Stereo Vision: Technical Report

This report provides a detailed analysis of the stereo vision implementation, explaining each function, its parameters, and how it works to generate disparity maps from stereo image pairs.

Table of Contents

Introduction to Stereo Vision

Stereo vision is a technique for extracting 3D information from 2D images captured by two cameras positioned with a small lateral displacement. The fundamental concept behind stereo vision is that objects appear at different horizontal positions in the left and right images, and this displacement (called disparity) is inversely proportional to the depth of the object.

The code provided implements several approaches to compute disparity maps from stereo image pairs:

- Block matching using Sum of Absolute Differences (SAD)
- Block matching using Sum of Squared Differences (SSD)
- Dynamic programming (DP) approach for stereo matching

Image Preprocessing

Function: `convert_to_grayscale`

Parameters:

- `image`: A color image in BGR format (the default format for OpenCV)

How it works: This function converts a color image to grayscale using OpenCV's `cvtColor` function. Processing grayscale images is more efficient and often provides sufficient information for disparity calculation, as the disparity calculation primarily relies on intensity differences rather than color information.

The conversion reduces the dimensionality of the image from three channels (Blue, Green, Red) to a single channel representing intensity values, which simplifies subsequent computations.

Function: `show_image`

Parameters:

- `image`: The image to display
- `title`: Optional title for the displayed image (default is "Image")

How it works: This function displays an image using matplotlib. Since OpenCV loads images in BGR format while matplotlib expects RGB format, the function converts the color space using `cv2.cvtColor(image, cv2.COLOR_BGR2RGB)` before displaying. The function also sets a title for the image and disables axis labels for a cleaner visualization.

Block Matching Algorithms

Block matching is a technique used in stereo vision to find correspondences between the left and right images. The idea is to take a block (window) of pixels from the left image and find the most similar block in the right image. The horizontal displacement between the positions of these two blocks is the disparity.

Function: `block_matching_SAD`

Parameters:

- `left_image`: The grayscale left image
- `right_image`: The grayscale right image
- `max_disparity`: The maximum disparity value to search for
- `window_size`: The size of the window (block) used for matching

How it works: The function computes a disparity map using the Sum of Absolute Differences (SAD) method:

1. It initializes an empty disparity map with the same dimensions as the left image.
2. For each pixel in the left image (within valid bounds considering the window size):
    - It extracts a patch of size `window_size × window_size` centered at the current pixel.
    - It searches for the best matching patch in the right image by trying different disparity values from 0 to `max_disparity`.

    - For each disparity value, it computes the SAD cost, which is the sum of absolute differences between the pixel values in the left and right patches.
    - It tracks the disparity value that yields the minimum cost.
3. It scales the disparity value to the range [0, 255] for visualization and stores it in the disparity map.
4. It returns the completed disparity map.

The SAD metric is calculated as: Σ|left_pixel - right_pixel| for all pixels in the patches.

Function: `block_matching_SSD`

Parameters:

- `left_image`: The grayscale left image
- `right_image`: The grayscale right image
- `max_disparity`: The maximum disparity value to search for
- `window_size`: The size of the window (block) used for matching

How it works: This function is similar to `block_matching_SAD` but uses the Sum of Squared Differences (SSD) method instead. The key difference is in the cost calculation:

- SAD: `cost = np.sum(np.abs(left_patch - right_patch))`
- SSD: `cost = np.sum((left_patch - right_patch) ** 2)`

The SSD metric is calculated as: Σ(left_pixel-right_pixel)² for all pixels in the patches.

The SSD metric penalizes larger differences more heavily than SAD, which can lead to more accurate disparity estimates in some scenarios. However, it is also more sensitive to outliers and can be computationally more expensive due to the squaring operation.

Dynamic Programming Approach

Dynamic programming offers an alternative approach to stereo matching that can handle occlusions more effectively than simple block matching.

Function: `dp_scanline`

Parameters:

- `Il`: A 1D array representing a row from the left image
- `Ir`: A 1D array representing the corresponding row from the right image
- `sigma`: A parameter controlling the sensitivity of the matching cost (default: 2)
- `c0`: The cost of skipping a pixel (occlusion cost) (default: 1)

How it works: The `dp_scanline` function implements a dynamic programming approach to find optimal matches between pixels in a scanline (row) of the left and right images:

1. **Initialize cost matrices**:
    - `d`: A matrix of squared differences between pixel values in the left and right images, normalized by `sigma²`.
    - `D`: A cost matrix for the dynamic programming algorithm.
2. **Set initial conditions**:
    - `D[0, 0] = 0` (no cost for matching empty sequences)
    - `D[i, 0] = D[i-1, 0] + c0` (cost for skipping i pixels in the left image)
    - `D[0, j] = D[0, j-1] + c0` (cost for skipping j pixels in the right image)
3. **Fill the cost matrix** using dynamic programming:
    - For each cell `D[i, j]`, compute:
        - `term1 = D[i-1, j-1] + d[i-1, j-1]` (cost of matching pixels)
        - `term2 = D[i-1, j] + c0` (cost of skipping a pixel in the left image)
        - `term3 = D[i, j-1] + c0` (cost of skipping a pixel in the right image)
    - Take the minimum of these three terms as the cost for `D[i, j]`
4. **Backtrack** to find the optimal path:
    - Start from the bottom-right corner of the cost matrix
    - At each step, determine whether to match pixels, skip a pixel in the left image, or skip a pixel in the right image, based on which option gives the minimum cost
    - For matches, record the disparity (absolute difference between indices)
    - For occlusions (skips), set the disparity to 0
5. **Return** the disparity arrays for the left and right images

This approach elegantly handles occlusions by allowing pixels to be skipped in either image when there's no good match, instead of forcing potentially incorrect matches.

Function: `get_disparity_map_dp`

Parameters:

- `left_img`: The grayscale left image
- `right_img`: The grayscale right image
- `sigma`: Parameter for the cost calculation (default: 2)
- `c0`: The occlusion cost (default: 1)

How it works: This function applies the `dp_scanline` algorithm to each row of the left and right images to compute complete disparity maps:

1. It initializes empty disparity maps for both the left and right images

2. For each row in the images, it:
    - Applies the `dp_scanline` function to compute disparities for that row
    - Stores the resulting disparity arrays in the corresponding rows of the disparity maps
3. It returns both the left and right disparity maps

Visualization and Evaluation

Function: `plot_alignment`

Parameters:

- `Il`: A 1D array representing a row from the left image
- `Ir`: A 1D array representing the corresponding row from the right image
- `sigma`: A parameter controlling the sensitivity of the matching cost (default: 2)
- `c0`: The cost of skipping a pixel (occlusion cost) (default: 1)

How it works: This function visualizes the alignment between pixels in the left and right images as determined by the dynamic programming algorithm:

1. It computes the cost matrix using the same approach as `dp_scanline`
2. It traces the optimal path from the bottom-right to the top-left of the matrix
3. It plots the path with different colors indicating:
    - Black: Matches between pixels
    - Red: Skips in the left image (occlusions)
    - Blue: Skips in the right image (occlusions)

This visualization helps in understanding how the dynamic programming algorithm aligns the pixels and handles occlusions.

Conclusion

The code implements three different approaches to stereo matching:

1. **Block Matching with SAD**: A simple and efficient approach that computes disparities based on the sum of absolute differences between patches in the left and right images.

2. **Block Matching with SSD**: Similar to SAD but uses squared differences, which can be more sensitive to outliers but potentially more accurate in certain scenarios.

3. **Dynamic Programming**: A more sophisticated approach that can handle occlusions more effectively by allowing pixels to be skipped in either image when there's no good match.

Each approach has its advantages and limitations:

- **Block matching methods** (SAD and SSD) are simple and efficient but can struggle with occlusions and repeated patterns.
- **Dynamic programming** is more robust to occlusions but processes each scanline independently, potentially leading to horizontal streaking artifacts.

The quality of the disparity maps depends on several factors:

- The resolution and quality of the input images
- The choice of algorithm and its parameters
- The presence of textureless regions, occlusions, and repeated patterns in the scene

The implementation shows how these different approaches can be used to generate disparity maps from stereo image pairs, which are the foundation for 3D reconstruction and depth estimation in computer vision applications.