Image & Video Processing

Assignment 2 - Local Operations

Albert Cerfeda

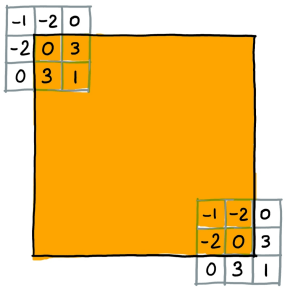
Contents

2 April 2023
Università della Svizzera italiana
Faculty of Informatics
Switzerland

1 Spatial Filtering [2 points]

Maximum and minimum values of the convoluted image.

We know that that each pixel is encoded with in a 6-bit integer format. That is, each pixel has 2^6 possible values and the minimum and maximum values representable are 0 and $2^6 - 1 = 63$ respectively.
To compute the max and min values of the convoluted image, let us consider the case where the filter is getting applied to the corners of the image. Let us employ the padding technique for dealing with pixels outside the bounds of the image (i.e we add artificial pixels with value 0).
If we consider an image where every pixel has value 63 as a result of the convolution the pixel in the top-left corner will have a value of $(0) * 63 + (3) * 63 + (3) * 63 + (1) * 63 = 441$ while the pixel in the bottom-right corner will have a value of $(-1) * 63 + (-2) * 63 + (-2) * 63 + (0) * 63 = -315$.



Post-filtering pixel transformation.
Knowing that each convoluted pixel ranges from -315 to 441, we can apply a linear transformation to the pixel values. By mapping each pixel intensity by mapping it $x \mapsto (x + 315)(\frac{63}{441-(-315)})$ we will map the pixel values to the range $[0, 63]$.

Separability of filter H

2 Combining linear operations [2 points]

We know that the unsharp masking operation can be represented with the following formula: i.e we add back to

$$g_{\text{sharp}} = f + \gamma\,(f - h_{\text{blur}} \star f)$$

Figure 1: Unsharp masking operation

the image the details obtained by subtracting the blurred image from the original image.

To compute the kernel reponsible for blurring the image, we can use the Gaussian filter. The gaussian filter computes a weighted average of the pixels in the neighborhood. The weights fall off as the distance increases from the center pixel, according to the gaussian distribution. We can control such gaussian distribution by changing the value of the standard deviation σ (i.e control the amount of blur). The size for the kernel matters: to give a good approximated representation of the gaussian function we choose the kernel size to be $4\sigma + 1$.

We notice how we can translate the unsharp masking operation [Figure 1] into a linear operation between kernels [Figure 2]. Notice how the kernel for the original image is just a zero matrix with a 1 in the center. The kernel for the blurred image is the gaussian filter. The difference between the two kernels is the image details.
Parameter γ serves as a control parameter for determining the amount of sharpening.
The code for computing the unsharp masking kernel is rather simple [Figure 3]. Notice how ideally more details would have been extracted with a higher σ value. It has been kept purposefully low for generating small matrices, in order to fit them in the report. The results are shown in [Figure 4].

3 Morphological operations A [2 points]

Morphological operations are a set of operations that are applied to binary images, in order to alter the represented shapes.
The exercise requires to choose one structuring element and perform multiple morphological operations on the binary in order to obtain the desired result.
By choosing structural element c) we can obtain the desired result by performing erosion and dilation in succession. Results are shown in Figure 5:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \gamma \underbrace{\left(\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Original pixel}} - \underbrace{\begin{bmatrix} 0.00 & 0.01 & 0.02 & 0.01 & 0.00 \\ 0.01 & 0.06 & 0.10 & 0.06 & 0.01 \\ 0.02 & 0.10 & 0.16 & 0.10 & 0.02 \\ 0.01 & 0.06 & 0.10 & 0.06 & 0.01 \\ 0.00 & 0.01 & 0.02 & 0.01 & 0.00 \end{bmatrix}}_{\text{Blurred pixel}} \right)}_{\text{Image details}}$$

(a) Kernel computation for unsharp masking.

$$\sigma = 1, \gamma = 8$$

$$\begin{bmatrix} -0.02 & -0.11 & -0.18 & -0.11 & -0.02 \\ -0.11 & -0.48 & -0.79 & -0.48 & -0.11 \\ -0.18 & -0.79 & 7.70 & -0.79 & -0.18 \\ -0.11 & -0.48 & -0.79 & -0.48 & -0.11 \\ -0.02 & -0.11 & -0.18 & -0.11 & -0.02 \end{bmatrix}$$

(b) Resulting unsharp masking kernel

Figure 2: Computing the unsharp masking kernel

```
% ...
% Create gaussian blur filter with a gamma parameter of 1
sigma = 1;
gamma = 5;
size = 4*sigma + 1;
blur_f = fspecial('gaussian', size, sigma);
% Identity kernel - leaves the pixel untouched
identity_f = zeros(size);
identity_f(ceil(size/2), ceil(size/2)) = 1;
% Unsharp filter
unsharp_filter = identity_f + gamma.*(identity_f - blur_f);
im_sharp = imfilter(im, unsharp_filter);
```

Figure 3: Matlab code computing the unsharp masking kernel.

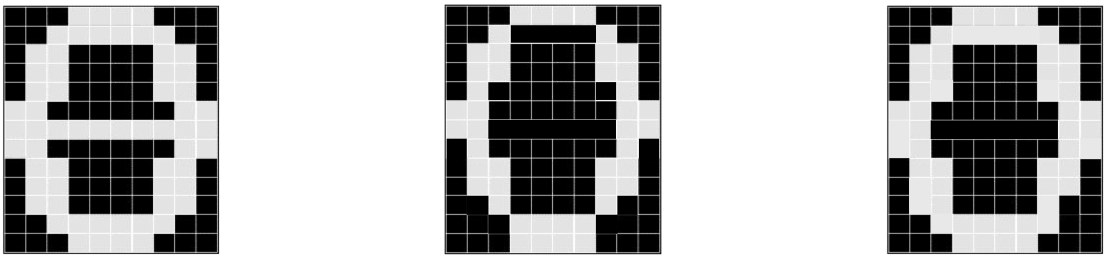
(a) Original image

(b) Sharpened image

Figure 4: Unsharp masking results.

4 Morphological operations B [2 points]

Having just one foreground pixel in the structuring element and applying erosion on a binary image results in the removal of the edges in a particular direction.
It can be easily visualized by thinking of shining a directed light on the shape, and by eroding the pixels that are in shadow.



(a) Original binary image (b) Erosion (c) Erosion + Dilation

Figure 5: Morphological operations using structuring element c)

5 Linear Motion Blur Filter [4 points]

The gaussian function is the same on every axis i.e it falls off equally in every direction. However, in the case of motion blur, each axis can have a separate gaussian function. This allows us to mimic the effect of motion blur. Additionally, we want to provide the angle in degrees for which we rotate the kernel. Let us implement function `gaussianKernel(sigma,angle)` for computing the anisotropic gaussian kernel.

```
function res = gaussianKernel(sigma, angle)
    if size(sigma) == 1
        sigma = [sigma, sigma];
    end
    angle = deg2rad(angle);
    kernelsize = 4*sigma+1;
    kernelsize = abs([cos(angle) sin(angle)]) .* kernelsize(1) + abs([sin(angle) cos(angle)]) .* kernelsize(2);

    [X,Y] = meshgrid(1:kernelsize(2),1:kernelsize(1));
    X = X - kernelsize(2)/2;
    Y = Y - kernelsize(1)/2;
    Xg = X * cos(angle) - Y * sin(angle);
    Yg = X * sin(angle) + Y * cos(angle);

    res = exp(-(Xg.^2/(2*sigma(2)^2) + Yg.^2/(2*sigma(1)^2))) ;
    res = res / sum(res(:));
end
```

Figure 6: Matlab function for computing an anisotropic gaussian kernel.

We can call the function like so:

```
gaussianKernel([10, 1], -45)
```

The function will return a gaussian kernel with $\sigma^y = 10, \sigma^x = 1, \theta = -45°$ i.e a standard deviation of 10 in the y direction and 1 in the x direction. Finally the kernel will be rotated by -45 degrees. [Results are shown in Figure 7c]

A regular isotropic gaussian kernel with $\sigma = 5$ can be obtained in the following way [Results are shown in Figure 7b]:

```
gaussianKernel(5, 0)
```

The function derives the optimal kernel size for the given θ, σ^x and σ^y. It first creates a non-rotated kernel with size $\begin{bmatrix} 4 * \sigma^x + 1 & 4 * \sigma^y + 1 \end{bmatrix}$ after which a "bounding box" containing the rotated kernel is created. This bounding box is then afterwards populated with the kernel values.

6 Iterative filtering [4 points]

Iterative filtering consists in approximating a bigger kernel filter by repeatedly applying a smaller kernel filter. In the case of Gaussian blur, we can perform iterative filtering to approximate a bigger Gaussian kernel.

(a) Original image $\sigma = 0$ (b) Regular isotropic gaussian blur $\sigma = 5$ (c) Anisotropic gaussian blur $\sigma^y = 10, \sigma^x = 1, \theta = -45°$

Figure 7: Gaussian kernel comparison

Gaussian filtering assigns the weights for each pixel based solely on spatial distance. We can intuitevely infer that since the distances between pixels stay the same, if we repeatedly apply a smaller Gaussian filter, the pixel values will eventually converge to the same values as if we had applied a bigger Gaussian filter.

Bilateral filtering additionally takes into consideration the color difference between pixels when assigning weights in order to preserve edges. As the pixel values change for each iteration, the color difference between pixels will also change. Therefore, we can not expect the pixel values to converge to the same values as if we had applied a bigger bilateral filter.

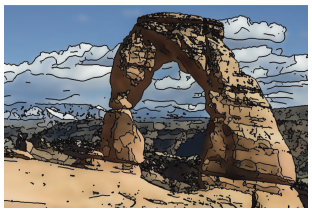
When inspecting an image obtained through iterative filtering with bilateral filtering, we also notice how there is an increase in aliasing artifacts. The cause for the effect is may due to the fact that by repeatedly applying the filter, a greater degree of smoothing is applied resulting in a loss of details. Additionally the bilateral filter tries to preserve edges. This results in sharper edges with a possible increase of aliasing artefacts. Results are shown in Figure 8c.

```
% ...
% Edge preserving bilateral filtering
im_bigbilat = imbilatfilt(im, 0.12, 6);
im_bigbilat2 = im;
for i = 1:45
    im_bigbilat2 = imbilatfilt(im_bigbilat2, 0.012, 0.6);
end
```

(a) Original image (b) `imbilatfilt(im,0.12,6)` (c) 45x `imbilatfilt(im,0.012,0.6)`

Figure 8: Gaussian kernel comparison

6.1 Image Stylization [4 points]



We are tasked in stylizing the image and mimic the appearance of the provided image. There are two noticeable effects present in the stylized image:

- The underlying image is blurred
- The edges of the image content are thickened and blackened

Let us implement the code for performing such stylization on image `delicate_arch.jpg`:

```
% 1. Find edges using Canny algorithm
im_bibilat = imbilatfilt(im, 0.9, 6);
edges = edge(rgb2gray(im_bibilat), 'canny');

% 2. Dilate edges
edges = imdilate(edges, strel('disk', 2));

% 3. Use the edges as a mask to stylize the image
im_stylized = im_bibilat.*~edges;
```

Notice how the Canny[1] algorithm has been used for computing the edges of the image. In my experience using the Canny algorithm yields the best results, as it tends to group regions of similar color values together, resulting in the desired cartoonish effect.

[1] Canny edge detection: Uses linear filtering with a Gaussian kernel to smooth noise and then computes the edge strength and direction for each pixel in the smoothed image. [Sources indiantechwarrior.com, mathworks.com]