Image Processing
lab 5

Klaas Kliffen Jan Kramer

January 22, 2016

Exercise 1 – Edge detection

a. The MarrHildreth edge detection algorithm implemented here follows the steps outlined in the book.

- convolve the image with a Gaussian filter
- take the Laplacian of the result
- detect zero crossings including a threshold

Note that we could also have chosen to convolve the Gaussian filter with the Laplacian mask and apply that to the image with the same result. In addition we could have evaluated the Laplacian Gaussian function to create the filter. This last method has the drawback that the sum of the created filter is not necessarily zero, which is a requirement. Our implemtation is as follows:

```
function im = IPMarrHildreth (x, sigma)
  fx = double(x);
  N = ceil(6*sigma);
  % correct for meshgrid result size
  if (mod(N,2) == 1)
    N = N - 1;
  end
  [U, V] = meshgrid(-N/2:N/2, -N/2:N/2);
  % calculate a gaussian filter
  rs = (U .* U + V .* V)/(2*sigma^2);
  g = (1/(2*pi*sigma^2))*e.^-rs;
  % pad the image
  xd1 = size(x, 1);
  xd2 = size(x, 2);
  bd1 = size(g, 1);
  bd2 = size(g, 2);
  a = zeros(size(x) + 2 * size(g));
  a(bd1+1:(bd1 + xd1), bd2+1:(bd2 + xd2)) = fx;
  %% apply the gaussian filter
  a = conv2(a, g, 'same');
  %% compute the Laplacian
  a = conv2(a, [1 1 1; 1 -8 1; 1 1 1], 'same');
  % unpad
  fx = a(bd1+1:(bd1 + xd1), bd2+1:(bd2 + xd2));
```

```
25    %% detect zero crossings
26    % prepare output image
27    im = zeros(size(x), 'logical');
28    % shift the result and pad edges with nan
29    nr = size(fx, 1);
30    nc = size(fx, 2);
31    x_u = [fx(2:nr, :); NaN([1 nc])];
32    x_d = [NaN([1 nc]); fx(1:(nr - 1), :)];
33    x_l = [fx(:, 2:nc) NaN([nr 1])];
34    x_r = [NaN([nr 1]) fx(:, 1:(nc - 1))];
35    x_ul = [x_u(:, 2:nc) NaN([nr 1])];
36    x_ur = [NaN([nr 1]) x_u(:, 1:(nc - 1))];
37    x_dl = [x_d(:, 2:nc) NaN([nr 1])];
38    x_dr = [NaN([nr 1]) x_d(:, 1:(nc - 1))];
39    % compute differences in the four directions
40    d_ud = abs(x_u - x_d);
41    d_lr = abs(x_l - x_r);
42    d_x1 = abs(x_ul - x_dr);
43    d_x2 = abs(x_ur - x_dl);
44    % calculate the threshold based on maximum value of the absolute
          differences
45    mud = max(max(d_ud));
46    mlr = max(max(d_lr));
47    mx1 = max(max(d_x1));
48    mx2 = max(max(d_x2));
49    mv = max([mud mlr mx1 mx2]);
50    t = 0.15 * mv;
51    % check zero crossings
52    im = (((x_u < 0 & x_d > 0)   | (x_u > 0 & x_d < 0))   & d_ud > t) |
          ...
53          (((x_l < 0 & x_r > 0)   | (x_l > 0 & x_r < 0))   & d_lr > t)
              | ...
54          (((x_ul < 0 & x_dr > 0) | (x_ul > 0 & x_dr < 0)) & d_x1 > t)
              | ...
55          (((x_ur < 0 & x_dl > 0) | (x_ur > 0 & x_dl < 0)) & d_x2 > t);
56  end
```

Note that meshgrid is used to make the Gaussian filter. To get a $n \times n$ filter as in the book the arguments to meshgrid should be in the interval $[n/2, n/2]$. However in the case that n is odd, then we get the wrong dimensions so one is subtracted to correct it. In addition n is based on the 6σ suggestion in the book. Next the image is padded with zeros to prevent artifacts at the edges and the filters are applied. The laplacian filter is the one given in to book, so there is not really something interesting there.

After this the image is unpadded, so we can start on the zero crossing detection. This is done by comparing the signs of values opposite of each other in a 3×3 neighborhood. So there are 4 directions that have to be checked: horizontal, vertical and two diagonal directions. The code is based on the shifting images technique to work with these neighborhoods efficiently. Note that thresholding is also incorporated. The threshold is based on the maximum absolute difference in all directions. Alternatively we could have used the maximum absolute value, but this seemed less natural because we are comparing it with the absolute difference. The overall result is a binary image with two value true on places where the edges are.

b. The results of applying the algorithm can be seen in Figure 1. The best result we got was the one in the upper right corner, for this one we used a σ of 0.6% of the smallest image dimension. While there is still some noise, most on the main edges are clearly visible. In the lower left corner we have a result with the same *sigma*, but without thresholding. The image now has way too much noise to be useful.

The image in the lower right corner has thresholding, but only a σ of 0.1% of the smallest image dimension. This image shows that if the σ is too low, some noise will show up in the results. Also note that there is an edge at the edge of the image. This is probably an implementation error, because in this case there should not be a straight edge in the air or at the right of the house at the bricks. One possible reason could be that the convolution is not implemented correctly. Another theory is that maybe this is the result of the Gaussian filter being cut off too much. Or maybe the Gaussian is not calculated correctly. Sadly we could not confirm that any of these were the cause of the issue.

The original image

$\sigma = 1.536$ and a threshold of 15%

$\sigma = 1.536$ and a threshold of 0%

$\sigma = 0.256$ and a threshold of 15%

Figure 1: MarrHildreth edge detection with various parameters.

Exercise 2 – Region splitting and merging

a. There are two main objects in the scene, the tiger and the sky. The tiger consists of a large range of grey levels and complex patterns. The sky however is smooth and consists of a well defined range of grey levels. Therefor it less complex to define a piece of sky. Opening Figure 2 in an image editor gave the value 107 as a lower bound for the dark regions and 168 for brighter regions. Therefor a suitable predicate would be that all pixels in the regions are with these bounds. For the actual implementation a slightly larger bound

is used, since some areas where still a bit brighter than measured and therefor the final bounds used are $[100, 180]$.

Figure 2: The original image with the tiger

b. The predicate implementation is quite simple. It mainly consists of two large matrices used to detect values above the lower bound an below the higher bound. The difference is taken and should sum to 0 if all values in the region are with the bounds. However, since the image is padded by the split and merge algorithm to a power of two. It is possible that around the border of an image some values are zero in a region. This is avoided by setting all pixels in the region with a zero grey level to 128 (a value in between the bounds).

```
function flag = IPpredicate(region)

  % set the boundary (zero values) to a value in the sky
  % this prevent the black line on the side
  boundary = zeros(size(region));
  boundary = region == boundary;
  boundary = boundary .* 128;

  % fix the boundary
  region += boundary;

  % 107 till 168 grey levels for the sky
  low = ones(size(region)) * 100;
  high = ones(size(region)) * 180;

  highpass = region > low;
  lowpass = region < high;

  diff = highpass .- lowpass;
  if (sum(sum(diff)) == 0)
    flag = true;
  else
    flag = false;
  end
end
```

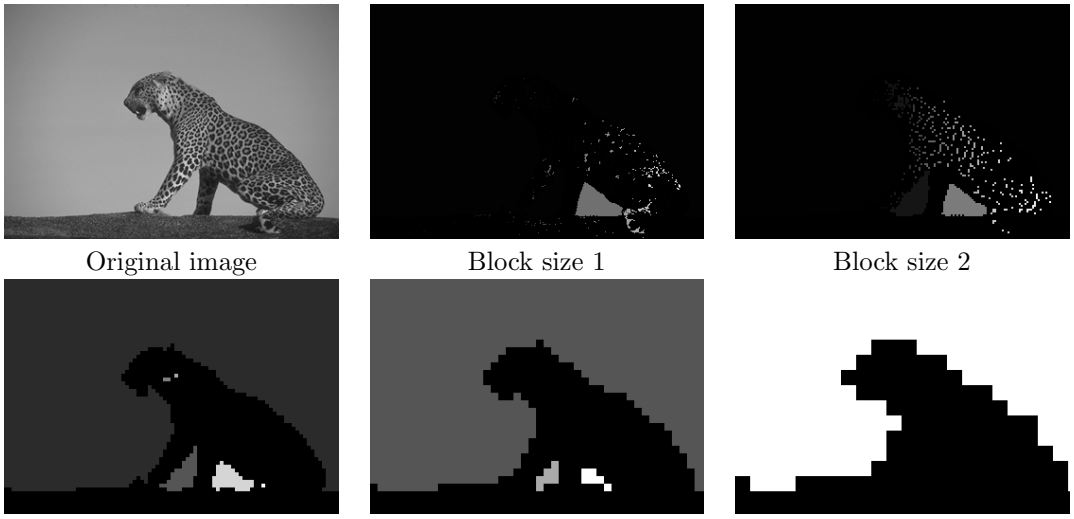


Figure 3: The partitioning scheme on the tiger image with different minimum block sizes

In Figure 3 the results can be seen with various minimum block sizes. The best block size for this particular image seems to be a size of 4. Anything smaller has too much connected blocks, where the tiger is completely invisible. When comparing the block size of 8 with 4, the tiger has no gaps, but the contour is less detailed. A block size of 4 still can be recognized as a tiger. A block size of 16 is useless, since it there is no more detail to be found in the image.

Exercise 3 – Fourier descriptors

a. The starting point of the boundary is determined by systematically scanning each horizontal row. It starts in the center and checks if the row has values other than the background value. If not, two new rows are checked, each at a quarter of the image, until there are no more rows to scan and it will throw an error. Having found the starting point, the `bwtraceboundary` function is called.

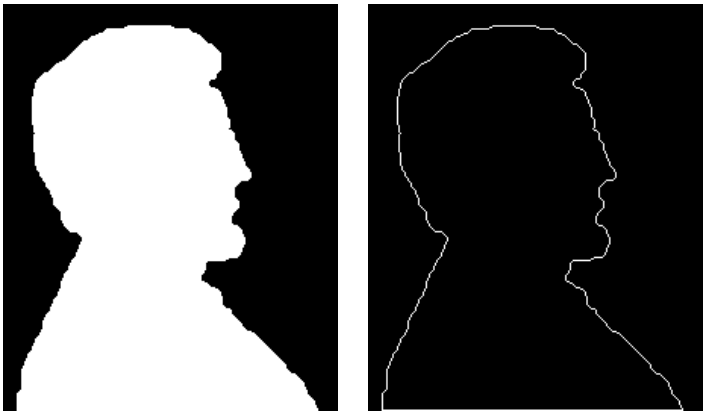
```
function [start,contour] = IPcontour(img)

% Find starting point
[~,h] = size(img);
starty = ceil(h/2);
searching = true;
stack = [];
startx = 0;
stack(end+1) = starty;

while (searching == true)
    starty = stack(1);
    stack(1)=[];
    if (max(img(starty,:)) > 0)
        % Found a point with an object
        startx = find(img(starty,:), 1);
```

```
17          searching = false;
18      else
19          % Do something with starty
20          half = round(starty/2);
21          if (starty-half == 0 || starty+half > h)
22              % reached the edge, no objects found, throw an error
23              error('Could not find any object');
24          end
25          % put two new values on the stack
26          % half the difference and subtract/add it from the current
                value
27          stack(end+1) = starty-half;
28          stack(end+1) = starty+half;
29      end
30  end
31
32  start = [starty, startx];
33  contour = bwtraceboundary(img,start,'N');
34
35  end
```

b. The starting point of the boundary using the method described above is at x: 25, y: 111. The complete boundary can be seen in Figure 4



The original image The contour

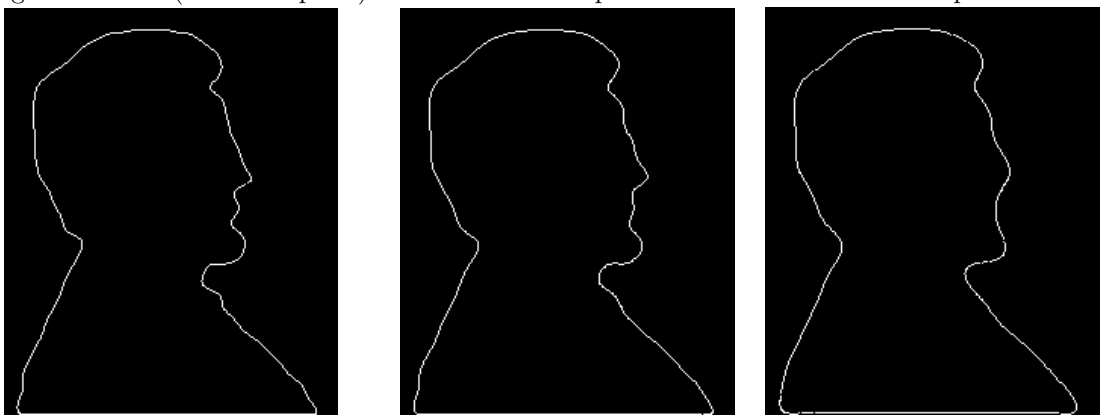
Figure 4: The contour of the silhouette of Lincoln

c. The Fourier descriptors are calculated by the build in function fft. Since the center of the array resemble the higher frequencies, the number of descriptors to be kept is at the start and the end of the array. With a simple for loop, each of the center values can be set to 0. The resulting approximating boundary is then retrieved by the rounding the real part of the inverse Fourier transformation.

```
1  function [contour] = IPfourierdescr(boundary,P)
2
3  transform = fft(boundary);
4  [l,~] = size(transform);
```

```
5
6  % Check P value
7  if (P > 1/2)
8     error('P too large! Must be <= length(boundary)/2');
9  end
10
11 % Loop over all values, set the high frequency
12 % fourier descriptors to zero.
13 for i = P : (1-P)
14       transform(i,:) = complex([0 0],0);
15 end
16 contour = round(real(ifft(transform)));
17 end
```

Original contour (410 descriptors) | 100 descriptors | 75 descriptors



50 descriptors | 38 descriptors | 25 descriptors

Figure 5: The contour of the Lincoln silhouette using different amounts of Fourier descriptors

As can be seen in Figure 5, the minimal number of descriptors needed for a recognizable silhouette is somewhere around 38. Using less descriptors will smooth the surface, where at 25 descriptors most features are smoothed and it takes effort to recognize the silhouette of Lincoln.

Task distribution

ex1	design	implementation	answers questions	writing report
Klaas	20%	10%	20%	30%
Jan	80%	90%	80%	70%

ex2	design	implementation	answers questions	writing report
Klaas	50%	80%	75%	75%
Jan	50%	20%	25%	25%

ex3	design	implementation	answers questions	writing report
Klaas	80%	75%	75%	80%
Jan	20%	25%	25%	20%