Image Processing and Pattern Recognition—Homework 3

Lukas Glaszner—01430043
Tobias Hamedl—11808141

January 8, 2021

1 Implementation

We implemented the coherence enhancing diffusion algorithm as described on the assignment sheet. Since the structure tensor $S(i, j)$ is a real symmetric matrix $\forall i, j$ we can use `numpy.eigh(S)` to perform the Eigendecomposition. Afterwards we need to flip the array of Eigenvectors along axis 2, because the function sorts for Eigenvalues in ascending order; we do not need to flip the array of Eigenvalues, since only the absolute value of the difference is of interest.

To check whether our implementation works properly, we run the algorithm on the given input image with the default parameters in Table 1; the result can be found in Figure 1. The result we obtain is reasonably similar to the one on the assignment sheet, the minor differences are due to different methods of calculating the derivatives.

We have also implemented `batch_coherence_enhancing_diffusion()`, a function which allows us to specify timesteps at which we would like to save intermediary results.

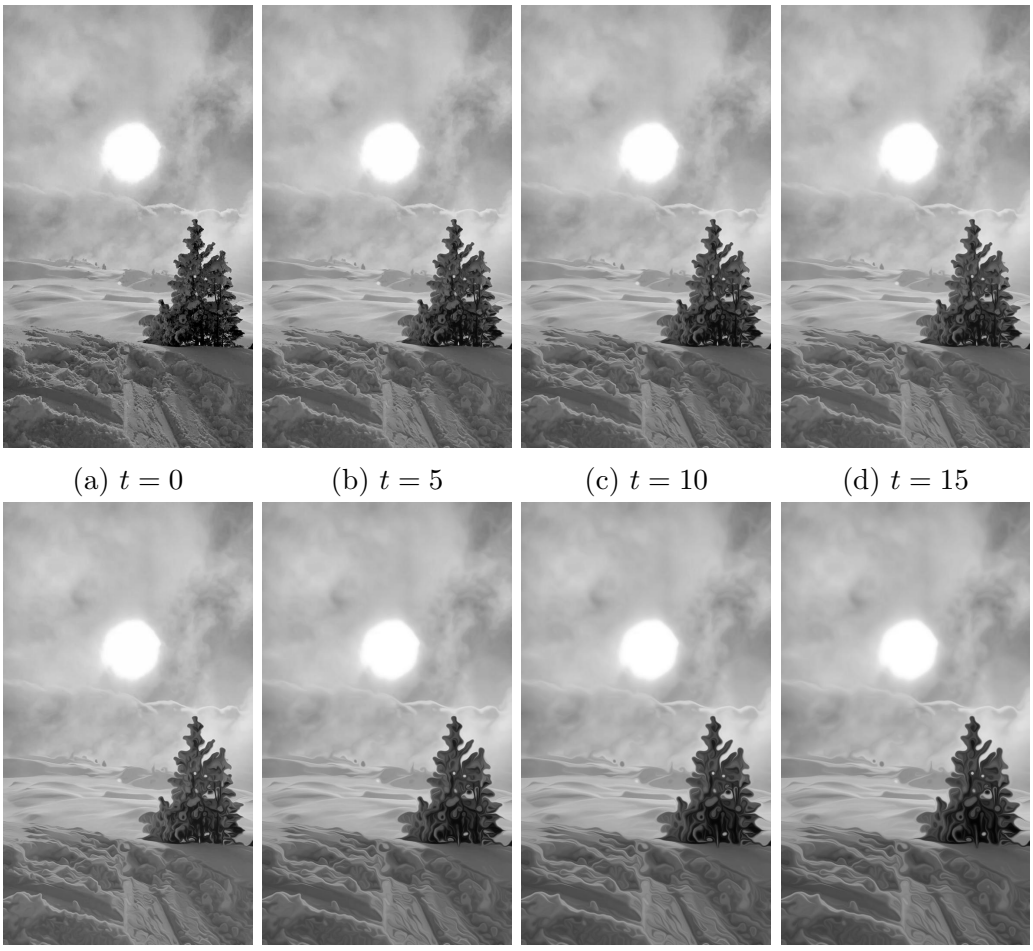
parameter	value
σ_1	0.7
σ_2	1.5
α	0.0005
γ	0.0001
τ	5
t_{end}	100

Table 1: The default parameters

Figure 1: The result of coherence enhancing diffusion on the given input image is very similar to the one on the assignment sheet.

2 Discussion

We now apply the algorithm to our own image taken with an iPhone XS and investigate the influence of various parameters—Figure 2 shows the output after various diffusion times for the parameters in Table 1.



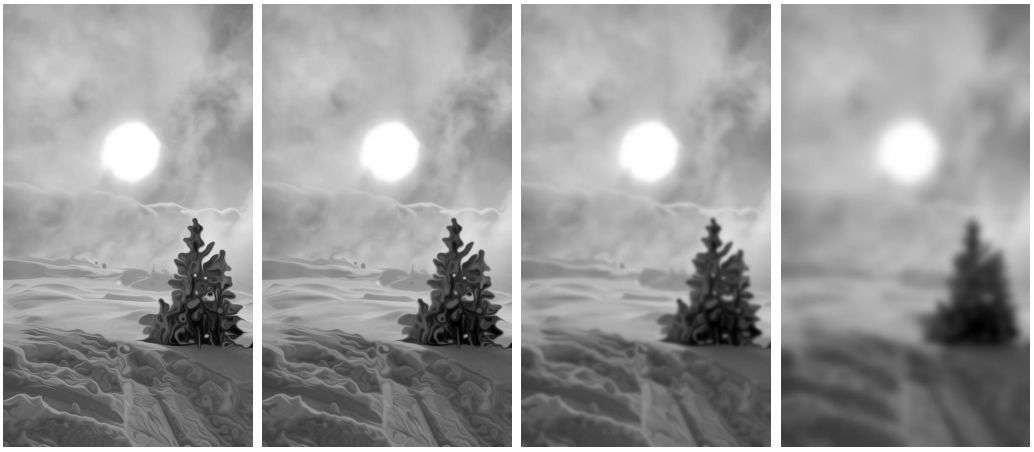
(a) $t = 0$ (b) $t = 5$ (c) $t = 10$ (d) $t = 15$

(e) $t = 20$ (f) $t = 60$ (g) $t = 100$ (h) $t = 140$

Figure 2: The longer the algorithm runs, the more pronounced the effect of coherence enhancing diffusion becomes.

Here we can see how the diffusion process evolves over time (Figure 2a corresponds to the image input). At the beginning fine details start to vanish and over time the typical style of CED becomes more apparent.

To investigate the influence of the parameter α we run the algorithm with a wide range of parameters—the resulting images are presented in Figure 3.



(a) $\alpha = 5 \cdot 10^{-4}$ (b) $\alpha = 5 \cdot 10^{-3}$ (c) $\alpha = 5 \cdot 10^{-2}$ (d) $\alpha = 5 \cdot 10^{-1}$

Figure 3: Any value below $5 \cdot 10^{-4}$ does not seem to have a significant influence; beyond this point, an increase in the value of α leads to more and more image blurring. The image style that created by is CED persists, however.

After performing Eigendecomposition of the per-pixel structure tensor $S(i, j)$, we compute the diffusion tensor as

$$D = \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} v_1^{\mathrm{T}} \\ v_2^{\mathrm{T}} \end{bmatrix}, \tag{1}$$

where

$$\begin{cases} \lambda_1 = \alpha, \\ \lambda_2 = \alpha + (1 - \alpha)\big(1 - g(|\mu_1 - \mu_2|)\big). \end{cases} \tag{2}$$

If there is an edge present—i.e. $\mu_1 \gg \mu_2$—the Eigenvector v_1 points in the direction perpendicular to the edge, while v_2 in the direction along the edge. Thus λ_1 defines the relative amount of diffusion across the edge; λ_2 along it.

This explains the behavior we observe very well: as α increases, the diffusion process becomes less anisotropic, because λ_1 increases and the edge preserving term $\big(1 - g(|\mu_1 - \mu_2|)\big)$ is weighted less in λ_2. In the extreme case of $\alpha = 1$, we would observe homogeneous isotropic diffusion, since the information about edges is completely discarded.

Analogously, we perform coherence enhancing diffusion with various values of γ and present the results in Figure 4.

This parameter can be found in the so-called edge-stopping function

$$g(x) = \exp\left(-\frac{x^2}{2\gamma^2}\right), \tag{3}$$

(a) $\gamma = 10^1$ (b) $\gamma = 10^2$ (c) $\gamma = 10^3$ (d) $\gamma = 10^4$

Figure 4: For any value $\gamma \leq 10$ the result is more or less the same; if it increases more, we can see that more fine details of the input image are preserved. This can be best observed at the branches of the trees. For very large values of γ the output resembles the input very closely.

where we recall that x is the difference between the two Eigenvalues. First we need to discuss how g influences the result: If g is close to one, there is little more diffusion in v_2 than in v_1. If g is close to zero, however, the second term in λ_2 becomes larger and there will be an increase in diffusion along v_2. These two cases occur if the numerator is much smaller than the denominator and if it is much larger than the denominator, respectively. It follows that γ is the threshold in the difference of the Eigenvalues is that controls what is seen as an edge.

If *gamma* is very large, however, g will be close to one for all pixels—thus $\lambda_2 \approx \lambda_1 = \alpha$, if alpha is very small, as is the case in the example, there will be very little diffusion at all. Therefore, the output will closely resemble the input, as is the case in Figure 4d.

To answer the question what filter kernel `spnabla_hp()` implements, we first look at the code to see what the returned matrix looks like. Since it simply combines `spnabla_x_hp()` and `spnabla_y_hp()` we look at the gradient in x-direction first:

$$\begin{bmatrix} -0.5 & 0.5 & & & & \cdots & -0.5 & 0.5 & & & & \\ & -0.5 & 0.5 & & & \cdots & & -0.5 & 0.5 & & & \\ & & \ddots & \ddots & & \cdots & & & \ddots & \ddots & & \\ & & & 0 & 0 & \cdots & & & & 0 & 0 & \\ & & & & \ddots & \ddots & & & & & \ddots & \ddots \end{bmatrix}, \quad (4)$$

where along the main, 1st, Nth and $(N+1)$th diagonal the $N \times 1$ vector

$$\pm \begin{bmatrix} 0.5 \\ 0.5 \\ \vdots \\ 0.5 \\ 0 \end{bmatrix} \tag{5}$$

is repeated $M-1$ times—the last vector is a zero vector of size N. The matrix product with the flattened image leads to

$$\nabla_x U^t = 0.5 \begin{bmatrix} U^t_{0,1} - U^t_{0,0} + U^t_{1,1} - U^t_{1,0} \\ U^t_{0,2} - U^t_{0,1} + U^t_{1,2} - U^t_{1,1} \\ \vdots \\ 0 \\ U^t_{1,1} - U^t_{1,0} + U^t_{2,1} - U^t_{2,0} \\ \vdots \end{bmatrix}, \tag{6}$$

where we can see that this is the rotation invariant derivative presented in the lecture. From this expression it also directly follows that the corresponding convolution kernel for a 2D convolution is

$$h_{\nabla_x} = 0.5 \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & -1 & 1 \end{bmatrix}; \tag{7}$$

analogously, the filter kernel for the directional derivative in y direction is

$$h_{\nabla_y} = 0.5 \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & 1 & 1 \end{bmatrix}. \tag{8}$$