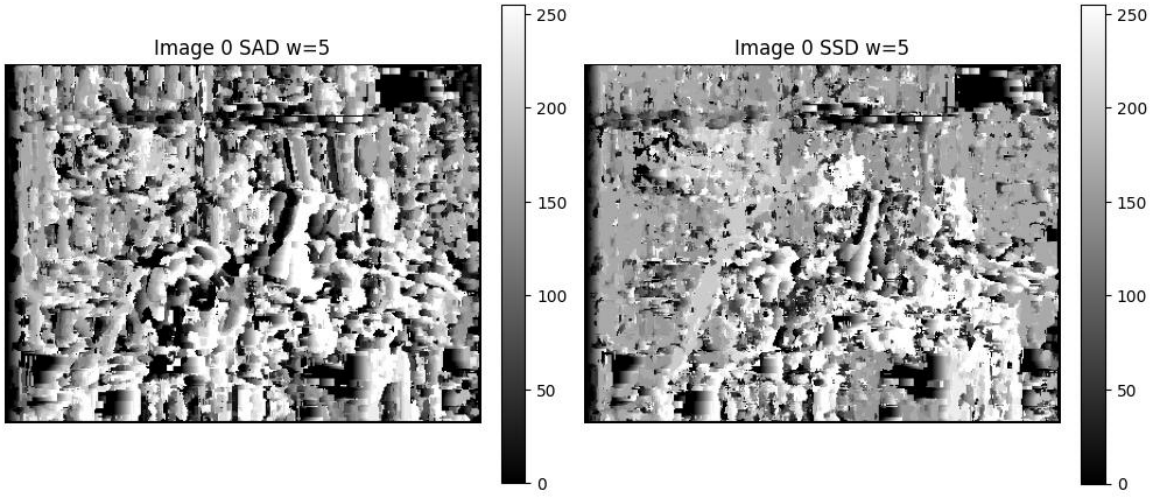
```python
def sad(left_patch, right_patch):
    return np.sum(np.abs(left_patch - right_patch))

def ssd(left_patch, right_patch):
    return np.sum((left_patch - right_patch) ** 2)
```

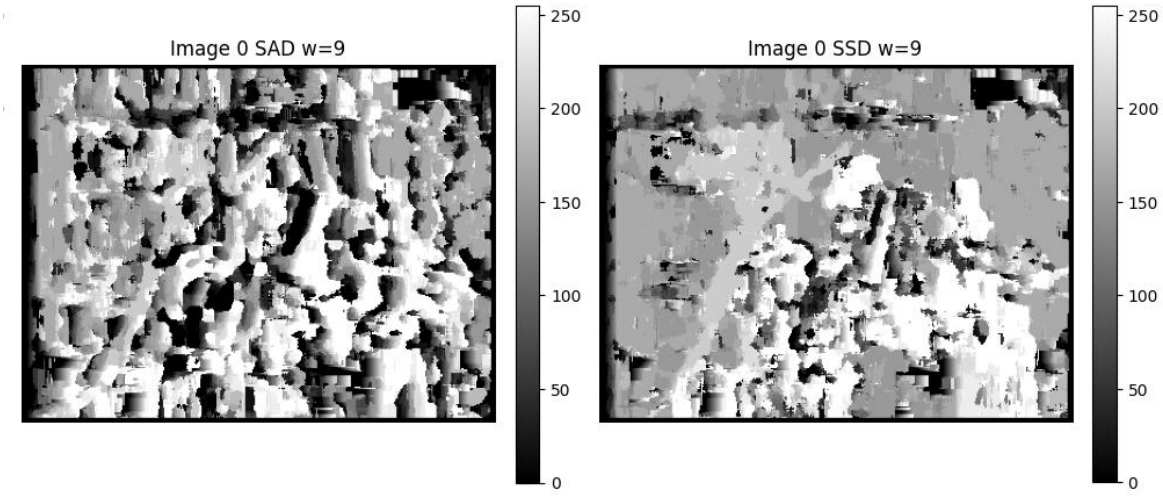
```python
def block_matching(left, right, window_size, method, disparity_range=DISPARITY_RANGE):
    h, w = left.shape
    disparity_map = np.zeros((h, w), dtype=np.uint8)
    half_w = window_size // 2

    for y in tqdm(range(half_w, h - half_w) , desc="Block matching ({method}, w={window_size})"):
        for x in range(half_w, w - half_w):
            min_cost = float('inf')
            best_disparity = 0
            for d in range(disparity_range):
                if x - d - half_w < 0:
                    continue
                left_patch = left[y - half_w:y + half_w + 1, x - half_w:x + half_w + 1]
                right_patch = right[y - half_w:y + half_w + 1, x - d - half_w:x - d + half_w + 1]
                cost = method(left_patch, right_patch)
                if cost < min_cost:
                    min_cost = cost
                    best_disparity = d
            disparity_map[y, x] = best_disparity * (255 // disparity_range)
    return disparity_map
```

```python
img_id = 0
disparity_maps = []
titles = []
left, right = load_images(image_paths[img_id][0],image_paths[img_id][1])
for w in WINDOW_SIZES:
    print(f'Computing disparity map for window size {w}...')
    print('SAD method...')
    sad_map = block_matching(left, right, w, sad)
    print('SSD method...')
    ssd_map = block_matching(left, right, w, ssd)
    cv2.imwrite(f'disparity/disparity_imag_{img_id}_sad_w{w}.png', sad_map)
    cv2.imwrite(f'disparity/disparity_imag_{img_id}_ssd_w{w}.png', ssd_map)
    disparity_maps.append(sad_map)
    titles.append(f'Image {img_id} SAD w={w}')
    disparity_maps.append(ssd_map)
    titles.append(f'Image {img_id} SSD w={w}')
```

Test Samples:







2. Dynamic Programming

The dynamic programming approach finds the optimal alignment between scanlines by minimizing a global cost function that considers both matching costs and occlusion penalties.

Implementation Steps:

For each scanline y:

Compute the pixel-wise cost matrix d_{ij} where:

$$d_{ij} = \frac{(I_l(i) - I_r(j))^2}{\sigma^2}$$

Initialize the dynamic programming matrix D

Fill the first row and first column

Fill the matrix using the recurrence relation:

$$D(i,j) = \min\left(D(i-1,j-1) + d_{ij}, D(i-1,j) + C_0, D(i,j-1) + C_0\right)$$

Backtrack from $D(N,N)$to $D(1,1)$ to find the optimal alignment path

Compute disparity maps for both left and right images based on the path

Calculate the alignment path

Parameters:

- $\sigma = 2$(pixel noise measure)
- $C_0 = 1$ (occlusion cost)

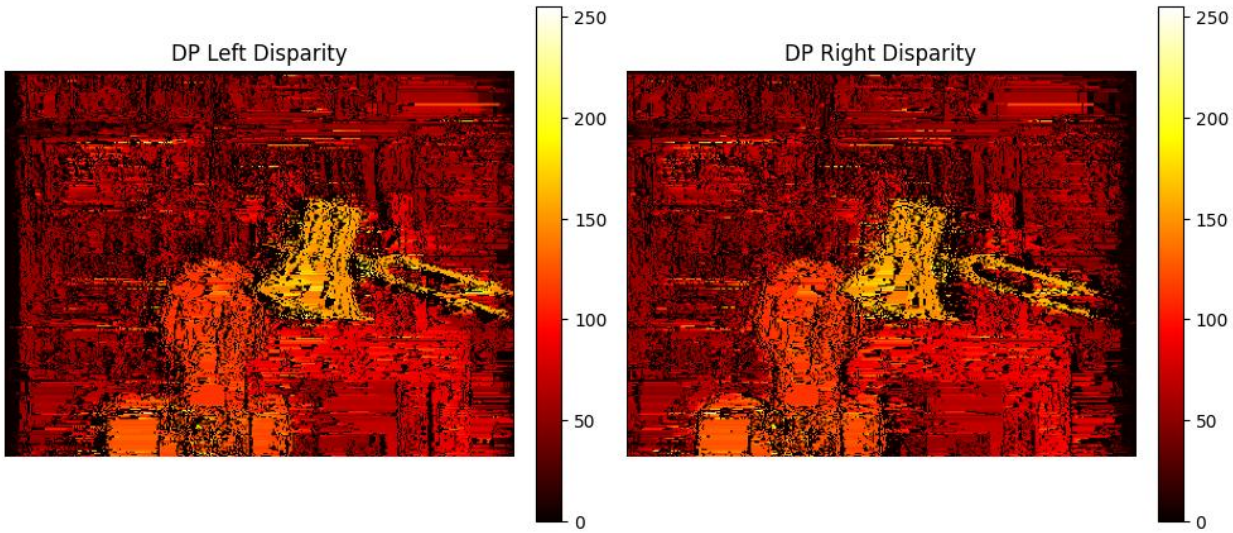
Code:

```
def dynamic_programming_stereo(left_img, right_img, sigma=2, c0=1):
    h, w = left_img.shape[:2]
    left_disparity_map = np.zeros((h, w), dtype=np.float32)
    right_disparity_map = np.zeros((h, w), dtype=np.float32)
    for y in tqdm(range(h), desc="Dynamic programming"):
        left_scanline = left_img[y, :].astype(float)
        right_scanline = right_img[y, :].astype(float)
        d_ij = [(left_scanline[i] - right_scanline[j]) ** 2 / (sigma ** 2)
                for i in range(w) for j in range(w)]
        d_ij = np.array(d_ij).reshape((w, w))
        D = np.zeros((w , w ), dtype=float)
        path = np.zeros((w , w ), dtype=np.int32)
        D[0, 0] = d_ij[0, 0]
        for j in range(1, w ):
            D[0, j] = D[0, j-1] + c0
            path[0, j] = 2
        for i in range(1, w ):
            D[i, 0] = D[i-1, 0] + c0
            path[i, 0] = 1
        for i in range(1, w ):
            for j in range(1, w ):
                cost_diag = D[i-1, j-1] + d_ij[i, j]
                cost_left = D[i, j-1] + c0
                cost_up = D[i-1, j] + c0
                min_cost = min(cost_diag, cost_left, cost_up)
                D[i, j] = min_cost
                if min_cost == cost_diag:
                    path[i, j] = 0
                elif min_cost == cost_left:
                    path[i, j] = 2
                else:
                    path[i, j] = 1
```

```
        i, j = w-1, w-1
        while i > 0 or j > 0:
            if path[i, j] == 0:
                left_disparity_map[y, i] = abs(i - j) * (255 // 16)
                right_disparity_map[y, j] = abs(i - j) * (255 // 16)
                i -= 1
                j -= 1
            elif path[i, j] == 1:
                left_disparity_map[y, i] = 0
                i -= 1
            else:
                right_disparity_map[y, j] = 0
                j -= 1
        alignment_path = plot_alignment_path(path, w)
    return left_disparity_map, right_disparity_map, alignment_path
```

```
img_id = 0
left_img, right_img = load_images(image_paths[img_id][0], image_paths[img_id][1])
left_disparity_map, right_disparity_map, alignment_path = dynamic_programming_stereo(left_img, right_img)

cv2.imwrite(f'disparity/disparity_imag_{img_id}_dp_left.png', left_disparity_map)
cv2.imwrite(f'disparity/disparity_imag_{img_id}_dp_right.png', right_disparity_map)

plt.imshow(left_disparity_map, cmap='hot')
plt.show()
plt.imshow(right_disparity_map, cmap='hot')
plt.show()

fig_dp = visualize_disparity_maps([left_disparity_map, right_disparity_map],
                                  ["DP Left Disparity", "DP Right Disparity"])
fig_dp.savefig("disparity/dynamic_programming_results.png")
plt.close(fig_dp)
```

Test Samples:



3. Alignment Path Visualization

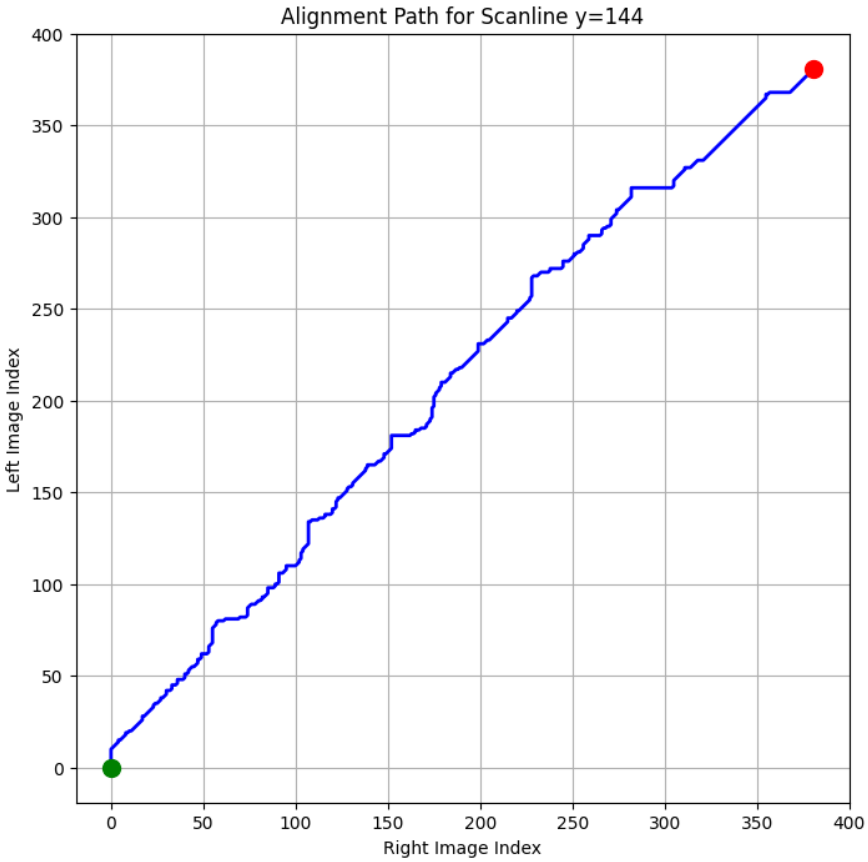
For the bonus section, we visualize the alignment path for a single scanline (The middle scanline). The implementation:

- Extracts the path during backtracking
- Plots it as a graph with the left image index on the vertical axis and the right image index on the horizontal axis
- Uses different line styles to represent matches, left occlusions, and right occlusions

Code:

```
def plot_alignment_path(path, n):
    i, j = n-1, n-1
    align_path = [(i, j)]
    while i > 0 or j > 0:
        if path[i, j] == 0:
            i -= 1
            j -= 1
        elif path[i, j] == 1:
            i -= 1
        else:
            j -= 1
        align_path.append((i, j))
    return align_path[::-1]
```

Test Samples:



Results and Analysis

Block Matching Results

The block matching algorithm produced six disparity maps, one for each combination of cost function (SAD, SSD) and window size (1, 5, 9).

Key Observations:

1. **Window Size Effects**:
    - w=1: Highly detailed but noisy disparity maps
    - w=5: Better noise reduction with reasonable detail preservation
    - w=9: Smoothest results but with loss of detail at object boundaries
2. **Cost Function Comparison**:
    - SAD: Generally faster computation
    - SSD: More sensitive to outliers but can provide slightly better results in areas with gradual intensity changes

Dynamic Programming Results

The dynamic programming approach produced left and right disparity maps simultaneously.

Key Observations:

1. The algorithm explicitly models pixel skipping, which helps handle occlusions between the stereo pair.
2. By considering the entire scanline, the approach produces more consistent disparity estimates compared to local methods.

Alignment Path Visualization

The alignment path visualization shows the matching between pixels in the left and right scanlines:

- Diagonal segments indicate matched pixels
- Horizontal segments indicate pixels skipped in the right image
- Vertical segments indicate pixels skipped in the left image