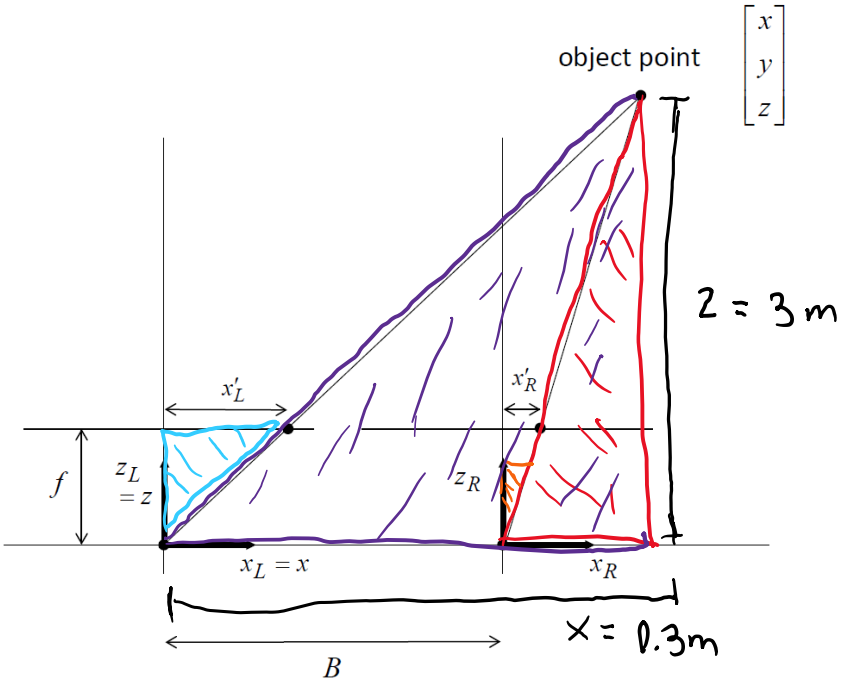
ECE 4554 / 5554: Computer Vision: Homework 4

Fall 2020

Instructions

- The assignment is due at Canvas before midnight on Dec. 6. Late submissions are allowed at the cost of 1 token per 24-hour period. A submission received only a minute after midnight will cost an additional token.
- Problems 1 through 4 are worth 10 points each. Notice that one of the problems is required for 5554 students, but is optional (extra-credit) for 4554 students. Problem 5 is worth 20 points.
- Prepare an answer sheet that contains all of your written answers in a single file named *LastName_FirstName*_HW4.pdf, using your own name. Handwritten solutions are permitted, but they must be easily legible to the grader. Do not place this pdf file inside the zip file that is described in the next bullet.
- Place all of your Python-related files (*.py, *.png, etc.) into a single zip file named *LastName_FirstName*_HW4.zip. Upload 2 files to Canvas: this zip file and the pdf file that is described above.
- For the Python implementation problems, try to ensure that the grader can run your code "out of the box". For example, do not encode absolute path names for input files; provide relative path names, assuming that the input files reside in the same directory as the source files.
- If any plots are required, include them in your answer sheet (the pdf file). Also, your Python code must display the plots when executed.
- After you have submitted to Canvas, it is your responsibility to download the files that you submitted and verify that they are correct and complete. *The files that you submit to Canvas are the files that will be graded.*

Problem 1. Consider the simple stereo imaging geometry that was introduced in class, as shown below. Both optical axes are parallel, and both cameras have the same focal length. In this view from above, the overall coordinate reference frame (x, y, z) is centered at the left camera.



a) Assume that all distances are given in units of meters. Let $f = 0.05$ and $B = 0.1$. For object point $(x, y, z) = (0.3, 0.0, 3.0)$, solve for horizontal locations x'_L and x'_R in the two images. (You may assume that the vertical locations y'_L and y'_R are 0 for this problem.)

b) Using your answer from (a), compute the value for horizontal disparity in the two images for the given object point.

c) Using your numerical answers from above, confirm the relation $z = Bf/d$ for this imaging geometry, where d is the horizontal disparity.

a.)

Similar triangles $\frac{x'_L}{f} = \frac{X}{Z}$ $\qquad \frac{x'_R}{f} = \frac{X-B}{Z}$

$x'_L = \left(\frac{X}{Z}\right) f$ $\qquad x'_R = \left(\frac{X-B}{Z}\right) f$

$x'_L = \left(\frac{0.3}{3}\right)(0.05)$ $\qquad x'_R = \left(\frac{0.3-0.1}{3}\right)(0.05)$

$x'_L = 0.005\text{m}$ $\qquad x'_R = 0.00\overline{3}\text{m}$

b.)

$$d \cong x'_L - x'_R = (0.005\ m - 0.003\overline{3}\ m) \approx 0.00167$$

c.)

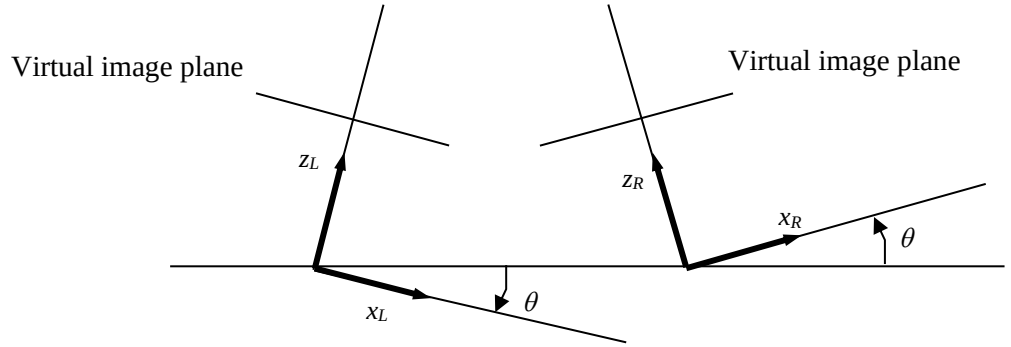
$$Z = \frac{Bf}{d} = \frac{(0.1)(0.05)}{0.00167} = 2.99\ m$$

Problem 2. Consider again the stereo imaging geometry from the previous problem. Now assume that you have implemented a stereo matching process that has produced an incorrect horizontal disparity value, $d + \Delta d$, where d is the correct disparity and Δd is an unknown error. Write an equation for the error in computed depth *z* that will result from this error in disparity. (Do not plug in numerical values for *B, F, z,* etc.)

$$Z = \frac{Bf}{(d + \Delta d)} \Rightarrow 2d + 2\Delta d = Bf \qquad 2\Delta d = Bf - 2d$$

$$\Delta d = \frac{Bf - 2d}{2}$$

Problem 3. Now assume that the 2 cameras from Problem 1 are allowed to rotate about their vertical axes, y_L and y_R . Suppose that both cameras rotate inward by the same angle θ , as shown in the diagram below. (When $\theta = 0$, we have same arrangement as for Problem 1. When the cameras rotate, the baseline *B* does not change. Assume that both cameras have focal length *f*.)



For this camera arrangement, solve for the locations of the 2 epipoles as function of θ. (Hint: it may help to consider a larger baseline and larger rotation angle than shown in the diagram.)

$$F = \begin{bmatrix} 0 & -B\sin\theta & 0 \\ 0 & 0 & 0 \\ 0 & B\cos\theta & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \quad \\ \quad \\ \quad \end{bmatrix}$$

$$-\sin(x)\cos(2x) \quad \cos(x)\sin(2x)$$

$$\sin(x)\sin(2x) - \cos(x)\cos(2x)$$

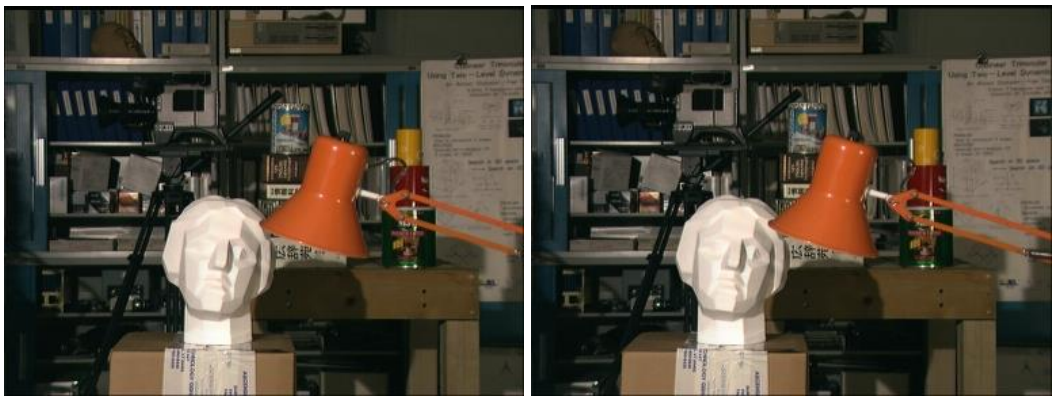
Problem 4. (For 5554 students, this problem is required. For 4554 students, this problem will count as extra credit. This topic is discussed in Section 7.2 of Szeliski's first edition, and in Section 11.3 of the second edition.)

a) Find the *essential matrix* that relates these 2 cameras, as a function of θ.

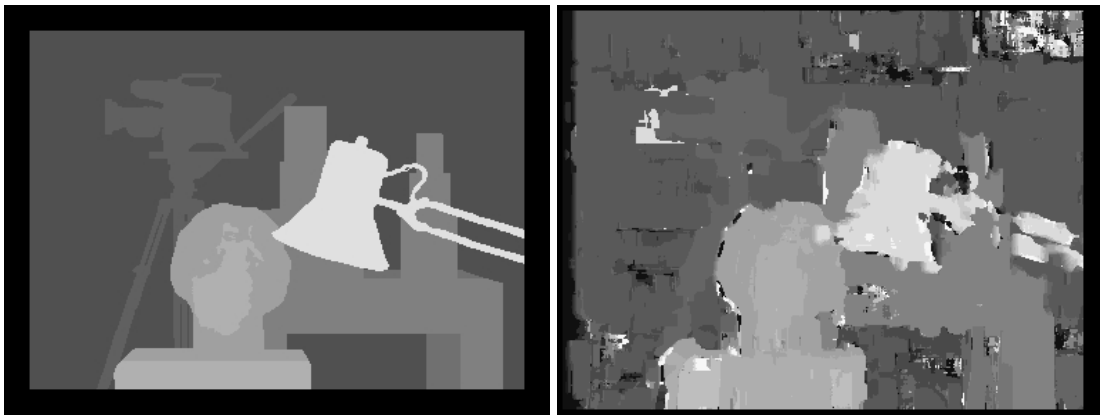
b) Use your answer to part (a) to solve for the locations of the 2 epipoles, and confirm that your answer agrees with your solution to Problem 3.

c) For any given point (x'_L, y'_L) in the left image, write an equation for the corresponding epipolar line in the right image.

Problem 5. (20 points.) Write a Python/OpenCV program that implements a stereo matching algorithm for rectified stereo pairs. An example image pair is shown below:



The output of your program should be a disparity map that is referenced to the image frame of the left camera. The ideal output is shown at the left below. (These stereo images and the ground-truth output were obtained from Middlebury College, https://vision.middlebury.edu/stereo/data/.) Because of the difficulty of stereo matching, you should not expect perfect results from your program. For example, a reasonable computed output is shown at the right below.



Your code should perform window-based matching. For each pixel in the first (reference) image (except near the image boundary), pick a window around that pixel and then search the corresponding row in the second image for the location having the best match.

You should experiment with the following settings and parameters. In your solution sheet, write a brief description to describe the following and to present your results:

- **Window size:** Show disparity maps for 2 or more window sizes and describe the trade-offs between the different window sizes. Which case gave you the best results? How did the computation time depend on window size?

- **Matching criterion:** Try at least 2 different matching criteria, including sum of squared differences (SSD) and any other criteria that you choose. Describe what differences you observed in the results, if any.

- **Disparity limits:** Most stereo matching algorithms do not search an entire epipolar line in order to find one corresponding point, but instead search within a range of possible disparity values. Examine the given stereo pair to determine the minimum and maximum disparity values that makes sense. In your write-up, describe the settings that you decided to use in your program.

- **Other constraints:** Most stereo matching algorithms use a variety of constraints for reduce the amount of search. Several of these constraints have been discussed in lectures. In your write-up, describe which of those constraints (if any) you used in your solution to this problem.

- **Not required, but worth considering:** Suppose feature descriptors such as SIFT or ORB were used in developing your script. What are some advantages and disadvantages?

Additional requirements: Use the file name `P5.py` for your script. Paste this script into your answer sheet, and also submit the source file. Assume that the input files are named `image_left.png` and `image_right.png`, and hard-code these names into your program. The output of your program should be an image file named `disp_map.png`, and it should be the same size as one of the input images. Submit a copy of `disp_map.png` that contains the best output from your program. In your write-up, show this best output along with a few other representative disparity maps that you computed. Briefly discuss the strengths and shortcomings of your algorithm. Where do the estimated disparity maps look good, and where do they look bad? What additional work could be done to produce better results?

For this problem, do not use any OpenCV/NumPy/etc. functions except for the following: for loading/saving image files, and for basic math and matrix operations. (If you decide to use keypoint descriptors such as ORB, you <u>are</u> allowed to use canned library functions for detection/matching.)

Window Size:

Figure 1: Window Size of 3

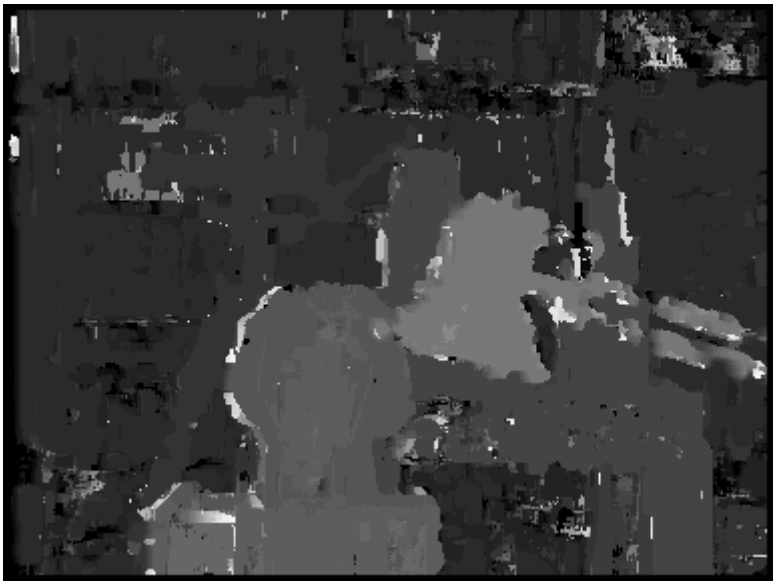


Figure 2: Window Size of 6

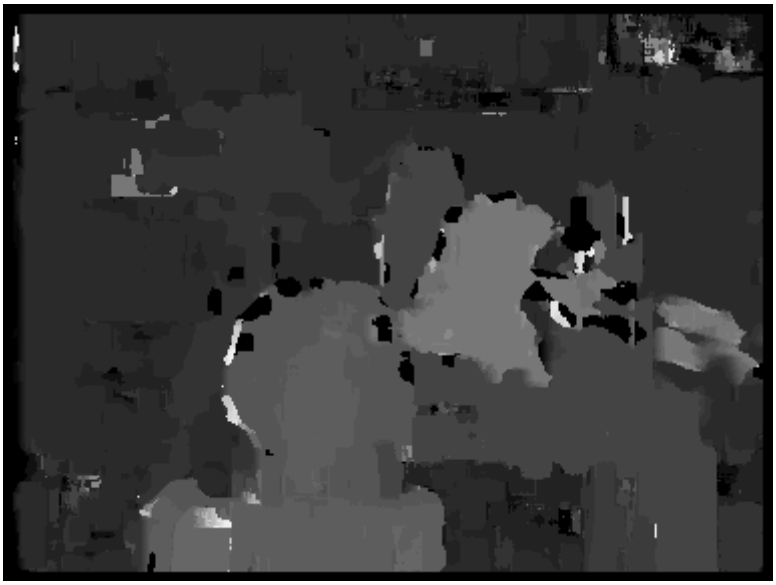


Figure 3: Window Size of 10

In the images above, I am utilizing the SSD matching criterion while each image represents a different window size. Based on the results above, it seems that around a window size of 6 produces the best results for detail, smoothness of the disparity map, and eliminating noise. It is also important to note that the bigger your window size, the bigger the computation time.

Matching Criterion:

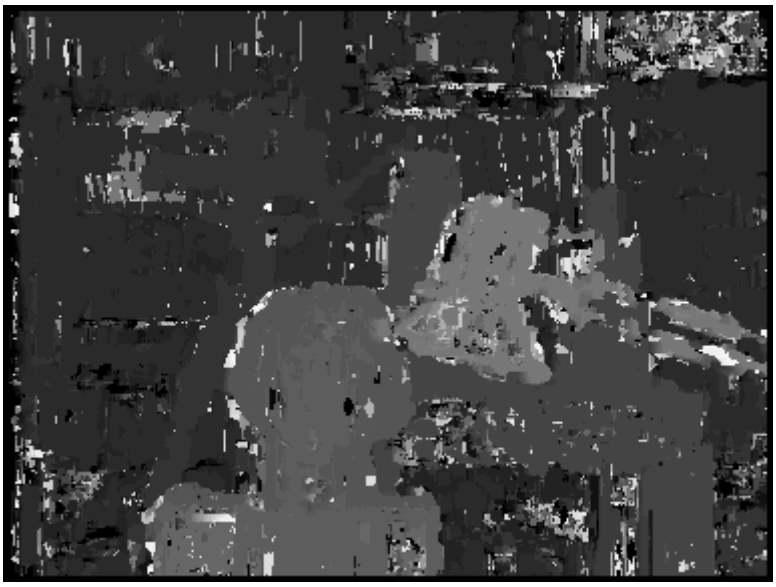


Figure 4: Using the NCC Matching Criterion

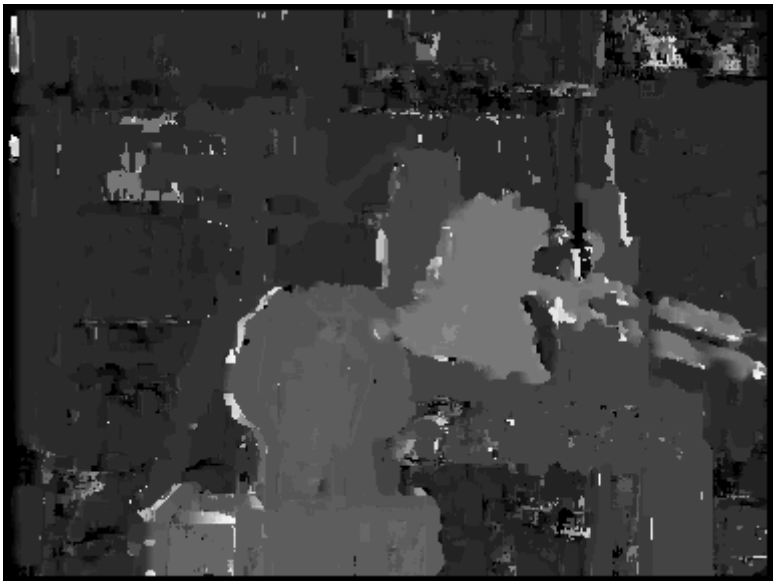


Figure 5: Using the SSD Matching Criterion

In the above images show the results of running the stereo matching using SSD and NCC while maintaining the same parameters (a window size of 6 and same disparity limit). As can be seen in the images, the stereo matching result using the NCC matching criterion contains more details, but also produces more noise in the result. In the stereo matching algorithm that uses SSD as its matching criterion, the resulting image has a smoother looking disparity map but contains less details compared to its NCC counterpart. It also important to note that the SSD matching criterion allowed my code to run faster while the NCC matching criterion greatly increased computation time.

Disparity Limits:

In my algorithm, I only search within 30 pixels search range and this seems to produce decent results.

Other Constraints:

I do not believe I have utilized any of the constraints mentioned in class. If I did choose to do so, then I would probably see better results from the matching criterions SSD and NCC.

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
from PIL import Image

def stereo_match_SSD(left_img, right_img, kernel, max_offset):
    # Load in both images, assumed to be RGBA 8bit per channel images, convert
 to gray scale
    left_img = Image.open(left_img).convert('L')
    left = np.asarray(left_img)
    right_img = Image.open(right_img).convert('L')
    right = np.asarray(right_img)
    w, h = left_img.size  # assume that both images are same size

    # Depth (or disparity) map
```

```
    depth = np.zeros((w, h), np.uint8)
    depth.shape = h, w

    kernel_half = int(kernel / 2)
    offset_adjust = 255 / max_offset  # this is used to map depth map output t
o 0-255 range

    for y in range(kernel_half, h - kernel_half):
        print(".", end="", flush=True)  # let the me know that something is ha
ppening

        for x in range(kernel_half, w - kernel_half):
            best_offset = 0
            prev_ssd = 65534

            for offset in range(max_offset):
                ssd = 0
                ssd_temp = 0

                # v and u are the x,y of our local window search, used to ensu
re a good
                # match- going by the squared differences of two pixels alone
is insufficient,
                # we want to go by the squared differences of the neighbouring
 pixels too
                for v in range(-kernel_half, kernel_half):
                    for u in range(-kernel_half, kernel_half):
                        # iteratively sum the sum of squared differences value
 for this block
                        # left[] and right[] are arrays of uint8, so convertin
g them to int saves
                        # potential overflow, and executes a lot faster
                        ssd_temp = int(left[y+v, x+u]) - int(right[y+v, (x+u)
- offset])
                        ssd += ssd_temp * ssd_temp

                # if this value is smaller than the previous ssd at this block
                # then it's theoretically a closer match. Store this value aga
inst
                # this block..
                if ssd < prev_ssd:
                    prev_ssd = ssd
                    best_offset = offset

            # set depth output for this x,y location to the best match
            depth[y, x] = best_offset * offset_adjust

    # Convert to PIL and save it
    Image.fromarray(depth).save('disp_map_SSD_Window_10.png')
    cv2.imshow("The Disparity Map with SSD", depth)
```

```
def stereo_match_NCC(left_img, right_img, kernel, max_offset):
    # Load in both images, assumed to be RGBA 8bit per channel images, convert
 to gray scale
    left_img = Image.open(left_img).convert('L')
    left = np.asarray(left_img)
    right_img = Image.open(right_img).convert('L')
    right = np.asarray(right_img)
    w, h = left_img.size  # assume that both images are same size

    # Depth (or disparity) map
    depth = np.zeros((w, h), np.uint8)
    depth.shape = h, w

    kernel_half = int(kernel / 2)
    offset_adjust = 255 / max_offset  # this is used to map depth map output t
o 0-255 range

    for y in range(kernel_half, h - kernel_half):
        print(".", end="", flush=True)  # let the me know that something is ha
ppening

        for x in range(kernel_half, w - kernel_half):
            best_offset = 0
            prev_ncc = 65534

            for offset in range(max_offset):
                ncc = 0
                l_mean = 0
                r_mean = 0
                n = 0

                # v and u are the x,y of our local window search, used to ensu
re a good
                # match- going by the normalized cross correlation of two pixe
ls alone is insufficient,
                # we want to go by the normalized cross correlation of the nei
ghbouring pixels too
                for v in range(-kernel_half, kernel_half):
                    for u in range(-kernel_half, kernel_half):
                        #calculate the cumulative sum
                        l_mean += left[y+v, x+u]
                        r_mean += right[y+v, (x+u) - offset]
                        n += 1

                l_mean = l_mean/n
                r_mean = r_mean/n

                l_r = 0
                l_var = 0
```

```
                r_var = 0

                for v in range(-kernel_half, kernel_half):
                    for u in range(-kernel_half, kernel_half):

                        l = int(left[y+v, x+u]) - l_mean
                        r = int(right[y+v, (x+u) - offset]) - r_mean

                        l_r += l*r
                        l_var += l**2
                        r_var += r**2

                        ncc = -l_r/np.sqrt(l_var*r_var)


                # if this value is smaller than the previous ncc at this block
                # then it's theoretically a closer match. Store this value aga
inst
                # this block..
                if ncc < prev_ncc:
                    prev_ncc = ncc
                    best_offset = offset

            # set depth output for this x,y location to the best match
            depth[y, x] = best_offset * offset_adjust

    # Convert to PIL and save it
    Image.fromarray(depth).save('disp_map_NCC.png')
    cv2.imshow("The Disparity Map with NCC", depth)


if __name__ == '__main__':
    stereo_match_SSD("image_left.png", "image_right.png", 6, 30)  # 6x6 local
search kernel, 30 pixel search range

    #Uncomment bottom line if you want to run stereo matching with NCC instead
    #stereo_match_NCC("image_left.png", "image_right.png", 6, 30)  # 6x6 local
 search kernel, 30 pixel search range

    cv2.waitKey(0)
    cv2.destroyAllWindows()
```