Save Luna
Aerial Robotics Kharagpur - Perception Task 1

***Abstract*— The project features Luna, a robot which needs help travelling in her environment. Given two images, left.png and right.png, we are expected to generate a depth map in which closer objects appear red and farther objects appear blue. Using calibrated stereo measurements, we obtain depth values and then plot them.**

I. INTRODUCTION AND PROBLEM STATEMENT

Luna does not have the ability to interpret its surroundings. Unless she is able to recognize relative distances between two points, she is not able to move around in the environment, and autonomous navigation is impossible. Given two stereo images (left and right views), the objective is to generate a depth map that highlights depth variations across the scene.

We have to measure the shift of corresponding elements in both images. The depth estimation algorithm should effectively compute these disparities and translate them into a heatmap representation, where closer objects are highlighted in red and distant objects appear in blue. The implementation must be done from scratch.

Fig. 1: left.png

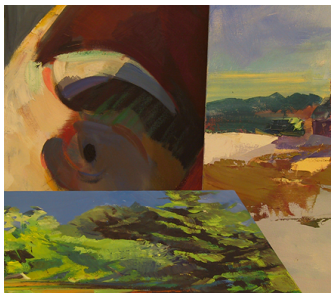


Fig. 2: right.png

II. RELATED WORK

The problem that we are dealing with is calibrated stereo, which is fairly easy, but a more general problem is uncalibrated stereo, where the intrinsic and extrinsic parameters of the cameras are unknown. It requires fundamental matrix estimation and epipolar geometry constraints to align the images for depth computation. Rectification techniques are often required before depth estimation.

III. FINAL APPROACH

```
import cv2
import matplotlib.pyplot as plt
import numpy as np
```

I defined a function to give all the details about the image

```
def image_details(url):
    img = cv2.imread(url, cv2.IMREAD_GRAYSCALE)
    print(img)
    print(f"Image Size : {img.size}")
    print(f"Image Shape : {img.shape}")
    plt.imshow(img, cmap="gray")
    plt.show()
    return img
```

We then perform template matching to determine each pixel's disparity values. We define a window with length WINDOW_LENGTH and height WINDOW_WIDTH. We place this at the top left corner in the left image, then we slide this through the right image in the same row, as we know that there is no change in the height of the cameras as they are placed horizontally parallel to the ground. The location where the error is minimal is where the pixel of interest in the left image has traversed (though this is not entirely correct, it works fairly well). Also, we do not traverse the complete row in the right image; we have defined a boundary called max_disparity, so we search only between the (col in a left image - max_disparity) and (col in the left image + max_disparity) in the right image.

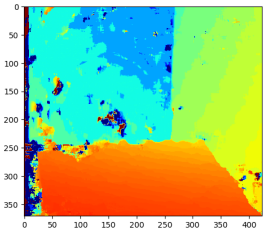
```
def template_matching(left_image, right_image,
    window_length, window_width, max_disparity=32):
    height, width = left_image.shape[0],left_image.
        shape[1]

    disparity = np.zeros((height - 2 * (
        window_width // 2), width - 2 * (
        window_length // 2)))

    half_window_length = window_length // 2
    half_window_width = window_width // 2

    for i in range(half_window_width, height -
        half_window_width):
        for j in range(half_window_length, width -
            half_window_length):
            left_patch = left_image[i -
                half_window_width:i +
                half_window_width + 1, j -
                half_window_length:j +
                half_window_length + 1]
```

```
            min_error = np.inf
            best_match = j

            for d in range(min(max_disparity, j -
                half_window_length)):
                col = j - d

                if col - half_window_length < 0 or
                    col + half_window_length + 1 >
                    width:
                    continue

                right_patch = right_image[i -
                    half_window_width:i +
                    half_window_width + 1, col -
                    half_window_length:col +
                    half_window_length + 1]

                error = np.sum((left_patch -
                    right_patch) ** 2)

                if error < min_error:
                    min_error = error
                    best_match = col

            disparity[i - half_window_width, j -
                half_window_length] = abs(j -
                best_match)

    return disparity
```

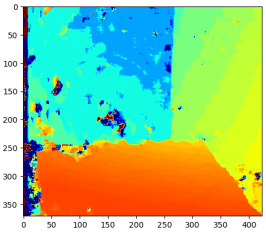
Now we decide upon the optimal window size and max_disparity value for well generation of heatmap.

```
ans = template_matching(left_image,right_image,
    WINDOW_LENGTH,WINDOW_WIDTH)

original_map  = plt.cm.get_cmap('jet')
map = original_map.reversed()
plt.imshow(ans, cmap=map)
plt.show()

depth = 1/ans
depth[ans == 0] = 0
k = np.max(depth.ravel())
factor = 5
depth[depth > k/factor] = k/factor
plt.imshow(depth, cmap=map)
plt.show()
```

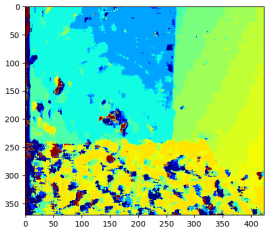
First, we change the max_disparity, keeping the window size and factor fixed.



(a) max_disparity = 64



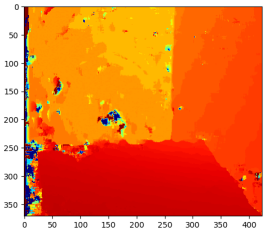
(b) max_disparity = 32



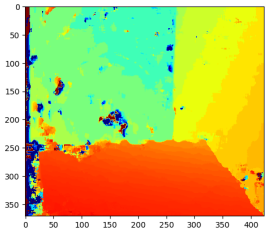
(c) max_disparity = 16

So, a max_disparity of 32 or 64 is fine. So let us keep it 64 only; if there is a requirement for better speed, we will reduce it to 32 as it will take almost half the time it took for 64.

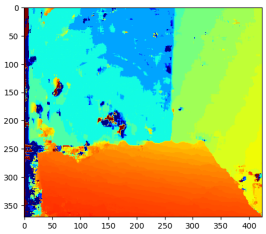
Now let us change the factor, keeping window size and max_disparity fixed. This factor is used to clamp extremely large values to a max value so that there is a fair distribution of depth values and there are no outliers, and eventually, we get to see a clean, smooth heatmap.



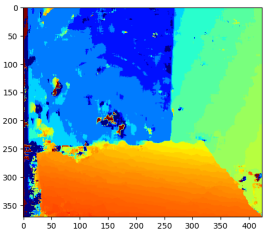
(a) factor = 2



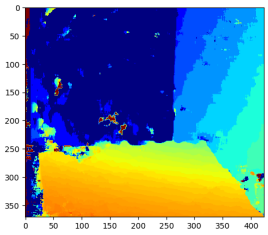
(b) factor = 4



(c) factor = 5



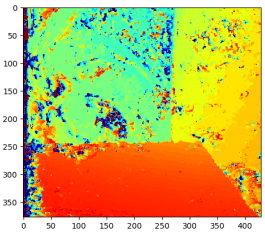
(d) factor = 6



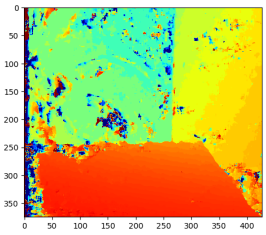
(e) factor = 8

A factor of 4 or 5 is acceptable; let us keep it to 4 only,

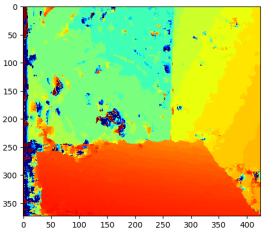
as it looks better.
Now, let us change the window size; let us keep the window square for simplicity.



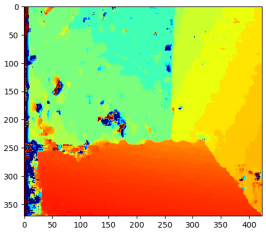
(a) size = 5



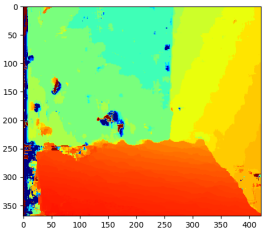
(b) size = 7



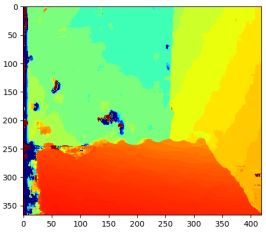
(c) size = 9



(d) size = 11



(e) size = 13



(f) size = 15

As there is not much improvement in the reduction of noise after window size = 13, and we also have started to lose the boundary from window size = 13, let us take it to be 13.
Observing all the results let us finally present out heatmap with factor=4, max_disparity = 64 and window size = 13

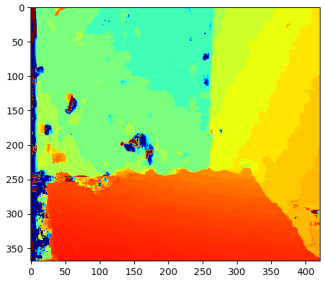
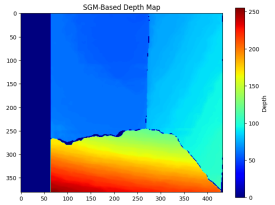


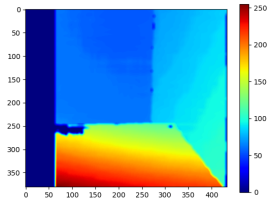
Fig. 6: Window Size = 13, max_disparity = 64, factor = 4

IV. Results and Observation

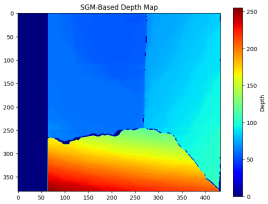
Depth Maps from various other algorithms are attached below.



(a) SGM



(b) SBGM



(c) Graph Cut

These algorithms tend to perform better than our basic stereo calibration model both in terms of noise handling and edge preservation.

V. Future Work

There lies a huge scope for improvement in my code in many regards.
1. Noise: As we saw, the other models handle noise very beautifully, but our algorithm still has a lot of noise even after tweaking the parameters.
2. Edge Preservation: Our algorithm is not able to produce a distinction between surfaces; meanwhile, other models like SBGM have clear edges visible.
3. One point to note is the formation of gradients in colour bands. I am not sure if this is because the actual surface is tilted or if there are some limitations in the models.
4. We calculate the disparity by computing how much a pixel traverses, but this is not valid because we assume that a 2D area moves as a whole, but if that area originally contained elements with varying depth values, it will not move as a whole. We could devise some methods to tackle this.
These are some of the points which require further study.

Conclusion

In this project, we developed a depth estimation system to help Luna, a robot with stereo cameras, perceive its environment and avoid obstacles. Using stereo vision techniques, we generated a depth map highlighting closer objects in red and farther ones in blue. Various methods were implemented and compared with each other.

The final depth map gives Luna a structured understanding of her surroundings, enabling safe navigation. While effective, challenges such as occlusions, textureless regions, and computational efficiency remain areas for improvement.

This project demonstrates the importance of vision-based navigation for autonomous systems.

References

[1] **First Principles of Computer Vision** , https://www.youtube.com/@firstprinciplesofcomputerv3258

[2] **OpenCV**, https://docs.opencv.org/4.x/dd/d53/tutorial_py_depthmap.html
[3] https://dibyendu-biswas.medium.com/stereo-camera-calibration-and-depth-estimation-from-stereo-images-29d87bc702f3
[4] **Numpy Documentation**, https://numpy.org/doc/
[5] **ChatGPT**, https://chatgpt.com/