Example Solutions for Homework Assignment 12 (H12)

Problem 1 (Lucas and Kanade Method for Colour Sequences)

(a) We are given the colour image sequence

$$\boldsymbol{f}(x,y,z) = (f_R(x,y,z), f_G(x,y,z), f_B(x,y,z))^\top$$

in RGB colour space. We assume that corresponding image structures should have the same colour value, i.e. the values in the individual colour channels should be the same. Thus, we obtain for the optic flow:

$$\begin{aligned} f_R(x+u, y+v, z+1) &= f_R(x,y,z)\,, \\ f_G(x+u, y+v, z+1) &= f_G(x,y,z)\,, \\ f_B(x+u, y+v, z+1) &= f_B(x,y,z)\,. \end{aligned}$$

If the values of u and v are small and f varies slowly, we can assume that the Taylor expansion around (x,y,z) gives a good approximation. This yields the linearised optic flow constraints

$$\begin{aligned} f_{R,x}u + f_{R,y}v + f_{R,z} &= 0\,, \\ f_{G,x}u + f_{G,y}v + f_{G,z} &= 0\,, \\ f_{B,x}u + f_{B,y}v + f_{B,z} &= 0\,. \end{aligned}$$

In matrix-vector notation we obtain

$$\begin{pmatrix} f_{R,x} & f_{R,y} \\ f_{G,x} & f_{G,y} \\ f_{B,x} & f_{B,y} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -f_{R,z} \\ -f_{G,z} \\ -f_{B,z} \end{pmatrix}\,.$$

(b) We have an overdetermined system with three equations and two unknowns. In general, overdetemined systems are almost always inconsistent, which means that they do not have a solution because of contradicting equations.
Three cases can be distinguished:

Case 1: All equations are pairwise linearly dependent.
The solution lies on a line. The aperture problem persists and we can only compute the normal flow.

Case 2: The three equations are linearly dependent (but not pairwise linearly dependent).
There is a unique solution.

Case 3: The three equations are linearly independent.
No exact solution exists. There are still ways to obtain an approximation, for example by using the method of ordinary least squares.

(c) In a corresponding Lucas-and-Kanade-like approach, we penalise deviations from the optic flow constraint quadratically in every channel. We obtain

$$E(u,v) = \frac{1}{2} \int_{B_\rho(x_0,y_0)} \sum_{c \in \{R,G,B\}} (f_{c,x} u + f_{c,y} v + f_{c,z})^2 \, dx \, dy \, .$$

(d) Computing the partial derivatives with respect to u and v gives

$$0 \stackrel{!}{=} \frac{\partial E}{\partial u} = \int_{B_\rho} \sum_{c \in \{R,G,B\}} f_{c,x} (f_{c,x} u + f_{c,y} v + f_{c,z}) \, dx \, dy$$

$$0 \stackrel{!}{=} \frac{\partial E}{\partial v} = \int_{B_\rho} \sum_{c \in \{R,G,B\}} f_{c,y} (f_{c,x} u + f_{c,y} v + f_{c,z}) \, dx \, dy$$

Problem 2 (Variational Optic Flow)

(a) The invariance of the gradient under additive global illumination changes follows directly from the rules of differentiation:

Let $g(x, y, t) = f(x, y, t)+a$ be the sequence we get by adding a constant value a to our original sequence f. Taking the gradient leads to

$$\boldsymbol{\nabla} g = \begin{pmatrix} \partial_x f + \partial_x a \\ \partial_y f + \partial_y a \\ \partial_t f + \partial_t a \end{pmatrix} = \begin{pmatrix} \partial_x f \\ \partial_y f \\ \partial_t f \end{pmatrix} = \boldsymbol{\nabla} f \ .$$

Thus, the gradient of an image is invariant under additive illumination changes.

(b) We can now formulate a new optic flow constraint based on the assumption that corresponding structures within the sequence will have the same spatial gradient. That is, the optic flow field $(u, v)^\top$ between frame t and $t + 1$ satisfies:

$$\boldsymbol{\nabla}_2 f(x, y, t) = \boldsymbol{\nabla}_2 f(x + u, y + v, t + 1) \ .$$

After linearisation by Taylor expansion, we get:

$$\partial_x \boldsymbol{\nabla}_2 f(x, y, t) \cdot u + \partial_y \boldsymbol{\nabla}_2 f(x, y, z) \cdot v + \partial_t \boldsymbol{\nabla}_2 f(x, y, t) = 0 \ .$$

or equivalently:

$$\begin{aligned} f_{xx} u + f_{xy} v + f_{xt} &= 0 \\ f_{xy} u + f_{yy} v + f_{yt} &= 0 \ . \end{aligned}$$

(c) If we take a look at our optic flow constraint, we see that we have two equations for two unknowns, in contrast to just one equation for brightness constancy. Therefore we can solve these equations for an unique, local solution, given that both are linearly independent. Nevertheless as soon as the system of equations is under-determined (i.e. at least one of the eigenvalues of the system matrix is equal to zero) we get again the aperture problem. So it is possible that there are locations where the flow can be computed in normal flow direction only and there are even locations, namely homogeneous regions, where the flow cannot be computed at all.

(d) To embed this constraint in a variational approach, we can just substitute the data term in the Horn-and-Schunck functional. Utilising a quadratic penaliser for our terms, this yields:

$$E(u,v) = \int_\Omega (f_{xx}u+f_{xy}v+f_{xt})^2+(f_{xy}u+f_{yy}v+f_{yt})^2+\alpha(|\boldsymbol{\nabla} u|^2+|\boldsymbol{\nabla} v|^2)\,dx\,dy\ .$$

(e) The minimiser of the functional

$$E(u,v) = \int_\Omega F(x,y,u,v,u_x,u_y,v_x,v_y)\,dx\,dy$$

has to satisfy the Euler-Lagrange-Equations:

$$\begin{aligned} \partial_x F_{u_x} + \partial_y F_{u_y} - F_u &= 0\ , \\ \partial_x F_{v_x} + \partial_y F_{v_y} - F_v &= 0\ , \end{aligned}$$

with Neumann boundary conditions.

In our case, we have

$$F = (f_{xx}u + f_{xy}v + f_{xt})^2 + (f_{xy}u + f_{yy}v + f_{yt})^2 + \alpha(|\boldsymbol{\nabla} u|^2 + |\boldsymbol{\nabla} v|^2),$$

with partial derivatives:

$$\begin{aligned} F_u &= 2f_{xx}(f_{xx}u + f_{xy}v + f_{xt}) + 2f_{xy}(f_{xy}u + f_{yy}v + f_{yt}) \\ F_v &= 2f_{xy}(f_{xx}u + f_{xy}v + f_{xt}) + 2f_{yy}(f_{xy}u + f_{yy}v + f_{yt}) \\ F_{u_x} &= 2\alpha u_x \\ F_{u_y} &= 2\alpha u_y \\ F_{v_x} &= 2\alpha v_x \\ F_{v_y} &= 2\alpha v_y\ . \end{aligned}$$

After plugging in, we get the Euler-Lagrange-Equations:

$$\begin{aligned} \Delta u - \frac{1}{\alpha}(f_{xx}(f_{xx}u + f_{xy}v + f_{xt}) + f_{xy}(f_{xy}u + f_{yy}v + f_{yt})) &= 0 \\ \Delta v - \frac{1}{\alpha}(f_{xy}(f_{xx}u + f_{xy}v + f_{xt}) + f_{yy}(f_{xy}u + f_{yy}v + f_{yt})) &= 0\ . \end{aligned}$$

Problem 3 (Optic Flow Estimation, Lucas and Kanade)

The approach by Lucas and Kanade was discussed in detail in Lecture 23. We follow this lecture to complete the missing pieces of code here.

The first method `create_eq_systems` calculates the entries of the matrix and the right-hand side. This method is essentially the calculation of the structure tensor (without presmoothing the image). The new parts are the approximation of the time derivative and the calculation of the vector on the right-hand side.

The second method `lucas_kanade` distinguished the three cases with criteria given in the lecture. If the trace is smaller than ε, we have no information and set the flow to zero. If the determinant is small, we only calculate the normal flow, and otherwise we use Cramer's rule to solve the linear system of equations.

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

void create_eq_systems
(float    **f1,      /* frame 1, input */
 float    **f2,      /* frame 2, input */
 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    ht,        /* distance between two frames */
 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 */
 float    **dxz,     /* element of the right-hand side, output */
 float    **dyz)     /* element of the right-hand side, output */


/*
 Calculates the entries of the structure tensor and the
 right-hand side for the linear systems of equations.
*/

{
```

```
long    i, j;                  /* loop variables */
float   df_dx, df_dy, df_dz;   /* derivatives of f */
float   w1, w2, w3, w4, w5;    /* time savers */


/* ---- calculate gradient and its tensor product ---- */

dummies(f1, nx, ny);
dummies(f2, nx, ny);

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

         df_dx = w1 * (f1[i+1][j+1] - f1[i-1][j+1]
                       + f1[i+1][j-1] - f1[i-1][j-1])
               + w2 * (f1[i+1][j] - f1[i-1][j]);

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

         df_dy = w3 * (f1[i+1][j+1] - f1[i+1][j-1]
                       + f1[i-1][j+1] - f1[i-1][j-1])
               + w4 * (f1[i][j+1] - f1[i][j-1]);
         /* time derivative with forward difference, ht = 1 */
         w5 = 1.0 / ht;
         df_dz = w5 * (f2[i][j] - f1[i][j]);

         /* calculate matrix entries and right-hand side */
         dxx[i][j] = df_dx * df_dx;
         dxy[i][j] = df_dx * df_dy;
         dyy[i][j] = df_dy * df_dy;
         dxz[i][j] = df_dx * df_dz;
         dyz[i][j] = df_dy * df_dz;
     }
 }
/* ---- smoothing at integration scale, Dirichlet b.c. ---- */
```

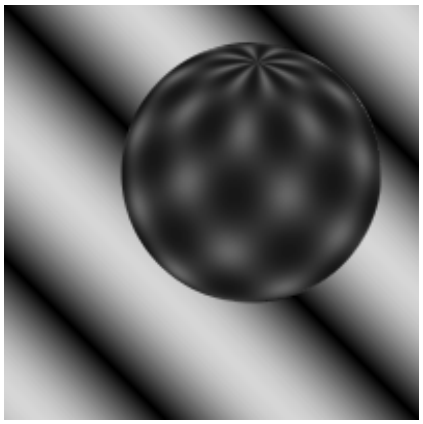
```
if (rho > 0.0)
   {
   gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dxx);
   gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dxy);
   gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dyy);
   gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dxz);
   gauss_conv (rho, nx, ny, hx, hy, 5.0, 0, dyz);
   }

return;
} /* create_eq_systems */

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

void lucas_kanade
(long     nx,          /* image dimension in x direction */
 long     ny,          /* image dimension in y direction */
 float    **dxx,       /* component of system matrix, input */
 float    **dxy,       /* component of system matrix, input */
 float    **dyy,       /* component of system matrix, input */
 float    **dxz,       /* component of right-hand side, input */
 float    **dyz,       /* component of right-hand side, input */
 float    eps,         /* threshold */
 float    **c,         /* confidence of the flow, output */
 float    **u,         /* x component of optic flow, output */
 float    **v)         /* v component of optic flow, output */

/*
   Performs optic flow estimation with Lucas-Kanade method.
*/

{
long i, j;       /* loop variables */
float trace;     /* trace of the matrix at pixel [i][j] */
float det;       /* determinant of the matrix at pixel [i][j] */

for(i=1; i<=nx; i++) {
    for(j=1; j<=ny; j++) {

        trace = dxx[i][j] + dyy[i][j];
        det   = dxx[i][j]*dyy[i][j] - dxy[i][j]*dxy[i][j];
```

```
        if( trace <= eps )
        {
            /* nothing can be said */
            u[i][j] = 0.0;
            v[i][j] = 0.0;

            c[i][j] = 0.0;
        } else if ( det <= eps )
        {
            /* we can only compute the normal flow */
            u[i][j] = - dxz[i][j] / trace;
            v[i][j] = - dyz[i][j] / trace;

            c[i][j] = 128.0;
        } else {
            /* we can solve the system */
            u[i][j] = (- dyy[i][j]*dxz[i][j] + dxy[i][j]*dyz[i][j])
                  / det;
            v[i][j] = (  dxy[i][j]*dxz[i][j] - dxx[i][j]*dyz[i][j])
                  / det;

            c[i][j] = 255.0;
        }
    }
}

return;
} /* lucas_kanade */

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

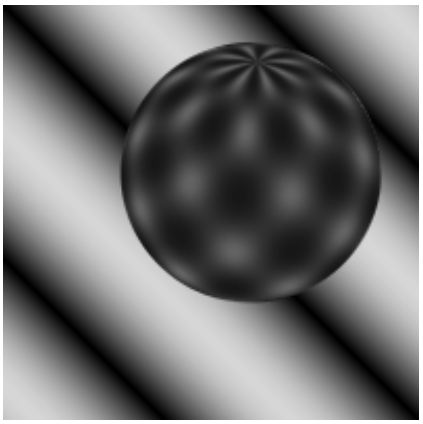
In Fig. 1 and 2 we show some results for the `pig` and `sphere` image pairs:

`pig1.pgm`

`pig2.pgm`



`sphere1.pgm`



`sphere2.pgm`

We see that the parameter ρ has several effects: For small ρ, we usually have large regions where we have the aperture problem or even have no flow information at all. Taking larger values of ρ allows to solve the system of equations almost in all pixels. Nevertheless, we can see the price for this regularisation in the flow magnitude: Edges in the flow field are blurred with larger ρ.

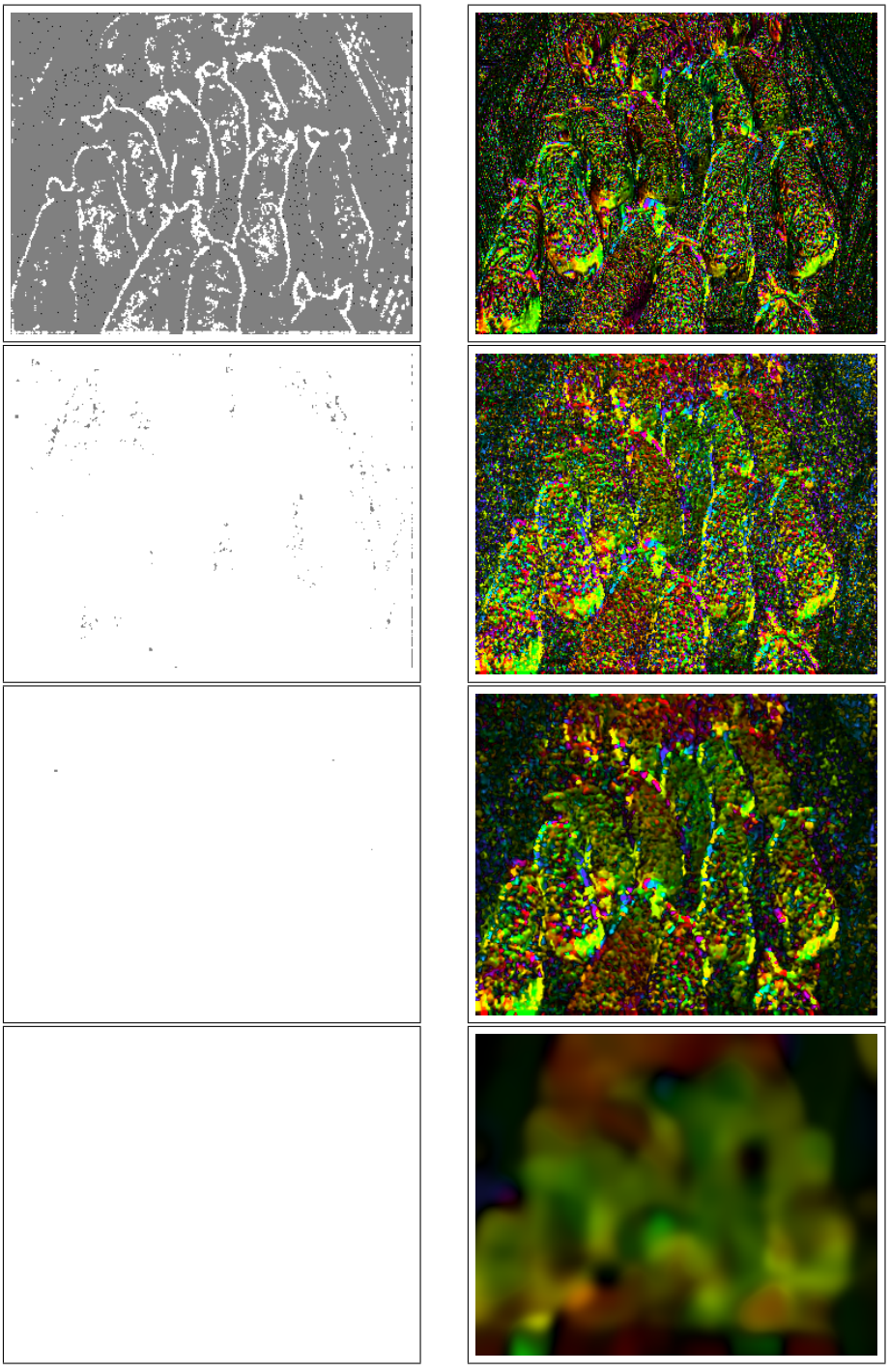


Figure 1: Lucas-Kanade approach for the pig image pair. **Left:** Flow classification. **Right:** colour coded flow. **From top to bottom:** $\rho = 0.2, 0.5, 1.0, 10.0$.

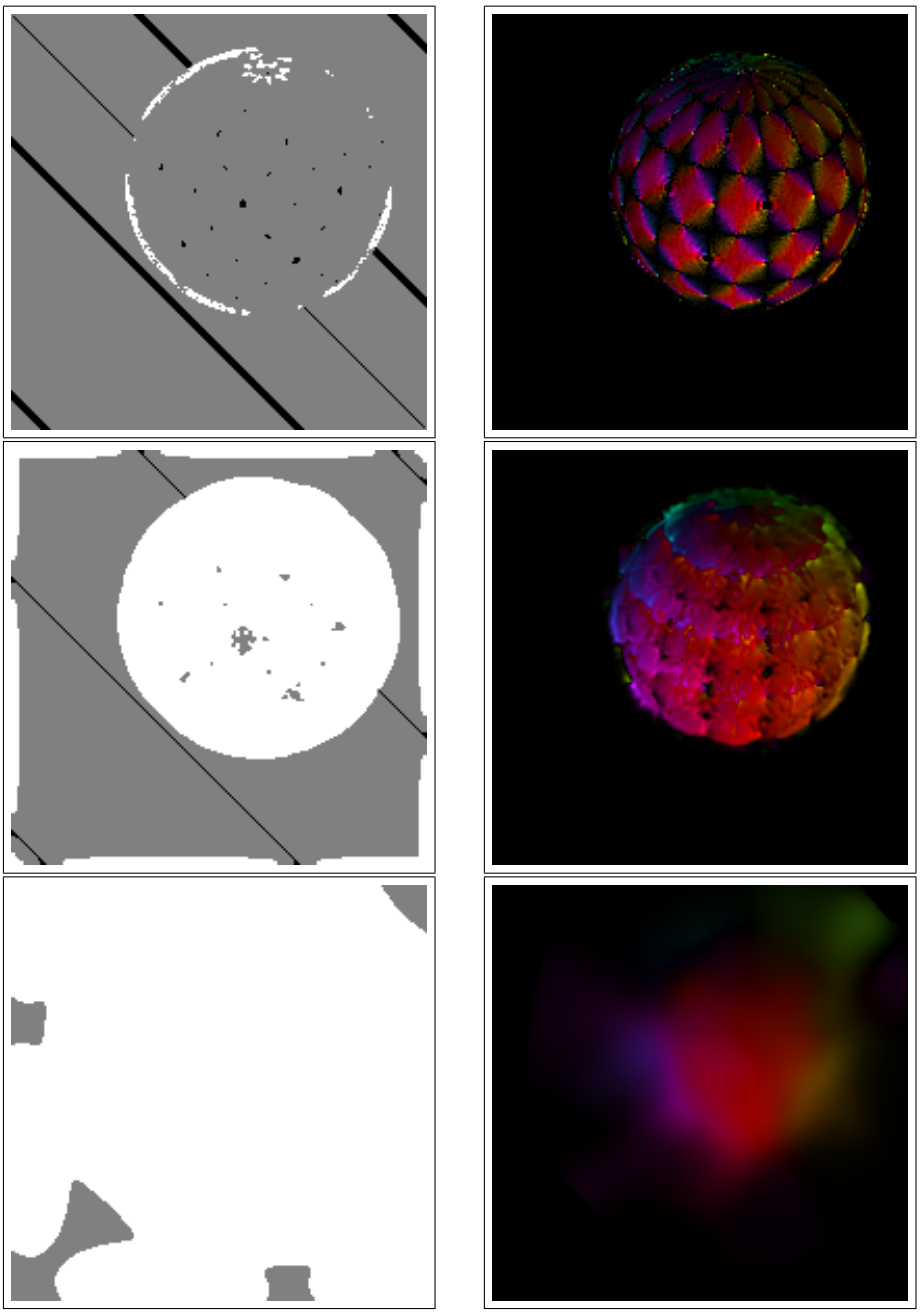


Figure 2: Lucas-Kanade approach for the sphere image pair. **Left:** Flow classification. **Right:** colour coded flow. **From top to bottom:** $\rho = 0.2, 1.0, 10.0$**.**

Problem 4 (Variational Optic Flow Estimation)

In the template file `hornschunck.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:

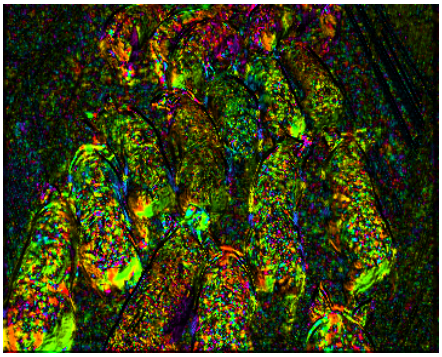
```
help = 1.0 / alpha;

/* 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]);
}
```

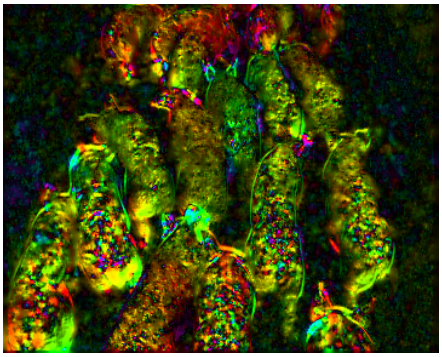
The disadvantage is that the risk of implementation errors is much higher then. So it is a matter of taste which way of implementation one prefers.

Now we turn our attention to the experimental results for the `pig` image pair. We show some optic flow results to demonstrate the dependence of the parameters α and the number of iterations.

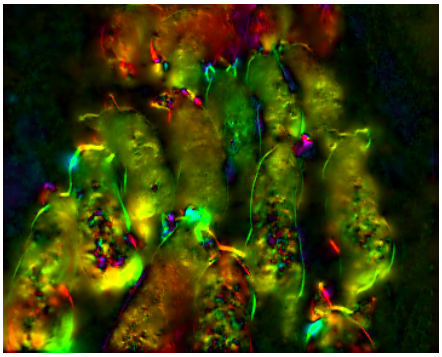
First we note that the results get smoother with higher values for α in accordance with the interpretation of α as smoothness parameter. With higher values of α one can also see the *filling-in effect* mentioned on the assignment sheet. In one iteration of our iterative scheme a pixel is only influenced 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.



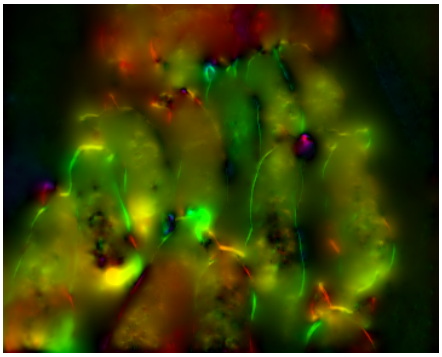
$\alpha = 1$, 10 iterations



$\alpha = 10$, 100 iterations

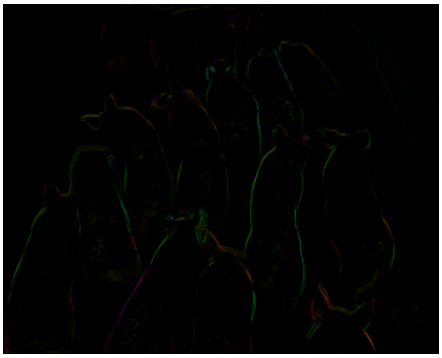


$\alpha = 100$, 1000 iterations

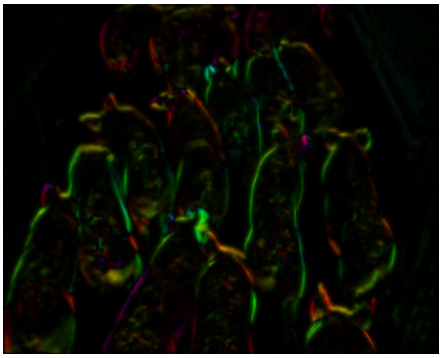


$\alpha = 1000$, 1000 iterations

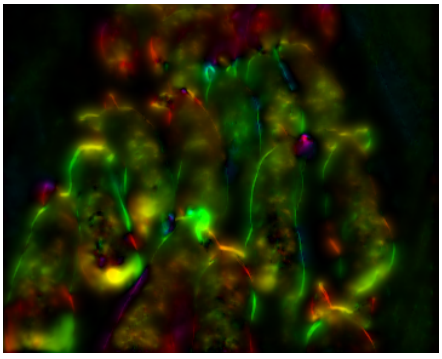
Finally we fix the value of $\alpha = 1000$ and show how the result behaves for different iteration numbers. We see that for such a Jacobi scheme the number of iterations can also heavily influence the result, and thus an appropriate choice of this number is important. With higher values of α and increasing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



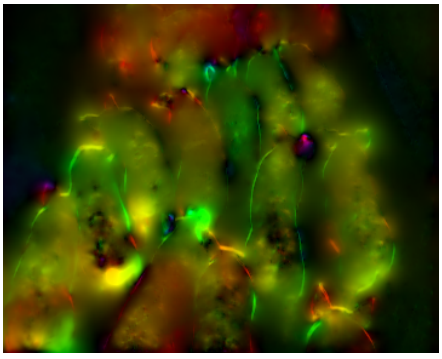
1 iteration



10 iterations



100 iterations



1000 iterations