A 12-week project in

Image Analysis

by Vedrana Andersen
vedrana@itu.dk
(130274-xxxx)

Project supervisor:
Marleen de Bruijne

IT University of Copenhagen
September – November 2005.

Contents

1 Introduction

Image Analysis was among the courses I wanted to take in the second semester of my studies at the IT University in Copenhagen. But the course was in the autumn semester 2005 combined with the Signal Processing course to form the new Image and Signal Processing course. At the same time, the project cluster in Image Analysis was formed, consisting mostly of students that were planning to attend the Image Analysis course. I decided to join the project cluster and carry out a 12-week project in Image Analysis.

The main goal of the project was to cover the complete curriculum of the old Image Analysis course: reading material, exercises, and assignments. We attended the lessons and exercises of the new Image and Signal Processing course, but we had to compare the old and the new curriculum, read up on the missing parts and do the missing exercises. Around the middle of the semester we started having weekly project cluster meetings to discuss the more advanced topics that were left out in the new course. We studied the materials ourselves and in turn prepared the presentations for the rest of the group.

The supervisor of the project and Image and Signal Processing teacher, Marleen de Bruijne, was exceptionally helpful during the whole project period and always open for suggestions or discussion. Her guidance during the project cluster meetings was invaluable, and many of our dilemmas would stay unsolved without Marleen's explanations.

Topics covered by the Image Analysis project introduced us to the basic concepts, methods and algorithms of digital manipulation, analysis, and understanding of images. In the first couple of lessons we covered the basics of image processing: local and global histograms, gray-scale transformations, spatial and frequency filtering, convolution theorem. In the next meetings we covered one topic (sometimes two) per meeting: restoration, feature detection using scale space, morphology, segmentation, shape and texture, pattern recognition, color, motion. All of the lessons were supplemented by MATLAB exercises, where we learned to use build-in tools and we implemented our own functions.

This project report consists of the selection of exercises I completed during the project period. Three of the exercises were mandatory assignments in the old Image Analysis course curriculum. The rest was selected in such a way that there is one exercise per topic. I presented the exercises in the chronological order, as I was completing them. Most of the exercises consist of a problem formulation, short explanation, solution that includes MATLAB code, generated images and my comments. The depth of the explanation and the extent of my comments vary from exercise to exercise, depending mostly on how interesting I found it to be. There is usually no connection between exercises, so it is not a perfectly coherent material.

Finally, I want to say that the work on the Image Analysis project has more than fulfilled my expectations. Not only I fell I obtained theoretical insight and practical experience on the covered matter, I have also found an area around which I want to focus the rest of my studies.

2 Histogram equalization

From Week 2 — Simple Image Transformations

Enhance the contrast by histogram equalization. Implement your own function and compare it (speed, look of result, histogram) to the built-in MATLAB *function.*

It is often desirable to have an image with the uniform histogram because such images have high contrast and show a great deal of gray-level detail. To obtain the image with the uniform histogram, we can use histogram equalization.

Histogram equalization is a gray-level transformation, i.e. the intensity level of a certain pixel in the processed image depends only on the intensity value of the corresponding pixel in the input image. Gray values are processed using the transformation function $s = T(r)$, which maps pixel value r into pixel value s. Transformation function T for the histogram equalization has to be monotonically increasing, so that the intensities never get inverted. Most importantly, the result of transforming the image using the transformation function T has to produce an image with the uniform histogram.

If we consider the image with the continuous gray level values r in the interval $[0,1]$, and with the probability density function (normalized histogram) $p(r)$, it can be shown (GW [2], page 91) that the the transformation function for the histogram equalization is the cumulative distribution function

$$s = T(r) = \int_0^r p(w)dw$$

In the continuous case the resulting image will always have uniform histogram.

In the case of images with discrete gray level values, the discrete transformation will generally not produce an image with the uniform histogram, but the histogram will be redistributed and spread, achieving the desired enhancement effect.

MATLAB function `histogram_equalization` is my implementation of the histogram equalization and we can compare it with the built-in MATLAB function `histeq`.

```
function HE = histogram_equalization(I)
% takes as an input a gray-scale image (256 gray levels)
% and enhances the contrast by histogram equalization
%---------------------------------------------------------

% finding histogram (shifted by 1, for Matlab's sake)
for i = 1:256;
    h(i) = length(find(I==i-1));
end

% finding cumulative distribution function
CDF(1) = h(1);
for i = 2:256;
```

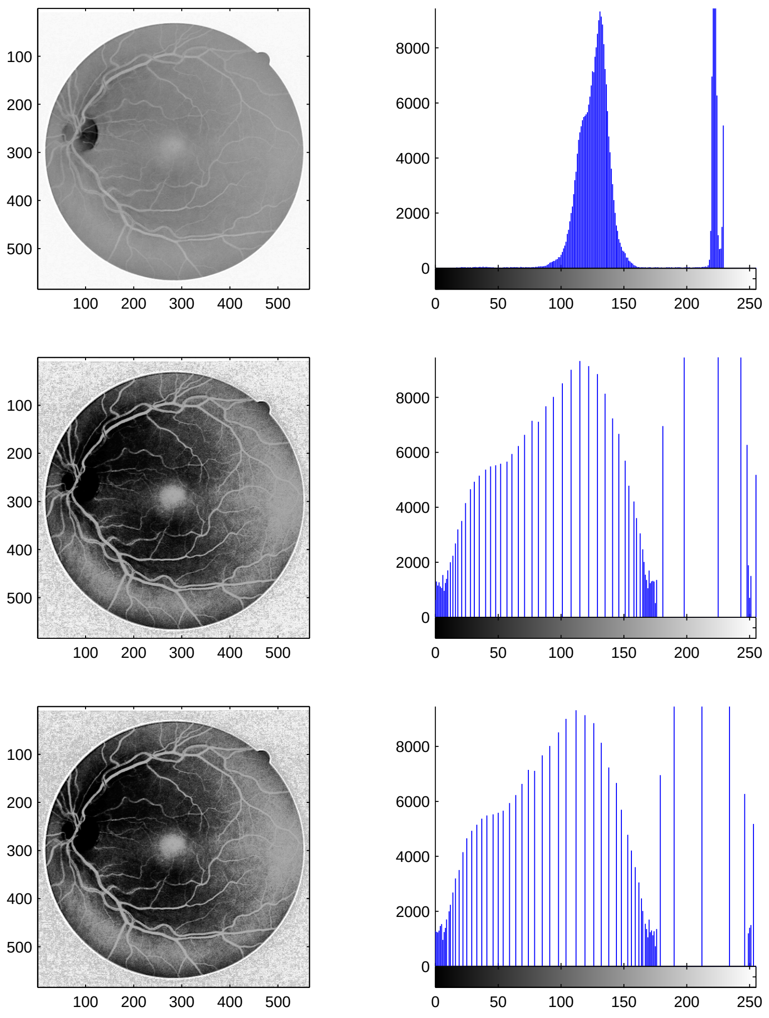


Figure 1: From top to bottom: original image of the retina, image enhanced using my `histogram_equalization` function and image enhanced using built-in MATLAB function `histeq`. Corresponding histograms on the right.

```
    CDF(i) = CDF(i-1)+h(i);
end

% equalizing and scaling
s = 255/CDF(256);
HE = uint8(CDF(I+1)*s);
```

On the figure 1 we have the original image and the images obtained using the two histogram equalization functions, together with the corresponding histograms. When using the built-in MATLAB function `histeq` I chose the number of gray values of the output image to be 256, in order to have comparable results. The default value is 64, which results with flatter histogram. Let's first look at the effect of histogram equalization, and than compare the two functions.

The original image is rather light with just a small dark area. Histogram equalization has darkened the image and the overall contrast has increased. There are some drawbacks too—the small area that was very dark in the original image is in the equalized image blended with the background.

Comparing the histograms of the original and the equalized image, we can see that the histogram equalization in the discrete case acts as a kind histogram spreading. Gray levels that are highly represented will be more spread than the levels with just a few pixels, so we have approximately equal number of pixels in the gray level intervals of the same length.

There is no visible difference between the image equalized with my function and the one equalized with built-in MATLAB function `histeq`. The histograms of those two images are very similar, but not the same—look for example the region around gray value of 250 and the three high spikes between the value 200 and 230. MATLAB does spread the gray levels better. The most significant difference is in the speed of the two functions. My histogram equalization took 5.9 seconds, while it took only 0.05 seconds to perform the built-in MATLAB function `histeq`. MATLAB is 100 times faster!

3 Averaging Filter

from week 3: Spatial Filtering and Fourier Analysis

> *Implement an averaging filter in* MATLAB*. Use* `cameraman.tif` *standard test image for displaying the result of applying the filter. Do not use the built-in* MATLAB *filter functions (such as* `filter2` *and* `conv2`*). Let the size of the filter be a parameter for a function that does the filtering. Measure the run-time for the function when the filter size is varied. Plot the run-times (possibly using* `tic` *and* `toc`*).*

Averaging filter is an example of spatial filtering. The gray-level value of each pixel in filtered image is calculated from gray-level values of pixels in the certain neighborhood of the corresponding pixel in the input image. The size of this neighborhood is the size of the filter.

In case of linear spatial filtering, the gray-level in the filtered image is the linear combination of the gray-levels in the neighborhood of the original image. The coefficients of this linear combination are the filter coefficients. Linear filtering can be visualized as moving a filter mask containing the filter coefficients across the image, multiplying the filter coefficients with the pixel values and summing those products to obtain the new pixel value.

Averaging filter is linear, and all the coefficients of the averaging filter are $\frac{1}{k^2}$, where $k \times k$ is the size of the filter. When using this filter, the value of each pixel is replaced by the average of the gray values in the neighborhood.

The effect of averaging filtering is smoothing and blurring the image. It can be used for noise reduction, but the edges will also get blurred.

MATLAB function `averaging_filter` is my implementation of the averaging filter. It uses zero-padding to treat the problem of the image borders, so the filtered image is of the same size as the original. We can look at the results of applying it to the standard `cameraman.tif` image.

```
function AF = averaging_filter(I,k)
% takes as an input a gray-scale image (256 gray levels)
% and filter size k (must be odd)
%--------------------------------------------------------

% zero padding
[m,n] = size(I);
e = (k-1)/2;
I = [zeros(e,n+2*e); zeros(m,e), I, zeros(m,e); zeros(e,n+2*e)];

% averaging, rounding and cropping to original size
for i = (e+1):(e+m)
    for j = (e+1):(e+n)
        AF(i-e,j-e) = uint8((1/k^2)*...
                        sum(sum(I((i-e):(i+e),(j-e):(j+e)))));
    end
end
```

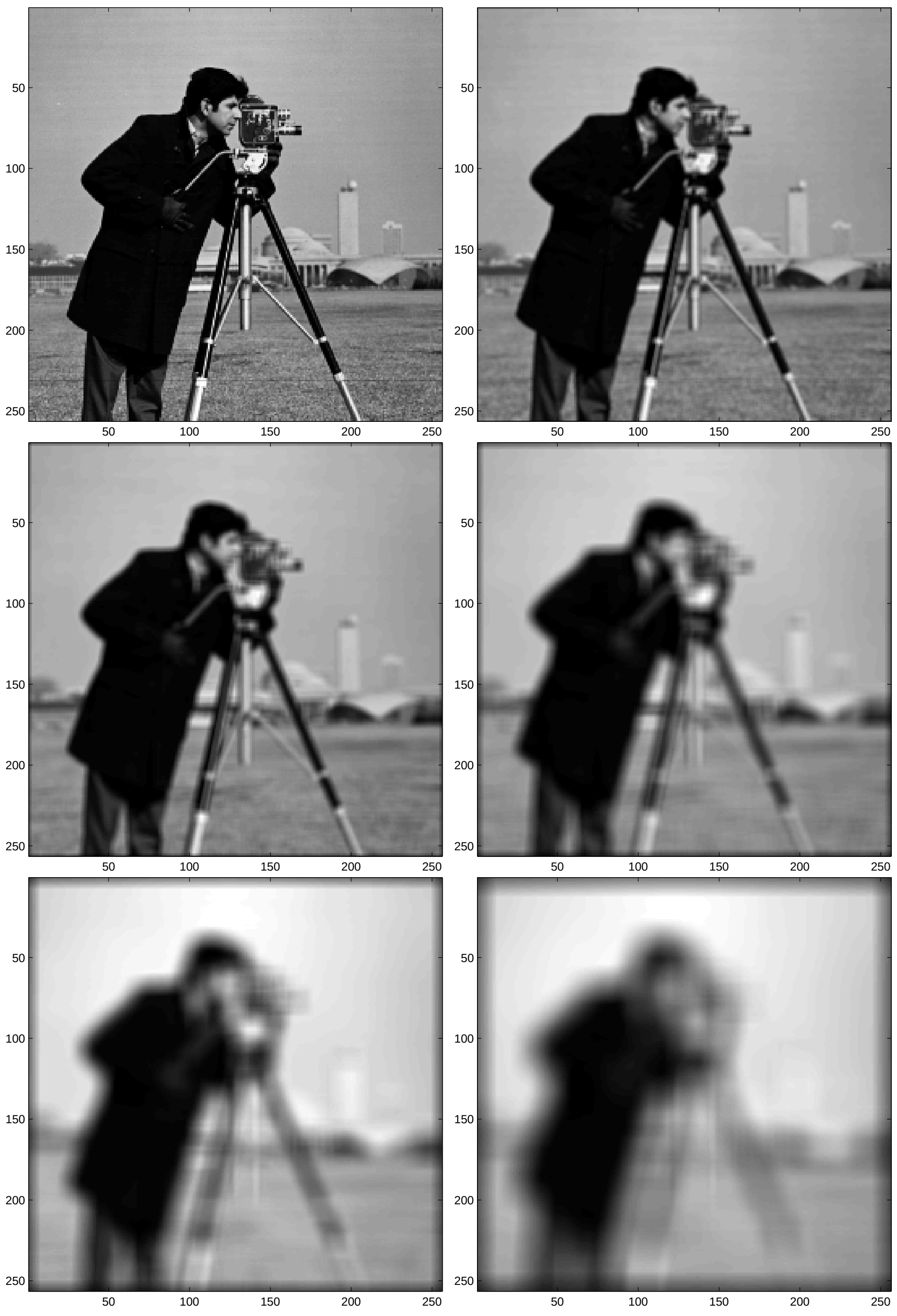


Figure 2: Left to right, top to bottom: original image, of size 256×256 pixels, and images smoothed using `averaging_filter` function with the filter sizes $k = 3, 5, 9, 1, 25$ pixels.

Figure 2 shows original image and the images obtained using `averaging_filter` function with the filter sizes $k = 3, 5, 9, 1, 25$ pixels. We can see that the image is more and more blurred as the filter size increases. The details that are small with respect to the filter size are lost.

On the figure 3 we can see the run-times of the function `averaging_filter` when the filter size k is varied. We can see that it is pretty time-consuming procedure, and that the run-time increases dramatically when the filter grows in size.

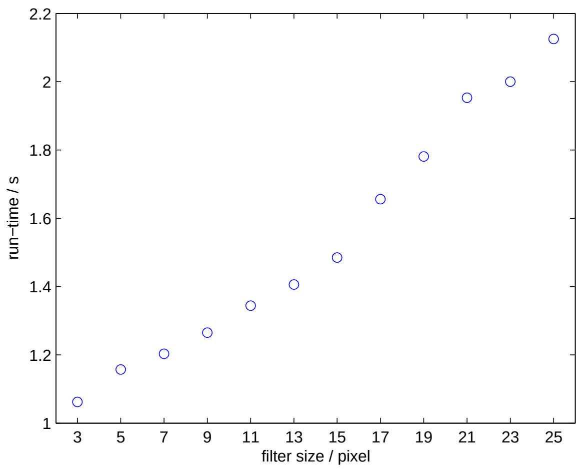


Figure 3: The plot of the run-times of the MATLAB function `averaging_filter.m` when the filter size k is varied.

4 Fourier Transform

from week 3: Spatial Filtering and Fourier Analysis

> *Recreate the illustration of the Fourier transformation of the box function from figure 4.3 in GW [2] (also shown in the lecture slides). Feel free to use the auxiliary functions you desire.*

We need to recreate an image of a 20×40 white rectangle on a black background of size 512×512 pixels, together with it's centered Fourier spectrum shown after application of the log transformation. This can be done by using the build-in MATLAB functions `fft2` and `fftshift`, as in the following commands.

```
n = 512;
box = zeros(n,n);
box(n/2-10:n/2+11,n/2-20:n/2+21) = 255;

FTbox = fftshift(fft2(box));

figure(1), imagesc(box), colormap gray, axis image
figure(2), imagesc(log(abs(FTbox)+1)), colormap gray, axis image
```

The function `fft2` returns the two-dimensional Fourier transform of the input matrix (image). The function `fftshift` shifts the zero-frequency component to center of spectrum. For matrices (images), `fftshift` swaps the first and third quadrants and the second and fourth quadrants. It is useful for visualizing the Fourier transform with the zero-frequency component in the middle of the spectrum.

Recreated images are shown in the figure 4. The spectrum vas processed prior to displaying by using the log transformation to enhance gray-level detail.

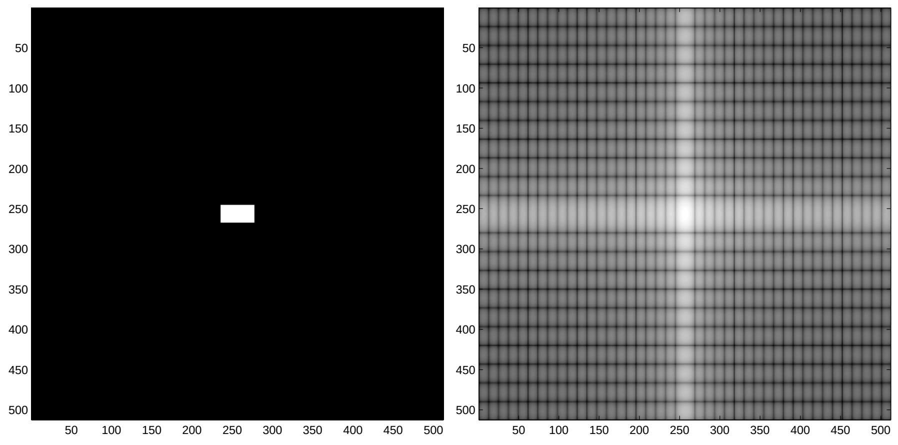


Figure 4: Image of a 20×40 white rectangle on a black background of size 512×512 pixel, together with it's Fourier spectrum.

5 Homomorphic Filter

from week 4: Frequency Domain Filtering

Implement a homomorphic filter in MATLAB*. Use formula 4.5-13 from GW [2] for the specific definition of the filter. Your filter should have value 0.25 at the center and then increase linearly to 1 in a distance D_0 from the origin. Test your homomorphic filter on the* `cameraman.tif` *image from the* MATLAB *image toolbox collection. Can you enhance details in the image?*

According to the illumination-reflectance modes, an image $f(x,y)$ can be expressed as the product of illumination and reflectance components. The goal of the homomorphic filtering is to gain control over illumination and reflectance components, for example to amplify the contribution of reflectance (high frequencies), and to decrease the contribution of illumination (low frequencies).

The central part of homomorphic filtering is homomorphic filter function that affects the low- and high-frequency components of the Fourier transform in a different way. To make it possible for homomorphic filter to operate separately on the components of illumination and reflectance, a logarithm of the original image is taken prior to filtering, since logarithm breaks products into sums. The final result is obtained by exponential operation after filtering. The whole sequence of operations can be illustrated as

$$f(x,y) \Rightarrow \boxed{\ln} \Rightarrow \boxed{\text{DFT}} \Rightarrow \boxed{H(u,v)} \Rightarrow \boxed{(\text{DFT})^{-1}} \Rightarrow \boxed{\exp} \Rightarrow g(x,y)$$

For the homomorphic filter we can use a modified form of the Gaussian highpass filter given by

$$H(u,v) = (\gamma_H - \gamma_L)[1 - e^{-c(D^2(u,v)/D_0^2)}] + \gamma_L$$

where parameters γ_L and γ_H are gains at low and high frequencies, D_0 is the cutoff frequency and $D(u,v)$ is the distance from the point(u,v) to the center of the Fourier

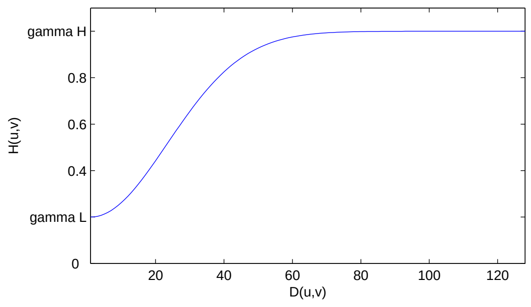


Figure 5: Cross section of a homomorphic filter function for $\gamma_L = 0.2$, $\gamma_H = 1$ and $c = 10$. $D(u,v)$ is the distance from the origin of the centered Fourier transform.

transform

$$D(u,v) = \sqrt{(u - M/2)^2 + (v - N/2)^2}$$

Cross section of such a filter is shown at the figure 5

MATLAB function `homomorphic_filtering` is my implementation of the homomorphic filtering. The results of applying it to the standard `cameraman.tif` image can be seen at the figure 6. It has succeeded in enhancing the details in the image—look for example at cameraman's coat.

```
function FI = homomorphic_filtering(I,d,gl,gh,c)
% Filters image I using homomorphic filtering, where:
% d  - cutoff distance,
% gl - low-frequency response,
% gh - high-frequency response,
% c  - sharpness of the cutoff slope.
%-----------------------------------------------

% creating filter
[m,n] = size(I);
H = zeros(m,n);
cm = floor(m/2+1);
cn = floor(n/2+1);
for u=1:m
    for v=1:n
        H(u,v) = (gh-gl)*(1-exp(-c*(((u-cm)^2+(v-cn)^2)/d^2)))+gl;
    end
end

% performing filtering
FI = exp((ifft2(ifftshift(fftshift(fft2(log(double(I)))).*H))));
```

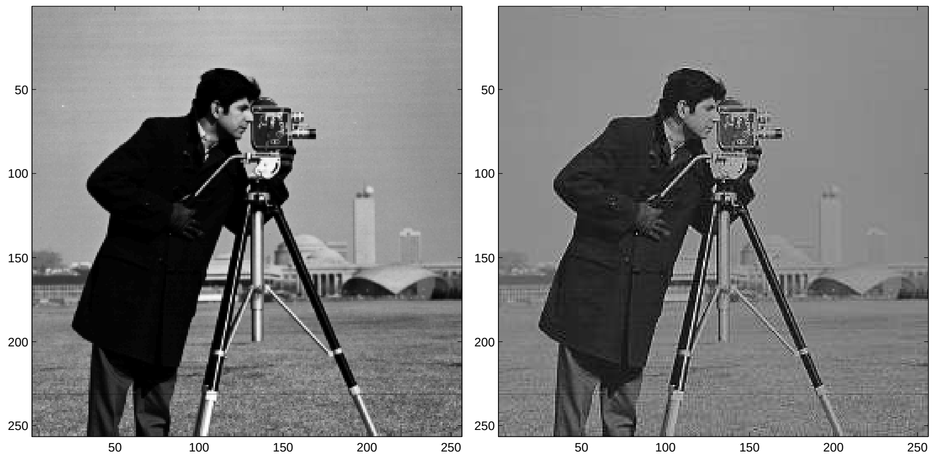


Figure 6: An example of homomorphic filtering. Left: original image, right: the result of applying the homomorphic filter with $\gamma_L = 0.25$, $\gamma_H = 1$, $D_0 = 150$ and $c = 1$.

6 Fourier Analysis, Phase, and Magnitude

First Mandatory Assignment

As the first lectures and exercises have shown, images can be represented in the Fourier domain. In this domain, there exist simple formulations for several image analysis methods—examples include among others Butterworth filter for noise reduction, or homomorphic filters for contrast enhancement. Both these filters are zero-phase filters, i.e. they modify real and imaginary parts of the frequency domain in exactly the same manner.

This assignment explores the role of the phase further. Instead of representing the complex numbers in the frequency domain by a real and an imaginary part, they can be represented by the phase and magnitude (GW [2], equations 4.2-11 and 4.2-10, page 152). The task is to pick up two images of the same size, and do the following in MATLAB*:*

- *Calculate the phase and magnitude description of the Fourier representation.*
- *Reconstruct the original images from the phase and magnitude information exclusively.*
- *Switch the phase and magnitude information between the two images and reconstruct these mixed images.*

Mathematical explanation

The discrete Fourier transform of an image $f(x,x)$ of size $M \times N$ is given by

$$F(u,v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(ux/M-vy/N)}$$

for $u = 0,1,2\ldots M-1, v = 0,1,2\ldots N-1$. It is a sum of $M \times N$ elements, each of which is a complex number because

$$e^{j\phi} = \cos\phi + j\sin\phi$$

So, the result of the summation will also be a complex number, and we can write

$$F(u,v) = R(u,v) + jI(u,v)$$

where $R(u,v)$ and $I(u,v)$ are real and imaginary parts of $F(u,v)$, given by

$$R(u,v) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos(2\pi(ux/M - vy/N))$$

$$I(u,v) = -\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\sin(2\pi(ux/M - vy/N))$$

We can now calculate magnitude $|F(u,v)|$ and phase $\phi(u,v)$ of Fourier transform

$$|F(u,v)| = \sqrt{R^2(u,v) + I^2(u,v)}$$

$$\phi(u,v) = \tan^{-1}\left[\frac{I(u,v)}{R(u,v)}\right]$$

which leads us to the Fourier transform written in polar form

$$F(u,v) = |F(u,v)|e^{j\phi(u,v)}$$

Having magnitude and phase, we can get real and imaginary part of the Fourier transform

$$R(u,v) = |F(u,v)|\cos\phi(u,v)$$

$$I(u,v) = |F(u,v)|\sin\phi(u,v)$$

Given $F(u,v)$, we can get back to original image using the inverse discrete Fourier transform

$$f(u,v) = \sum_{u=0}^{M-1}\sum_{v=0}^{N-1} F(u,v)e^{j2\pi(ux/M - vy/N)}$$

for $x = 0,1,2\ldots M-1$,$y = 0,1,2\ldots N-1$.

MATLAB code

Reconstructing image just from phase or magnitude, and mixing magnitude and phase description can be done in MATLAB using following commands.

```
C = imread('cameraman.tif');
R = imread('retina_resized.bmp');

% calculating Fourier representations
CF = fftshift(fft2(double(C)));
Rf = fftshift(fft2(double(R)));

% reconstructing using just magnitude or just phase
Cmag = real(ifft2(ifftshift(abs(CF))));
Cpha = real(ifft2(ifftshift(exp(i*angle(CF)))));
Rmag = real(ifft2(ifftshift(abs(Rf))));
Rpha = real(ifft2(ifftshift(exp(i*angle(Rf)))));

% reconstructing mixed images
CmagRpha = real(ifft2(ifftshift(abs(CF).*exp(i*angle(Rf)))));
RmagCpha = real(ifft2(ifftshift(abs(Rf).*exp(i*angle(CF)))));
```

Illustrations of selected original images, their reconstructions form the phase and magnitude description, and the two images reconstructed from the mixed phase and magnitude description can be seen at the figure 7

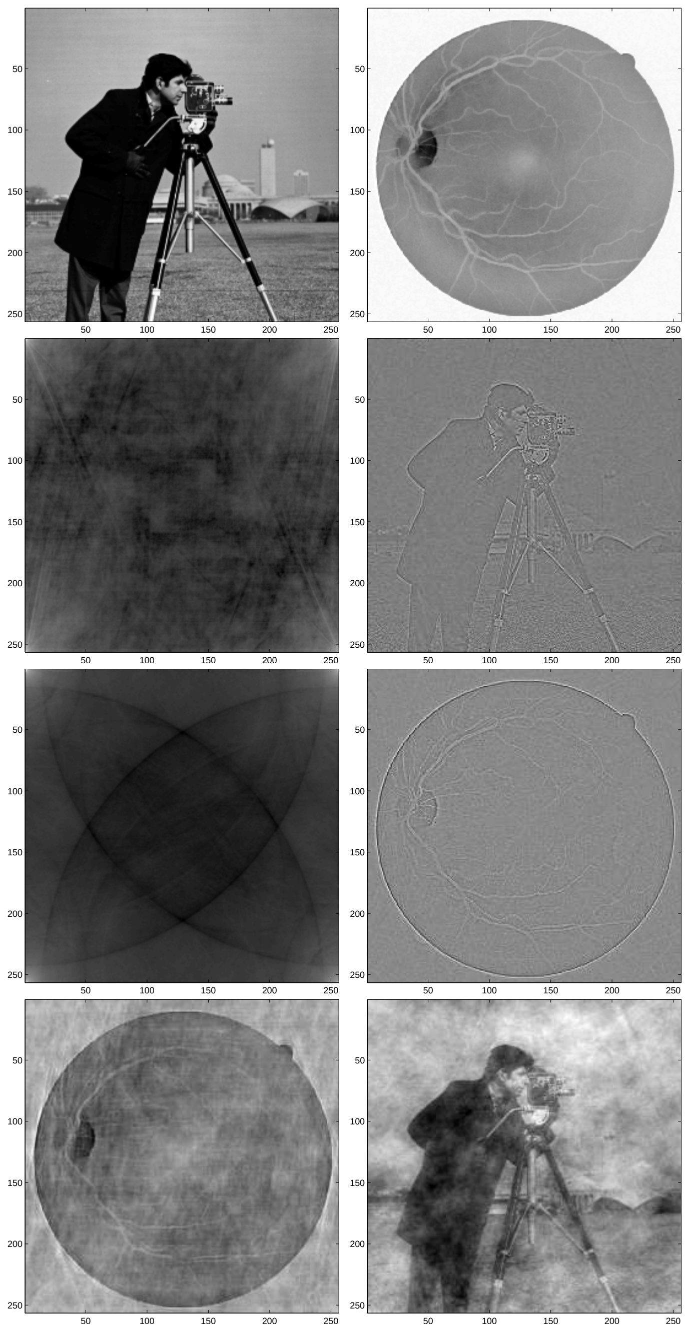


Figure 7: Top: original images, cameraman (left) and retina (right). Both images have the size of 256×256 pixels. Second row: cameraman, log of image reconstructed from magnitude (left), the image reconstructed from phase (right). Third row: retina, log of image reconstructed from magnitude (left), the image reconstructed from phase (right). Bottom: mixed images, magnitude of cameraman and phase of retina (left) magnitude of retina and phase of cameraman (right).

Comments

Images reconstructed using just magnitude of the Fourier transform bear no resemblance to the original images. Those images are symmetric – the consequence of magnitude spectrum being real. As well as the magnitude spectrum, the images reconstructed using just magnitude spectrum are symmetric and have high dynamic range with just few very light pixels in the corners, while the rest is dark. To enhance the gray detail, the logarithmic transformation was applied when displaying those images.

Images reconstructed using just phase of the Fourier transform bear clear resemblance to the original images, but the quality of the images has degraded – they appear grayish. As well as the phase angle, which is limited to the range from $-\pi$ to π, the dynamic range of images reconstructed using just phase angle is low. Both the areas that were very dark and those that were very light in the original image turn equally gray at the images reconstructed form phase angles, but the edges are well preserved.

The mixed images look surprisingly well, taking in consideration how they were obtained. It is the phase information that determines the looks (the contours) of the images. The magnitude spectrum of the other (wrong) image has even somehow contributed to the quality of the resulting image by filling it in with color, not always at correct places though.

It is obvious from the exercise that both magnitude and phase of Fourier transform are needed when reconstructing the image. The magnitude description carries the information about which frequency components are present in the image, and it gives the amount of the certain frequency component. The phase description locates where in the images is the certain frequency component present.

Trying to reconstruct the image without the phase information is impossible. To start with, images reconstructed just from magnitude are always symmetric, the consequence of magnitude spectrum being real. When reconstructing just from magnitude, we have all the right frequency components , but they are all starting from the edges of the image making it unrecognizable—that's the reason behind light corners.

Reconstructing the image using just phase information means setting the magnitude of all frequency components to 1. Since it is in general low frequencies that have larger magnitudes than high frequencies, setting all magnitudes to 1 deemphasizes low and emphasizes high frequencies, acting as a kind of high-pass filter. The features of the image are preserved, especially edges.

7 Interpolation

from week 6: Restoration

Program a MATLAB *function that rotates an image 45° (or $\frac{\pi}{4}$). Start by re-reading section 5.11.2 in GW [2] and then choose your favorite interpolation method. Test your function on a natural image (such as much used camera man) and an artificial image with straight horizontal and vertical lines.*

Rotating an image is easiest done if we first transfer the Cartesian coordinates into polar coordinates, do the rotation, and then transfer coordinates back to Cartesian. Whit this procedure is each pixel of rotated image mapped to a point in the original image. The rotation for 45° will generally result in noninteger values, i.e. pixels of rotated image will be mapped to points where original image is not defined, so we need to do gray-level interpolation.

The simplest gray-level interpolation is zero-order interpolation, where instead of noninteger values we use the nearest integer neighbor. Implementing this method is based on rounding non-integer values. This method is simple, but often distorts straight edges in images.

Bilinear interpolation uses four nearest neighbors to determine the gray-level of noninteger point. This method assumes that gray-level function is linear in x and y direction, i.e. that the gray-level values can be described by

$$v(x,y) = ax + by + cxy + d$$

The coefficients a, b, c and d are calculated using the known gray-level value at four integer neighbors, and than are those coefficients used to calculate gray-level for noninteger value. Four nearest neighbors are obtained by finding floors $\lfloor x \rfloor$ and $\lfloor y \rfloor$, and ceilings $\lceil x \rceil$ and $\lceil y \rceil$ of coordinates x and y, and combining those values in four coordinate pars.

Instead of solving the four equations with four unknowns for each interpolated value, we can use the equivalent alternative implementation of bilinear interpolation. The bilinear interpolation for noninteger point (x,y) is weighted average of gray-levels in the four nearest neighboring integer points, where weights are obtained from distances in x and y direction

$$\begin{aligned} v(x,y) = {} & (\lceil x \rceil - x)(\lceil y \rceil - y)v(\lfloor x \rfloor, \lfloor y \rfloor) + \\ & + (\lceil x \rceil - x)(y - \lfloor y \rfloor)v(\lfloor x \rfloor, \lceil y \rceil) + \\ & + (x - \lfloor x \rfloor)(\lceil y \rceil - y)v(\lceil x \rceil, \lfloor y \rfloor) + \\ & + (x - \lfloor x \rfloor)(y - \lfloor y \rfloor)v(\lceil x \rceil, \lfloor y \rfloor) \end{aligned}$$

In the MATLAB function `rotation` I implemented both the zero-order and bilinear interpolation.

```
function R = rotation(I,mode)
% takes as an input a gray-scale image (256 gray levels)
% and rotates it for 45 deg, using either zero-order or
% bilinear interpolation (bilinear interpolation uses
% alternative implementation - not calculating coeficients,
% but using distances)
%--------------------------------------------------------

I = double(I);
% adding white border (padding with 255)
% so that the result is the square image
[n, m] = size(I);
k = ceil((m+n)/2^0.5);
I = [255*ones(floor((k-n)/2),k);
     255*ones(n,floor((k-m)/2)), I, 255*ones(n,ceil((k-m)/2));
     255*ones(ceil((k-n)/2),k)];

[n, m] = size(I);
R = 255*ones(size(I));
for i=1:n
    for j=1:m
        % coordinates relative to the center
        x = (n+1)/2 - i;
        y = (m+1)/2 - j;
        r = (x^2 + y^2)^0.5;
        theta = atan2(x,y);
        % rotating
        x0 = (r*(sin(theta - pi/4)));
        y0 = (r*(cos(theta - pi/4)));
        % back to image coordinates
        n0 = (n+1)/2 - x0;
        m0 = (m+1)/2 - y0;

        if mode==0 % zero-order interpolation mode
            n0 = round(n0);
            m0 = round(m0);
                if n0>=1 & n0<=n & m0>=1 & m0<=m
                    R(i,j)=I(n0,m0);
                end
        end

        if mode==1 % bilinear interpolation mode
            % finding the four neighbours
            n1 = floor(n0);
            n2 = ceil(n0); if(n1==n2), n2=n2+1; end
            m1 = floor(m0);
            m2 = ceil(m0); if(m1==m2), m2=m2+1; end
            if n1>=1 & n2<=n & m1>=1 & m2<=m
                R(i,j)=round(((n2-n0+m2-m0)*I(n1,m1)+(n2-n0+m0-m1)*I(n1,m2)...
                    +(n0-n1+m2-m0)*I(n2,m1)+(n0-n1+m0-m1)*I(n2,m1))/4);
            end
        end
    end
end
```

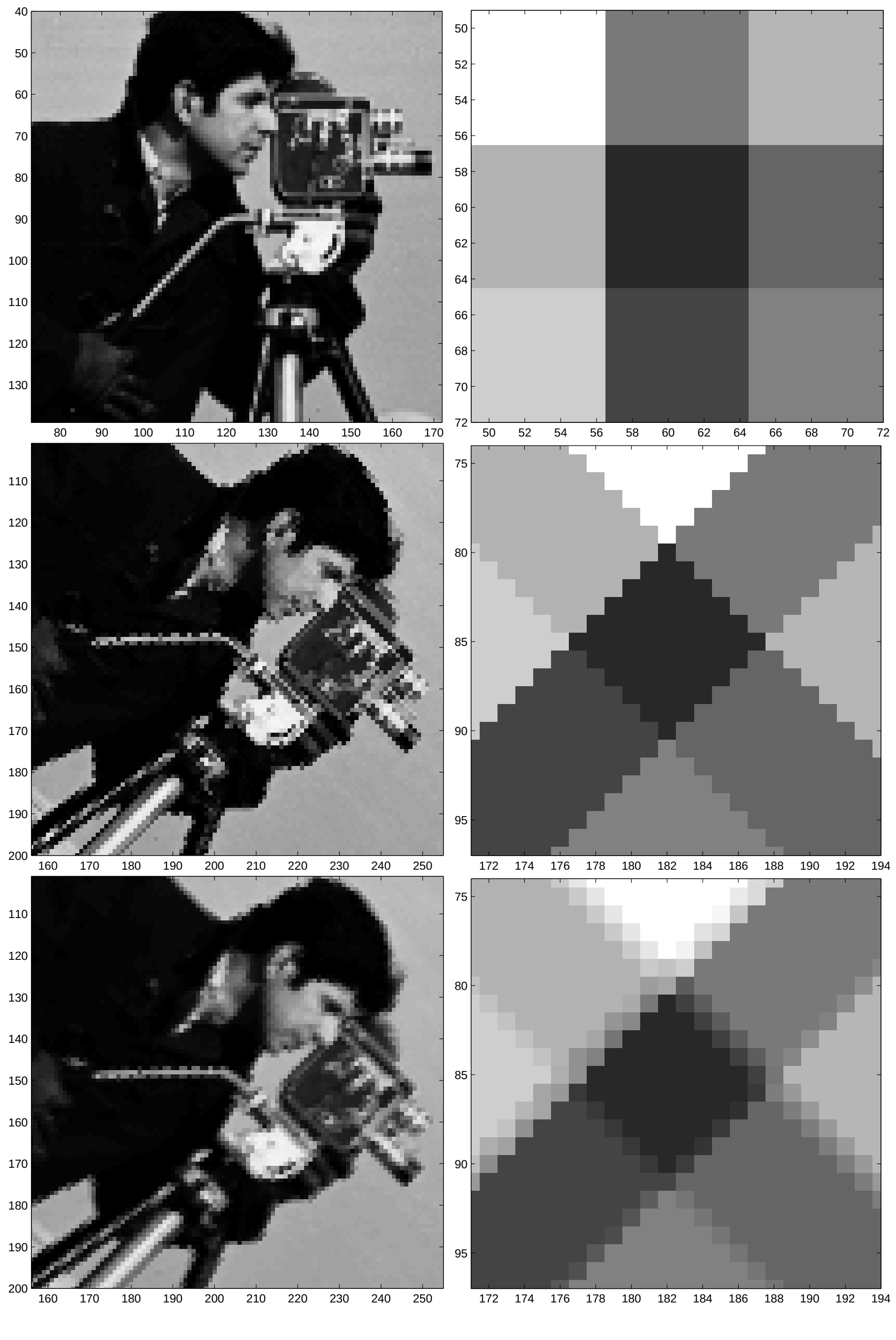


Figure 8: Top row: the details of original images, cameraman (left) and an artificial image with straight horizontal and vertical lines (right). Middle row: corresponding details of images obtained by rotation using zero-order interpolation. Bottom row: corresponding images obtained by rotation using bilinear interpolation.

Results of applying function `rotation` can be seen at the figures 8 and 9. Figure 9 shows general appearance of the rotated image. At the figure 8, which shows details of rotated images, we can compare the two interpolation methods.

Both interpolation methods did reasonable well with the `cameraman` image, but the bilinear interpolation is a giving better (smoother) result. The difference between zero-order and bilinear interpolation can best be seen at the rotations of image with straight horizontal and vertical lines. Zero-order interpolation distorts the edges, while bilinear interpolation smooths the distortion a bit, introducing gray levels that are not present at the original image.

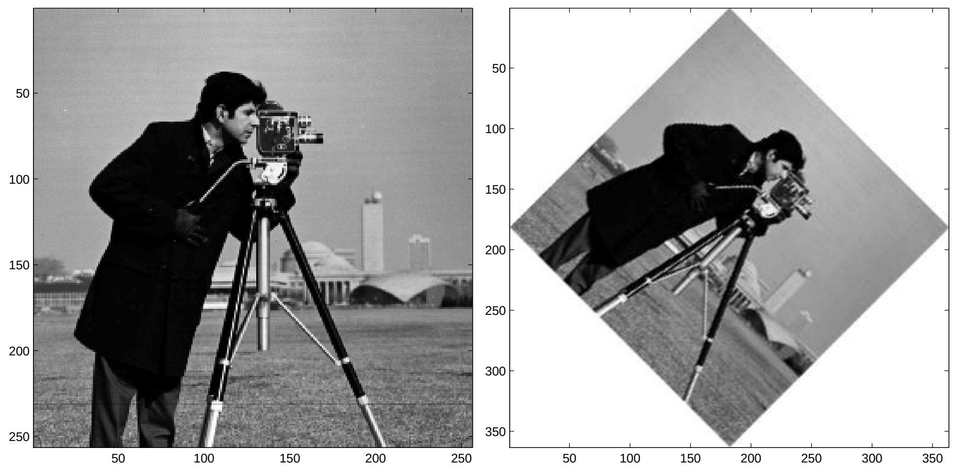


Figure 9: Original image and the image rotated for 45°, using bilinear interpolation as the interpolation method.

8 Multi-Scale Blob Detection

from week 8: Scale Spaces, Blob Detection

Blob Detector: The absolute value of the Laplacian can be used as a blob detector. Where the absolute value of the Laplacian has a spatial local maxima, there is a blob in the image. Use the `LocalMaxima.m` *function (download it from the exercise home page) to find these local maxima for a number of different scales (say, from 1 to 10) for the test image circlesquare.tif (again, you can download it from the home page). Compare the blob feature strength responses from the feature detector for the square and the circle at different scales. Which is the most blob-like feature at low scale and at high scale?*

Normalized Blob Detector: The highest blob feature response comes from the square at a low scale. At higher scales, the circle has a higher response that the square—but much lower than the square at low scales. Why? Hint: The responses are local maxima. At higher scales, there is more smoothing/blurring. What happens to the values of extrema during blurring? The normalized blob detector at scale σ is $\sigma^2\nabla^2_\sigma L(x,y) = \sigma^2(L_{xx}(x,y,\sigma)+L_{yy}(x,y,\sigma))$. Implement this function in MATLAB*. Repeat the experiment, now looking at the normalized blob feature responses at different scales for the square and the circle. Which has the highest response at low scales? Which has the highest response at high scales? Which has the highest response overall?*

Feature detection at a given scale consists of two steps: blurring an image with a Gaussian of a given scale (e.i. given standard deviation σ) and taking appropriate derivative, depending on features we want to detect. Because of commutativity and associativity of convolution, those steps can be combined into filtering the image with the appropriate derivative of the Gaussian. To detect blobs we need to find local extremes of Laplacian.

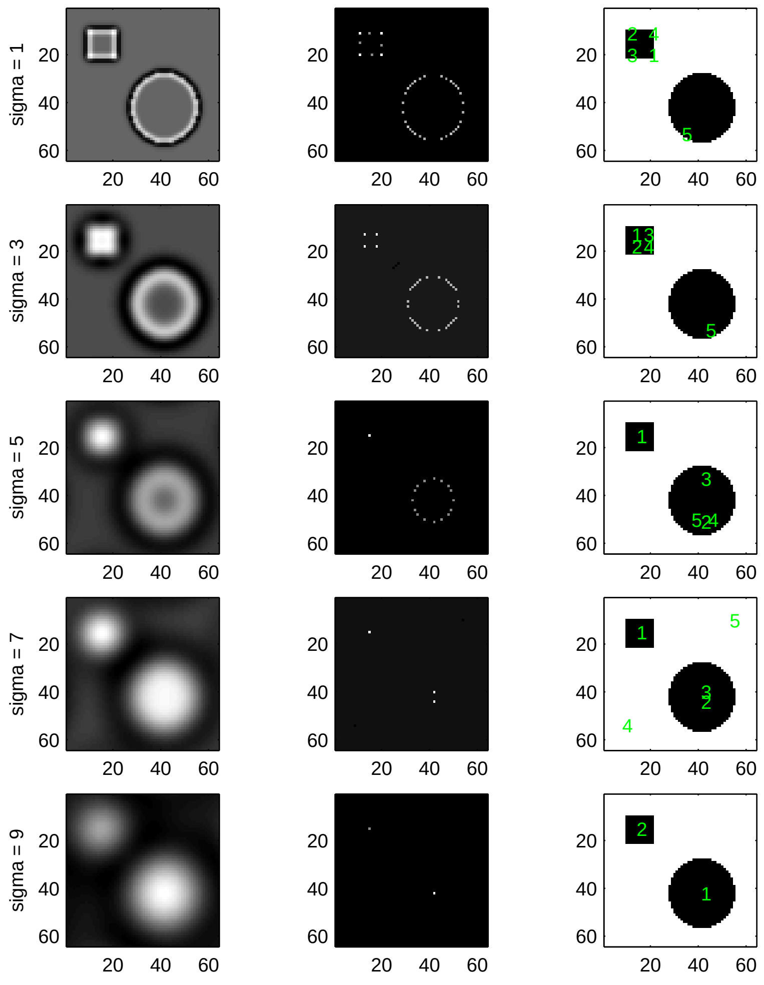
When detecting blobs, we can for each scale find the certain number (or certain percentage) of the highest responses. If we compare the responses at different scales we will notice that the highest responses come from lowest scales. The higher the scale—the more blurring, and blurring flattens local extremes.

Therefore, if we want to consider all scales simultaneously as in automatic scale selection (L [4]), we need to normalize responses to counteract the smoothing. The derivatives at each scale are multiplied by a scale-dependent factor, which also depends on the features we want to detect. The normalized blob detector at scale σ is $\sigma^2\nabla^2_\sigma f(x,y)$.

MATLAB function `blob_detector` is my implementation of the described scheme. I modified it a bit for the second part of the task.

```
function blob_detector(I,scale,r,R)
% scale - vector containing scales
% r - number of best responses in each scale
% R - number of best responses overall
%--------------------------------------------

k = length(scale);
[m,n] = size(I);

% initializing overall best responses
bestvalues = [];
bestx = [];
besty = [];

for i=1:k
    % finding normalized Laplacian, and all local maxima for each scale
    lap = scale(i)^2*real(ifft2((scale2(fft2(I),scale(i),2,0)) + ...
            scale2(fft2(I),scale(i),0,2)));
        figure(i), subplot(131), imagesc(lap), colormap gray, axis image
    %[values, x, y] = LocalMaxima(abs(lap)) % to detect also white blobs
    [values, x, y] = LocalMaxima(lap);
    max = sparse(x,y,values,m,n);
        figure(i), subplot(132), imagesc(max), colormap gray, axis image

    % finding r best responses for each scale
    [svalues,map] = sort(values);
    svalues = flipud(svalues);
    sx = flipud(x(map));
    sy = flipud(y(map));
    r = min(r,length(svalues));
        figure(i), subplot(133), imagesc(I), colormap gray, axis image,
            hold on, plot(sy(1:r),sx(1:r),'o'), hold off

    % memorizing to find overall best
    bestvalues = [bestvalues; svalues];
    bestx = [bestx; sx];
    besty = [besty; sy];
end

% finding R best responses ovarall
[bestvalues,map] = sort(bestvalues);
bestvalues = flipud(bestvalues);
bestx = flipud(bestx(map));
besty = flipud(besty(map));
R = min(R,length(bestvalues));
    figure(k+1), imagesc(I), colormap gray, axis image
        hold on, plot(besty(1:R),bestx(1:R),'o'), hold off
```

At the figure 10 we can analyze the process of blob detection, and the automatic blob detection. The image used is artificial image `circlesquare.tiff` with a larger circle and smaller square. We detected blobs for the scales $\sigma = 1, 3, 5, 7, 9$. The circle is a perfect blob at a larger scale, so it has the strongest response at the largest scale. The square is (less perfect) blob at smaller scale so it has the highest



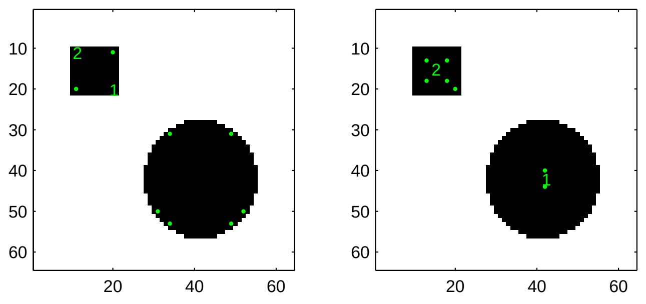


Figure 10: Applying `blob_detector` to the image `circlesquare.tiff`. Rows 1–5 correspond to scales $\sigma = 1, 3, 5, 7, 9$: Laplacian (left), all local maxima of Laplacian (middle) and 5 highest responses for a given scale (right). Bottom: 10 highest responses over all scales (1, 3, 5, 7 and 9) without normalization (left) and with normalization (right).

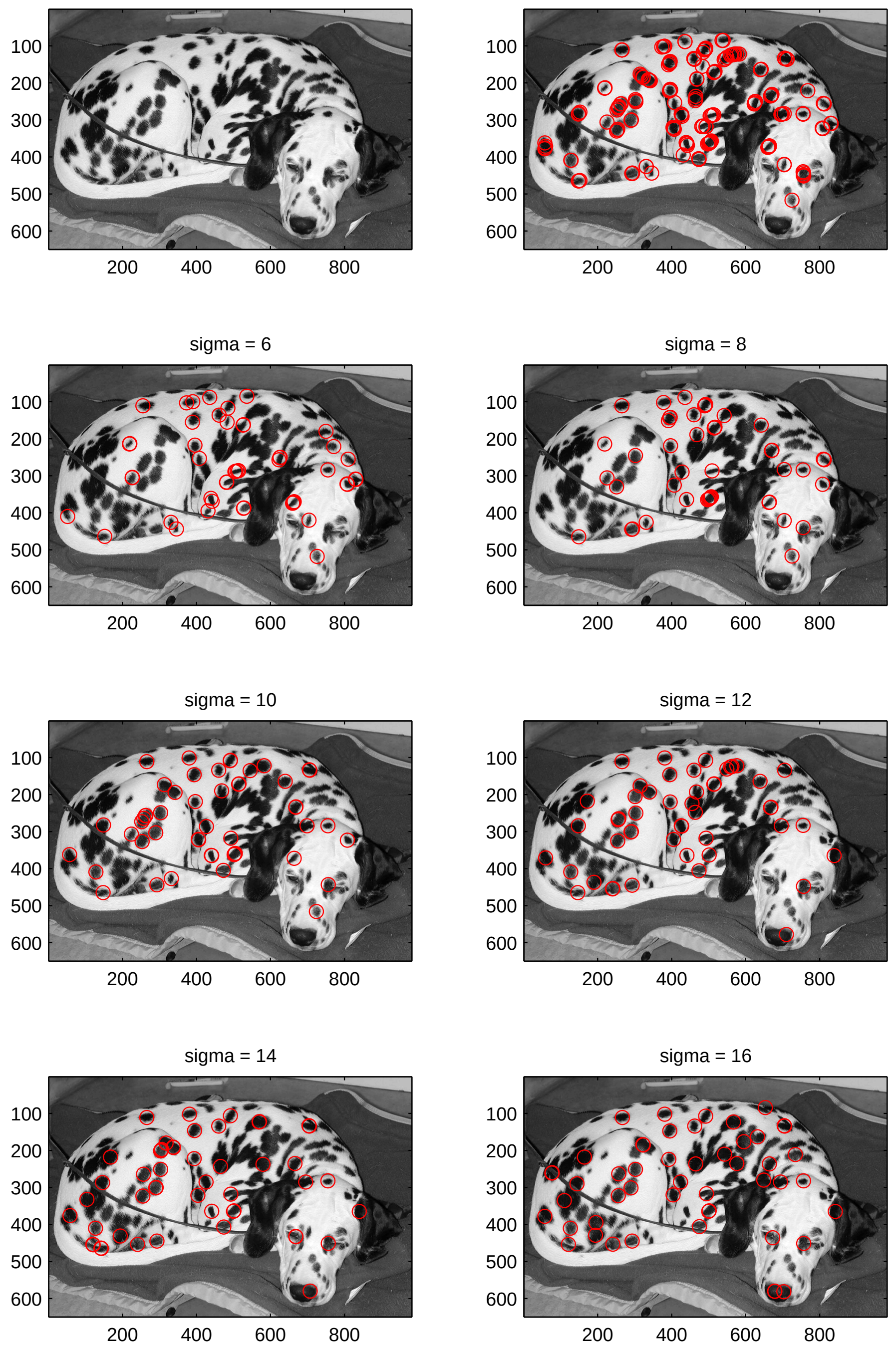


Figure 11: Applying `blob_detector` to the `dog.jpeg` image. Top left: original image, top right: 200 highest (normalized) responses over scales $\sigma = 6, 8, 10, 12, 14, 16$. Rows 2–4: 50 highest responses for a given scale.

response at smaller scale. At very small scale the corners of the square also give high responses, since they are also blob-like.

We want the highest response over all scales to be that of a circle, since it is a perfect blob. Still, without normalization the highest response over all scales is not the circle—responses in the smallest scale are the strongest and the corners of the square seem to be best blobs, even though we have a perfect blob in the image (at a larger scale). Blurring has flattened extremes at the larger scales so we need to normalize the responses. After normalization we get the desired result—the highest response for circle, and the second-best response for the square.

At the figure 11 we can see the result of applying slightly modyfied function `blob_detector` to a more realistic image `dog.jpg` [1]. The scales over which the blobs were detected are $\sigma = 6, 8, 10, 12, 14, 16$. For each scale 50 best responses is displayed. The final result is 200 best responses over all scales. It is evident, that the blob detector is quite successful in detecting some blobs, but many blobs (big and small) remained undetected.

[1] I've borrowed this image — it was used for the feature detection exercise of ITU's Signal and Image Processing course, fall 2005.

9 Morphological Skeleton

from week 7: Morphology

The purpose of this exercise is to illustrate the complications in extracting the skeleton of a given shape. Use the example image `w09man.tif` *(a true work of art) provided on the homepage for the exercise. Implement the morphological skeletonisation method illustrated in figure 9.24 in GW [2] and extract the skeleton of the man. Possibly experiment with the choice of structuring element, and compare with the results of on* `w09man2.tif` *where the man has grown another finger and gained a bit of weight.*

A skeleton $S(A)$ of a set A is a collection of all points z from A, such that z has more than one closest boundary points. We can think about the process of obtaining skeleton as off setting the borders of the image 'on fire' and looking where two (or more) fire fronts meet.

This process can be expressed in terms of successive erosions (advancing fire) followed by an opening (to find protrusions—places where fire fronts meet). Erosion continues until A erodes completely, and skeleton is union of skeleton parts

$$S(A) = \bigcup_{k=0}^{K} S_k(A)$$

where

$$S_k(A) = (A \ominus kB) - (A \ominus kB) \circ B$$

and $(A \ominus kB)$ indicates k successive erosions of A with structuring element B.

I implemented morphological skeletonisation in MATLAB function `skeleton`.

```
function S = skeleton(I,B)
% morphologic skeleton of a BW image,
% uses B as the structuring element
%-----------------------------------

S = zeros(size(I));
im_e = I;
while any(any(im_e))
    im_e = imerode(im_e,B);
    S = (S|(im_e~=imopen(im_e,B)));
end
```

Results of applying the function `skeleton` on the two images `w09man.tif` and `w09man2.tif` can be seen at the figure 12. I used two different structuring elements: 3×3 square and a + formed element contained in 3×3 square.

Final skeletons are thicker than needed and not connected. One should notice how sensitive skeletonisation is to very small changes in original set—it introduced some new branches after a minor change in original image, when using square structuring element. It is also obvious that the directions of the branches in the head were influenced by the structuring element.

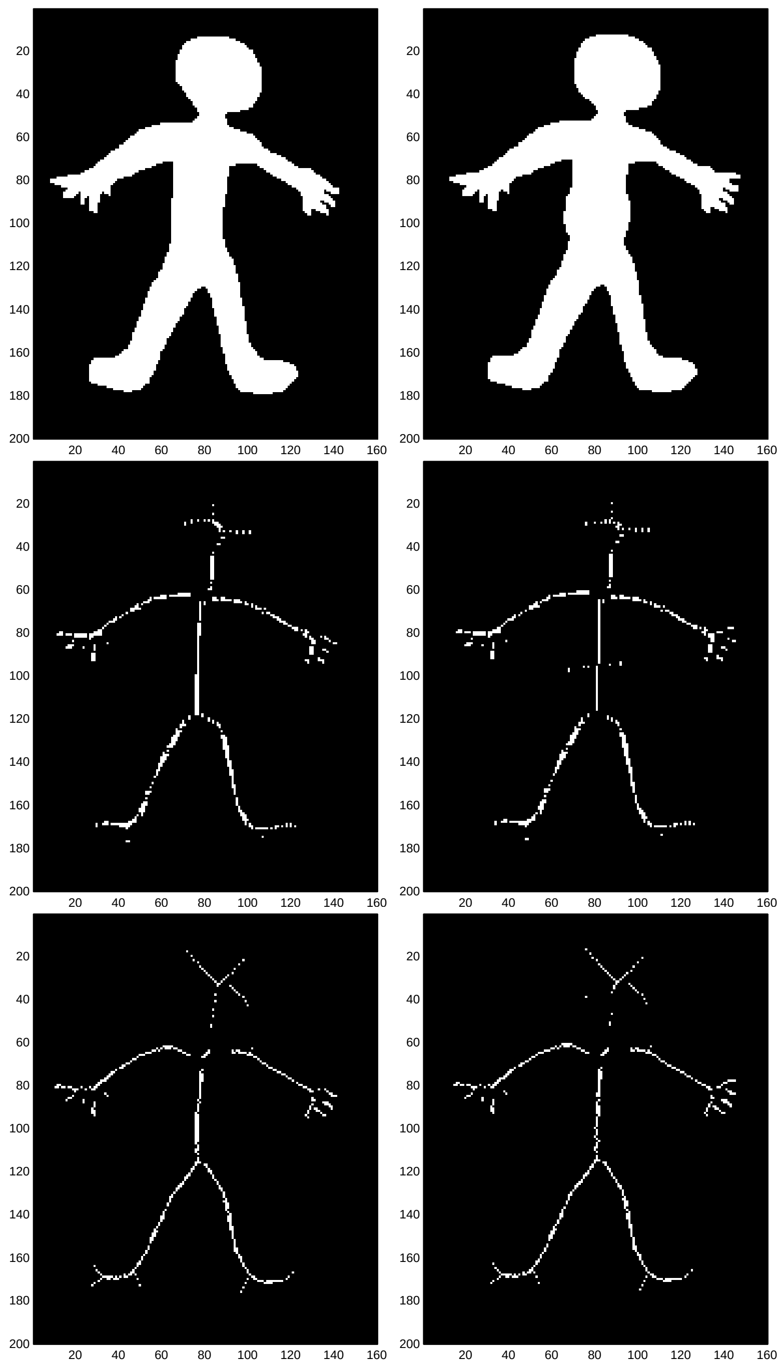


Figure 12: Morphological skeletons of the two images. Top: original images, middle: skeletons obtained using 3×3 square as a structuring element, bottom: skeletons obtained using + formed structuring element.

10 Morphology in 3D

Second Mandatory Assignment

In medical CT scans it is quite easy to locate bone structures. The bone has a high absorbtion rate of the x-rays used for the scanning. Thereby the CT scan gets a very high intensity at bones. The bone structures can almost be segmented by simple thresholding. There are two main sources of errors in the segmentation we will be using:

- *Contrast enhancement: In this particular scanning, a contrast agent have been injected into the blood of the patient. Thereby some of the blood vessels and part of the kidney is visible as well.*
- *Noise: Due to noise and imprecision in the imaging process, there are small irregularities in the segmentation. This can be seen as small nodules lying around and small holes in the bone.*

This assignment focuses on getting rid of the small holes in the bone. Since a 3D data set is a relatively large amount of data, we will only use part of the original data set. The purpose of this assignment is to get rid of the small holes in the bones by means of morphological operators. You should program the following morphological operators: dilate, erode, and closing. The structuring element could for instance be a $3 \times 3 \times 3$*-cube. Note that the corresponding operators in* MATLAB *will not do the job, since they are designed for 2D images only.*

Description of the algorithms:

Functions `dilate` and `erode` were implemented directly from the definitions for dilation and erosion, applying it to 3D. Functions return the result of dilation/erosion, taking the volume V and structuring element S as the arguments.

Definition of the dilation of the set (volume) V with the structuring element S is

$$V \oplus S = \{x \mid (\hat{S})_x \cap V \neq \emptyset\}$$

The condition $S \cap V \neq \emptyset$ written in MATLAB language is

```
any(V&S)=1
```

so, the 3D version of that condition was checked in three for-loops moving the volume V over all coordinates x, y and z.

Definition of the erosion of the set (volume) V with the structuring element S is

$$V \ominus S = \{x \mid (S)_x \subseteq V\}$$

The condition $S \subseteq V$ is equivalent to $V \cap S = S$ which can be written in MATLAB language as

```
all(V&S)=S
```

so, similarly as in dilation, the 3D version of this condition was checked in three for-loops moving the V over all coordinates x, y and z.

The problem of boundaries of the data volume vas solved by padding, so the resulting volume is of the same size as the original. In case of dilation I used zero-padding, but in case of erosion I padded with ones, to prevent the bones being eroded from the boundaries of the volume.

Functions `close` and `open` were implemented directly from the definitions for closing and opening, using the functions `dilate` and `erode`. Definition of morphological closing and opening are

$$V \bullet S = (V \oplus S) \ominus S$$

$$V \circ S = (V \ominus S) \oplus S$$

All implemented functions can take structuring element S of any size, not necessarily symmetric.

MATLAB code:

Here is the MATLAB code of the four morphological operators, which can be applied to 3D binary data V, using S as the structuring element.

```
function D = dilate(V,S)
% morphological dilation of volumen V with structuring element S
% --------------------------------------------------------------

% flipping S to be consistant with the definition
S = flipdim(flipdim(flipdim(S,1),2),3);

[Vx, Vy, Vz] = size(V);
[Sx, Sy, Sz] = size(S);

% zero-padding
e = ceil((size(S)-1)./2);
VP = zeros(size(V)+size(S)-1);
VP(e(1)+1:e(1)+Vx,e(2)+1:e(2)+Vy,e(3)+1:e(3)+Vz) = V;

% dilating
for i=1:Vx
    for j=1:Vy
        for k=1:Vz
            D(i,j,k) = any(any(any(...
                VP(i:i+Sx-1,j:j+Sy-1,k:k+Sz-1) & S)));
        end
    end
end
```

```
function E = erode(V,S)
% morphological erosion of volumen V with structuring element S
% ---------------------------------------------------------------

[Vx, Vy, Vz] = size(V);
[Sx, Sy, Sz] = size(S);

% padding with ones
e = ceil((size(S)-1)./2);
VP = ones(size(V)+size(S)-1);
VP(e(1)+1:e(1)+Vx,e(2)+1:e(2)+Vy,e(3)+1:e(3)+Vz) = V;

% eroding
for i=1:Vx
    for j=1:Vy
        for k=1:Vz
            E(i,j,k) = all(all(all((...
                VP(i:i+Sx-1,j:j+Sy-1,k:k+Sz-1) & S) == S)));
        end
    end
end

function C = close(V,S)
% morphological closing of volumen V
% with structuring element S
% ---------------------------------

C = erode(dilate(V,S),S);

function O = open(V,S)
% morphological opening of volumen V
% with structuring element S
% ---------------------------------

O = dilate(erode(V,S),S);
```

The results of using those four morphological operators with $3 \times 3 \times 3$ cube as the structuring element can be seen at figure 13.

Comments

The goal of the assignment was to get rid of the small holes in the bones by means of morphological operators. To achieve this goal, one should use morphological closing, since is smoothes the contours and closes small holes.

As far as I could see, all the operators used in this assignment did what they were supposed to do. Applying `close` did fill in small holes at the surface of the bones, and also small holes in the bones, so I'm satisfied with the result.

I tried using different shapes as a structuring element, but even the simple $3 \times 3 \times 3$ cube gave satisfying results.

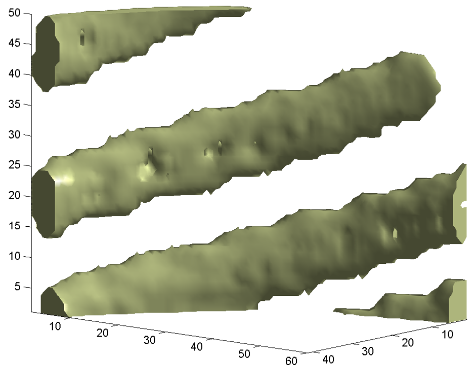
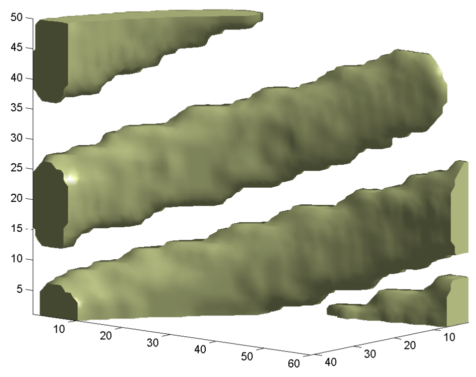
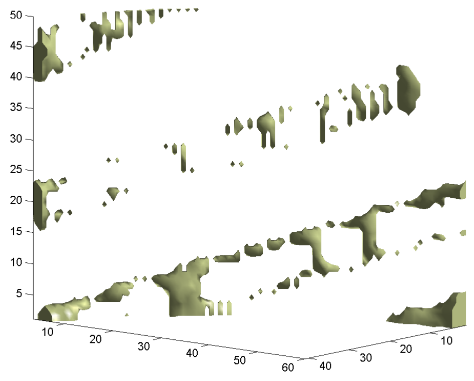
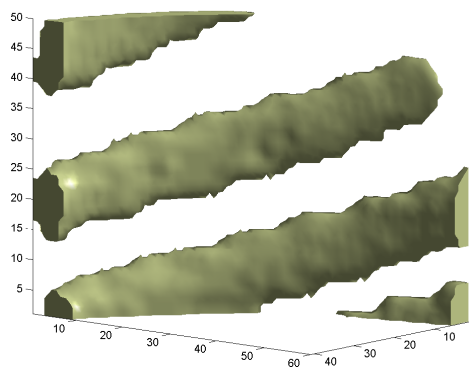
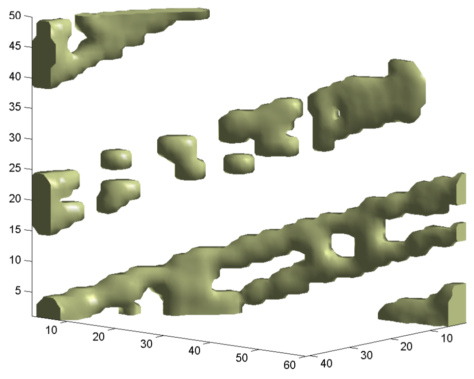






Figure 13: Visualization of the original bone structure together with the results of applying different morphological operators, all using $3 \times 3 \times 3$ cube as the structuring element. Top: original data, middle left: after single dilation, middle right: after single erosion, bottom left: after closing, bottom right: after opening.

Single erosion

What happens when you apply a single erosion to the ribs in the data set, and why?

Applying a single erosion thins the ribs from all sides. Since the ribs are sometimes hollow (according to the data set), they get eroded both from inside and outside—this can be nicely seen at the result of morphological opening.

An unexpected effect of eroding the ribs is an introduction of vertical slices. This can be explained by the shape of the ribs—a rib is long, thin and curved in the (x,y) plane, so the erosion with the $3 \times 3 \times 3$ cube often leaves just distinct squares in (x,y) plane. Since the rib is not curved in z direction, erosion reduces the rib to distinct rods in z direction. Figure 14 illustrates this effect in 2D.

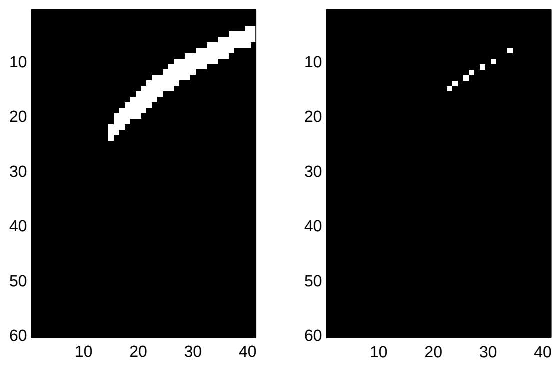


Figure 14: A slice trough the data in the plane parallel with (x,y) plane at z=49. Left: original bone structure, right: after single erosion with $3 \times 3 \times 3$ cube as the structuring element.

11 Hough Transform

from week 10: Segmentation

Describe an algorithm for finding circles using a Hough transform approach. Do not implement this in Matlab.

Hough transform is a global segmentation method for detecting lines, circles or any other shape with simple parametrical representation. For example, a line in an image can be represented with two parameters, so a line will correspond to a point in 2D parametric space. Since many lines pass trough each point, each point in the image will correspond to a line (or curve, depending on parametrization) in the parametric space. Detecting lines in the image is than equivalent to finding intersections of lines (or curves) in parametric space.

To use Hough transform for finding circles, we first need to choose parametrization. A point (x,y) lying on circle with the center in the point (c_x, c_y) and with radius r satisfies the equation

$$(x-c_x)^2+(y-c_y)^2=r^2$$

We need 3 parameters c_x, c_y and r to describe a circle in the image, therefore we also need 3D parametric space with coordinates (c_x, c_y, r). Each circle in the image corresponds to a point in the parametric space.

Each point (x,y) in the image lies on many circles. Actually, whichever center point (c_x, c_y) we choose, we can find radius r in such a way that point (x,y) lies on the circle. So, a point (x,y) in the image corresponds to a 2D conical surface in parametric space described with

$$r=\sqrt{(x-c_x)^2+(y-c_y)^2}$$

Intersection of two such surfaces yields a line, and intersection of three surfaces yields a point, which than corresponds to a circle in original image.

Algorithm for detecting circles using Hough transformation is then described by following steps.

1. Use an edge detector to find the set of edge points (x_i, y_i).

2. Initiate (discrete) 3D parameter space P. The size of P in (c_x, c_y) plane depends on the size of the image, while the size in r dimension depends on the choice of the maximal radius to be detected.

3. Scan the image, and for each edge point (x_i, y_i) do:

    - Scan the (c_x, c_y) plane in parameter space, and for each par (c_x, c_y) do:
        - Calculate $r=\sqrt{(x-c_x)^2+(y-c_y)^2}$ and round it to $[r]$.
        - Increment the value of $P(c_x, c_y, [r])$.

4. Find maxima in the parameter space. A maximum at position $P(c_x, c_y, r)$ corresponds to a circle $(x-c_x)^2+(y-c_y)^2=r^2$ in the original image.

12 Statistical Shape Models

from week 10: Representations & Descriptions / Shape & Texture

In this exercise you will make a simple shape model of human spine data. Implement this in MATLAB. *You may find the commands* `eig` *and* `cov` *useful. Download the spine shape dataset. It contains a* 200×25 *matrix shapes. Each column is one 2D shape with 100 landmark points. Elements 1–100 of each column are the* x*-coordinates and 101–200 the* y*-coordinates. Do the following:*

- *Divide the shapes into a separate training and test set, for instance 24 train shapes and 1 test shape.*
- *Model construction, use the training set for this!*
  - *Plot all shapes on top of each other.*
  - *Compute the mean shape and plot it with the other shapes.*
  - *Compute the covariance matrix and visualize it, e.g. using* `imagesc`. *Explain what you see.*
  - *Compute the modes of variation (the principal components) as the eigenvectors of the covariance matrix. Visualize the first few modes, for instance by drawing the mean shape plus and minus 3 standard deviations of the deformation.*
- *Select a small number of modes in your model and project the test shape(s) on the model subspace using* $\mathbf{x} = \bar{\mathbf{x}} + \mathbf{P}\mathbf{b}$ *and thus* $\mathbf{b} = \mathbf{P}^T(\mathbf{x} - \bar{\mathbf{x}})$. *Plot the original shape together with the projected shape. Try this for models of different dimensionality (different number of modes).*

Statistical shape modeling (C [1]) is a 'top-down' approach, where we make use of a prior model of what is expected in an image. To build a model, one must decide on suitable landmark points $\{(x_i, y_i)\}$, and one must have a training set of annotated typical images, so each image contributes with $2n$ element vector

$$\mathbf{x} = (x_1, \ldots, x_n, y_1, \ldots, y_n)^T$$

The set of vectors $\mathbf{x}_k$ forms a distribution in the $2n$ dimensional space which will have $2n \times 2n$ covariance matrix. By modeling this distribution, we can generate new vectors, similar to those from the original training set.

We can reduce dimensionality of the data cloud by finding eigenvectors and eigenvalues of covariance matrix, and deciding to use just $t < 2n$ main axes of the data cloud—those axes for which the variance is greatest. We can than approximate vector **x** using

$$\mathbf{x} \approx \bar{\mathbf{x}} + \mathbf{P}\mathbf{b}$$

where **P** is a $2n \times t$ submatrix of the covariance matrix containing it's t eigenvectors and **b** is a t dimensional vector given by

$$\mathbf{b} = \mathbf{P}^T (\mathbf{x} - \bar{\mathbf{x}})$$

I made a simple shape model of human spine data using following MATLAB code.

```
load shapes
training = [shapes(:,1:12),shapes(:,14:25)];
test = shapes(:,13);

figure(1) % plotting all training shapes
    shapeplot(training,'g'), fixaxis(0), hold on
    m = mean(training,2);
    shapeplot(m,'b'), shapeplot(m,'b.'), hold off

CM = cov(training'); % covariance matrix
[P,D] = eig(CM);
figure(2), imagesc(CM), axis image

figure(3) % plotting first 3 modes
for mode=200:-1:198
    shape_min = m - 3*(D(mode,mode))^0.5*P(:,mode);
    shape_max = m + 3*(D(mode,mode))^0.5*P(:,mode);
    subplot(3,3,(200-mode)*3+1), shapeplot(shape_min,'r'), fixaxis(1)
        ylabel(sprintf('mode %d',200-mode+1))
    subplot(3,3,(200-mode)*3+2), shapeplot(m,'b'), fixaxis(1)
    subplot(3,3,(200-mode)*3+3), shapeplot(shape_max,'r'), fixaxis(1)
end

figure(4) % projectng test shape, using 2,3 or 7 modes
NOM = [2 3 7];
for i=1:length(NOM)
    Psub = P(:,200-NOM(i)+1:200);
    b = Psub'*(test-m);
    model = m+Psub*b;
    subplot(1,length(NOM),i)
        plot(test(1:100),test(101:200),'b'),
            title(sprintf('%d modes',NOM(i))), fixaxis(0), hold on
        plot(model(1:100),model(101:200),'r'), hold off
end

function fixaxis(wide)
    axis image, axis([-61-20*wide,72+20*wide,-113,122])

function shapeplot(shape,color)
    plot(shape(1:100,:),shape(101:200,:),color)
```

On the figure 15 we can see the 24 spine shapes that were used as training shapes, together with the mean shape with 100 landmark points. We can also see visualization of the covariance matrix for the distribution of the training shapes.

From the covariance matrix we can see that the covariances have largest absolute values in the upper left quadrant, corresponding to *x*-coordinates. From the two red areas of large covariances, we can conclude that if a landmark in an highest (or lowest) part of a spine moves to left or right, neighboring landmarks will generally move in the same direction. From the two dark blue areas of negative covariances, we can conclude that if the highest part of a spine moves to left or right, lowest part will generally move in an opposite direction. In all, we can conclude that some kind of rotation is the principal variation of the shapes.

Covariances between *y*-coordinates are relatively small. Covariances between *x*- and *y*-coordinates display a pattern, reflecting the the fact that spine consists of 4 vertebrae. Knowing that the rotation is the principal variation, we can interpret that pattern: when the spine rotates a bit in a clockwise direction, left part of it (with half of the landmarks) will move down, and right part of it (with another half of the landmarks) will move up.

On the figure 16 we can see three principal modes of variation. This confirms that rotation is an important factor in the principal mode of variation. Second mode involves scaling in *y*-direction. Variations of the third mode cause less significant and more local changes.

In this particular shape model the first mode is rather dominant—it describes 73% of the variance in the training set. Already with the first 3–4 modes we can describe approximately 90% of the variance, and with the first 7 modes we cover 95% variance of the training set.

On the figure 17 shape modeling is used for generating new shapes, and for

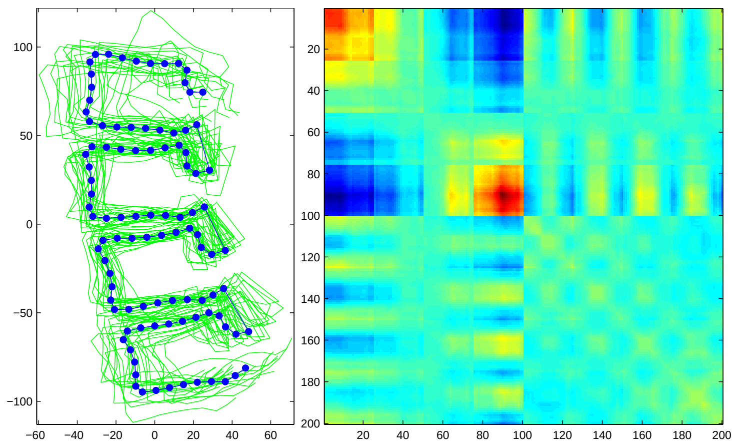


Figure 15: Left: 24 training shapes of human spine data (green) and the mean shape with 100 landmark points (blue). Right: Covariance matrix for the distribution of training shapes.

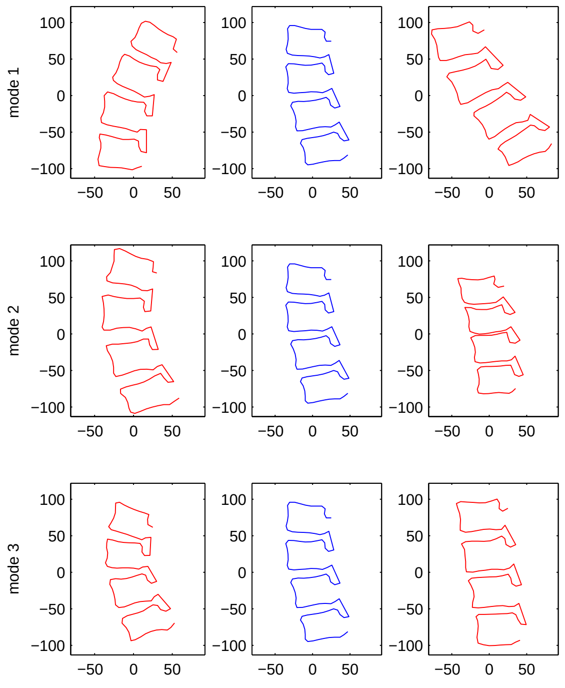


Figure 16: Effect of varying first three modes. Top line: first mode, middle line: second mode, bottom line: third mode. Modes are varied between -3 and $+3$ standard deviation from the mean shape, with the middle column being mean shape.

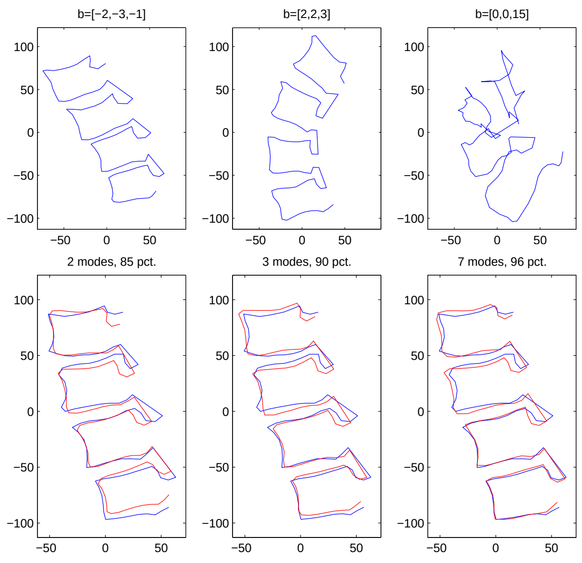


Figure 17: Top line: Three new spine shapes generated by varying the parameters of the shape model. Bottom line: Test shape (blue) and the shapes obtained by projecting test shape on the model subspace for different number of modes (red).

fitting the model to test shape. Of the new generated shapes, two were obtained using shape parameters not bigger than 3 times standard deviation for the each mode. Those two shapes look plausible. The third shape was obtained by varying the third mode for 15 times standard deviation for the third mode. This shape looks very distorted—we moved too far away from our data cloud. By keeping the parameters close to standard deviation, we ensure that the generated shapes are similar to those in the original set.

Fitting the model to test shape by projecting test shape on the model subspace can also be seen at the figure 17. Projecting was done for 2, 3 and 7 modes, corresponding to 85%, 90% and 96% of the total variance. Already 2 modes give good approximation, and it slightly improves with increasing number of modes.

13 Co-occurrence Matrix

from week 10: Representations & Descriptions / Shape & Texture

For this exercise you need the images `texture1.bmp`–`texture5.bmp` *and the function* `cooc.m` *from the exercises homepage. Compute a co-occurrence matrix for each of the five texture images with the function* `cooc` *and visualize the matrices, e.g. using* `imagesc`*. Explain the differences. Experiment with another distance for the co-occurrence and possibly also with different directions. Compute some of the standard measures on co-occurrence matrices, e.g. maximum probability, element difference moment, and entropy. Can you discriminate between the different textures on basis of these measures?*

We can look at the co-occurrence matrix as of a 2D histogram that carries the information regarding the relative position of pixels with respect to each other. Relative position of pixels can be described with, for example 'two pixels down and on pixel to the right'. Co-occurrence matrix for a given relative position is than obtained by counting how many times did the pixel with the relative value z_i occur relative to the pixel with the relative value z_j. Each element c_{ij} of the co-occurrence matrix C gives us probability of finding intensity z_i relative to z_j. For an image with 256 gray levels, co-occurrence matrix for a given relative position will have size 256×256.

Some of the descriptors that can be extracted from co-occurrence matrix and which can be used to characterize the content of it are:

1. Maximum probability
$$\max_{i,j}(c_{ij})$$
2. Element difference moment of order k
$$\sum_i \sum_j (i-j)^k c_{ij}$$
3. Uniformity
$$\sum_i \sum_j c_{ij}^2$$
4. Entropy
$$-\sum_i \sum_j c_{ij} log_2 c_{ij}$$

We can use those descriptors in pattern recognition, when trying to distinguish between textures on the basis of descriptor values.

Function `cooc` can be used to obtain co-occurrence matrix. I modified this function a bit, to allow negative directions needed to check '1 down, 1 left' direction. I also made the function direction-dependent [2] to be consistent with the definition from GW [2]. The other function `descriptors` is used to extract the descriptors from the co-occurrence matrix.

```
function CM = cooccurrence_matrix(I,x,y);
% Co-occurrence matrix of a gray-scale image I (265 gray levels)
% where x and y are distance parameters:
% x - up(x<0) and down(x>0)
% y - left(y<0) and right(y>0)
%---------------------------------------------------------------
[n,m] = size(I);
CM = zeros(256,256);

for i=1-min(0,x):n-max(0,x)
    for j=1-min(0,y):m-max(0,y)
        CM(I(i+x,j+y)+1,I(i,j)+1) = CM(I(i+x,j+y)+1,I(i,j)+1)+1;
    end
end

CM = CM/sum(sum(CM));

function [MP,EDM,uniformity,entropy] = descriptors(CM)
% descriptors from co-occurrence matrix CM:
% maximum probability, element differnce moment of order 2,
% uniformity and entropy
%-----------------------------------------------------------

MP = max(max(CM));
EDM = sum(sum(toeplitz(0:size(CM,1)-1).^2.*CM));
uniformity = sum(sum(CM.^2));
entropy = -sum(sum(CM.*log2(CM+eps)));
```

On the figure 18 we have 5 test textures and the co-occurrence matrices for some selected directions. Co-occurrence matrices of the texture 1 have high probabilities shattered around the whole matrix, the result of compared pixels often having very different and uncorrelated values. On the contrary, co-occurrence matrices for texture 2 have high probabilities on the main diagonal because neighboring pixels often have similar intensities. Both for texture 1 and 2 we can see that the similarity between pixels generally decreases with the distance.

Texture 3 is an example where co-occurrence matrix doesn't change much for different relative positions on the same distances. On the other hand, co-occurrence matrices of textures 4 and 5 are different even though the relative positions are on the same distance. Following the direction of stripes results in higher probabilities closer to main diagonal. From the co-occurrence matrices for the texture 5 we can also conclude that the histogram of the texture has highest peak around the intensity level 100, and another high peak around 225.

[2]Function `cooc` is bi-directional, not making distinction between '1 down, 1 left' and '1 up, 1 right'. Some authors ignore the direction when defining co-occurrence matrix.

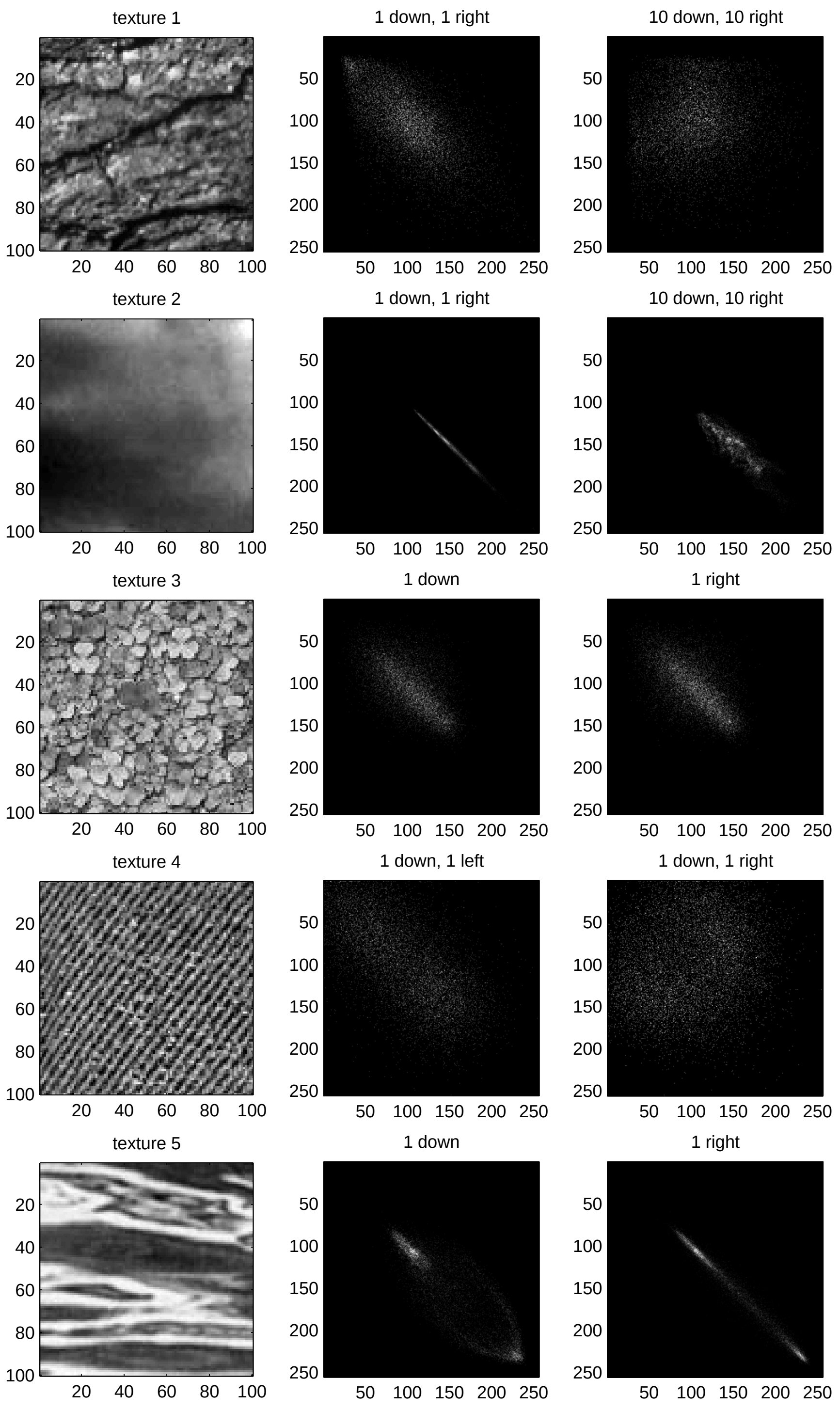


Figure 18: Five test textures and two co-occurrence matrices for each texture. Relative positions are written above each matrix.

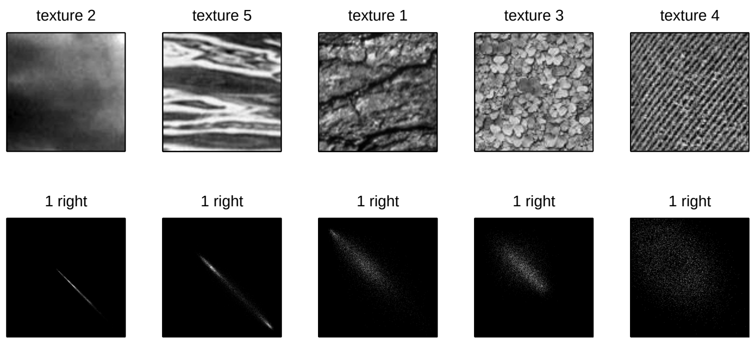
I have used the function `descriptors` to extract the four descriptors from co-occurrence matrices for all the textures. It turned out that it was possible to partially sort the textures by those descriptors. On the figure 19 we can see the textures sorted by

1. decreasing maximum probability,
2. increasing element difference moment,
3. decreasing uniformity and
4. increasing entropy.

It was only the ordering of textures 1 and 3 that could not be resolved by this scheme.

We can see that we have more uniform and smoother textures on one side, and more coarse textures on other. In terms of co-occurrence matrices we have those with the high values near main diagonal on the one side, and those with the high values scattered across the matrix on the other side.

Descriptors would make it possible to discriminate between some of the textures. For example texture 5 has relatively high element difference moment, so this value could be used to recognize that texture. Texture 1 could also be easily recognized due to its high uniformity. On the other hand, it would be hard to discriminate between textures 1 and 3 on the basis of those 4 descriptors—for those textures are the high probabilities similarly distributed in co-occurrence matrix.



	texture 2	texture 5	texture 1	texture 3	texture 4
Max. probability$/10^{-3}$	7.8	2.5	1.0	0.9	0.5
El. diff. moment (or. 2)	4	48	464	612	2804
Uniformity$/10^{-4}$	26	6.0	2.2	2.4	1.4
Entropy	9.2	11.2	12.4	12.3	13

Figure 19: Five test textures, their corresponding co-occurrence matrices and the four descriptors for each matrix. Textures are partially sorted depending on the descriptor values.

14 Pattern Classification

Third Mandatory Assignment

For this assignment, you will have to implement the minimum distance classifier (GW [2], 12.2.1).

Download the data `pr.mat`*. Start with dataset A. The dataset has two classes,* `setA_class1` *and* `setA_class2`*. There are 1000 samples of each class and the pattern vectors are 2-dimensional (each sample is described by 2 feature values). Plot the data (like is done for the iris data in GW [2], figure 12.1).*

Take a small subset of the dataset (e.g. 10 samples for each class) and plot the decision boundary of the minimum distance classifier trained on the subset. Repeat this several times for different subsets of the same size. Explain your findings.

Use 500 samples of each class to test your classifier and train the classifier on subsets of the other 2×500 *samples. Evaluate how the performance (the error rate = the percentage of misclassified pixels) depends on the size of the training set; draw the curve of the error rate as a function of the number of training samples. Draw in the same plot also the curve of the error rate on the training sets that you used. Explain your findings. How would the curves have looked for a 1-NN classifier? Why?*

How do you think this data is generated? Why? From the classifiers that you learned about in the lecture, what would be the best classifier for this kind of data? Why?

Plot the distribution of the 2 classes of dataset B. From the classifiers that you learned about in the lecture, what would be the best classifier for this kind of data? Why?

Explanation of implementation

A pattern is an arrangement of certain descriptors, and a pattern class is a family of patterns that share some common properties. Pattern classification is a technique of assigning patterns to their respective classes. An often used scenario for pattern classification involves designing a classification algorithm based on a set of pre-classified (training) patterns, and than applying the algorithm on unknown (testing) patterns.

The patterns used in this assignment have two descriptors so we can represent each pattern as the point in a plane. The classifier we need to use is minimum distance classifier. This classifier first calculates a mean pattern for each class of training patterns. An unknown pattern is than assigned to that class whose mean pattern it is closest to. The boundary between two classes (decision boundaries) is in that case a line (plane or hyperplane for more dimensions). More precisely, decision boundary is perpendicular bisector of a line segment joining the means of the classes.

Since the patterns from our assignment live in a plane, it was possible to find the decision boundary using plane analytic geometry. The boundary between the two classes can be found using the equation of a straight line trough a given point and perpendicular to a given straight line

$$y - y_p = -\frac{1}{a_l}(x - x_p)$$

where a_l is the slope of a line connecting the two mean points, i.e. $a_l = \frac{y_1 - y_2}{x_1 - x_2}$, and (x_p, y_p) are coordinates of a midpoint between two means $(x_p, y_p) = (\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$. Solving this equation for y results in slope-intercept form of the decision boundary [3]

$$y = -\frac{x_1 - x_2}{y_1 - y_2}x + \frac{x_1 + x_2}{2} \cdot \frac{x_1 - x_2}{y_1 - y_2} + \frac{y_1 + y_2}{2}$$

We have a slope of the decision boundary given by

$$a = -\frac{x_1 - x_2}{y_1 - y_2}$$

and the intercept of a decision boundary given by

$$b = \frac{x_1 + x_2}{2} \cdot \frac{x_1 - x_2}{y_1 - y_2} + \frac{y_1 + y_2}{2}$$

where (x_1, y_1) and (x_2, y_2) are coordinates of the mean patterns for the two classes. Alternatively, one could write the decision boundary in a vector form, as in equation (12.2-6) GW [2]. I wrote a function `boundary.m` that for a pair of mean points returns the slope and the intercept of a decision boundary.

Assigning the unknown pattern with coordinates (x_u, y_u) to one or another class is now equivalent to checking if the point (x_u, y_u) lies under or above the decision boundary, i.e. if $y_u < ax_u + b$ than (x_u, y_u) belong to one class, otherwise to the other. In case of equality we have a point on the decision boundary. I wrote a function `test.m` that counts how many of the testing patterns ended on the wrong side of the boundary. When implementing this function, I used the fact that we actually know to which class a certain testing pattern belongs, and that we know that class 1 lies under, and class 2 above the decision boundary.

[3]This form of the line equation can not be used for vertical lines, but in our particular case I knew that the decision boundary is not a vertical line, so I saw no problem in using slope-intercept form.

MATLAB code:

Assignment task can than be solved using the following MATLAB code.

```
load pr;
% plotting all data (set A)
figure(1), classplot(setA_class1,setA_class2)

% finding the decision boundary for 5 subsets of dataset
for k=1:5
    training_class1 = setA_class1(10*(k-1)+1:10*k,:);
    training_class2 = setA_class2(10*(k-1)+1:10*k,:);
    figure(k+1)
        classplot(training_class1,training_class2), hold on
        m1 = mean(training_class1); m2 = mean(training_class2);
        plot(m1(1),m1(2),'bo',m2(1),m2(2),'ro')
        [a,b] = boundary(m1,m2);
        x = -3:5; y=a*x+b;
        plot(x,y,'g')
end

% finding error rate for training sets of different size
testing_class1 = setA_class1(501:1000,:);
testing_class2 = setA_class2(501:1000,:);
k = 10:10:500;
for i=1:length(k)
    training_class1 = setA_class1(500-k(i)+1:500,:);
    training_class2 = setA_class2(500-k(i)+1:500,:);
    m1 = mean(training_class1); m2 = mean(training_class2);
    [a,b] = boundary(m1,m2);
    err_rate(i) = test(a,b,testing_class1,testing_class2);
    err_tr(i) = test(a,b,training_class1,training_class2);
end
figure(7), plot(k,err_rate,'b',k,err_tr,'g'), axis([0 502 0 20])

% plotting dataset B
figure(8), classplot(setB_class1,setB_class2), axis auto,

function classplot(class1, class2)
% function for plotting datasets
    plot(class1(:,1),class1(:,2),'b.',class2(:,1),class2(:,2),'r.')
    axis([-3 5 -3.5 5])

function [a,b] = boundary(m1,m2)
% slope and intercept of the decision boundary between two means
    a = -(m1(1)-m2(1))/(m1(2)-m2(2));
    b = 0.5*(m1-m2)*(m1+m2)'/(m1(2)-m2(2));

function err = test(a,b,class1,class2)
% finding error rate for given decision boundary
    t1 = a*class1(:,1)+b;
    t2 = a*class2(:,1)+b;
    err_rate = 100*(length(find(class1(:,2)>t1))+...
                length(find(class2(:,2)<t2)))/length([class1;class2]);
```

Explanation of experiments

The experiments in this assignment are based on training the minimum distance classifier on a certain subset of dataset, and than testing the performance of the classifier on an other subset of dataset. Training and testing subsets have to be disjunct.

On the figure 20 we have a plot of the two classes, each consisting of 1000 patterns. Each class will later be divided into training and testing part.

On the figure 21 we have four different training sets, the mean pattern for each class and the decision boundaries of minimum distance classifiers trained on these sets. All training set have 10 patterns from each class.

We can see that the positions of the mean patterns vary from plot to plot, and consequently the decision boundary also varies. This shows the importance of having enough of training patterns. The more training patterns one has available, the better statistical description of the class can be obtained. Just few training patterns will not capture statistical behavior of the class. To illustrate this with an extreme case we can consider having just two training patterns. Depending on the position of those two patterns, decision boundary could take any possible direction since classes are partially intermixed.

We can also notice that already for the training sets on the figure 21 we have some occurrences of the patterns ending on the wrong side of the decision boundary. This is a consequence of classes being partially intermixed—it is not possible to find a single line (or even single curve) that perfectly separates class 1 and class 2.

In the next experiment, I gradually increased the size of the training set from 10 to 500 of each class, by successively adding 10 patterns of each class to the training

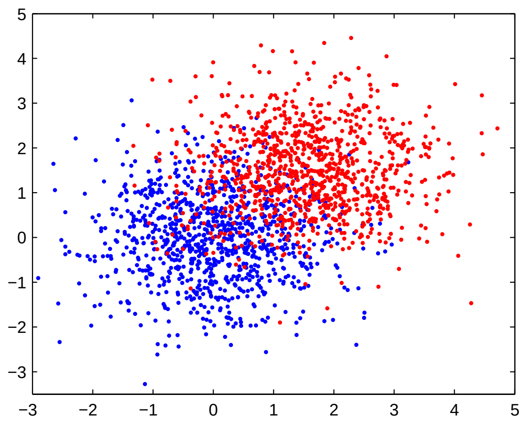


Figure 20: A plot of the two classes: class 1 (blue) and class 2 (red). Each class consists of 1000 patterns.

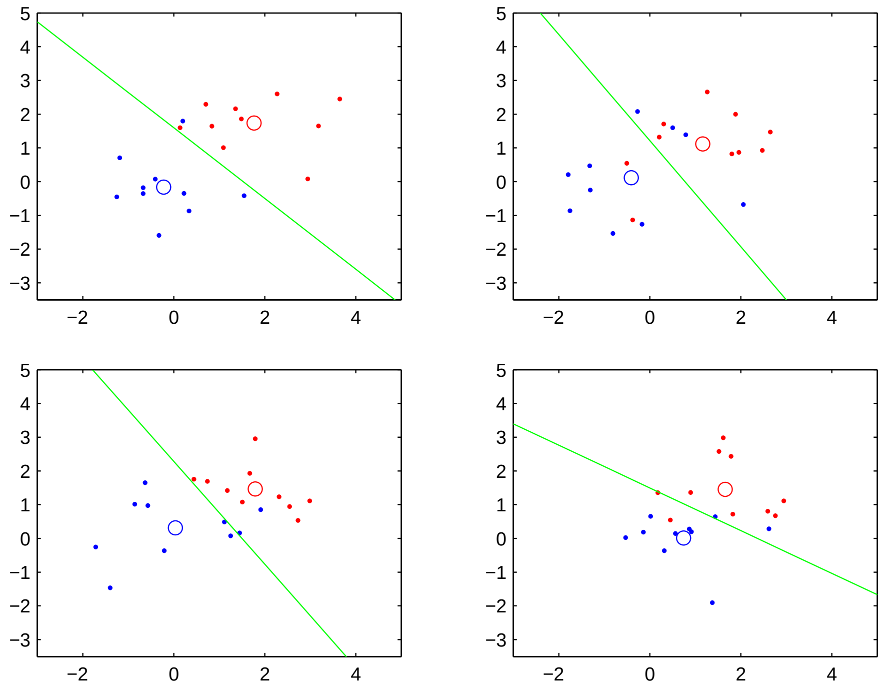


Figure 21: Four different training sets of two classes, the mean pattern for each class and the decision boundaries of minimum distance classifiers trained on these sets.

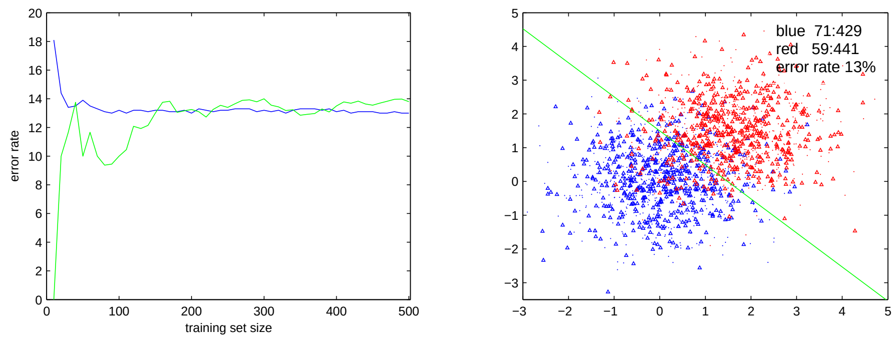


Figure 22: Left: Error rate (in percent) for different sizes of training set. Right: Training set containing 500 patterns from each class (small circles), decision boundary, and the testing set containing 500 patterns from each class (small triangles). Hit vs. miss rate for each class is also displayed, together with error rate.

set. For each training set the decision boundary was calculated and tested on the (unchanged) testing set of $500+500$ patterns. The plot on the figure 22 shows how does the error rate depend on the training set size. The error rate on the training set is also included in the plot.

We can see that the error rate is relatively high for very small training sets (under 100 patterns from each class), but for the bigger training sets the error rate is rather stabile around 13%. We can conclude that when the training set has reached a certain size it describes statistical properties of the classes well enough and the decision border is more or less fixed. The error rate can not decrease further because the classes are partially intermixed.

The training set shows small error rate for the small sets, since the decision border is defined by the means of the training set and it was possible to divide the small sets nicely. For bigger training sets, the error rate for training set meets the error rate for the testing set—both those error rates are caused by the fact that classes are partially intermixed.

On the figure 22 we can also see a detailed plot of the last test from the previous experiment. It is the case where training set includes 500 of patterns from each class.

Instead of using minimum distance classifier, we could have used 1-NN (nearest neighbor) classifier. In that scenario an unknown pattern is put in the same class as its nearest neighbor from training patterns. Error rate dependency on training set size would be different in that case. The error rate for training set would be 0% for all training set sizes, since the 1-NN classifier always does perfectly on the training set. It is hard to imagine how would the error rate for test set look like, I assume it would be a bit worse than minimum difference classifier because of intermixed classes.

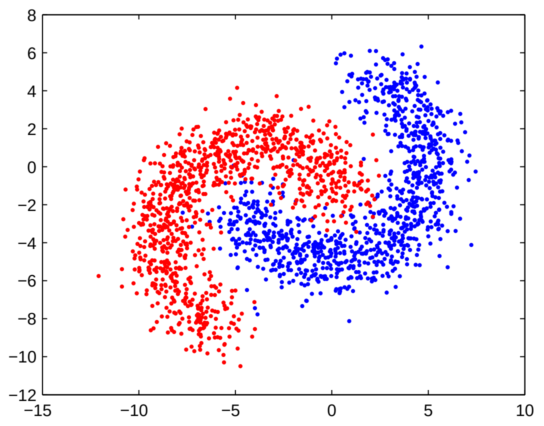


Figure 23: Another plot of the two classes: class 1 (blue) and class 2 (red). Each class again consists of 1000 patterns.

I think that the classes we used in this assignment look like two round [4] clouds, therefor I would assume that each of the classes is a Gaussians having the same standard deviation in all directions.I would even say that the two Gaussians have same standard derivation, and differ only in the mean value. If this is the case, it is a perfect setting for minimum distance classifier—no other classifier would yield a smaller error rate. That is because Bayes classifier for Gaussian pattern classes reduces to minimum distance classifier when classes have covariance matrices equal to the identity matrix and all classes are equally likely to occur. The error rate is in this case caused by the fact that the distance between means is not very large compared to standard deviation.

On the figure 23 we have a plot of the another two classes, again consisting of 1000 patterns each. Those two classes are not significantly intermixed—it is not hard to imagine the curve that separates the two classes, but it would be hard to define that curve analytically. Alternatively, we can say that those classes can not be approximated with a Gaussian, so using Bayers classifier even in its most general form would not yield a success.

I believe that 1-NN classification could do well on the case like this. The classes are not very intermixed and there should not be a lot of outliners in the training set that could cause wrong classification later. Using for example 5-NN classification would further improve the result.

[4]Plots were not made using `axis equal`, so circles look like ellipses.

15 Face Detection

from week 5: Color

Use the `children.tif` *test image for this exercise. Extract a region in the image (the larger the better) exclusively filled with facial skin. Make a histogram for each of the three color channels (red, green, and blue). For those channels with a distinctive peak in the histogram, establish a suitable range of values that correspond to skin. Combine these ranges in an expression like 'A pixel with color values (r,g,b) is skin if $0.6 < r < 0.8$ and $0.1 < g < 0.2$'. Try to use this expression to segment the faces from the test image. Is it possible?*

We need to segment the faces from the image by separately thresholding red, green and blue channel of the RGB image. The initial idea for thresholds should be found by looking at the three histograms of the image part containing facial skin. I used following MATLAB code to try to solve this problem.

```
C = imread('children.tiff');

% regions filled with facial skin
F = vertcat(horzcat(vertcat(C(61:80,101:160,:),C(131:170,221:280,:),...
    C(191:210,226:285,:)),C(131:210,221:250,:)),C(111:130,71:160,:));

figure(1), subplot(121), imagesc(C), axis image
figure(1), subplot(122), imagesc(F), axis image

% histograms for three chanels
figure(2), for i=1:3, subplot(3,1,i), imhist(F(:,:,i)), end

Sr = zeros(size(C,1),size(C,2));
Sg = zeros(size(C,1),size(C,2));
Sb = zeros(size(C,1),size(C,2));

% thresholding
Sr(find(150<C(:,:,1) & C(:,:,1)<250))=1;
Sg(find(90<C(:,:,2) & C(:,:,2)<210))=1;
Sb(find(40<C(:,:,3) & C(:,:,3)<190))=1;

figure(3), subplot(221), imagesc(Sr), colormap gray, axis image
figure(3), subplot(222), imagesc(Sg), colormap gray, axis image
figure(3), subplot(223), imagesc(Sb), colormap gray, axis image
figure(3), subplot(224), imagesc(Sr&Sg&Sb), colormap gray, axis image
```

On the figure 24 we can see the test image, and its three channels: red, green and blue. An image made by extracting the regions exclusively filled with facial skin and patching those together is on the figure 25, together with the corresponding histograms. By looking at the histograms, I chose initial values for thresholds

$$150 < r < 250, \quad 75 < g < 190, \quad 30 < b < 150$$

After trying different values, I adjusted the upper thresholds for green and blue channel, allowing more green and blue, otherwise were bright spots on the foreheads segmented out. I also made the lower threshold values for green and blue channel higher, hoping to eliminate as much of background as possible.

The results of thresholding and the final threshold values can be seen at the figure 26. We can see that it is the red channel contributing the most to the final segmentation. Trying to narrow allowed range for green and blue didn't yield desired result—blue and green channel are not contributing a lot to segmentation.

Looking at the final result, we can see that the faces were mostly segmented correctly, but there is still quite a lot of surroundings that sneaked in, especially the yellow clothes of a smaller child. The arms were of course also recognized as the facial skin.

When segmenting by thresholding the tree color channels separately, we ignore the relationship between the color intensities. If we, for example want to segment both dark and bright shades of the same color, we need to allow wide intensity range for all channels. Instead, we could say 'Allow a lot of blue, only if all three colors have high intensity values'. This could be done easier for HSI images, or maybe by thresholding both RGB and HSI components of an image.

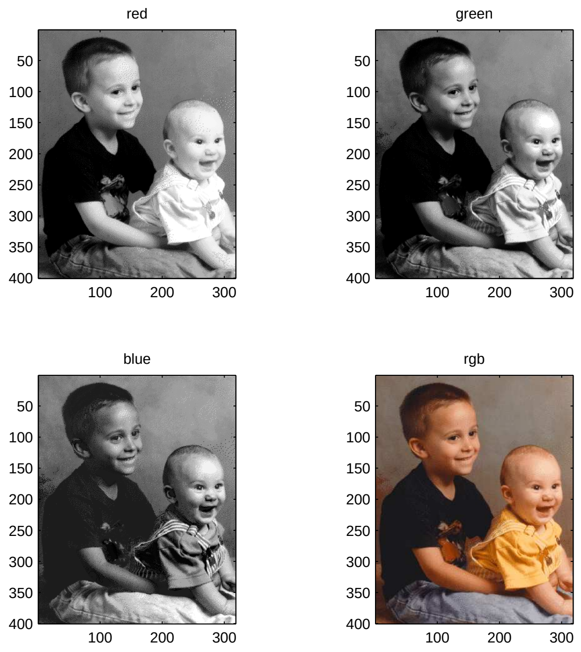


Figure 24: Red, green and blue channel of an color image, and the corresponding RGB image.

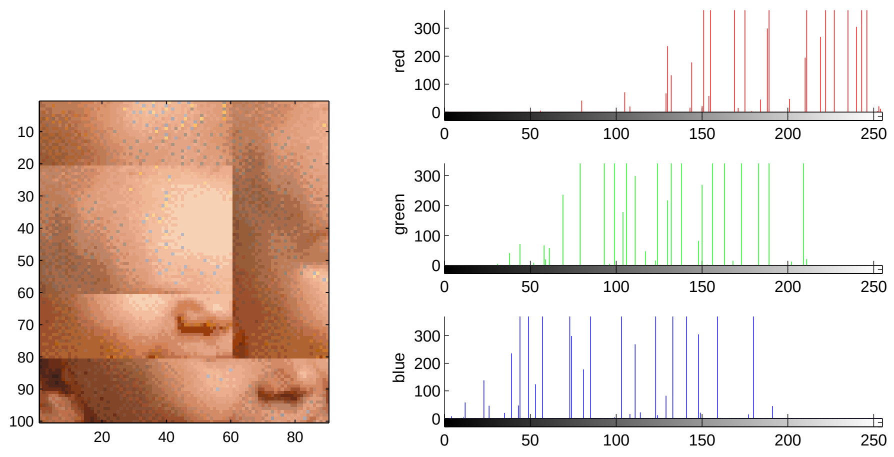


Figure 25: An RGB image made by patching the regions filled with facial skin, together with the histograms of red, green and blue channel.

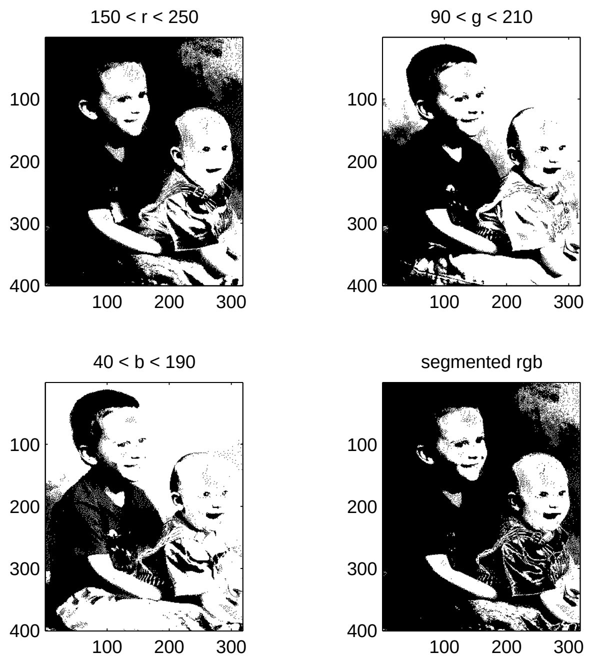


Figure 26: Results of segmentation by thresholding the three color channels separately, and the final result which is the product of the three segmented images.

16 Horn-Schunck Algorithm

from week 12: Motion

The very last batch of exercises:

- *Read Horn's chapter 12.*
- *Download the zip-file of a rotating sphere.*
- *Unzip the files to an appropriate place.*
- *Implement the Horn and Schunck's method to optic flow estimation.*

Horn-Schunck algorithm is an algorithm for estimating optical flow velocities. The algorithm is based on minimizing a combination of two errors, departure of smoothness e_s and the error in optical flow constrain e_c, i.e we minimize the sum

$$e_s + \lambda e_c$$

where λ is the weighting parameter.

If we looked just at optical flow constrain, we couldn't determine optical flow in, for example, uniformly bright regions or in the direction of the edges. Including the departure of smoothness in the minimization ensures that for the regions where the optic flow can not be found locally, it will be interpolated from the optical flow velocities in surrounding areas.

As shown in H [3], chapter 12, the minimization problem can be solved iteratively, using the following steps.

1. Estimate the spatial and time derivatives of brightness E_x, E_y and E_z, by using first differences in a $2 \times 2 \times 2$ cube of brightness.

2. Initialize optical flow components u and v (corresponding to x and y direction) by zeros.

3. Iterate a certain number of times over:

   - Calculate local averages $\bar{u}$ and $\bar{v}$ of u and v by looking at the first four neighbors.
   - Update the optical flow components as

$$u = \bar{u} - \frac{\lambda(E_x\bar{u} + E_y\bar{v} + E_t)}{1 + \lambda(E_x^2 + E_y^2)} E_x$$

$$v = \bar{v} - \frac{\lambda(E_x\bar{u} + E_y\bar{v} + E_t)}{1 + \lambda(E_x^2 + E_y^2)} E_y$$

The following MATLAB function is an implementation of the Horn-Schunck algorithm.

```
function [u,v] = Horn_Schunck(M,lambda,it)
% Horn-Schunck algorithm for motion estimation
% lambda - weighting parmeter, it - number of iterations
% [u v] - optical flow velocities in x and y direction

M=double(M);
[X,Y,T] = size(M);

% estimating partial derivates
x1=1:X-1; x2=2:X; y1=1:Y-1; y2=2:Y; t1=1:T-1; t2=2:T;
Ix = (M(x2,y1,t1)+M(x2,y1,t2)+M(x2,y2,t1)+M(x2,y2,t2)...
    -M(x1,y1,t1)-M(x1,y1,t2)-M(x1,y2,t1)-M(x1,y2,t2))/4;
Iy = (M(x1,y2,t1)+M(x1,y2,t2)+M(x2,y2,t1)+M(x2,y2,t2)...
    -M(x1,y1,t1)-M(x1,y1,t2)-M(x2,y1,t1)-M(x2,y1,t2))/4;
It = (M(x1,y1,t2)+M(x1,y2,t2)+M(x2,y1,t2)+M(x2,y2,t2)...
    -M(x1,y1,t1)-M(x1,y2,t1)-M(x2,y1,t1)-M(x2,y2,t1))/4;

u = zeros(X,Y,T);
v = zeros(X,Y,T);

% iteratively finding flow velocities
for m=1:it
    for t=1:T-1
        for x=2:X-1
            for y=2:Y-1
                ubar(x,y,t) = (u(x-1,y-1,t)+u(x-1,y+1,t)...
                    +u(x+1,y-1,t)+u(x+1,y+1,t))/4;
                vbar(x,y,t) = (v(x-1,y-1,t)+v(x-1,y+1,t)...
                    +v(x+1,y-1,t)+v(x+1,y+1,t))/4;
                alpha(x,y,t) = lambda*(Ix(x,y,t)*ubar(x,y,t)...
                    +Iy(x,y,t)*vbar(x,y,t)+It(x,y,t))...
                    /(1+lambda*(Ix(x,y,t)^2+Iy(x,y,t)^2));
                u(x,y,t) = ubar(x,y,t)-alpha(x,y,t)*Ix(x,y,t);
                v(x,y,t) = vbar(x,y,t)-alpha(x,y,t)*Iy(x,y,t);
            end
        end
    end
end
```

On the figure 27 we can see the results of applying Horn-Schunck algorithm to an image sequence showing a rotating sphere. Figure shows the x and the y component of the estimated optical flow after 5 and 50 iterations.

We can immediately see that the algorithm correctly estimated background to be still [5] and the sphere to be moving. One could actually easily segment the sphere by thresholding the absolute value of the optical flow.

[5] The difference in the color of the background is introduced by scaling the intensity range of the image, so zero was moved from the center of the range.

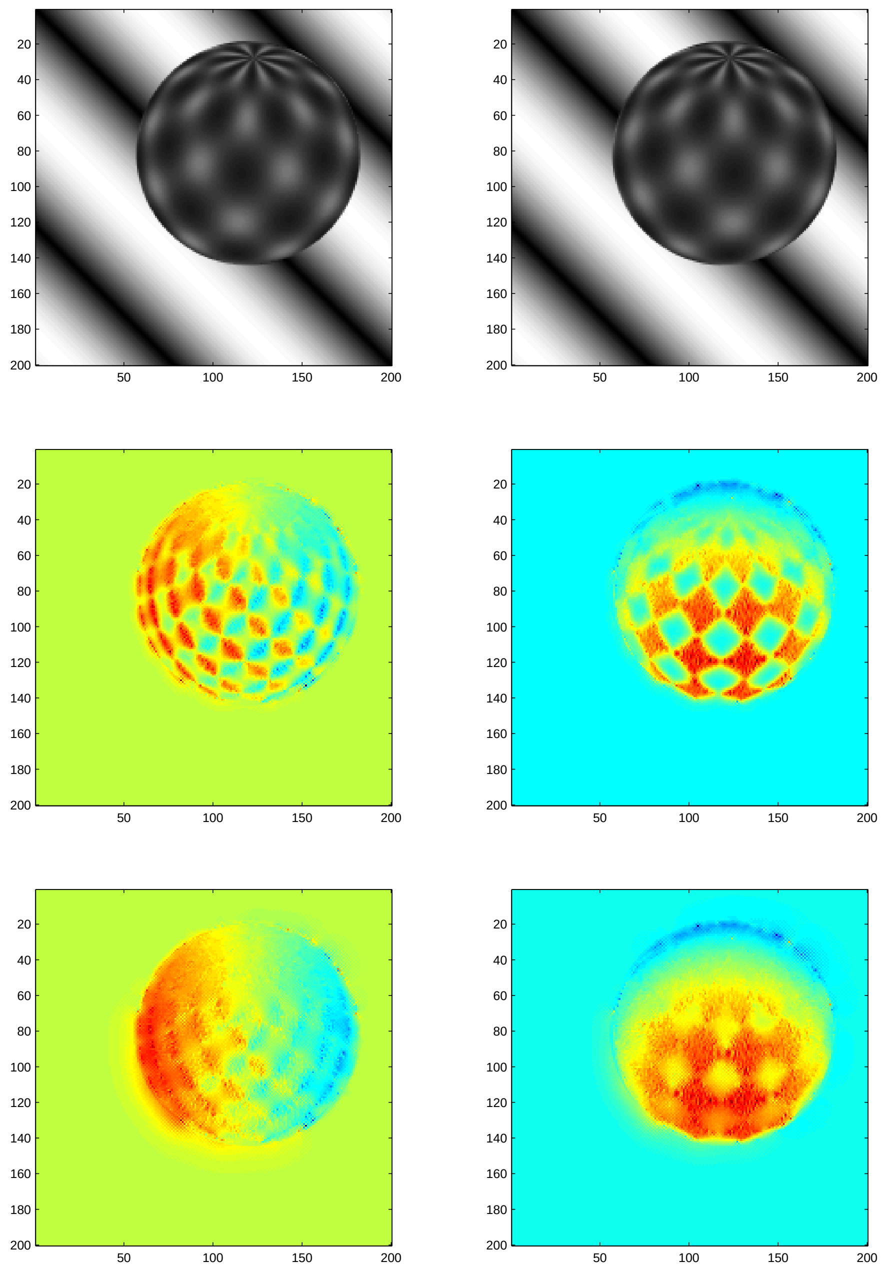


Figure 27: The first and the fifth image of a sequence showing a rotating sphere (first line), together with estimates of the optical flow between the second and the third image of the sequence, obtained using Horn-Schunck algorithm: after 5 iterations (middle line) and after 50 iterations (bottom line). On the left are the optical flow velocities in *x* direction (up–down), and on the right are the flow velocities in the *y* direction (left–right).

We can also see that the directions of the movement are generally estimated correctly: left part of the sphere is moving down (red) and left is moving up (blue), the front is moving right (red), and the bottom left (blue).

However, the uniformly bright parts of the sphere are after 5 iterations estimated to be still. As we can see, after 50 iterations the algorithm has smoothed the optical flow of the sphere's surface, but it also smoothed the optical flow out of its silhouette introducing unwanted effects. The solution to that problem would be to incorporate the segmentation into the iterative solution for the optical flow.

References

[1] Tim Cootes. An introduction to active shape models. 2000.

[2] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing*. Prentice-Hall, second edition, 2002.

[3] Berthold K. P. Horn. *Robot Vision*. The MIT Press, 1986.

[4] Tony Lindberg. Scale-space: A framework for handling image structures at multiple scales. *Proc. CERN School of Computing*, 1996.

[5] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice-Hall, 1998.