Image Processing and Computer Vision 2007/08
Example Solutions for Practical Assignments 6 (P6)

Problem 1 (Corner Detection)

(a) 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.

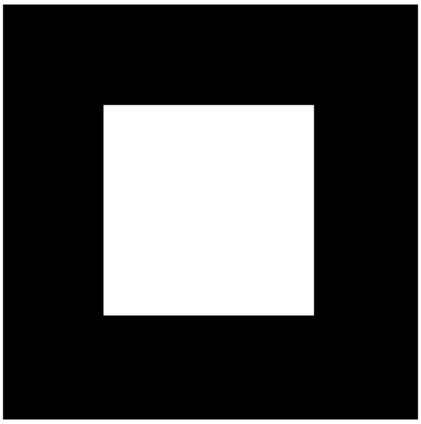
```
/* ------------------------------------------------------------ */
void struct_tensor
 (float    **v,    /* image !! gets smoothed on exit !! */
  long     nx,     /* image dimension in x direction */
  long     ny,     /* image dimension in y direction */
  float    hx,     /* pixel size in x direction */
  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;
}
/* ---------------------------------------------------------------- */
```

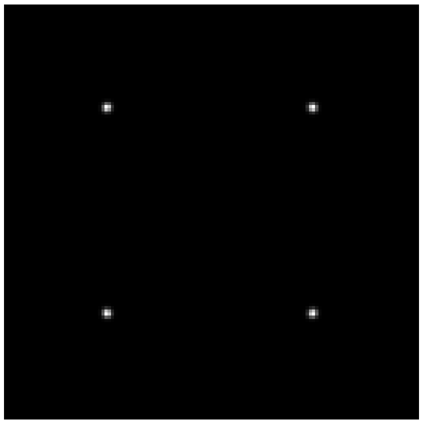
(b) The cornerness can for instance be implemented as the determinant of the structure tensor here. Then the code looks as follows:

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

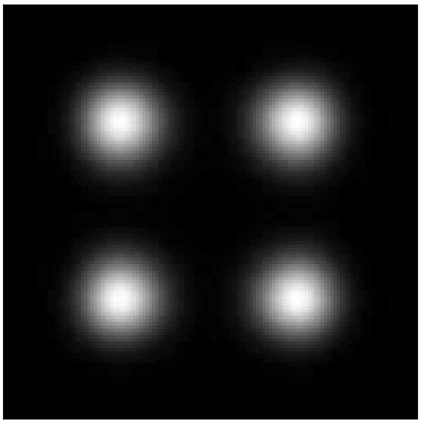
(c) We demonstrate the influence of the parameters σ and ρ at the first example `square.pgm`. Higher values of both parameters lead to fuzzy edge detections.



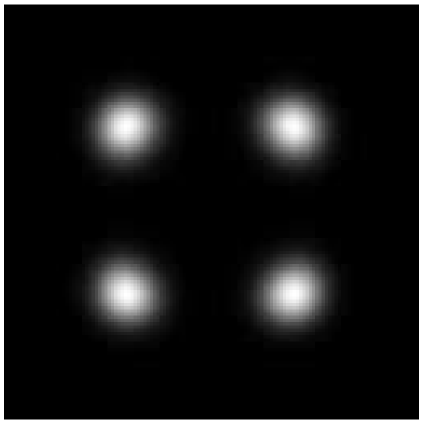
Original image `square.pgm`



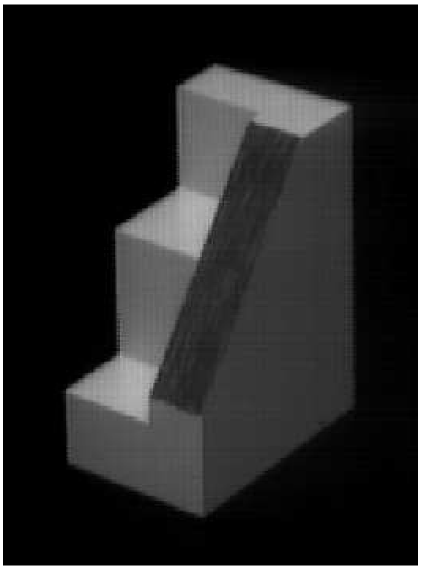
Corner detection, $\sigma = 0.0$. $\rho = 1.0$



$\sigma = 0.0$, $\rho = 10.0$



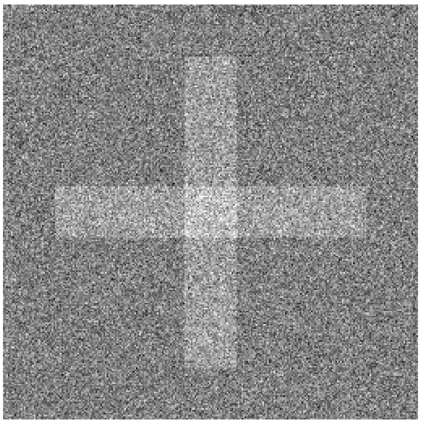
$\sigma = 10.0$, $\rho = 1.0$



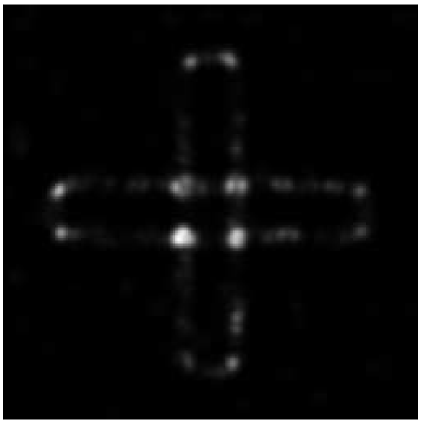
Original image `stairs.pgm`

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, $\sigma = 3.0$, $\rho = 3.0$

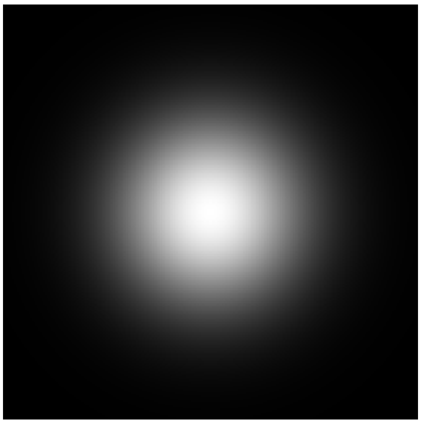
Problem 2 (Double Thresholding)

(a) For the double threshold method we start with determining which pixel values exceed t_1, and t_2, respectively. The values higher than t_2 serve as seed points and are iteratively expanded to the final segmentation. A dilation is used to mark the current region and all its direct neighbours. All pixels of this region that exceed t_1 are joined with the current region. If nothing has happened in the last step, the iterative procedure stops.

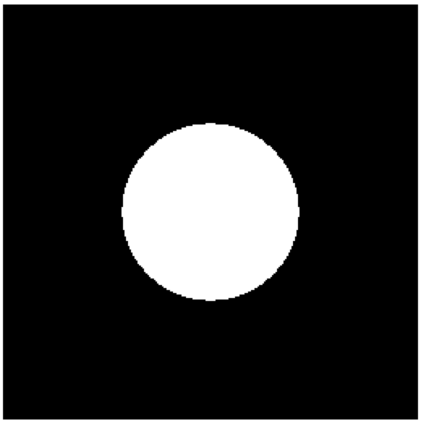
```
/* ------------------------------------------------------------- */
void double_thresholding

     (double  **u,        /* image, changed on output */
      long    nx,         /* size in x direction */
      long    ny,         /* size in y direction */
      double  t1,         /* smaller threshold */
      double  t2)         /* larger threshold */

/*
 double thresholding with the threshold pair (t1,t2).
*/

{
long    i, j;    /* loop variables */
long    stop;    /* stop iterations? */
double  **uold;  /* previous iteration value of growing image u */
double  **v;     /* image thresholded at t2 */

/* allocate storage for v and uold */
alloc_matrix (&v, nx+2, ny+2);
alloc_matrix (&uold, nx+2, ny+2);

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

/* threshold u at t2, and v at t1 */
for (i=1; i<=nx; i++)
  for (j=1; j<=ny; j++)
    {
      if (u[i][j] <= t2) u[i][j] = 0.0; else u[i][j] = 255.0;
```

```
      if (v[i][j] <= t1) v[i][j] = 0.0; else v[i][j] = 255.0;
    }

 /* expand seed objects in u until they reach the object */
 /* boundaries of v                                       */
 do
   {
    /* copy u into uold */
    for(i=1; i<=nx; i++)
      for(j=1; j<=ny; j++)
        uold[i][j] = u[i][j];

    /* add direct neighbours with dilation */
    dilation(uold, nx, ny, u);

    /* remove pixels in neighbourhood which are smaller than t1 */
    for(i=1; i<=nx; i++)
      for(j=1; j<=ny; j++)
        if(v[i][j] < u[i][j]) u[i][j] = v[i][j];

    /* check if something has changed */
    stop = 1;
    for(i=1; i<=nx; i++)
      for(j=1; j<=ny; j++)
        if(u[i][j] > uold[i][j]) stop = 0;
   }
 while (stop == 0);

 /* free storage */
 disalloc_matrix (v, nx+2, ny+2);
 disalloc_matrix (uold, nx+2, ny+2);

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

(b) The first image `gauss.pgm` shows a symmetric Gaussian. The level sets of this image are concentric circles. Thus the double threshold method yields the same result as a simple threshold method with threshold t_1.

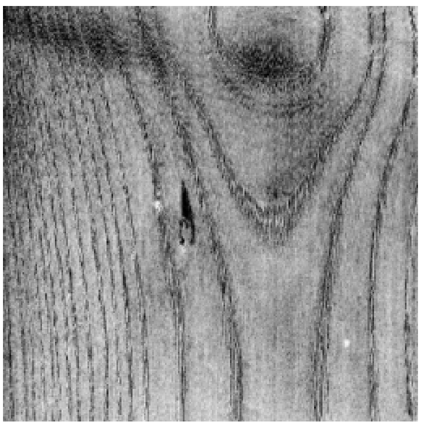


Original image `gauss.pgm`



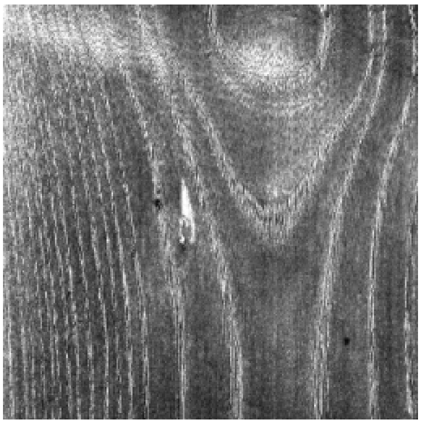
Double thresholding with $t_2 = 200$, $t_1 = 100$

(c) With the double threshold technique as implemented in our program we are only able to detect bright structures. In the third example we want to find a dark defect in a wood surface. To this end in a first step we invert the image such that dark structures turn into bright ones. Grey value images can be inverted either in the code directly or using `xv`'s color editor (which can be opened by pressing `'e'` with focus on the image window. The button `RevVid` inverts the image. We then apply double thresholding to the inverted image.

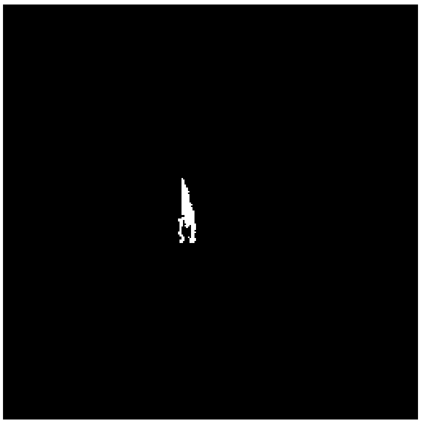


Original image `wood.pgm`

Single threshold $t = 24$



Inverted image



Double threshold of the inverted image with $t_2 = 254$, $t_1 = 150$.