Image Processing and Computer Vision 2005/06
Example Solutions for Practical Assignments 6 (P6)

Problem 1

In the template file `hsTemplate.c` the code for the Jacobi iteration was missing. The corresponding iterative scheme was explicitly given in the lecture as

$$\begin{aligned} u_i^{(k+1)} &= \frac{\sum\limits_{j\in\mathcal{N}(i)} u_j^{(k)} - \frac{1}{\alpha} f_{xi}\left(f_{yi}v_i^{(k)} + f_{zi}\right)}{|\mathcal{N}(i)| + \frac{1}{\alpha}f_{xi}^2} \\ v_i^{(k+1)} &= \frac{\sum\limits_{j\in\mathcal{N}(i)} v_j^{(k)} - \frac{1}{\alpha} f_{yi}\left(f_{xi}u_i^{(k)} + f_{zi}\right)}{|\mathcal{N}(i)| + \frac{1}{\alpha}f_{yi}^2} \quad . \end{aligned}$$

In the text of the assigment the hint was given that one has to make sure that the boundary is treated correctly. The postition of a pixel comes into play with the set $\mathcal{N}(i)$ of neighbouring pixels in the above formula. In the numerators there are sums over all neighbouring pixels, and in the denominator the number of neighbours is needed.

In general one can distinguish three cases: The first one are inner pixels which have a full set of four direct neighbours. The second case are boundary pixels which only have three neighbours, and the third one are corners with only 2 direct neighbours.

To simplify the implementation we remember the fact that the arrays in our programs are always allocated with on outer boundary of size 1. In the last exercises we have used this additional memory to mirror the boundary of images with the `dummies` function. Now we can set this outer boundary to zero. This assures that we can use the same code for the computation of the sums shown above: For boundary or corner pixels the neighbours not belonging to the image are zero and does not contribute to the sum. Now we only have to make sure that the number of neighbours is computed correctly. The following code sample shows how this can be done:

```
void flow
 (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    **fx,    /* x derivative of image */
  float    **fy,    /* y derivative of image */
  float    **fz,    /* z derivative of image */
  float    alpha,   /* smoothness weight */
  float    **u,     /* x component of optic flow */
  float    **v)     /* v component of optic flow */

/*
 Performs one Jacobi iteration for the Euler-Lagrange equations
 arising from the Horn and Schunck method.
*/

{
long    i, j;             /* loop variables */
long    nn;               /* number of neighbours */
float   help;             /* 1.0/alpha */
float   **u1, **v1;       /* u, v at old iteration level */


/* ---- allocate storage ---- */

alloc_matrix (&u1, nx+2, ny+2);
alloc_matrix (&v1, nx+2, ny+2);


/* ---- copy u, v into u1, v1 ---- */

for (i=1; i<=nx; i++)
 for (j=1; j<=ny; j++)
     {
     u1[i][j] = u[i][j];
     v1[i][j] = v[i][j];
     }


/* ---- perform one Jacobi iteration ---- */

/* initialise outer boundary of u1 and v1 to zero */
for(i=0; i<=nx+1; i++) {
    u1[i][0]    = 0.0;
    u1[i][ny+1] = 0.0;
```

```
    v1[i][0]    = 0.0;
    v1[i][ny+1] = 0.0;
}

for(j=0; j<=ny+1; j++) {
    u1[0][j]    = 0.0;
    u1[nx+1][j] = 0.0;
    v1[0][j]    = 0.0;
    v1[nx+1][j] = 0.0;
}

help = 1.0 / alpha;

for (i=1; i<=nx; i++)
 for (j=1; j<=ny; j++)
 {
     /* compute number of neighbours */
     nn = 4;
     if( i == 1 || i == nx ) nn--;
     if( j == 1 || j == ny ) nn--;

     /* formula from the iterative scheme */
     u[i][j] = (u1[i-1][j] + u1[i+1][j] + u1[i][j-1] + u1[i][j+1]
           - help * fx[i][j] * (fy[i][j] * v1[i][j] + fz[i][j]))
         / (nn + help * fx[i][j] * fx[i][j]);

     v[i][j] = (v1[i-1][j] + v1[i+1][j] + v1[i][j-1] + v1[i][j+1]
           - help * fy[i][j] * (fx[i][j] * u1[i][j] + fz[i][j]))
         / (nn + help * fy[i][j] * fy[i][j]);
 }


/* ---- disallocate storage ---- */

disalloc_matrix (u1, nx+2, ny+2);
disalloc_matrix (v1, nx+2, ny+2);
return;

} /* flow */
```

It is clear that it is also possible to distinguish the cases and treat them separately. The most efficient way implements the formula for all cases independently and thus does not need any `if` conditions. We show one example of each pixel group for an efficient way of implementation:

```
help = 1.0 / alpha;

/* upper left corner */
u[1][1] = (u1[1][2] + u1[2][1]
       - help * fx[1][1] * (fy[1][1] * v1[1][1] + fz[1][1]))
     / (2 + help * fx[1][1] * fx[1][1]);

v[1][1] = (v1[1][2] + v1[2][1]
       - help * fy[1][1] * (fx[1][1] * u1[1][1] + fz[1][1]))
     / (2 + help * fy[1][1] * fy[1][1]);

/* upper and lower boundary */
for (i=2; i<nx; i++)
 {
     /* upper boundary */
     u[i][1] = (u1[i][2] + u1[i-1][1] + u1[i+1][1]
           - help * fx[i][1] * (fy[i][1] * v1[i][1] + fz[i][1]))
         / (3 + help * fx[i][1] * fx[i][1]);

     v[i][1] = (v1[i][2] + v1[i-1][1] + v1[i+1][1]
           - help * fy[i][1] * (fx[i][1] * u1[i][1] + fz[i][1]))
         / (3 + help * fy[i][1] * fy[i][1]);

     /* lower boundary */
     u[i][ny] = (u1[i][ny-1] + u1[i-1][ny] + u1[i+1][ny]
           - help * fx[i][ny] * (fy[i][ny] * v1[i][ny] + fz[i][ny]))
         / (3 + help * fx[i][ny] * fx[i][ny]);
     v[i][ny] = (v1[i][ny-1] + v1[i-1][ny] + v1[i+1][ny]
           - help * fy[i][ny] * (fx[i][ny] * u1[i][ny] + fz[i][ny]))
         / (3 + help * fy[i][ny] * fy[i][ny]);
 }

/* inner pixels */
for (i=2; i<nx; i++)
 for (j=2; j<ny; j++)
 {
```

```
    u[i][j] = (u1[i-1][j] + u1[i+1][j] + u1[i][j-1] + u1[i][j+1]
          - help * fx[i][j] * (fy[i][j] * v1[i][j] + fz[i][j]))
        / (4 + help * fx[i][j] * fx[i][j]);

    v[i][j] = (v1[i-1][j] + v1[i+1][j] + v1[i][j-1] + v1[i][j+1]
          - help * fy[i][j] * (fx[i][j] * u1[i][j] + fz[i][j]))
        / (4 + help * fy[i][j] * fy[i][j]);
}
```

The disadvantage is that the risc of implementation errors is much higher then. So it is a matter of taste which way of implementation one prefers.

Now we turn our attention to the experimental results. The images given for the optic flow computation are shown below.

pig1.pgm

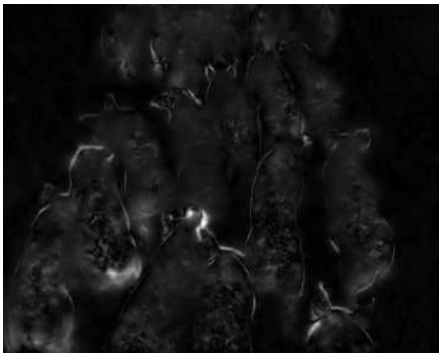
pig2.pgm

We show some optic flow results to demonstrate the dependence of the parameters α and the number of iterations.

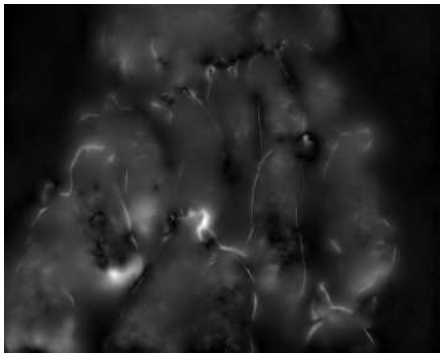
First we note that the results get smoother with higher values for α in accordance with the interpretation of α as smoothness parameter. With higher values of α one can also see the *filling-in effect* mentioned on the assignment sheet. In one iteration of our iterative scheme a pixel is only influenced by its four direct neighbours. Thus it is clear that it takes a number of iterations since information is propagated over a long distance in the flow field. This makes it plausible that a noticeable filling-in effect needs not only a high value of α (of about 1000) but also a high number of iterations.

$\alpha = 1$, 10 iterations

$\alpha = 10$, 100 iterations

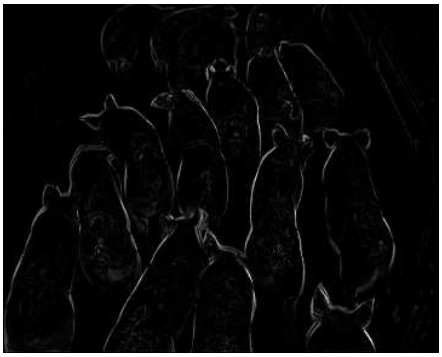


$\alpha = 100$, 1000 iterations

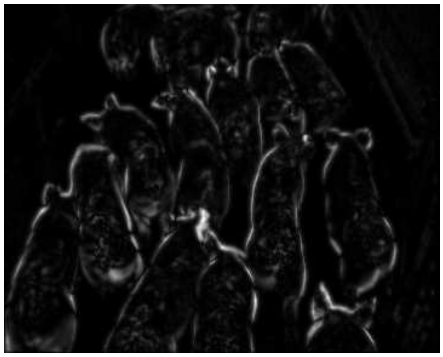


$\alpha = 1000$, 1000 iterations

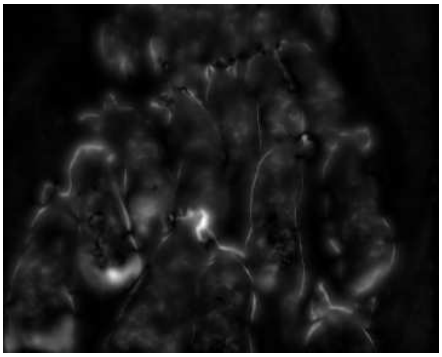
Finally we fix the value of $\alpha = 1000$ and show how the result behaves for different iteration numbers. We see that for such a Jacobi scheme the number of iterations can also heavily influence the result, and thus an appropriate choice of this number is important. With higher values of α and increasing smoothness the need for communication between pixels demands a higher iteration number, too.



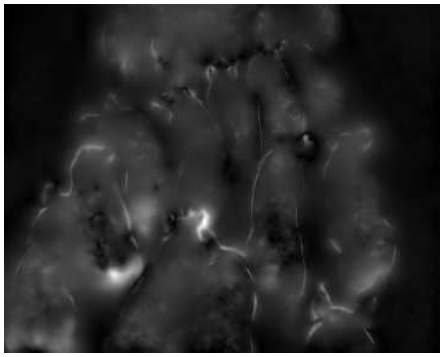
1 iteration



10 iterations



100 iterations



1000 iterations