3D DATA PROCESSING - LAB 1

Topic: Semi Global Stereo Matching with Monocular Disparity Initial Guess.
Goal: Disparity maps estimation of stereo images.

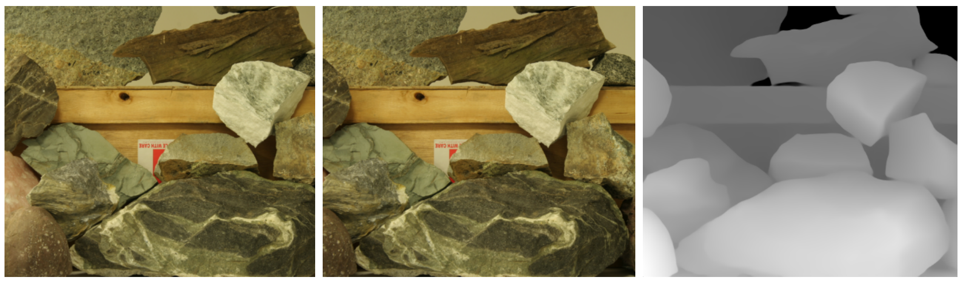

Fig. 1: Example of input data item: left and right images (left and center) ; initial guess of the disparity map computed with [1] (right).

In this assignment, you will have to complete the code of a basic implementation of the Semi Global Block Matching (SGM) stereo matching algorithm. Furthermore, in addition to the pair of input stereo images (e.g., Fig. 1, left and center), an initial guess of the disparity map is provided (e.g., Fig. 1, right), calculated using a very recent data-driven monocular depth estimation method [1]. This initial guess is generally very accurate but defined up to a scalar factor. Once you have computed the disparity map with SGM, you can compute this scalar factor and use the scaled initial guess disparity map to refine/complete the disparity map computed with SGM.

What is provided

An incomplete source code of a basic SGM algorithm is provided, defined as a C++ `SGM()`class within the `sgm.h` and `sgm.cpp` source files. Basic OpenCV structure and functions are used here. The `main.cpp` source file defines the main function that calls the `SGM()` class and computes the quantitative results (i.e., the MSE errors) on the sample data item provided as input argument. Five example data items are provided inside the `Example/` folder. Each one includes the left and right input images (`left.png` and `right.png` files), the right and left initial guess of the disparity map computed with [1] (`right_mono.png`, `left_mono.png` files), and the right ground truth disparity map used to evaluate the results (`rightGT.png` file). Please refer to the `ReadMe.md` file for instructions on how to build and execute the code on the provided data items.

The provided code already implements:

- The **right-to-left** cost volume computation (for all possible, positive disparities), obtained by summating the hamming distances between right and left 3x3 census transforms inside a support window around each pixel. The resulting cost volume is stored inside the **`cost_`** tensor and it is defined for each possible pixel `x`, `y`, and possible disparity `d` (e.g. `cost_[y][x][d]`). Total size is (`height_`, `width_`, `disparity_range_`). **Note**: Variables with names ending with an underscore '_' represent class member variables, defined in `sgm.h`.

- Part of the cost aggregation calculation. Aggregated costs are stored inside the **`aggr_cost_`** tensor computed for all the defined paths. As the **`cost_`** tensor it is defined for each possible pixel `x, y`, and possible disparity `d`.
- Right-to-left disparity and confidence final computation. The disparities are stored inside the **`disp_`** 2D matrix, disparities values are 8-bit integers (e.g., to query the disparity for the pixel `x, y`, use the standard OpenCV method `at<>()`, e.g. `disp_.at<uchar>(y, x)`),
- Quantitative assessment, by means of the Mean Squared Error (MSE) computation of the right-to-left disparity map.

What you need to do

You will have to implement some missing code fragments in order to calculate the right-to-left disparity map with SGM (remember that in the right-to-left case, the disparities will be positive). Moreover, you will have to take advantage of the initial guess of the right disparity map (`right_mono.png` file) computed with [1] to improve the quality of the SGM disparity map (take into account that the provided initial guess is defined up to a scalar factor, that should be estimated before using it).
Finally **try to leverage `left_mono.png` for the disparity estimation**. Observing improvements in metrics is not required after implementing this last step.

Descriptions of what is required are reported directly inside the code. Chunks of code to be completed are marked with the preamble:

```
//////////////////// Code to be completed (x/y) //////////////////
```

followed by a short description of the required functionalities.

Specifically, you have to complete the following tasks:

1. Complete the **`compute_path_cost()`** function that, given:

   i) a single pixel **`p`** defined by its coordinates **`cur_x`** and **`cur_y`**;
   ii) a path with index **`cur_path`** (**`cur_path=0,1,..., PATHS_PER_SCAN - 1`**, a path for each direction), and;
   iii) the direction increments **`direction_x`** and **`direction_y`** associated with the path with index **`cur_path`** (that are the `dx,dy` increments to move along the path direction, both can be -1, 0, or 1),

   should compute the path cost for **`p`** for all the possible disparities **`d`** from **0** to **`disparity_range_`** (excluded, already defined). The output should be stored in the tensor (already allocated) **`path_cost_[cur_path][cur_y][cur_x][d]`**, for all possible **`d`**. That is, at each call of **`compute_path_cost()`**, given a pixel with coordinates (**cur_x**, **cur_y)** and a path with index **cur_path**, all the **disparity_range_** disparity entries should be computed. The matching cost (data term) should be recovered from the already computed cost volume (see previous section), e.g. **cost_[cur_y][cur_x][d]**. To update the path cost, use the already defined class variables **p1_** and **p2_** which represent respectively the small and big penalty added factors. Take into account that the smoothness term (i.e., the penalty term) is not considered in the starting and end points. Use the `pw_` structure, which holds the minimum and maximum horizontal and vertical coordinates, to verify this special case (see in the code below the condition of the first `if()`).

```
void SGM::compute_path_cost(int direction_y, int direction_x, int cur_y, int
cur_x, int cur_path)
{
 //use this variables if needed
 unsigned long prev_cost, best_prev_cost, no_penalty_cost,
               penalty_cost, small_penalty_cost, big_penalty_cost;

 // if the processed pixel is the first:
 if( cur_y == pw_.north || cur_y == pw_.south ||
     cur_x == pw_.east || cur_x == pw_.west)
 {
   //Please fill me!
 }

 else
 {
   //Please fill me!
 }
}
```

2. In the function **aggregation()**, set the variables **start_x**, **start_y**, **end_x**, **end_y**, **step_x** and **step_y** are used to define the cycles where **compute_path_cost()** is called. Step and direction are closely related. The step variables control the scan order over all image pixels when computing the path cost, while directions define the next (and previous) pixel on the current path. Consider here the already defined vector **paths_**, containing all **PATHS_PER_SCAN** paths encoded by their scan direction (for example direction_x = -1 and direction_y = 0 means a right to left horizontal path, east to west).

3. Be careful to instantiate the values of `step_x` and `step_y` in order to scan the cost volume in the correct order (hint: check the value of dir_x and dir_y**,** for instance, dir_x = -1 and dir_y = 0 means a right to left horizontal path).

```
void SGM::aggregation()
{
 //for all defined paths
 for(int cur_path = 0; cur_path < PATHS_PER_SCAN; ++cur_path)
 {

   int dir_x = paths_[cur_path].direction_x;
   int dir_y = paths_[cur_path].direction_y;

   int start_x, start_y, end_x, end_y, step_x, step_y;

   //TO DO: initialize the variables start_x, start_y, end_x, end_y,
next_dim_x, next_dim_y with the right values
 }
//TO DO: aggregate the costs for all direction into the aggr_cost_ tensor
}
```

3. In the **`compute_disparity()`** function, use high confidence disparities to compute the linear coefficients that should be applied to the monocular disparity values in order to scale them. In other words, add here the "good" disparities estimated with SGM and the corresponding unscaled disparities from the right-to-left initial guess of the disparity map (stored in `mono_`, it can be accessed e.g., with `mono_.at<uchar>(row, col)`) to the pool of disparity pairs to be used to estimate the unknown scale factor. Namely, given a pair of disparities of the same pixel d_{sgm} and d_{mono} representing the disparity estimated with SGM and the (unscaled) initial guess disparity, find h and k such that:

   $$d_{sgm} = h * d_{mono} + k$$

   Recall from the Camera Calibration Camera lecture that the solution for the least squares problem for a nonhomogeneous system $Ax = b$ is expressed as:

   $x = (A^T A)^{-1} A^T b$

   Where $x = [h\,k]^T$, $b = d_{sgm}$, $A = [d_{mono,} \bar{1}]$.

   with d_{sgm} and $d_{mono,}$ are nx1 vectors holding the n "good" disparities estimated with SGM and the corresponding unscaled disparities from the right-to-left initial guess of the disparity map, respectively.

4. Finally, again in the the **`compute_disparity()`** function, scale the initial guess disparities and use them to improve/replace the low-confidence SGM disparities.

SAMPLE OUTPUT

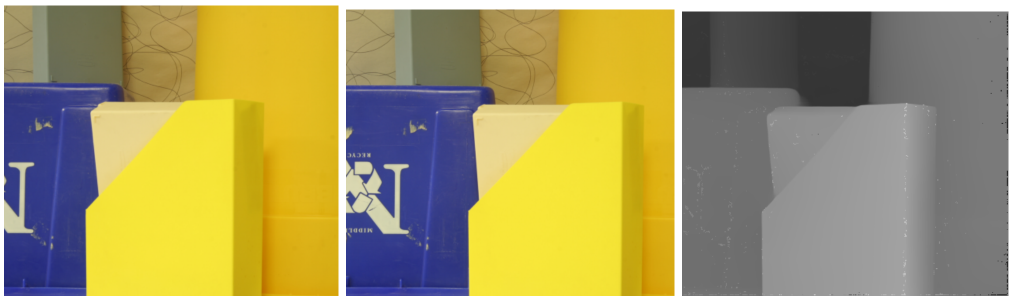

Fig. 2: Example of input data item: left and right images (left and center); an example of estamated right-to-left disparity map computed with SGM and refined with scaled initial guess disparities.

Once all the code sections have been correctly completed, by running the ./sgm executable on a data time, the calculated disparity map will be saved as a png image (e.g. see Fig. 2) and compared with the ground truth, reporting to the terminal the Mean Squared Error (MSE). Below are some reasonable values of MSE that can be obtained with a good implementation by using the refinements step (task 4):

Data item	Aloe	Cones	Plastic	Rocks1
MSE	13.78	16.36	341.73	32.71

What you need to deliver

- Source code (without objects and executables)
- A short written report that includes:
  - A brief description of the work done, and the problems encountered, if any;
  - The quantitative results (i.e., the MSE errors) for each of the four provided data items in the Example/ folder with and without the refinement step (task 4 above) and some qualitative results.

[1] Lihe Yang, Bingyi Kang, Zilong Huang, Xiaogang Xu, Jiashi Feng, Hengshuang Zhao "Depth Anything: Unleashing the Power of Large-Scale Unlabeled Data", CVPR 2024
For more information, please refer also to the paper website: https://depth-anything.github.io

FAQ

1) In the description of the third task (3/4) it is mentioned that we have to add the disparity pairs (sgm and mono) of high confidence pixels to the disparity pool, but there is no disparity pool data structure defined. Are we supposed to define it on our own or is it defined in some part of the code?

The data structure to gather disparities with good confidence, must be defined by you (use Eigen)

2) I have a doubt about how the first pixel is processed because with the current initialization of the variables :

pw_.north = window_height_/2;
pw_.south = height_ - window_height_/2;
pw_.west = window_width_/2;
pw_.east = width_ - window_height_/2

the borders of the image are not considered in the computation of path costs. In order to consider the pixels on the borders, I would initialize the variables in the following way:

pw_.north = 0;
pw_.south = height_ - 1;
pw_.west = 0;
pw_.east = width_ - 1

Is it necessary to modify these variables?

You should leave the original initialization, as costs on the cost matrix are not computed on the borders (see how census transform is computed). If you want to change those parameters you can pad the image or apply more complex solutions, but it's out of the scope for the required task.

3) In the provided code, inside the compute_path_cost function, the condition of being on the border has been checked in order to check if the processed pixel is the first pixel in the path, However, isn't it possible that the processed pixel is not the first, but the last in the path, potentially leading to an incorrect cost assignment?

With the current implementation, a wrong cost is assigned to the last pixel. In order to avoid additional check conditions and considering that the error introduced is almost negligible we opted for this solution. If you want you can implement the additional check to improve the correctness of the algorithm, but it's obviously not required.