University of Girona - MIRS

Lab1 – Stereo

3D Perception and Sensor Fusion

Author: Ricard Campos
Sensor: optical stereo (passive)
Programming language: Python

1 INTRODUCTION

In this session, we will explore the stereo reconstruction pipeline. As you may know, a stereo camera is just the fusion of two regular optical cameras. By knowing the transformation between both cameras, we can benefit of a set of restrictions that allow us to reconstruct and sense 3D measurements.

You may recall the epipolar geometry concepts from other courses (see Figure 1 for a quick summary). This first session serves to recall what can be achieved with this type of passive sensor.

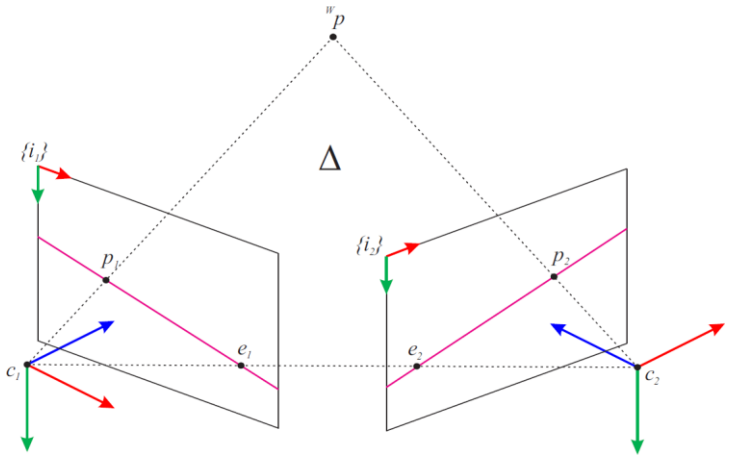


Figure 1. Epipolar geometry. The plane joining the 3D point ^{W}p with the center of the first camera c_1 (resp. p_1) and the center of the second camera c_2 (resp. p_2) define the epipolar constraints. Basically, this means that the match for p_1 can only be found along the epipolar line (pink segment in the image) in image 2 and viceversa. Also, if we know the pose of the cameras c_1 and c_2, and the projections of the point p_1 and p_2, we can *reconstruct* ^{W}p using triangulation.

In this hands-on laboratory session, we will perform all the steps required for sensing 3D data from a stereo camera, namely:

- Monocular calibration, to recover the intrinsics of each individual camera in the pair.
- Stereo calibration, to estimate the extrinsics of the camera, that is, the 3D transformation between cameras.
- Match points between images and reconstruct them based on the previous calibration. In this case, we will use two approaches:
    - Sparse reconstruction.
    - Dense reconstruction.

This first lab session is quite guided (don't get used to it), and you will find sample data to test all the pipeline and exercises proposed in this document.

However, in the *3D Perception and Sensor Fusion* laboratory sessions we will encourage you to work on real data **captured by you**. For this reason, you will find a ZED2 stereo camera in the lab for you to use whenever you want.

The idea is for you to follow this document and implement/run all the steps first with the sample data we provide. Then you are supposed to **capture some data yourself**, both for the calibration and for reconstruction, and re-execute the code you developed on this new data. You will have two days for this assignment. Ideally, you should work on the lab the first day, continue working at home, and leave the second session just for last-minute doubts and for capturing data and check that your pipeline works (*NOTE: this paragraph applies to all the labs in this course!*).

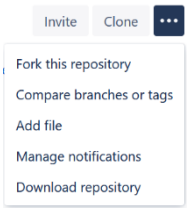
2 PREPARING THE ENVIRONMENT

You will find the code and sample data to work in this assignment in the Moodle page of the course. Please download the sample data and go to the link provided to gather the code.

As you can see, the code is hosted in the following GIT repository:

https://bitbucket.org/ricardcd/3dpsf_lab_1_stereo/

Please download the code either by doing a "git clone" (if you are familiar to GIT) or directly downloading it from Bitbucket as a zip file. You will find the download option at the upper-right corner of the repository page, by clicking at the button next to "clone":



This project uses **Python 3.7**. We recommend working in a virtual environment. You can create one with the following command:

```
python -m venv venv
```

This will create a virtual environment on the "venv" folder. Then, to activate it in Windows (PowerShell):

```
.\venv\Scripts\Activate.ps1
```

Alternatively, to activate it in linux:

```
source ./venv/bin/activate
```

WARNING: if working on PowerShell on the PCs of the lab, the running of scripts may be disabled by default. To enable it, execute the PowerShell as administrator, and run the following command:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

If you managed to run the "activate" script, you should be now within the virtual environment. The packages you install in this command line will just be available in this environment. You can now install all the requirements using pip:

```
pip install -r requirements.txt
```

If you worked with Python in the past, you will notice that what we provide is not a real Python package. In fact, what you will find in the code folder are a set of **Python scripts** that **are to be executed from within the same directory they are located**, for instance:

```
python <script_name.py>
```

The main libraries used in these scripts are OpenCV (most of the processing), Numpy (OpenCV images and any manipulation on them uses this library, when working with Python) and Open3D (for visualizing and saving point sets only in this session, but we will use it more intensively in the next labs). You can refer to their documentation whenever you have a doubt of the meaning/functionality of one of the functions used in the provided scripts.

3 MONOCULAR CAMERA CALIBRATION

In this first step we will perform the individual calibration of each camera. We aim at recovering the intrinsic parameters of the camera, which are those related to the pinhole geometry (i.e., the relations allowing the projection of 3D points into the image plane), as well as the distortion coefficients. By observing an object of known dimensions from different perspectives, we will be able to infer the camera matrix and the distortion parameters of each camera. For this purpose, we will use the "monocular_calibration.py" script. Please open the file with an editor and inspect its contents.

This Python script uses OpenCV functions to calibrate the camera automatically. We will now overview its most important parts. While we will only do this with this first script, **you should open and inspect carefully every script provided in this session before executing it**. This script starts with the input parameter handling:

```
def monocular_calib():
    # Handle input parameters
    parser = argparse.ArgumentParser(description="Monocular camera calibration")
    parser.add_argument('--row_corners', dest='pattern_row_corners', action='store', type=int,
required=True,
                        help='Number of internal corners in a row of the chessboard pattern')
    parser.add_argument('--col_corners', dest='pattern_col_corners', action='store', type=int,
required=True,
                        help='Number of internal corners in a column of the chessboard pattern')
    parser.add_argument('--squares_size', dest='square_size', action='store', type=float,
required=True,
                        help='Side length of the squares of the chessboard pattern')
    parser.add_argument('--images_dir', '-i', dest='imgs_dir', action='store', type=str,
default="./images", required=True,
                        help='Folder containing the input images')
    parser.add_argument('--out_calib', '-o', dest='out_calib_file', action='store', type=str,
required=True,
                        help='Output calibration file containing the intrinsic parameters of the
camera')
    parser.add_argument('--debug_dir', dest='debug_dir', action='store', type=str, default="",
                        help='If not empty, it will generate the specified directory and debug
data will be stored there')
    param = parser.parse_args()
```

We use the **argparse** module to handle the input arguments. You can check the functionality of the module at https://docs.python.org/2.7/library/argparse.html. Basically, as you can observe in the code, it provides an easy way of listing and getting the values of the input parameters. After defining each parameter *name*, *type*, *default value* and *help* string using the *add_argument()* function, we use the *parse_args()* in order to get an object (*param*) with each input parameter as an attribute. In this way, we can access them using *param.<parameter_name>* in the code. Also, when running the python script in the command line, we are able to get the required parameters and their meaning using **"-h" or "--help" parameters**. Note that this method for gathering the user parameters from the command line will be used in all the labs in this course.

The real processing starts after this part. In order to assess the camera parameters, we need to observe an object of known dimensions, and see how the object projects on the image. By observing the object in different positions and orientations on the image, we can infer all the camera parameters. The object, in this case, is a chessboard-like pattern. We will detect the inner corners defined by the black/white squares in the image and compare them to the known reference, that is, the 3D points of the real object. The following piece of code creates the known reference for the pattern given the number of squares, in width and height, and the square size.

```
    # Create the corner points of the chessboard of the calibration pattern
    pattern_size = (param.pattern_row_corners, param.pattern_col_corners)
```

```
    pattern_points = np.zeros((np.prod(pattern_size), 3), np.float32)
    pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2)
    pattern_points *= param.square_size
```

Then, we list all the images within the specified folder and file extension (note that the "list_image_files()" function is defined at the top of the file):

```
    # List the files in the specified input folder
    files = list_image_files(param.imgs_dir)
    files.sort() # Just to list the files in order
    if not files:
        print('[ERROR] No images found in the specified folder')
        return
```

Next, we basically run over each image in order to detect the chessboard:

```
    # Run over all the images in the folder and try to detect the chessboard corners on each
image
    obj_points = []
    img_points = []
    h, w = 0, 0
    images_used = []
    for file in files:
        print('- Detecting corners on image %s... ' % file, end='')
```

For each file, we load the image:

```
        # Load the image in gray scale
        img = cv2.imread(file, cv2.IMREAD_GRAYSCALE)
        if img is None:
            print("[WARNING] Unable to load image %s!", file)
            continue
```

And then we try to detect the chessboard at pixel level:

```
        # Try to find the chessboard on the current image
        h, w = img.shape[:2]
        found, corners = cv2.findChessboardCorners(img, pattern_size)
        if found:
            print('Chessboard detected')
```

In case it is detected, the following lines refine the initial corner detection at subpixel level:

```
            # If found, refine the corners to subpixel accuracy
            term = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.1)
            cv2.cornerSubPix(img, corners, (5, 5), (-1, -1), term)
```

We then add the refined detected points and its reference in two separate lists:

```
        # Add the detected corners and a set of reference 3D points to the lists
        img_points.append(corners.reshape(-1, 2))
        obj_points.append(pattern_points)
```

Finally, after collecting all the samples, we are ready to calibrate using the corresponding OpenCV function:

```
    # Compute the camera parameters given all the samples collected
    print('- Found pattern in %d/%d images' % (len(images_used), len(files)))
    print('- Calibrating...')
    error, K, dist, rvecs, tvecs = cv2.calibrateCamera(obj_points, img_points, (w, h), None,
None)
```

We end by saving the results to a YAML file (using OpenCV's FileStorage class) for later use:

```
     # Save the parameters to a file
    cv_file = cv2.FileStorage(param.out_calib_file, cv2.FILE_STORAGE_WRITE)
    cv_file.write("camera_matrix", K)
    cv_file.write("distortion_coefficients", dist)
```

```
cv_file.write("mean_error", error)
cv_file.release()
```

You can open the resulting YAML file with any text editor to check its contents.

3.1 EXERCISE 1

Knowing that the square size is of 2cm (set it to **0.02m**, so that the calibration is in meters), use the calibration script to **calibrate the two cameras individually**. The data to calibrate each camera is provided in:

`sample_data/calibration/left`

`sample_data/calibration/right`

You can count the number of inner corners in X and Y from one of the images. Remember that you can use the following command in order to see the parameters required for executing the calibration (and their meaning):

```
$ ./monocular_calibration.py –h
```

Run the calibrations with the "--debug_dir" parameter. This will generate some debug images, including the pattern detection and the reprojection of the pattern points following the calibration you just computed. Comment the results:

- Is the pattern correctly detected in all images?
- From the information printed on screen and the debug images generated, how can you tell if the calibration is accurate?

4 STEREO CALIBRATION

So far, we have found the intrinsic parameters of the camera. Now we go for the extrinsics, i.e., the rotation/translation between the two cameras.

The calibration procedure is similar, but in this case we need to see the pattern in both images at the same time. To calibrate, we will use the "stereo_calibration.py" script. If you open it, you will observe that it is very similar in essence to the monocular case, since the main scope of the script is to find the chessboard in two images instead of just one. The only difference is the *cv2.stereoCalibrate* function, which performs the calibration:

```
error, left_camera_calib, left_dist_coefs, right_camera_calib, right_dist_coefs, R, T, E, F =
cv2.stereoCalibrate(obj_points, left_img_points, right_img_points, left_K, left_dist, right_K,
right_dist, (w, h), criteria=criteria, flags=flags)
```

4.1 EXERCISE 2

Use the "stereo_calibration.py" script to perform the stereo calibration for our camera. The data is provided in data/calibration/stereo. Again, use the "-h" parameter to get the usage of the method, and the "--debug_dir" parameter to check that the chessboard detection method behaved correctly.

If you check the documentation for the *cv2.stereoCalibrate* function, you will see that it is not required to perform the monocular calibration and then the stereo one, as done in this document: you can perform both calibrations at the same time. However, what are the advantages of doing it separately?

5 3D RECONSTRUCTION

Once the camera is calibrated, we can use it to perform 3D reconstruction from the images. In order to ease the computations, the images are usually rectified in a first step. Then, if we can match points in both images, we can triangulate their 3D position. For this matching, we will explore both sparse and dense matching pipelines.

6 RECTIFICATION

When working with stereo data, we usually rectify the images in order to reduce the computational effort required to match points among stereo pairs.

The rectification procedure warps the input images to emulate the perfect stereo configuration. That is, two fronto-parallel cameras whose centers are vertically aligned. As you may know from other courses, in this configuration the epipolar lines coincide with the horizontal pixel lines in each image. That is, as shown in Figure 2, the Y coordinates of matching points should be the same (up to some tolerance in real data, as calibration is never perfect).

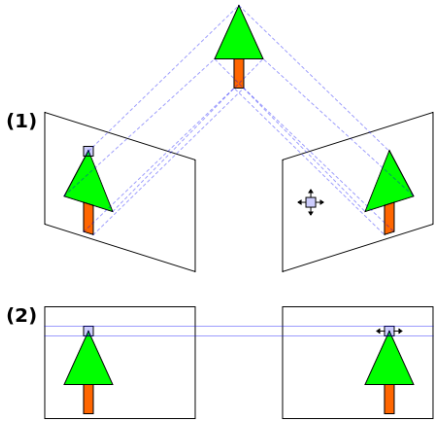


Figure 2. Stereo rectification. (1) shows the original stereo pair, and (2) the rectified stereo: an "ideal" fronto-parallel stereo where corresponding points lay on the same Y coordinate in both images.

Thus, given a point in an image, its corresponding point in the other image of the stereo pair is bounded to lay on the same Y coordinate. This effectively reduces the search space for matchings from 2D (i.e., the full image) to 1D (i.e., a line in the image).

The script “rectify_stereo_dataset.py” computes the rectified image pairs from a given dataset. Given the input stereo calibration, it computes a rectification map encoding the warping to be applied to the left/right images:

```
# Pre-compute rectification parameters
print("- Pre-computing the rectification parameters...")
im_size = (stereo_calib["image_width"], stereo_calib["image_height"])
R1, R2, P1, P2, Q, roi1, roi2 = cv2.stereoRectify(stereo_calib["left_camera_matrix"],
                                                  stereo_calib["left_distortion_coefficients"],
                                                  stereo_calib["right_camera_matrix"],
                                                  stereo_calib["right_distortion_coefficients"],
                                                  im_size,
                                                  stereo_calib["stereo_rotation"],
                                                  stereo_calib["stereo_translation"],
                                                  alpha=param.alpha)
left_maps = cv2.initUndistortRectifyMap(stereo_calib["left_camera_matrix"],
                                        stereo_calib["left_distortion_coefficients"],
                                        R1, P1, im_size, cv2.CV_16SC2)
right_maps = cv2.initUndistortRectifyMap(stereo_calib["right_camera_matrix"],
                                         stereo_calib["right_distortion_coefficients"],
```

```
                                        R2, P2, im_size, cv2.CV_16SC2)
```

Then we just apply these maps to the input images, to get a new set of rectified images:

```
# Rectify all the images
for i in range(len(left_files)):
    path, ext = os.path.splitext(os.path.basename(left_files[i]))
    full_out_img_path = os.path.join(param.left_out_dir, path + ext.lower())
    print("- Rectifying left image " + left_files[i])
    iml = cv2.imread(os.path.join(param.left_imgs_dir, left_files[i]))
    iml_rect = cv2.remap(iml, left_maps[0], left_maps[1], cv2.INTER_LANCZOS4)
    cv2.imwrite(full_out_img_path, iml_rect)

for i in range(len(right_files)):
    path, ext = os.path.splitext(os.path.basename(right_files[i]))
    full_out_img_path = os.path.join(param.right_out_dir, path + ext.lower())
    print("- Rectifying right image " + right_files[i])
    imr = cv2.imread(os.path.join(param.right_imgs_dir, right_files[i]))
    imr_rect = cv2.remap(imr, right_maps[0], right_maps[1], cv2.INTER_LANCZOS4)
    cv2.imwrite(full_out_img_path, imr_rect)
```

Since we are creating "virtual" cameras following this perfect fronto-parallel configuration, we are in fact **changing the camera calibration**. Therefore, we save this new calibration for later use:

```
# Save the new camera parameters
cv_file = cv2.FileStorage(param.out_calib_file, cv2.FILE_STORAGE_WRITE)
cv_file.write("left_R", R1)
cv_file.write("right_R", R2)
cv_file.write("left_P", P1)
cv_file.write("right_P", P2)
cv_file.write("Q", Q)
cv_file.release()
```

6.1 EXERCISE 3

Rectify all the stereo pairs in the provided dataset (reconstruction/left and reconstruction/right in the sample_data folder) using the "rectify_stereo_dataset.py" script. Observe the difference in the output images when using parameter *alpha = 0* and *alpha = 1*. What is the meaning of this parameter?

7 SPARSE RECONSTRUCTION

In this section, we will use standard feature detector/descriptors in order to reconstruct the keypoints in the scene in 3D. For this purpose, we will use the "sparse_matching_simple.py" script, which we will overview in the following. While here we just comment the parts with some processing involved, note that the script shows several intermediate steps on screen if the "--debug" flag is set.

In the script, we take two images of a stereo pair as input and convert them to grayscale (the feature detector/descriptor does not require colour information):

```
# Load the images
print('- Loading images...')
imgL = cv2.imread(param.left_img_file)
imgR = cv2.imread(param.right_img_file)

# Convert to greyscale (detectors do not use color)
imgLG = cv2.cvtColor(imgL, cv2.COLOR_BGR2GRAY)
imgRG = cv2.cvtColor(imgR, cv2.COLOR_BGR2GRAY)
```

After that, we load the calibration parameters (remember, these should correspond to the **output after rectification**):

```
# Load the rectified stereo camera calibration
if param.rect_stereo_calib_file:
    print('- Loading the rectified stereo calibration...')
    cv_file = cv2.FileStorage(param.rect_stereo_calib_file, cv2.FILE_STORAGE_READ)
    left_P = cv_file.getNode("left_P").mat()
    right_P = cv_file.getNode("right_P").mat()
```

Then, we extract the features and compute the descriptors, both using SIFT in this case:

```
    sift = cv2.SIFT_create()
    kpL, descL = sift.detectAndCompute(imgLG, None)
    kpR, descR = sift.detectAndCompute(imgRG, None)
```

Once computed, we match them using a brute force approach. That is, we find the best match given the Euclidean distance between the descriptors:

```
    # Match the descriptors
    print('- Matching the features...')
    bf = cv2.BFMatcher() # Create the matcher
    matches = bf.match(descL, descR)
```

Now we collect the 2D points of the matches on each image:

```
    # Reconstruct the 3D points if the camera calibration is available
    if param.rect_stereo_calib_file:
        print('- Computing the sparse 3d point cloud...')
        # Collect the 2D projections on each image
        p2dL = np.array([kpL[m.queryIdx].pt for m in matches]).reshape(-1, 1, 2)
        p2dR = np.array([kpR[m.trainIdx].pt for m in matches]).reshape(-1, 1, 2)
```

And we triangulate their position in 3D using the computed rectified stereo parameters (more precisely, the projection matrices for left/right cameras). After triangulation, we de-homogenize the points (remember that in computer vision points/transformations are always assuming homogeneous coordinates!):

```
       # Triangulate the points
        p3dh = cv2.triangulatePoints(left_P, right_P, p2dL, p2dR)
        # De-homogenize
        p3dh = p3dh / p3dh[3]
        p3d = p3dh[:3].T # And transpose, for convenience
```

We also remove obvious outliers before displaying the results. That is, we remove points that are "behind" the cameras (i.e., their Z is less than zero) or too far from it (more than 10 meters away, in this case):

```
        # Remove obvious outliers:
        #  - points behind the camera
        p3d = p3d[np.where(p3d[:,2] > 0)]
        #  - points too far from the camera (>10 meters, assuming the calibration was done in
meters)
        p3d = p3d[np.where(p3d[:,2] < 10)]
```

Finally, we will use Open3D to visualize/save the point sets:

```
        # Convert the points to Open3D for visualization/saving
        pcd = o3d.geometry.PointCloud()
        pcd.points = o3d.utility.Vector3dVector(p3d)

        if param.debug:
            # Visualize the points
            print('- Showing the reconstructed 3D points, press ''q'' or close the window to
continue...')
            o3d.visualization.draw_geometries([pcd])

        # Write the result to a PLY file
        print('- Saving the 3d point cloud...')
        o3d.io.write_point_cloud(param.out_ply, pcd)
```

7.1 EXERCISE 4

Use "sparse_matching_simple.py" to reconstruct the features detected on the rectified stereo pairs. We provide two different stereo pairs for you to test. Comment on the results (remember to set the "--debug" flag!).

7.2 EXERCISE 5

Notice that in “sparse_matching_simple.py” we are just performing brute force matching in order to find the correspondences between the 2D features. This is very generic, and is not enforcing in any way the known geometry of the stereo camera, and specially the rectified configuration of our pairs.

Extend the “sparse_matching_simple.py” script to filter those matches not following the epipolar constraint (you can copy the script with another name to compare both later on). I.e., loop over each match and check that the 2D projection of each feature on each image is approximately on the same Y line in both images. Write your code in the section marked with a **### Ex. 5** comment.

7.3 EXERCISE 6

You can test other feature detector/descriptors to check the performance of the different methods available in OpenCV. See the following reference to learn about the different algorithms available in the library:

https://docs.opencv.org/4.x/d5/d51/group__features2d__main.html#ga15e1361bda978d83a2bea629b32dfd3c

Look for the **### Ex. 6** comment in the code and change the detector/description by another method, and compare the results with those provided by SIFT. Moreover, you can test different parameters for each detector (e.g., change the contrastThreshold parameter in SIFT).

8 DENSE RECONSTRUCTION

Since the rectified stereo configuration greatly simplifies the matching problem, we can try to reconstruct not only feature points, but ALL the points on the image (or as much as possible). That is, for each 2D point in the left image, we will seek for its best match on the right image along the line of pixels with the same Y coordinate (that is, the epipolar line).

The script “dense_matching.py” applies the dense matcher algorithm called Block Matching from the OpenCV library:

```
stereo = cv2.StereoBM_create(numDisparities=num_disp, blockSize=block_size)
disp = stereo.compute(imgLG, imgRG).astype(np.float32) / 16.0
```

Basically, this method takes each point of a given image, and computes Sum of Squared Differences (SSD) similarity with the points in the same Y line in the other image of the pair. Then, it takes the one with the lowest score as the best match. The number of points tested is bounded by the *numDisparities* parameter, i.e., the search for a match starts at the same (x, y) position on the second image, and extends up to (x-numDisparities, y). The second parameter, *blockSize*, defines the window size that is opened around each candidate match to compute the SSD score.

The results of this function are stored in the form of a disparity map. This map consists of an image in which we encode the differences in Y defining the match in the other image. So disparity(x, y) = d encodes that point (x; y) in the left image matches with (x-d, y) in the right image.

This representation can be converted to a 3D point set using the following function:

```
points = cv2.reprojectImageTo3D(disp, Q)
```

The function uses the Q matrix, which is a perspective transform encapsulating all the stereo calibration parameters needed to convert disparity into depth, i.e., it transforms the disparity image into a 3D image.

After reconstruction, we remove obvious outliers and visualize/save the 3D point set, as done in the previous script.

8.1 EXERCISE 7

Use the "dense_matching.py" script to reconstruct the provided pairs in the dataset. Change the parameters for each pair in order to obtain the best results (look for the **### Ex. 7** comment in the code). Since it may be a bit difficult to set the parameters on the "dense_matching.py" script directly, we provide a GUI version called "dense_matching_gui.py", where you can play with the parameters of the algorithm until you have values that work for each pair. You can then set these values in the "dense_matching.py" script in order to reconstruct the 3D point set.

Note that there are some restrictions for the values allowed for some of the parameters (e.g., *num_disparities* must be positive and divisible by 16, while *block_size* must be odd). For the rest of the parameters, a description of their meaning can be found in the comments of the code, and you can check the documentation at:

https://docs.opencv.org/4.6.0/d9/dba/classcv_1_1StereoBM.html

8.2 EXERCISE 8

The Block Matching algorithm is suitable for real-time application due to its simplicity and its highly parallelizable scheme. However, we can apply more complex approaches reducing the number of outliers. OpenCV provides a variant of Block Matching called Semi-Global Block Matching. Check the documentation of this function and change the "dense_matching.py" script to use this function.

Compare the behaviour/parameterization of this function in front of the Block Matching method.

9 COLLECTING NEW DATA

As presented at the beginning of this document, you can perform all the steps/exercises in this document with the provided sample data. However, we want you to experiment the problems that arise when working with real data (collected by you!). Thus, you will find a ZED2 stereo camera in the lab, that you can use at any time to capture new data.

These cameras have their own software to capture and process the data, but we will use them just as regular USB cameras. You can run the "capture_stereo_images_zed.py" script, where you need to specify the output directory where the stereo pair of images will be stored (in <out_dir>/left and <out_dir>/right sub-directories):

```
python capture_stereo_images_zed.py <out_dir>
```

Should not be the case in the PCs of the lab, but in case you are working on your own laptop and have more than a camera connected, you may need to specify the port where the ZED camera is connected. You can get a list of the ports corresponding to connected cameras with the "list_ports.py" script:

```
python list_ports.py
```

You can then specify this port to the "capture_stereo_images_zed.py" script with the "-p" option.

Note that the camera used to provide the sample data is not the same camera as the one in the lab. Therefore, **you must collect both new calibration images as well as new images for reconstruction**. Then, use this new data to re-execute all the exercises/steps in this document.

10 TO DELIVER

You are expected to deliver a report including the solution to the different exercises and questions listed in the sections above. Include results using both the sample data provided and the data you collected at the lab. You should list all the problems you encountered during the development of this lab session, and especially the differences you found between using our sample data and the data you collected. You can also send the code, if you feel it necessary (otherwise, describe it correctly in the report).

The report is to be delivered one week after the end of the second day in the lab (i.e., the day before starting the next practical session) via Moodle.