Stereo Vision

1 Disparity Maps

In this task, you will implement a simple algorithm to compute disparity map from RGB images. The program runs the code on two images from a Middelbury stereo dataset. The algorithm to implement is a simple version of computing disparity maps and then applying post processing filtering steps to generate a smooth disparity map. The algorithm itself is implemented in the function `bin/run_ex4.py` and consists of the following steps:

Algorithm

1. Loading a pair of images; `src` taken from a camera and `dst` taken by translating the camera in horizontal direction only.
2. Computing the maximum horizontal `translation` (measured in pixels) of the camera for this pair of images.
3. Constructing a stack of all reasonable `disparity` hypotheses. (Understanding: Each disparity hypothesis is associated with a depth plane. For example: A disparity of zero assumes that the points are at infinity)
4. We apply a 3D Gaussian filter on the stack of disparities from the above step to enforce a local coherence between location (x-axis) and disparity hypotheses (z-axis)
5. For each pixel, we choose the disparity hypothesis that fits the `dst` image to `src` image.
6. We then apply a median filter to enhance local consensus (similar to removing salt-pepper noise) from the disparity map.
7. Check consistency: Given the `src` image and `disparity map`, we should be able to estimate `dst` image.

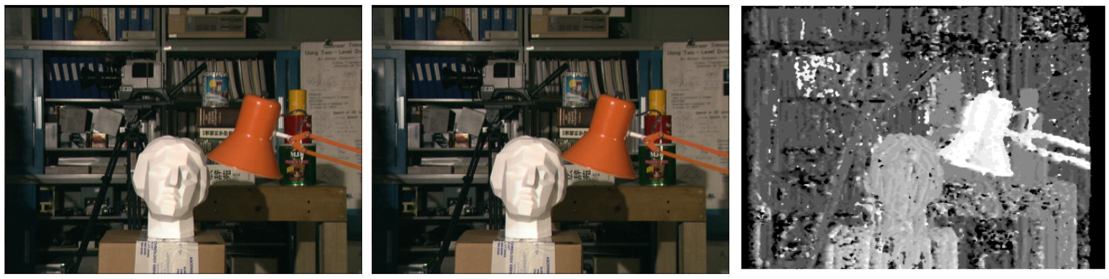


Figure 1: The two input images of the *Tsukuba* dataset and the disparity map computed in this task.

1.1 Estimating maximum translation

The first thing to do in this exercise is to compute the furthest translation of the pixel from `src` and `dst`. This will be later used as an estimate to how much neighborhood should be checked for finding the optimized disparity map. In order to do this, we will use the `extract_features` and `filter_and_align_descriptors` to generate the well aligned points. Using these aligned points, we can compute the furthest translation between them. A disparity of zero can be associated with pixels at infinity and all other disparities should be lying between zero and the furthest translation. If the `src image` is to the right of the `dst image`, then all the translations should be positive and otherwise negative. Note: The resulting translation is rounded in such a way that it's absolute value will be the greatest.
Implement the function `get_max_translation` in `ex4/functions.py`. Make sure that the test case succeeds before continuing with the next task.

1.2 Render disparity hypothesis

Given a pair of images (or their shifted versions), we need to render the disparity between them. Essentially we need to compute the agreement between any given `src` image and the `dst` image. In order to compute this agreement, we use the euclidean norm of the RGB values between the shifted `src` image and the `dst` image. The resulting maps should be an indication of how much does a disparity hypothesis satisfy each pixel.
Implement the function `render_disparity_hypothesis` in `ex4/functions.py`.

1.3 Find the best/minimum disparity map

There are various ways in which one can estimate the disparity map given a pair of images. In steps 3-6 from the Intro section, we describe a simple way of implementing it. Implement `disparity_map` in `ex4/functions.py` to realize this part of the exercise.

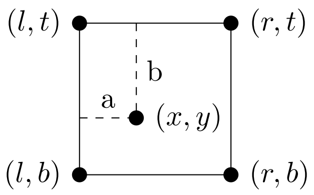
Additional Resources (Extra but No Credits) If you are interested to explore this further and want to generate better and enhanced disparity maps using highly engineered method, here are a few more resources to read. Here's the `patch match` paper and the resources related to the same. You can also read them for the general understanding of disparity maps. Note that, disparity maps are different from depth maps.

- Patch Match [Wiki], Patch Match [Paper], Patch Match [Video]
- Patch Match Stereo [Paper], Patch Match Stereo [Video]

Note: If you choose to implement an alternative algorithm (such as above), and diverge from the given exercise structure, the given test cases might be less helpful.

1.4 Bilinear interpolation

In this exercise (and for many other computer vision applications) we will need to access an image at positions between two pixels. The value is computed from the surrounding pixels using bilinear interpolation:



$$\begin{aligned} I(x,y) =&(1-a)(1-b)\cdot I(l,t)\\ &a(1-b)\cdot I(r,t)\\ &(1-a)b\cdot I(l,b)\\ &ab\cdot I(r,b) \end{aligned}$$

Figure 2: Bilinear interpolation of image I at position (x,y)

Implement the function `bilinear_sample_grid` in `ex4/functions.py` using the following steps:

- Step 1: Estimate the left, top, right, bottom integer parts (l, r, t, b) and the corresponding coefficients (a, b, 1-a, 1-b) of each pixel
- Step 2: Take care of out of image coordinates
- Step 3: Produce a weighted sum of each rounded corner of the pixel
- Step 4: Accumulate and return all the weighted four corners

This function applies a sampling field of continuous (float) values on a `src` image to produce a `dst` image with the same size as the sampling field. The sampling field consists of a pair of x-axis and y-axis coordinates (each one with the same shape as `dst` image).