<u>final project – 2 שיטות מתמטיות לעיבוד לניתוח תמונות</u>

Barak Meiri 308001346

In this document I will explain my final project, which deals with the problem of stereo matching estimation.

The reference paper I based on is:

Stereo Matching with Mumford-Shah Regularization and Occlusion Handling by Rami Ben-Ari and Nir Sochen [2].

I my project we try to estimate the disparity between two images which are different only by a translation in the horizontal axis.

There are many ways to solve this problem, some use classical methods and some use learning methods to estimate different stages in the process of the disparity estimation.

In this project we focus on the method that is presented in the reference paper, and try to combine optimizations to get the disparity results.

Description and summarization of the reference paper:

This paper introduces a novel approach to solving correspondence in binocular stereo vision. It uses a variational framework with unique regularization terms for discontinuity preservation and occlusion handling. The method provides accurate disparity maps with sharp boundaries and occlusion maps. Experimental tests show significant improvements compared to other methods, making it one of the top-ranked stereo matching algorithms on the Middlebury stereo benchmark.

This paper addresses the problem of correspondence in binocular stereo vision, aiming to establish a disparity map showing pixel differences between two images. It discusses the importance of dense disparity maps for detailed information and categorizes dense matching algorithms into local and global methods. Variational methods, which offer mathematical soundness and continuous solutions, are highlighted as effective for correspondence establishment. The Middlebury stereo benchmark is mentioned as a ranking method for stereo algorithms, and the paper proposes a method that promotes variational approaches in this benchmark. The paper also introduces a data-fidelity term that incorporates image gradients and color information in the objective functional.

This paper addresses the ill-posed nature of the stereo vision minimization problem by introducing a regularization term based on the Mumford-Shah (MS) functional, focusing on piecewise smooth modeling with abrupt disparities. It also evaluates discontinuity maps and introduces additional constraints to improve disparity/depth boundary allocation.

Unlike some previous variational stereo methods, this approach considers half-occlusions in the stereo problem and suggests a novel method for their extraction. The method's performance is assessed

on various datasets, including the Middlebury benchmark, showing superior results compared to recent variational methods.

The paper extends previously published work with several notable differences, including a comprehensive introduction to related work, improved discontinuity function calculation, extensive experimental evaluation, and comparisons between L2 and L1 regularizers in the Mumford-Shah functional.

The paper is organized into sections covering related work, the Mumford-Shah framework, the baseline method, an inhomogeneous image-based approach, a novel variational model handling half-occlusions and discontinuities, minimization strategies, and detailed experimental results and comparisons, followed by a conclusion.

Now we get to the loss and the math behind the method described in the reference paper. They used the functional theory and minimized two functional to get the two gradient steps for each of them.
In our method we use the same functionals, but we optimize them to achieve their minima.

Formulas derivation:

The Mumford-Shah functional, outlined in [31], focuses on denoising and segmenting gray-level images. It represents an image as smooth segments separated by distinct contours. The goal is to find smooth curves C_i and an optimal piecewise smooth approximation I for the observed image I_0, ensuring smooth variation within segments Ω/C_i and discontinuity across edges C_i.

$$E(I, C_i) = \int_{\Omega} (I - I_0)^2 dA + \beta_i \int_{\Omega \setminus C_i} \|\nabla I\|^2 dA + \alpha_i \int_{C_i} d\sigma, \tag{1}$$

The last term in equation (1) represents a line integral that penalizes the length of the set C_i, with α_i and β_i as fixed parameters determining the weights for smoothness and edge length penalization. The index i corresponds to the image cue. In homogeneous regions, linear diffusion is enforced, inhibiting smoothing across discontinuities. The main challenge in minimizing (1) lies in dealing with the unknown discontinuity set C_i, making classical calculus of variation methods unsuitable. Therefore, De Giorgio proposed an approximation of the Mumford-Shah segmentation functional using regular functionals within the framework of Γ-convergence [12].

The central concept of Γ-convergence involves approximating the irregular functional $E(I, C_i)$ with a sequence of regular functionals $E_{\epsilon_i}(I)$ such that:

$$\lim_{\epsilon_i \to 0} E_{\epsilon_i}(I) = E(I, C_i)$$

and the minimizers of $E_{\epsilon_i}(I)$ approximate the minimizer of $E(I, C_i)$.

Ambrosio and Tortorelli applied the Γ-convergence framework to minimize the Mumford-Shah functional, representing the discontinuity set using a characteristic function$(1 - \chi^{C_i}$. They introduced an approximation for χ^{C_i} using a smooth auxiliary function $v_i(x, y)$, where v_i is 0 if (x, y) is in C_i and 1 otherwise. The solution can then be approximated by minimizing the resulting functional:

$$\begin{aligned} E^{MS}(I, v_i) = &\int_\Omega (I - I_0)^2 dA + \beta_i \int_\Omega \|\nabla I\|^2 v_i^2 dA \\ &+ \alpha_i \int_\Omega \epsilon_i \|\nabla v_i\|^2 + \frac{(v_i - 1)^2}{4\epsilon_i} dA. \end{aligned} \tag{2}$$

The Total Variation Mumford-Shah functional was suggested by Shah:

$$\begin{aligned} E^{MSTV}(I, v_i) = &\int_\Omega (I - I_0)^2 dA + \beta_i \int_\Omega \|\nabla I\| v_i^2 dA \\ &+ \alpha_i \int_\Omega \epsilon_i \|\nabla v_i\|^2 + \frac{(v_i - 1)^2}{4\epsilon_i} dA, \end{aligned} \tag{3}$$

In the following we discuss our specific task and derivations.

Baseline approach:

Given a normalized image pair I_j, where j takes values in the set $\{l, r\}$ representing left and right images, respectively. For each point $x = (x, y)$ in the left image domain (l), the corresponding point in the right image domain (r) is described as $g_l = (x - d_l(x), y)$, where d_l is the left disparity map.

The brightness constancy assumption is commonly made when relating corresponding pixels. However, practical approaches often incorporate additional attributes, such as intensity gradients, to enhance correspondence and reduce ambiguity. Another option is using a robust data term that employs a modified L1 norm to ensure constancy in both brightness and brightness gradients:

$$e_d(s_d^2) = \sqrt{s_d^2 + \eta^2}, \qquad (4)$$

where:

$$s_d^2 = \|I_r(\mathbf{g}_l) - I_l(\mathbf{x})\|^2 + \lambda\|\nabla I_r(\mathbf{g}_l) - \nabla I_l(\mathbf{x})\|^2. \qquad (5)$$

The parameter η is a small positive constant acting as a stabilizing factor for the modified L1 norm. This norm retains the robustness to outliers of the L1 norm while preventing singularity at zero in the resulting partial differential equation. When $s_d^2 << \eta^2$, the norm tends toward the L2 norm by Taylor expansion rule.

the smoothness prior in this approach is the (modified) TV regularizer:

$$e_s = \beta\sqrt{\|\nabla d_l\|^2 + \eta^2} \qquad (9)$$

where β is a weight parameter.

By considering the data and smoothness terms in equations (4) and (9), the resulting energy functional for disparity evaluation is obtained:

$$E(d_l) = \int_{\Omega_l} \sqrt{\|I_r(\mathbf{g}_l) - I_l(\mathbf{x})\|^2 + \lambda\|\nabla I_r(\mathbf{g}_l) - \nabla I_l(\mathbf{x})\|^2 + \eta^2} + \beta\sqrt{\|\nabla d_l\|^2 + \eta^2}. \tag{10}$$

The image edge map is initially derived by minimizing the energy functional (2) concerning the discontinuity function v_i, with β_i set to 1 without loss of generality. The **Euler-Lagrange** equations dictate the conditions that v_i must satisfy in this process:

$$\frac{\delta E_{\epsilon_i}}{\delta v_i} = \|\nabla I\| v_i + \alpha_i \frac{(v_i - 1)}{4\epsilon_i} - \alpha_i \epsilon_i \nabla^2 v_i = 0, \tag{11}$$

Subsequently, the value v_i, constrained within the range [0, 1], is employed as an inhomogeneous diffusion factor in the energy functional for the estimation of disparity d_{l_0}:

$$E(d_{l_0}) = \int_{\Omega_l} \sqrt{\|I_r(x - d_{l_0}, y) - I_l(x, y)\|^2 + \eta^2} + \widehat{\beta} v_i^2 \sqrt{\|\nabla d_{l_0}\|^2 + \eta^2} dA, \tag{12}$$

where $\hat{\beta} > 0$ is a weight parameter.

So far, we discussed the numeric method that they used to solve the disparity.

From now on we dive into our specific method that has been used in this project.

Let us start by specifying and defining the formulas of which serves for the losses development.

Our work in this project includes the following steps:

We start from two stereo images.

Initializing several neural networks following the concepts in the reference paper:

- $\phi(x, y)$ - determines if pixel (x,y) is occlude or not.
- $v(x, y)$ - discontinuity function which determines if (x,y) is a pixel on the contours of the disparity levels.
- $d(x, y)$ - disparity map which tells the disparity for each pixel in the image.

We try to learn these functions (\mappings) on the pixel space of the two images.

We cite the reference paper equation of which we use to determine the loss:

The Heaviside function is used to binarize the occluded and not occluded regions. $\phi(x, y) \geq 0$ means visible pixel.

$$H(\phi_i) = \begin{cases} 1, & \phi \geq 0 \\ 0, & otherwise \end{cases}$$

$$H_\epsilon(\phi) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan\left(\frac{\phi}{\epsilon}\right)\right)$$

$u(x)$ gives the rational that the disparities of the right image and the left image should be the negative of each other:

$$u_l(\mathbf{x}) := d_l(\mathbf{x}) + d_r(\mathbf{g}_l), \quad \forall \mathbf{x} \in \Omega_l, \tag{15}$$

e_{occl} is the occlusion map that measure the inconsistency in $u(x)$:

$$e^l_{occl} = -\ln\left(\epsilon_{occl} + (1-\epsilon_{occl})e^{-|u_l|}\right). \tag{16}$$

The first loss is the loss on the consistency in $u(x)$:

$$\begin{aligned} E^l_{occl}(\phi_l) = \int_{\Omega_l} & \left(e^l_{occl} * G_\sigma\right) H(\phi_l) + t(1-H(\phi_l)) \\ & + \nu \|\nabla H(\phi_l)\| dA, \end{aligned} \tag{19}$$

We wish to have small values of $u(x)$ which indicates the consistency in the disparities of the two images.

Now we define the loss on the two images.
Starting with s_d (5) which represents the differences between the images after considering the current disparity between them. If the disparity is correct, we should get small values of s_d.

From s_d, e_d, e_s as mentioned before, we derive the second loss which takes care of the visible parts, where we can guess the disparity between the two images.:

$$\begin{aligned} E_L\left(d^{\mathbf{v}}_l, v_l\right) = \int_{\Omega_l} & \left[e_d + e_s v_l^2 + \widehat{\alpha}(v_l - 1)^2\right] H(\phi_l) \\ & + \widehat{\epsilon} \|\nabla v_l\|^2 dA. \end{aligned} \tag{23}$$

Now that we have the two losses for the stereo optimization problem we can state the algorithm.

The Algorithm:

Given two stereo images (l, r):

1. initialize NNs: ϕ, v, d
2. calculate e_{occ}
3. calculate E_{occ} loss (loss 1)
4. calculate E_L loss (loss 2)
5. Backward the gradients to the NNs and change their params accordingly.
6. Repeat 2-5 until some condition\convergence.

Experiments:

Example 1:

I used the cones data images:

left image

right image

The outputs logs from the algorithm code seems like:

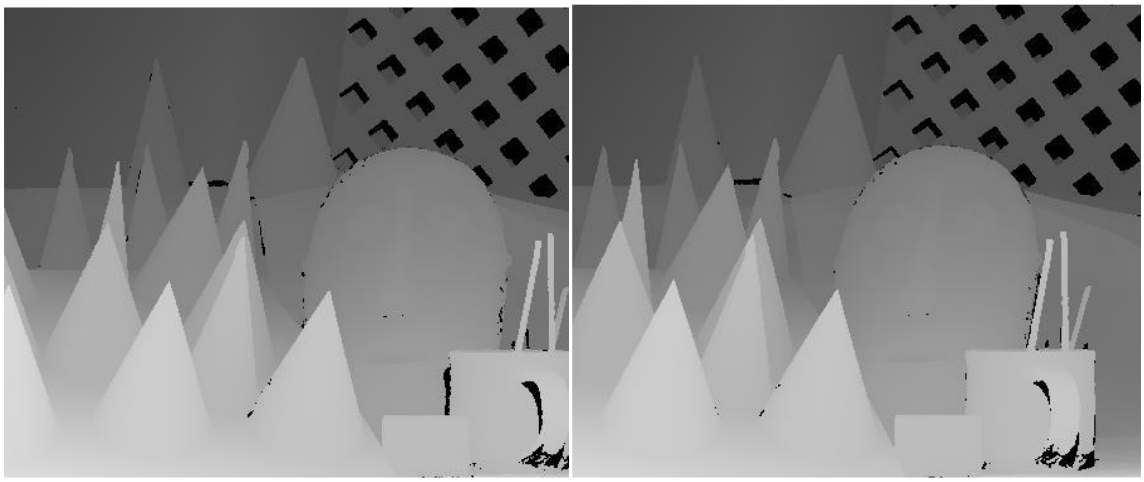
```
Ep: 2   L1-L: 99.35 L1-R: 1.43  L2-L: 570.08    L2-R: 93654.52  es_l: 55.05 es_r: 26.83 ed_l: 102066    ed_r: 94893 phi_l: -6.30    phi_r: 2.95 d_l: 0.98   d_r: 0.99
Ep: 3   L1-L: 99.31 L1-R: 1.32  L2-L: 567.22    L2-R: 93789.14  es_l: 49.37 es_r: 9.89  ed_l: 102066    ed_r: 94928 phi_l: -6.31    phi_r: 3.08 d_l: 0.98   d_r: 1.00
Ep: 4   L1-L: 99.26 L1-R: 1.25  L2-L: 564.07    L2-R: 93860.12  es_l: 44.80 es_r: 9.72  ed_l: 102066    ed_r: 94928 phi_l: -6.33    phi_r: 3.18 d_l: 0.98   d_r: 1.00
Ep: 5   L1-L: 99.22 L1-R: 1.19  L2-L: 560.99    L2-R: 93915.47  es_l: 41.16 es_r: 9.58  ed_l: 102066    ed_r: 94928 phi_l: -6.35    phi_r: 3.27 d_l: 0.98   d_r: 1.00
Ep: 6   L1-L: 99.18 L1-R: 1.14  L2-L: 558.01    L2-R: 93960.65  es_l: 38.31 es_r: 9.45  ed_l: 102066    ed_r: 94928 phi_l: -6.37    phi_r: 3.36 d_l: 0.98   d_r: 1.00
Ep: 7   L1-L: 99.14 L1-R: 1.10  L2-L: 555.12    L2-R: 93998.70  es_l: 36.25 es_r: 9.34  ed_l: 102066    ed_r: 94928 phi_l: -6.39    phi_r: 3.44 d_l: 0.98   d_r: 1.00
Ep: 8   L1-L: 99.09 L1-R: 1.06  L2-L: 552.32    L2-R: 94031.50  es_l: 34.73 es_r: 9.24  ed_l: 102066    ed_r: 94928 phi_l: -6.41    phi_r: 3.51 d_l: 0.98   d_r: 1.00
Ep: 9   L1-L: 99.05 L1-R: 1.03  L2-L: 549.61    L2-R: 94060.20  es_l: 33.46 es_r: 9.15  ed_l: 102066    ed_r: 94928 phi_l: -6.43    phi_r: 3.58 d_l: 0.98   d_r: 1.00
Ep: 10  L1-L: 99.01 L1-R: 1.00  L2-L: 546.98    L2-R: 94085.66  es_l: 32.32 es_r: 9.07  ed_l: 102066    ed_r: 94928 phi_l: -6.45    phi_r: 3.65 d_l: 0.98   d_r: 1.00
```

Which tells the first and second losses values and the intermediate values of different meaningful metrics.

First, I tried to estimate the disparity in a single side. Later when I saw that it didn't work, I did the symmetric calculations also for the other side.

In order to do a sanity check, I tried also to insert the ground truth disparity to check that the losses have smaller values then without inserting the ground truth – and it did happen.

The disparity for the sanity test was:



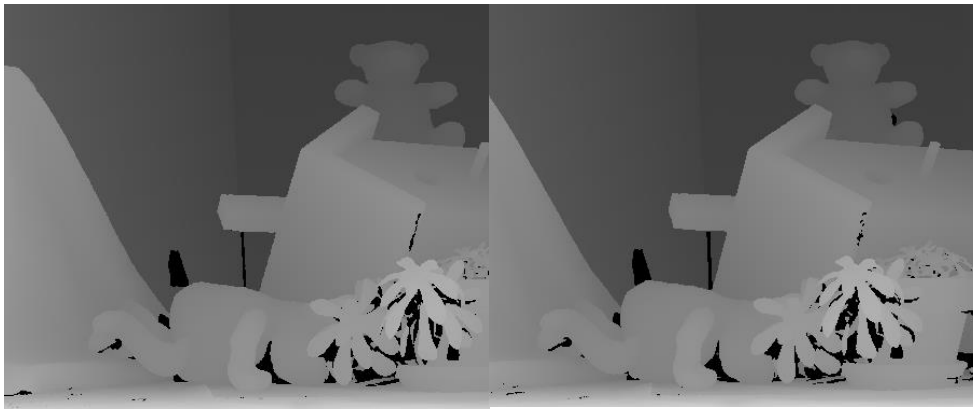
Left disparity right disparity

As we can see also the GT is not perfect since there are occluded regions.

Example 2:

Teddy data:

Image left image right



Disparity left Disparity right

We can observe here that the part of the leaves has large hidden space, which poses a problem for the disparity estimation.

Logs sample:

```
Ep: 118 L1-L: 94.64 L1-R: 58.38 L2-L: 92.31 L2-R: 37375.20  es_l: 12.00 es_r: 573.23    ed_l: 131164    ed_r: 97324 phi_l: -46.49   phi_r: -6.52    d_l: 1.00   d_r: 0.62
Ep: 119 L1-L: 94.60 L1-R: 85.09 L2-L: 92.22 L2-R: 9317.33   es_l: 12.01 es_r: 723.15    ed_l: 131164    ed_r: 91998 phi_l: -46.53   phi_r: -14.92   d_l: 1.00   d_r: 0.46
Ep: 120 L1-L: 94.56 L1-R: 83.95 L2-L: 92.14 L2-R: 10720.11  es_l: 12.01 es_r: 750.12    ed_l: 131164    ed_r: 94954 phi_l: -46.56   phi_r: -14.60   d_l: 1.00   d_r: 0.49
Ep: 121 L1-L: 94.51 L1-R: 84.08 L2-L: 92.05 L2-R: 10292.45  es_l: 12.01 es_r: 627.40    ed_l: 131164    ed_r: 92636 phi_l: -46.60   phi_r: -14.59   d_l: 1.00   d_r: 0.54
Ep: 122 L1-L: 94.47 L1-R: 83.27 L2-L: 91.97 L2-R: 10915.13  es_l: 12.02 es_r: 580.27    ed_l: 131164    ed_r: 91478 phi_l: -46.64   phi_r: -14.35   d_l: 1.00   d_r: 0.49
Ep: 123 L1-L: 94.43 L1-R: 82.03 L2-L: 91.89 L2-R: 11987.54  es_l: 12.02 es_r: 641.26    ed_l: 131164    ed_r: 90774 phi_l: -46.67   phi_r: -14.00   d_l: 1.00   d_r: 0.43
Ep: 124 L1-L: 94.38 L1-R: 73.60 L2-L: 91.80 L2-R: 20741.12  es_l: 12.02 es_r: 680.02    ed_l: 131164    ed_r: 93839 phi_l: -46.71   phi_r: -12.26   d_l: 1.00   d_r: 0.55
```

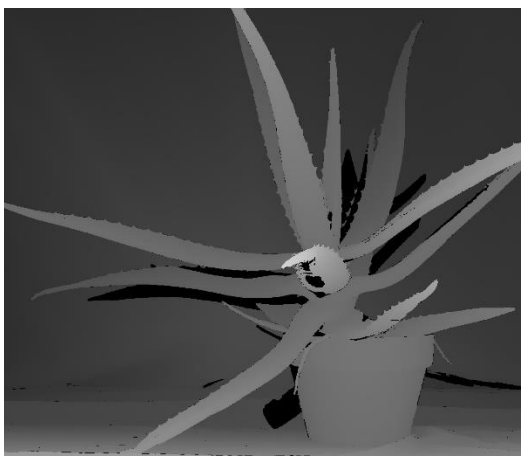
Example 3:

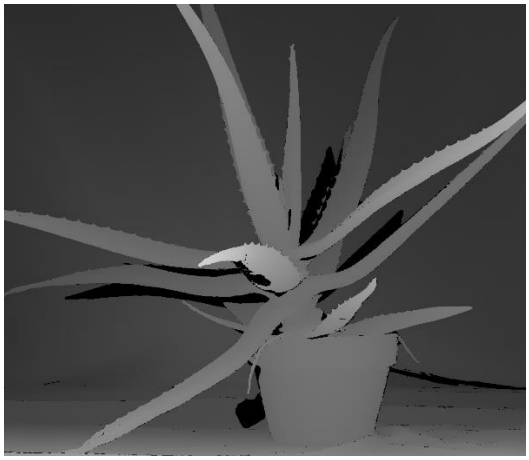
Aloe data:

Here we use the aloe data which consists of high brightness plant on the sofa.

Image left image right

The disparity images:





Disparity left Disparity right

The sofa has some features on it, and it lays in the background, while the foreground is the long plant leaves that hide the sofa and makes the disparity prediction in these areas poor.

Conclusions:

- As we can see the optimization process may tend to converge to some local minima or switching dots around this minima.
- There was a use in composing one loss from the two of them, while weighting each of them in a different scalar for balancing the advancements in the two directions which are dictated by the two losses.
- As for this project I can tell that if there is an efficient method for solving the functional and it can be mathematically formulated, it can be a better choice to use that instead of an optimization process.
- There are disadvantages in the optimization process regards to the hyper-parameters that this method offer. Some of them are the learning rates, losses balance parameters, stopping criteria\maximum iterations number.

To summarize, my project deals with the stereo matching problem and I aim to solve it with optimizations over the Mum-Shah losses and the loss for the occlusions.

Code:

My code for the project is shown here:

```
import torch
from torchvision import transforms
import numpy as np
from PIL import Image
from torch.utils.data import Dataset
from torch.utils.data.dataloader import DataLoader
import matplotlib.pyplot as plt
from torch import nn
import torch.nn.functional as F


def apply_gaussian_blur(image, kernel, kernel_size):
    # Expand dimensions for convolution
    image = image.unsqueeze(0)
    kernel = kernel.unsqueeze(0).unsqueeze(0)

    # Apply convolution with Gaussian kernel
    blurred_image = F.conv2d(image, kernel, padding=kernel_size//2)

    # Remove the extra batch dimension and return the blurred image
    return blurred_image.squeeze(0)


def create_gaussian_kernel(kernel_size, sigma):
    # Create a 1D Gaussian kernel
    kernel_1d = torch.exp(-(torch.arange(kernel_size) - kernel_size // 2)
** 2 / (2 * sigma ** 2))

    # Normalize the kernel
    kernel_1d = kernel_1d / kernel_1d.sum()

    # Expand dimensions to create a 2D kernel
    kernel_2d = torch.outer(kernel_1d, kernel_1d)

    return kernel_2d


def H(phi):
    return 0.5 * (1 + 2 * torch.arctan(phi / 0.1) / torch.pi)


def d_dx(f):
    return f[:, 1:] - f[:, :-1]


def d_dy(f):
    return f[1:, :] - f[:-1, :]


def get_pixel_values(image, coordinates=None):
    # Expand dimensions for grid_sample
    image = image.unsqueeze(0)
    if len(image.shape) == 3:
        image = image.unsqueeze(0)

    if coordinates is None:
```

```
        # Create coordinate grid
        grid_x, grid_y = torch.meshgrid(torch.linspace(-1, 1,
image.shape[2]), torch.linspace(-1, 1, image.shape[3]))
        grid = torch.stack((grid_y, grid_x), dim=2).unsqueeze(0)

    else:
        # Normalize coordinates to [-1, 1]
        normalized_coords = coordinates * 2 - 1
        # normalized_coords = coordinates

        # Reshape the normalized coordinates to match the image size
        grid = normalized_coords.view(1, image.shape[2], image.shape[3], 2)

    # Perform grid_sample with bilinear interpolation
    interpolated_values = F.grid_sample(image, grid.to(image.device),
mode='bilinear', padding_mode='border')
    return interpolated_values


def grad(image, coords=None):
    # # Expand dimensions for grid_sample
    # image = image.unsqueeze(0)
    # if len(image.shape) == 3:
    #     image = image.unsqueeze(0)
    #
    # if coords is None:
    #     # Create coordinate grid
    #     grid_x, grid_y = torch.meshgrid(torch.linspace(-1, 1,
image.shape[2]), torch.linspace(-1, 1, image.shape[3]))
    #     grid = torch.stack((grid_y, grid_x), dim=2).unsqueeze(0)
    # else:
    #     # Normalize coordinates to [-1, 1]
    #     normalized_coords = coords * 2 - 1
    #
    #     # Reshape the normalized coordinates to match the image size
    #     grid = normalized_coords.view(1, image.shape[2], image.shape[3],
2)
    #
    # # Perform grid_sample with bilinear interpolation
    # interpolated_values = F.grid_sample(image, grid.to(image.device),
mode='bilinear', padding_mode='border')
    interpolated_values = get_pixel_values(image, coords)

    # Compute gradients using central differences
    gradients_x = (interpolated_values[:, :, 2:, :] -
interpolated_values[:, :, :-2, :]) / 2
    gradients_y = (interpolated_values[:, :, :, 2:] -
interpolated_values[:, :, :, :-2]) / 2

    # pad with zeros to stay in the same shape:
    gradients_x = F.pad(gradients_x, (0, 0, 1, 1), mode='constant',
value=0)
    gradients_y = F.pad(gradients_y, (1, 1, 0, 0), mode='constant',
value=0)

    return torch.cat([gradients_x.squeeze(0), gradients_y.squeeze(0)])


class EoccLoss(nn.Module):
    def __init__(self, epochs):
        super().__init__()
```

```
        weights = torch.linspace(0, 1, epochs)
        self.t = 100 * torch.exp(-weights * torch.log(torch.tensor([100 /
1])))  # 5 * torch.exp(-weights * torch.log(torch.tensor([5 / 1])))
        self.ni = 0.01
        # gaussian_kernel = transforms.GaussianBlur(kernel_size=5,
sigma=0.1)  # size is not mentioned in the paper.
        # self.G = gaussian_kernel.weight.unsqueeze(0).unsqueeze(0)
        self.kernel_size = 21  # size is not mentioned in the paper.
        sigma = 2  # in the paper, sigma=0.1. but this has no effect.
        self.G = create_gaussian_kernel(kernel_size=self.kernel_size,
sigma=sigma)
        self.G = self.G.to(device)

    def forward(self, e_occ, phi, epoch):
        # Eq. 19
        e_occ_map = e_occ.view(im_h, im_w)  # TODO: check correctness of
stacking
        phi = phi.view(im_h, im_w)  # TODO: check correctness of stacking
        e_occ_blurred = apply_gaussian_blur(e_occ_map, self.G,
kernel_size=self.kernel_size)
        H_phi = H(phi)
        out_map = e_occ_blurred * H_phi + self.t[epoch] * (1 - H_phi) +
self.ni * grad(H_phi).norm(p=1, dim=0)
        return out_map.mean()  # TODO: consider .sum(),


class ELLoss(nn.Module):
    def __init__(self, im_l, im_r):
        super().__init__()
        self.eta = 0.001
        self.im_l = im_l
        self.im_r = im_r
        self.lambda_ = 2
        self.beta = 1  # TODO: I dont know what should be the value here
        self.alpha_hat = 100.9375 * self.beta  # paper: 0.9375 * self.beta
        self.eps_hat = 0.25 * self.beta

    # def forward(self, g_l, xy, d_l, phi_l, v_l):
    #     # Eq. 23
    #
    #     # compute e_d
    #
    #     # plt.imshow(g_l[:, 0].view(im_h, im_w).detach()),
plt.colorbar(), plt.show()
    #
    #     Ir_gl = get_pixel_values(self.im_r, g_l)
    #     # plt.imshow(Ir_gl[0].detach().permute(1, 2, 0)), plt.show()
    #     Il_xy_l = get_pixel_values(self.im_l, xy)
    #     # plt.imshow(Il_xy_l[0].detach().permute(1, 2, 0)), plt.show()
    #     grad_Ir_gl = grad(self.im_r, g_l)  # TODO: if takes runtime,
consider calculating the grads images in advance
    #     # plt.imshow(grad_Ir_gl[0].detach()), plt.show()
    #     grad_Il_xy_l = grad(self.im_l, xy)
    #     sd_2 = (Ir_gl - Il_xy_l).norm(p=1) ** 2 + self.lambda_ *
(grad_Ir_gl - grad_Il_xy_l).norm(p=1) ** 2
    #     e_d = (sd_2 + self.eta ** 2).sqrt()
    #
    #     # compute e_s
    #     grad_d_l = grad(d_l.view(im_h, im_w))
    #     e_s = self.beta * (grad_d_l.norm(p=1) ** 2 + self.eta **
2).sqrt()
```

```
    #
    #     v_l = v_l.view(im_h, im_w)
    #     grad_v_l = grad(v_l)
    #     phi_l = phi_l.view(im_h, im_w)
    #
    #     out_map = (e_d + e_s * v_l ** 2 + self.alpha_hat * (v_l - 1) **
2) * H(phi_l) + self.eps_hat * grad_v_l.norm(p=1) ** 2
    #     return out_map.mean(), e_s, e_d

    def forward(self, g, xy, d, phi, v, im_side):
        if im_side == 'left':
            im_src = self.im_l
            im_ref = self.im_r
        else:
            im_src = self.im_r
            im_ref = self.im_l
        I1_xy = get_pixel_values(im_src, xy)
        # plt.imshow(I1_xy[0].detach().permute(1, 2, 0)), plt.show()
        I2_g = get_pixel_values(im_ref, g)
        # plt.imshow(I2_g[0].detach().permute(1, 2, 0)), plt.show()
        grad_I1_xy = grad(im_src, xy)
        grad_I2_g = grad(im_ref, g_l)  # TODO: if takes runtime, consider
calculating the grads images in advance
        # plt.imshow(grad_I2_g[0].detach()), plt.show()

        sd_2 = (I2_g - I1_xy).norm(p=1) ** 2 + self.lambda_ * (grad_I2_g -
grad_I1_xy).norm(p=1) ** 2
        e_d = (sd_2 + self.eta ** 2).sqrt()

        # compute e_s
        grad_d = grad(d.view(im_h, im_w))
        e_s = self.beta * (grad_d.norm(p=1) ** 2 + self.eta ** 2).sqrt()

        v = v.view(im_h, im_w)
        grad_v = grad(v)
        phi = phi.view(im_h, im_w)

        out_map = (e_d + e_s * v ** 2 + self.alpha_hat * (v - 1) ** 2) *
H(phi) + self.eps_hat * grad_v.norm(
            p=1) ** 2
        return out_map.mean(), e_s, e_d




class MyData(Dataset):
    def __init__(self, image_left, image_right, disparity_left,
disparity_right):
        super().__init__()
        self.image_left = np.array(Image.open(image_left), dtype='float32')
        self.image_right = np.array(Image.open(image_right),
dtype='float32')
        self.disparity_left = np.array(Image.open(disparity_left),
dtype='float32').flatten()/4
        self.disparity_left = self.disparity_left /
self.image_left.shape[1]  # convert from pixel coords to [0, 1]
        self.disparity_right = np.array(Image.open(disparity_right),
dtype='float32').flatten()/4
        self.disparity_right = self.disparity_right /
self.image_right.shape[1]
        x, y = np.meshgrid(range(self.image_left.shape[1]),
```

```
range(self.image_left.shape[0]))
        x, y = x.astype('float32'), y.astype('float32')
        self.xy = np.stack([x.flatten() / self.image_left.shape[1],
y.flatten() / self.image_left.shape[0]], axis=1)

        # self.disparity_right = self.disparity_right * 2 - 1
        # self.disparity_left = self.disparity_left * 2 - 1
        # self.xy = self.xy * 2 - 1
        self.to_tensor = transforms.ToTensor()

        # start from right and get the left
        # d = torch.stack([batch['xy'][:, 0] - batch['d_l'], batch['xy'][:,
1]], dim=1)
        # a = get_pixel_values(loss2.im_r, d)
        # plt.imshow(a[0].detach().permute(1, 2, 0)), plt.show()

        # start from left and get the right
        # d = torch.stack([batch['xy'][:, 0] + batch['d_r'], batch['xy'][:,
1]], dim=1)
        # a = get_pixel_values(loss2.im_l, d)
        # plt.imshow(a[0].detach().permute(1, 2, 0)), plt.show()

    def __len__(self):
        return len(self.xy)

    def __getitem__(self, idx):
        # TODO d = disparity_right?
        # TODO x, y in [0,1]?
        xy = torch.from_numpy(self.xy[idx])
        return {'xy': xy, 'd_l': self.disparity_left[idx], 'd_r':
self.disparity_right[idx]}


class Phi(nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.model = nn.Sequential(nn.Linear(2, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                   nn.Linear(hidden_dim, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                   nn.Linear(hidden_dim, 1))

        # self.model = nn.Linear(2, 1)

    def forward(self, xy):
        return self.model(xy)


class Disparity(nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.model = nn.Sequential(nn.Linear(2, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                   nn.Linear(hidden_dim, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                   nn.Linear(hidden_dim, 1), nn.Sigmoid())

        # self.model = nn.Linear(2, 1)
```

```
    def forward(self, xy):
        return self.model(xy)


class Pipeline(nn.Module):
    def __init__(self, eps_occl, hidden_dim=10):
        super().__init__()
        self.eps_occl = eps_occl
        self.hidden_dim = hidden_dim
        self.phi_l = Phi(hidden_dim)
        self.phi_r = Phi(hidden_dim)
        self.d_l_v = Disparity(hidden_dim)
        self.d_r_v = Disparity(hidden_dim)
        self.d_l_o = Disparity(hidden_dim)
        self.d_r_o = Disparity(hidden_dim)
        self.v_l = nn.Sequential(nn.Linear(2, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                 nn.Linear(hidden_dim, 1), nn.Sigmoid())
        self.v_r = nn.Sequential(nn.Linear(2, hidden_dim),
nn.BatchNorm1d(hidden_dim), nn.ReLU(),
                                 nn.Linear(hidden_dim, 1), nn.Sigmoid())

    # def forward(self, batch):
    #     phi_l_xy_l = self.phi_l(batch['xy'])
    #     H_l = H(phi_l_xy_l)
    #     dl = self.d_l_v(batch['xy']) * H_l + \
    #          self.d_l_o(batch['xy']) * (1 - H_l)
    #
    #     gl = torch.cat([batch['xy'][:, [0]] - dl, batch['xy'][:, [1]]],
dim=1)
    #     # g_l = torch.cat([dl, batch['xy'][:, [1]]], dim=1)
    #     H_r = H(self.phi_r(gl))
    #     dr_gl = self.d_r_v(gl) * H_r + \
    #          self.d_r_o(gl) * (1 - H_r)
    #
    #     # plt.imshow(dl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
    #     # plt.imshow(dr_gl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
    #     ul = dl + (-dr_gl)  # different from the paper since here the
base corrds are the same (xy_l == xy_r)
    #     e_occ_l = -torch.log(self.eps_occl + (1 - self.eps_occl) *
torch.exp(-torch.abs(ul)))
    #
    #     v_l = self.v_l(batch['xy'])
    #     return e_occ_l, g_l, phi_l_xy_l, d_l, v_l

    def forward(self, batch):
        return self.calculate_aux(batch)

    def calculate(self, xy, phi1, d_v1, d_o1, phi2, d_v2, d_o2, im_side):
        phi_xy = phi1(xy)
        H_phi = H(phi_xy)
        d = d_v1(xy) * H_phi + \
             d_o1(xy) * (1 - H_phi)

        if im_side == 'left':
            g = torch.cat([xy[:, [0]] - d, xy[:, [1]]], dim=1)
        else:
            g = torch.cat([xy[:, [0]] + d, xy[:, [1]]], dim=1)
```

```
        # g_l = torch.cat([dl, batch['xy'][:, [1]]], dim=1)
        H_g = H(phi2(g))
        dg = d_v2(g) * H_g + \
                d_o2(g) * (1 - H_g)

        # plt.imshow(d.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        # plt.imshow(dg.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        u = d + (-dg)  # different from the paper since here the base
corrds are the same (xy_l == xy_r)
        e_occ = -torch.log(self.eps_occl + (1 - self.eps_occl) *
torch.exp(-torch.abs(u)))

        return e_occ, g, phi_xy, d

    def calculate_aux(self, batch):
        e_occ_l, gl, phi_l_xy, dl = self.calculate(batch['xy'], self.phi_l,
self.d_l_v, self.d_l_o,
                                                   self.phi_r, self.d_r_v,
self.d_r_o, 'left')
        e_occ_r, gr, phi_r_xy, dr = self.calculate(batch['xy'], self.phi_r,
self.d_r_v, self.d_r_o,
                                                   self.phi_l, self.d_l_v,
self.d_l_o, 'right')
        vl = self.v_l(batch['xy'])
        vr = self.v_r(batch['xy'])

        return e_occ_l, gl, phi_l_xy, dl, vl, e_occ_r, gr, phi_r_xy, dr, vr

    def forward2(self, batch):
        phi_l_xy = self.phi_l(batch['xy'])
        Hl = H(phi_l_xy)
        dl = self.d_l_v(batch['xy']) * Hl + \
             self.d_l_o(batch['xy']) * (1 - Hl)

        gl = torch.cat([batch['xy'][:, [0]] - dl, batch['xy'][:, [1]]],
dim=1)
        # g_l = torch.cat([dl, batch['xy'][:, [1]]], dim=1)
        Hr_gl = H(self.phi_r(gl))
        dr_gl = self.d_r_v(gl) * Hr_gl + \
             self.d_r_o(gl) * (1 - Hr_gl)

        # plt.imshow(dl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        # plt.imshow(dr_gl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        ul = dl + (-dr_gl)  # different from the paper since here the base
corrds are the same (xy_l == xy_r)
        e_occ_l = -torch.log(self.eps_occl + (1 - self.eps_occl) *
torch.exp(-torch.abs(ul)))

        vl = self.v_l(batch['xy'])

        ################################################################
        # Symmetric direction:
        ################################################################
        phi_r_xy = self.phi_r(batch['xy'])
        Hr = H(phi_r_xy)
        dr = self.d_r_v(batch['xy']) * Hr + \
             self.d_r_o(batch['xy']) * (1 - Hr)
```

```
        gr = torch.cat([batch['xy'][:, [0]] + dr, batch['xy'][:, [1]]],
dim=1)
        # g_l = torch.cat([dl, batch['xy'][:, [1]]], dim=1)
        Hl_gr = H(self.phi_l(gr))
        dl_gr = self.d_l_v(gr) * Hl_gr + \
                self.d_l_o(gr) * (1 - Hl_gr)

        # plt.imshow(dl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        # plt.imshow(dr_gl.view(im_h, im_w).detach()), plt.colorbar(),
plt.show()
        ur = dr + (-dl_gr)  # different from the paper since here the base
corrds are the same (xy_l == xy_r)
        e_occ_r = -torch.log(self.eps_occl + (1 - self.eps_occl) *
torch.exp(-torch.abs(ur)))

        vr = self.v_r(batch['xy'])

        return e_occ_l, e_occ_r, gl, gr, phi_l_xy, phi_r_xy, dl, dr, vl, vr


######
# im_l(xy + d_r) == im_r
# im_r(xy_r - d_l) == im_l
######


seed = 7
torch.manual_seed(seed)
np.random.seed(seed)

# device = 'cuda' if torch.cuda.is_available() else 'cpu'
device = 'cpu'
data_name = 'teddy'  # 'cones'  # 'teddy'


dataset = MyData(f'dataset/{data_name}/im_l.png',
f'dataset/{data_name}/im_r.png',
                 f'dataset/{data_name}/disp_l.png',
f'dataset/{data_name}/disp_r.png')

im_w, im_h = 450, 375
batch_size = im_w * im_h
epochs = 10000
lr = 1
hidden_dim = 10

data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=False,
drop_last=False)
im_l = Image.open(f'dataset/{data_name}/im_l.png')
im_r = Image.open(f'dataset/{data_name}/im_r.png')
to_tensor = transforms.ToTensor()

eps_occl = 0.005  # 0.005 # 0.01 # 0.02
pipe = Pipeline(eps_occl=eps_occl, hidden_dim=hidden_dim).to(device)

optim = torch.optim.SGD(pipe.parameters(), lr=lr)
scheduler = torch.optim.lr_scheduler.MultiStepLR(optim, [100, 500, 1000],
0.5)
loss1 = EoccLoss(epochs)
```

```
loss2 = ELLoss(to_tensor(im_l), to_tensor(im_r))

best_loss = torch.inf
for epoch in range(epochs):
    for batch in data_loader:
        xy = batch['xy'].to(device)

        # update phi:
        e_occ_l, g_l, phi_l, d_l, v_l, e_occ_r, g_r, phi_r, d_r, v_r =
pipe(batch)

        # using GT:
        # d_l = batch['d_l'].unsqueeze(1)
        # g_l = torch.cat([xy[:, [0]] - d_l, xy[:, [1]]], dim=1)
        # d_r = batch['d_r'].unsqueeze(1)
        # g_r = torch.cat([xy[:, [0]] + d_r, xy[:, [1]]], dim=1)

        plt.imshow(d_l.detach().view(im_h, im_w), cmap='gray'),
plt.title('d_l'), plt.show()
        plt.imshow(d_r.detach().view(im_h, im_w), cmap='gray'),
plt.title('d_r'), plt.show()

        l1_l = loss1(e_occ_l, phi_l, epoch)
        l1_r = loss1(e_occ_r, phi_r, epoch)
        l2_l, es_l, ed_l = loss2(g_l, xy, d_l, phi_l, v_l, 'left')
        l2_r, es_r, ed_r = loss2(g_r, xy, d_r, phi_r, v_r, 'right')
        print(f'Ep: {epoch}\tL1-L: {l1_l:.2f}\tL1-R: {l1_r:.2f}\t'
              f'L2-L: {l2_l:.2f}\tL2-R: {l2_r:.2f}\t'
              f'es_l: {es_l:.2f}\tes_r: {es_r:.2f}\t'
              f'ed_l: {ed_l:.0f}\ted_r: {ed_r:.0f}\t'
              f'phi_l: {phi_l.mean():.2f}\tphi_r: {phi_r.mean():.2f}\t'
              f'd_l: {d_l.mean():.2f}\td_r: {d_r.mean():.2f}'
              )

        # if epoch % 10 == 0:
        #     loss = l2 / 50
        # else:
        #     loss = l1

        # loss = l2
        # loss = l1 + 0.0000385 * l2
        # loss = l1 + l2
        loss = l1_l + l1_r + (l2_l + l2_r) / 1000

        # if epoch < 100:
        #     loss = l1
        # else:
        #     loss = l1 + l2


        optim.zero_grad()
        loss.backward()
        optim.step()
        scheduler.step()

    if epoch > 0 and epoch % 1000 == 0 and loss < best_loss:
        best_loss = loss
        torch.save(pipe.state_dict(), f'checkpoints/model/best.pt')
        torch.save(optim.state_dict(), f'checkpoints/optim/best.pt')
        torch.save(scheduler.state_dict(),
f'checkpoints/scheduler/best.pt')
```

```
    # if epoch % 200 == 0:
    #     plt.imshow(d_1.detach().cpu().view(im_h, im_w)), plt.show()
```

References:

[1] D. Mumford and J. Shah, “Optimal Approximations by Piecewise Smooth Functions and Associated Variational Problems,” Comm. Pure and Applied Math., vol. 42, pp. 577-684, 1989.

[2] R. Ben-Ari and N. Sochen, ‘‘Stereo matching with Mumford–Shah regular-ization and occlusion handling,’’ IEEE Trans. Pattern Anal. Mach. Intell., vol. 32, no. 11, pp. 2071–2084, Nov. 2010.