Image Processing and Computer Vision (IPCV)

Prof. Dr. J. Weickert — Summer term 2017
Mathematical Image Analysis Group — Saarland University

Example Solutions for Homework Assignment 7 (H7)

Problem 1: (Structure Tensor Analysis)

(a) Since the matrix $\boldsymbol{J} \in \mathbb{R}^{n\times n}$ is symmetric with real components, we know from linear algebra that it has n real eigenvalues $\lambda_1, \dots, \lambda_n$, and there is an orthonormal basis of eigenvectors $\{\boldsymbol{w}_1, \dots, \boldsymbol{w}_n\}$. We assume that the eigenvalues are ordered such that $\lambda_1 \le \lambda_2 \le \dots \le \lambda_n$. We can write $\boldsymbol{J} = \boldsymbol{Q}^\top \boldsymbol{D}\boldsymbol{Q}$ with

$$\boldsymbol{D} = \operatorname{diag}(\lambda_1, \dots, \lambda_n) \quad \text{and} \quad \boldsymbol{Q}^\top = (\boldsymbol{w}_1, \dots, \boldsymbol{w}_n)\ .$$

The matrix $\boldsymbol{Q}$ is orthogonal here, which means that $\boldsymbol{Q}^\top\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}^\top = \boldsymbol{I}$ and $\|\boldsymbol{Q}^\top\boldsymbol{v}\|_2 = \|\boldsymbol{Q}\boldsymbol{v}\|_2 = \|\boldsymbol{v}\|_2$ for all $\boldsymbol{v} \in \mathbb{R}^n$. The quadratic form E can then be written as

$$E(\boldsymbol{v}) = \boldsymbol{v}^\top\boldsymbol{J}\boldsymbol{v} = \boldsymbol{v}^\top\boldsymbol{Q}^\top\boldsymbol{D}\boldsymbol{Q}\boldsymbol{v} = (\boldsymbol{Q}\boldsymbol{v})^\top\boldsymbol{D}\boldsymbol{Q}\boldsymbol{v}$$

We abbreviate $\boldsymbol{x} = \boldsymbol{Q}\boldsymbol{v}$ and know that $\|\boldsymbol{x}\|_2 = \|\boldsymbol{Q}\boldsymbol{v}\|_2 = 1$. Thus we can write

$$E(\boldsymbol{v}) \;=\; \boldsymbol{x}^\top\boldsymbol{D}\boldsymbol{x} \;=\; \sum_{i=1}^{n}\lambda_i x_i^2 \;\ge\; \sum_{i=1}^{n}\lambda_1 x_i^2 \;=\; \lambda_1\|\boldsymbol{x}\|^2 \;=\; \lambda_1$$

for all $\boldsymbol{v} \in \mathbb{R}^n$ with $\|\boldsymbol{v}\|_2 = 1$. Now we know that λ_1 is a lower bound for the range of the function E.

Setting $\boldsymbol{v} = \boldsymbol{Q}^\top(1, 0, \dots, 0)^\top$ we obtain $E(\boldsymbol{v}) = \lambda_1$ that means our bound is sharp. From the definition of $\boldsymbol{Q}^\top$ we see that $\boldsymbol{Q}^\top(1, 0, \dots, 0)^\top = \boldsymbol{w}_1$. Therefore, the quadratic form E is minimised among all unit vectors by the eigenvector corresponding to the smallest eigenvalue of $\boldsymbol{J}$.

If $\boldsymbol{J}$ is positive definite, we know that all eigenvalues are positive. Since the smallest eigenvalue is the minimum of E, the function E is positive in this case.

(b) In the case $\rho = 0$, the outer convolution in the definition of the structure tensor is left out:

$$\boldsymbol{J}_0 \;=\; K_0 * \left(\boldsymbol{\nabla} f \boldsymbol{\nabla} f^\top\right) \;=\; \boldsymbol{\nabla} f \boldsymbol{\nabla} f^\top \;.$$

We assume $\boldsymbol{\nabla} f \neq 0$ and take a look at the principle axis decomposition of $\boldsymbol{J}_0$:

$$\boldsymbol{J}_0 \;=\; \frac{1}{\|\boldsymbol{\nabla} f\|} \begin{pmatrix} f_x & f_y \\ f_y & -f_x \end{pmatrix} \begin{pmatrix} \|\boldsymbol{\nabla} f\|^2 & 0 \\ 0 & 0 \end{pmatrix} \frac{1}{\|\boldsymbol{\nabla} f\|} \begin{pmatrix} f_x & f_y \\ f_y & -f_x \end{pmatrix} \;.$$

We see that $\boldsymbol{J}_0$ has only one non-zero eigenvalue with corresponding eigenvector $\boldsymbol{\nabla} f$ (see also assignment C7, Problem 1). That means, without outer convolution, the structure tensor only yields the gradient of f as useful information. Thus it cannot be used for corner detection.

Problem 2 (Morphological Operators)

The defined group of operations are the so-called *morphological derivatives*:

External gradient: $\rho_B^+(f) := \quad A_B := (f \oplus B) - f$

- approximates $f'; \boldsymbol{\nabla} f$;
- thickness for a step edge is one pixel;
- enhances external boundaries of objects brighter than the background;

Internal gradient: $\rho_B^-(f) := \quad B_B := f - (f \ominus B)$

- approximates $f'; \boldsymbol{\nabla} f$;
- thickness for a step edge is one pixel;
- enhances internal boundaries of objects brighter than the background;

Beucher gradient: $\rho_B(f) := \quad C_B := (f \oplus B) - (f \ominus B)$

- approximates $f'; \boldsymbol{\nabla} f$;
- the thickness of detectable step edge is two pixels;
- outputs the maximum variation of the grey levels within the neighbourhood of the structuring element;
- invariant with respect to complementation;

Morphological Laplacian:
$$\Delta_B(f) := \quad D_B := \rho_B^+(f) - \rho_B^-(f) = (f \oplus B) - 2 \cdot f + (f \ominus B)$$

- approximates $f''; \Delta f$;

We apply those operations to the signal $\boldsymbol{f} = (\ldots, 0, 0, 0, 0, \mathbf{1}, 1, 1, 1, \ldots)$ using a symmetric structuring element of size $2m+1$ with $m = 1$. The i-th symbol is always given in bold face.

$(f \oplus B) = (\ldots, 0, 0, 0, 1, \mathbf{1}, 1, 1, 1, \ldots)$
$(f \ominus B) = (\ldots, 0, 0, 0, 0, \mathbf{0}, 1, 1, 1, \ldots)$

(a) $\rho_B^+(f) = (\dots, 0, 0, 0, 1, \mathbf{0}, 0, 0, 0, \dots)$

(b) $\rho_B^-(f) = (\dots, 0, 0, 0, 0, \mathbf{1}, 0, 0, 0, \dots)$

(c) $\rho_B(f) = (\dots, 0, 0, 0, 1, \mathbf{1}, 0, 0, 0, \dots)$

(d) $\Delta_B(f) = (\dots, 0, 0, 0, 1, \mathbf{-1}, 0, 0, 0, \dots)$

These results are similar to the results which are obtained by the derivative filters of Lecture 12:

- first derivative with backward differences with grid size $h = 1$
  $$\frac{f_i - f_{i-1}}{h} = (\dots, 0, 0, 0, 0, \mathbf{1}, 0, 0, 0, \dots)$$
- first derivative with forward differences with grid size $h = 1$
  $$\frac{f_{i+1} - f_i}{h} = (\dots, 0, 0, 0, 1, \mathbf{0}, 0, 0, 0, \dots)$$
- first derivative with central differences with grid size $h = 1$
  $$\frac{f_{i+1} - f_{i-1}}{2h} = (\dots, 0, 0, 0, 0.5, \mathbf{0.5}, 0, 0, 0, \dots)$$
- second derivative with central differences with grid size $h = 1$
  $$\frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} = (\dots, 0, 0, 0, 1, \mathbf{-1}, 0, 0, 0, \dots)$$

Problem 3 (Edge and Corner Detection)

(a) The code for the routine `getderivatives` is given by

```
/*---------------------------------------------------------------------*/

void getderivatives
(
    float **u,          // in:  image
    float **dx,         // out: derivatives in x-direction
    float **dy,         // out: derivatives in y-direction
    int nx,             // in:  x-dimension of u
    int ny              // in:  y-dimension of u
)

// compute the derivatives in x and y direction;
// the grid size h is assumed to be 1

{
    int i,j;

    // compute the derivatives for channel m using the sobel operator
    for (i=1; i<nx+1; ++i)
        for (j=1; j<ny+1; ++j)
        {
            dx[i][j] = (-     u[i-1][j-1] +     u[i+1][j-1]
                        - 2 * u[i-1][j]   + 2 * u[i+1][j]
                        -     u[i-1][j+1] +     u[i+1][j+1])
                        /8.0;

            dy[i][j] = (-     u[i-1][j-1] -  2 * u[i][j-1]
                        -     u[i+1][j-1] +      u[i-1][j+1]
                        + 2 * u[i][j+1]   +      u[i+1][j+1])
                        /8.0;
        }
}

/*---------------------------------------------------------------------*/
```

1) Figure 1 shows some edge images for `objects.pgm` with different parameter selection. It is not hard to find reasonable parameters due to the fact that the original image is a binary black-and-white image having clear sharp edges. Most of the parameters for T_1 and T_2 will create identical results as it can be seen in the left and middle image of figure 1. The image does not contain noise. Thus σ can be set to 0. In contrast, large values for σ will delocalise the edges. In addition, due to the smoothing, it is possible that for large values for σ edges are detected at positions where actually no edges are. This statement is confirmed by the right image of figure 1. Little white spots appear in between objects with narrow distance to each other.

2) In contrast to `objects.pgm`, the test image `pruebab1.pgm` is very noisy. Hence, setting σ to 0 cannot give reasonable results (see figure 2, left image). If we set σ to 2 we can recognise the shape of the triangle and the rectangle already (figure 2, middle). However there are still some edges detected where actually no edges are. So we increase σ further to 4 and get finally a relatively reasonable result (figure 2, right side). Due to the large value for σ the edges are not perfectly localised. Nevertheless this is the best result we can obtain and considering the given degradations caused by noise it is quite feasible. This shows that the canny edge detector is robust under noise. The parameters T_1 and T_2 are chosen such that the edges of the triangle and the reactangle appear as closed contours.

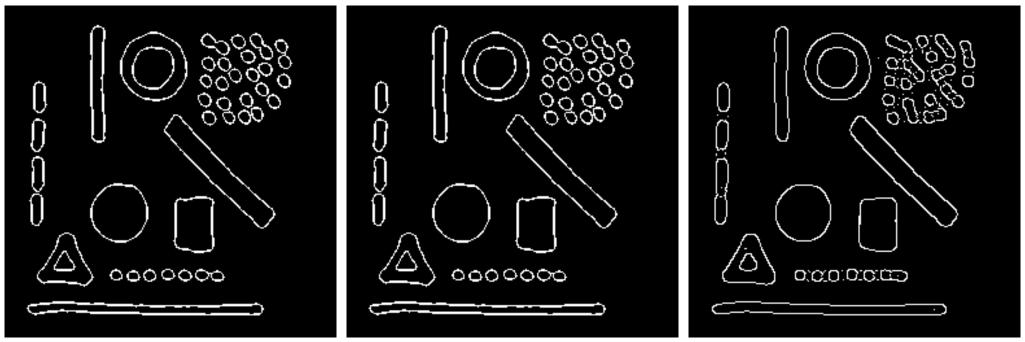


Figure 1: Edge images for `objects.pgm`: *Left:* $\sigma = 0$, $T_1 = 0$, $T_2 = 0.1$; *middle:* $\sigma = 0$, $T_1 = 46$, $T_2 = 120$; *right:* $\sigma = 3$, $T_1 = 0$, $T_2 = 0.1$.

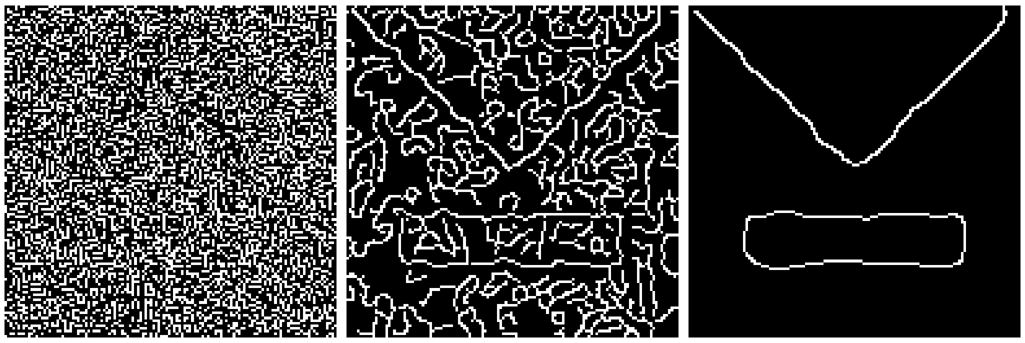


Figure 2: Edge images for `pruebab1.pgm`: *Left:* $\sigma = 0$, $T_1 = 2$, $T_2 = 6$; *middle:* $\sigma = 2$, $T_1 = 2$, $T_2 = 6$; *right:* $\sigma = 4$, $T_1 = 2$, $T_2 = 6$.

(b) To complete the implementation of the function `struct_tensor` we have to add the computation of the approximated partial derivatives. Here we have chosen to use Sobel operators for an approximation of v_x and v_y. We have to take care of boundary pixels: Before we compute the derivatives we mirror the boundaries (by calling the function `dummies`) to make sure that all pixels used during computation are filled with sensible values.

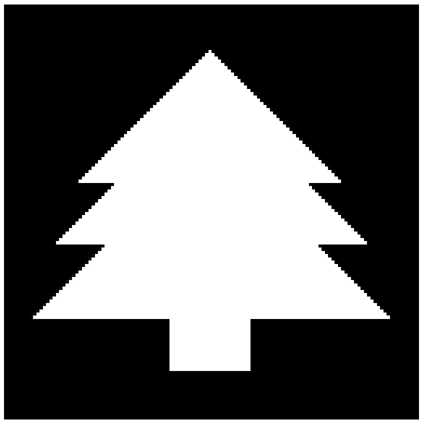
```
/* ---------------------------------------------------------------- */
void struct_tensor
 (float    **v,    /* image !! gets smoothed on exit !! */
  long     nx,     /* image dimension in x direction */
  long     ny,     /* image dimension in y direction */
  float    hx,     /* pixel size in x direction */
  float    hy,     /* pixel size in y direction */
  float    sigma,  /* noise scale */
  float    rho,    /* integration scale */
  float    **dxx,  /* element of structure tensor, output */
  float    **dxy,  /* element of structure tensor, output */
  float    **dyy)  /* element of structure tensor, output */

/* Calculates the structure tensor. */

{
long    i, j;                 /* loop variables */
float   dv_dx, dv_dy;         /* derivatives of v */
float   w1, w2, w3, w4;       /* time savers */

/* ---- smoothing at noise scale, reflecting b.c. ---- */

if (sigma > 0.0)
   gauss_conv (sigma, nx, ny, hx, hy, 5.0, 1, v);


/* ---- calculate gradient and its tensor product ---- */
dummies(v, nx, ny);

for(i=1; i<=nx; i++)
  for(j=1; j<= ny; j++)
    {
        /* compute the derivatives using Sobel operators */
        w1 = 1.0 / (8.0 * hx);
        w2 = 1.0 / (4.0 * hx);
        dv_dx = w1 * (  v[i+1][j+1] - v[i-1][j+1]
                      + v[i+1][j-1] - v[i-1][j-1])
              + w2 * (  v[i+1][j  ] - v[i-1][j  ]);

        w3 = 1.0 / (8.0 * hy);
        w4 = 1.0 / (4.0 * hy);

        dv_dy = w3 * (  v[i+1][j+1] - v[i+1][j-1]
                      + v[i-1][j+1] - v[i-1][j-1])
              + w4 * (  v[i  ][j+1] - v[i  ][j-1]);

         dxx[i][j] = dv_dx * dv_dx;
         dxy[i][j] = dv_dx * dv_dy;
         dyy[i][j] = dv_dy * dv_dy;
   }

/* ---- smoothing at integration scale, Dirichlet b.c. ---- */
if (rho > 0.0)
  {
     gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dxx);
     gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dxy);
     gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dyy);
  }

return;
}
/* ------------------------------------------------------------ */
```

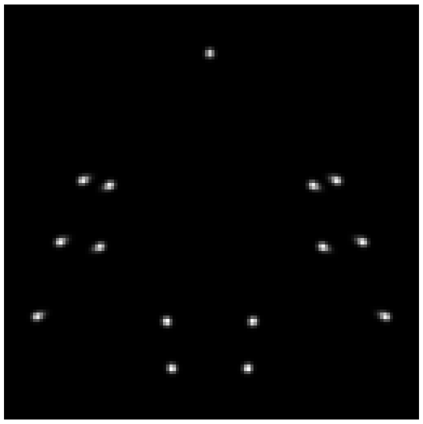
The cornerness can for instance be implemented as the determinant of the structure tensor here. Then the code looks as follows:

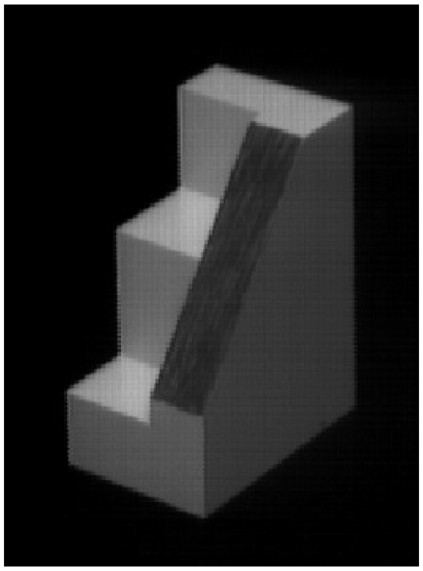
```
/* ------------------------------------------------------------ */
void cornerness
 (float    **u,    /* image !! gets smoothed on exit !! */
  long     nx,     /* image dimension in x direction */
  long     ny,     /* image dimension in y direction */
  float    hx,     /* pixel size in x direction */
  float    hy,     /* pixel size in y direction */
  float    sigma,  /* noise scale */
  float    rho,    /* integration scale */
  float    **v)    /* output */
```

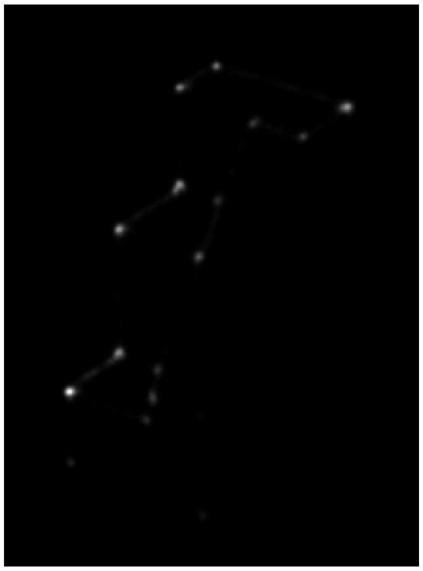
```
/*
 calculates cornerness in each pixel;
 it is evaluated as the determinant of the structure tensor;
*/

{
long    i, j;                   /* loop variables */
float   **dxx, **dxy, **dyy;   /* tensor components */

/* allocate storage */
alloc_matrix (&dxx, nx+2, ny+2);
alloc_matrix (&dxy, nx+2, ny+2);
alloc_matrix (&dyy, nx+2, ny+2);

/* calculate structure tensor */
struct_tensor (u, nx, ny, hx, hy, sigma, rho, dxx, dxy, dyy);

/* cornerness */
for(i=1; i <=nx; i++)
  for(j=1; j<=ny; j++)
     v[i][j] = dxx[i][j] * dyy[i][j] - dxy[i][j] * dxy[i][j];

/* free storage */
disalloc_matrix (dxx, nx+2, ny+2);
disalloc_matrix (dxy, nx+2, ny+2);
disalloc_matrix (dyy, nx+2, ny+2);

return;
}
/* ------------------------------------------------------------- */
```

Note that higher values of the parameters σ and ρ lead to fuzzy edge detections.

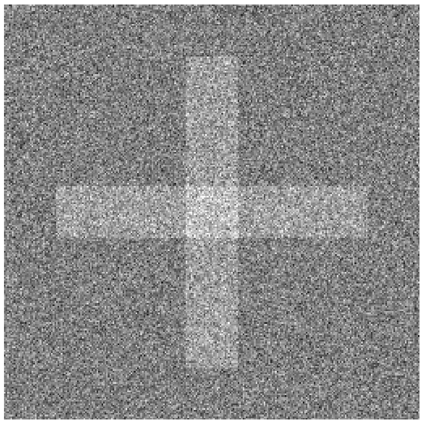

Original image `tree.pgm`

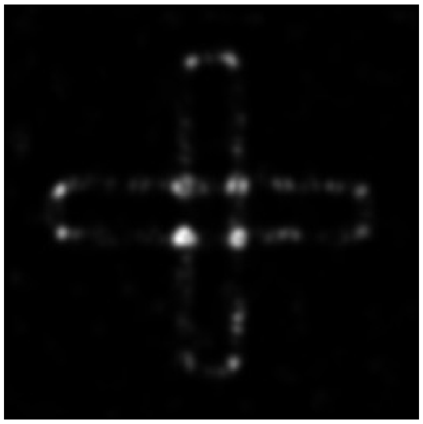

Corner detection, $\sigma = 0.0$, $\rho = 1.0$


Original image `stairs.pgm`


Corner detection, $\sigma = 1.0$, $\rho = 2.0$

The example `acros.pgm` is degraded with noise. We can reduce the influence of this noise by choosing a higher value σ.


Original image `acros.pgm`


Corner detection, $\sigma = 3.0$, $\rho = 3.0$

Problem 4 (Morphological Operations)

With the subroutines for erosion and dilation one can implement the missing subroutines according to the lecture notes.

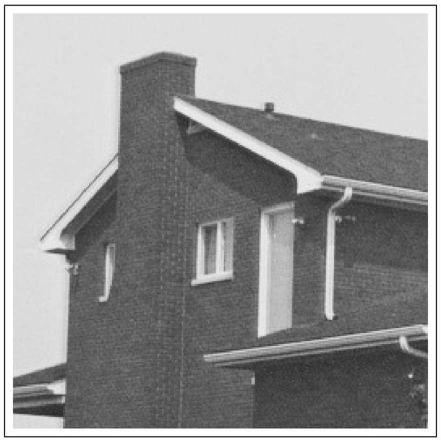
```
/*--------------------------------------------------------------------*/

void closing
 (long     nx,   /* image dimension in x direction */
  long     ny,   /* image dimension in y direction */
  long     m,    /* size of structuring element: (2m+1)*(2m+1) */
  float    **u)  /* input: original image; output: processed */

/*
 Closing with a square of size (2m + 1) * (2m + 1) as
 structuring element.
*/

{
dilation(nx, ny, m, u);
erosion(nx, ny, m, u);
return;
}

/*--------------------------------------------------------------------*/

void opening
 (long     nx,   /* image dimension in x direction */
  long     ny,   /* image dimension in y direction */
  long     m,    /* size of structuring element: (2m+1)*(2m+1) */
  float    **u)  /* input: original image; output: processed */

/*
 Opening with a square of size (2m + 1) * (2m + 1) as
 structuring element.
*/

{
erosion(nx, ny, m, u);
dilation(nx, ny, m, u);
return;
}

/*--------------------------------------------------------------------*/

void white_top_hat
 (long     nx,   /* image dimension in x direction */
  long     ny,   /* image dimension in y direction */
  long     m,    /* size of structuring element: (2m+1)*(2m+1) */
  float    **u)  /* input: original image; output: processed */

/*
 White top hat with a square of size (2m + 1) * (2m + 1) as
 structuring element.
*/

{
long    i, j;       /* loop variables */
float   **uo;       /* opening of input image */

alloc_matrix (&uo, nx+2, ny+2);

/* copy u into uo */
 for(i=1; i<=nx; i++)
     for(j=1; j<=ny; j++)
         uo[i][j] = u[i][j];

/* perform opening */
opening(nx, ny, m, uo);

/* subtract opening from u */
for(i=1; i<=nx; i++)
    for(j=1; j<=ny; j++)
         u[i][j] = u[i][j] - uo[i][j];

disalloc_matrix (uo, nx+2, ny+2);
return;

} /* white_top_hat */

/*--------------------------------------------------------------------*/
```

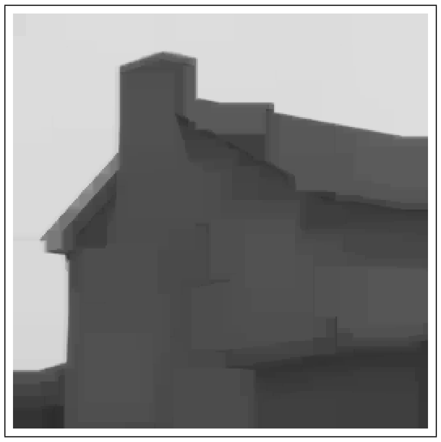
```
void black_top_hat
 (long     nx,   /* image dimension in x direction */
  long     ny,   /* image dimension in y direction */
  long     m,    /* size of structuring element: (2m+1)*(2m+1) */
  float    **u)  /* input: original image; output: processed */

/*
 Black top hat with a square of size (2m + 1) * (2m + 1) as
 structuring element.
*/

{
long    i, j;       /* loop variables */
float   **uc;       /* closing of input image */

alloc_matrix (&uc, nx+2, ny+2);

/* copy u info uc */
for(i=1; i<=nx; i++)
    for(j=1; j<=ny; j++)
        uc[i][j] = u[i][j];

/* perform closing */
closing(nx, ny, m, uc);

/* subtract u from closing */
for(i=1; i<=nx; i++)
    for(j=1; j<=ny; j++)
        u[i][j] = uc[i][j] - u[i][j];

disalloc_matrix (uc, nx+2, ny+2);
return;

} /* black_top_hat */

/*--------------------------------------------------------------------*/
```

```
void selfdual_top_hat
 (long     nx,   /* image dimension in x direction */
  long     ny,   /* image dimension in y direction */
  long     m,    /* size of structuring element: (2m+1)*(2m+1) */
  float    **u)  /* input: original image; output: processed */

/*
 Black top hat with a square of size (2m + 1) * (2m + 1) as
 structuring element.
*/

{
long    i, j;       /* loop variables */
float   **uc;       /* copy of input image */

alloc_matrix (&uc, nx+2, ny+2);

/* copy u into uc */
for(i=1; i<=nx; i++)
    for(j=1; j<=ny; j++)
        uc[i][j] = u[i][j];

closing(nx, ny, m, uc);
opening(nx, ny, m, u);

/* compute the difference between u and uc */
for(i=1; i<=nx; i++)
    for(j=1; j<=ny; j++)
         u[i][j] = uc[i][j] - u[i][j];

disalloc_matrix (uc, nx+2, ny+2);
return;

} /* selfdual_top_hat */

/*--------------------------------------------------------------------*/
```

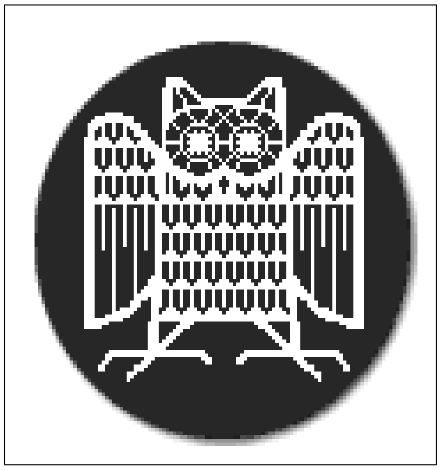
It is also possible to implement the selfdual top hat as the sum of black and the white top hat. The formula used here avoids to add and subtract the initial image f and thus needs less elementary operations.

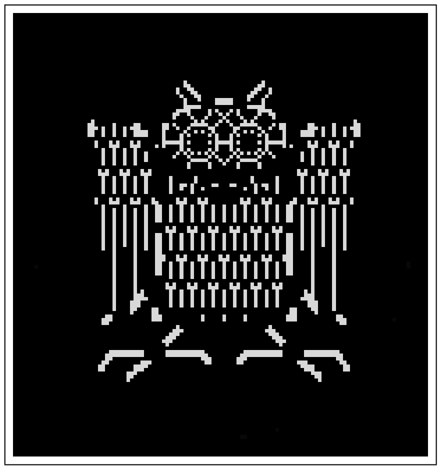
Now we show some examples how to apply this implementation on the problems given in the exercise sheet. Usually there are various ways to solve the problems. The parameters presented here should just give an idea how to proceed. For the removal of windows and doors it is possible to use an opening operation with a suitable structure element size.

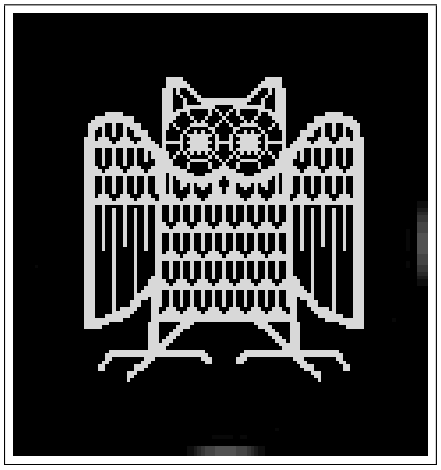

Original image

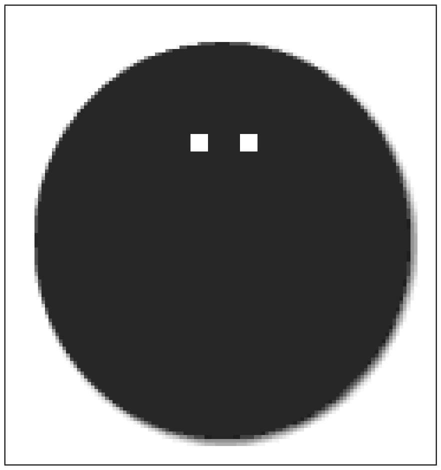

Opening ($m = 15$)

University owls are very shy at night – only a few people have ever seen one. We show different ways how one can imagine such a species:

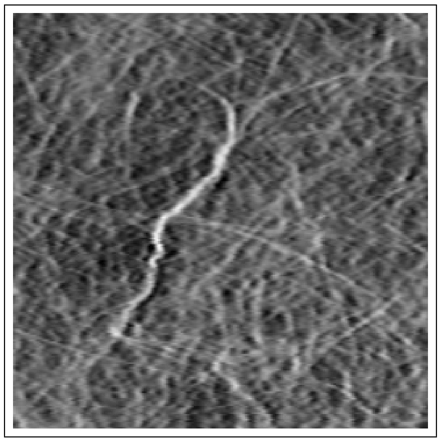

Original image

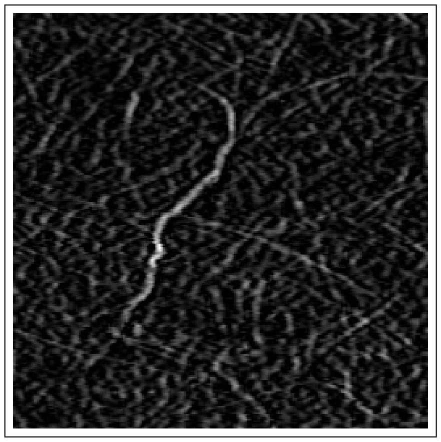

White top hat ($m = 1$)


White top hat ($m = 3$)

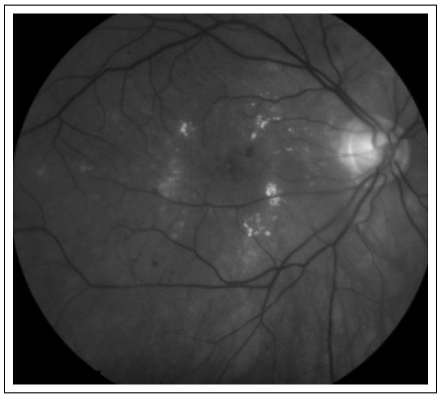

Opening ($m = 2$)

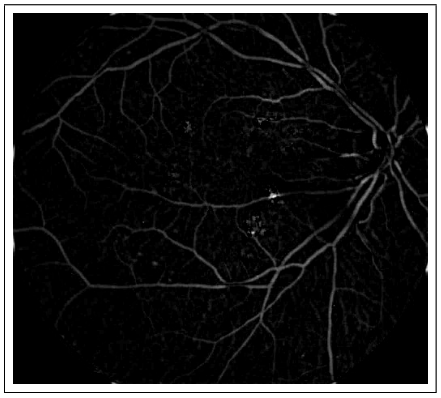
To separate small structures from the background some top hat is the method of choice. Here we have used a white top hat to emphasise the bright structures. For both the fabric image and the angiogram it is useful to normalise the results with xv.


Original image


White top hat ($m = 3$) normalised with xv

For the angiogram we aim to visualise the dark structures, that means we are applying a black top hat.


Original image


Black top hat ($m = 3$) normalised with xv
