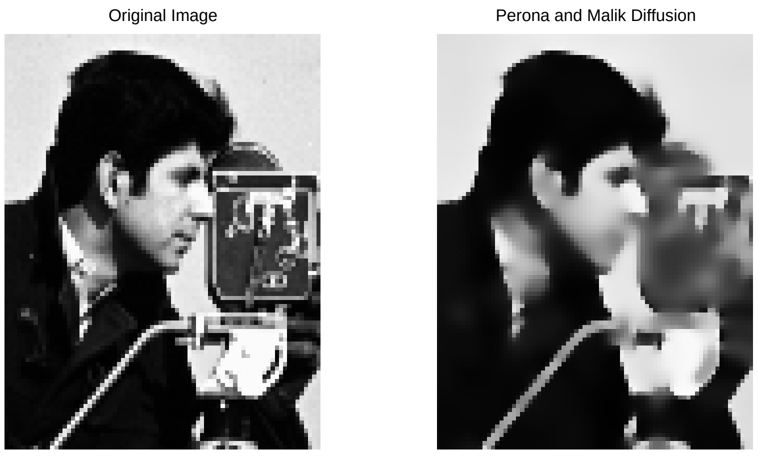
Intelligent Sensory Information Systems
University of Amsterdam
The Netherlands

isis

Algorithms for Non-Linear Diffusion
Matlab in a Literate Programming Style

Rein van den Boomgaard
Intelligent Sensory Information Systems
University of Amsterdam
The Netherlands



This report defines the Matlab code used for the course on non-linear diffusion in image processing. In this report you will find algorithms for *Gaussian convolutions*, *scalar nonlinear diffusion* (e.g. the Perona and Malik type of diffusion) and *tensor diffusion* (e.g. edge enhancing diffusion and coherence enhancing diffusion). A literate programming style is used for the specification of the Matlab code. This means that the actual code is derived from the (electronic version of the) text in this document.

Draft: please send your comments and suggestions to the corresponding author

Contents

Intelligent Sensory Information Systems
Department of Computer Science
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel: +31 20 525 7463
fax: +31 20 525 7490
`http://www.science.uva.nl/research/isis`

Corresponding author:
Rein van den Boomgaard
tel: +31(20)525 7560
`rein@science.uva.nl`
`http://www.science.uva.nl/~rein`

1 Introduction

This report defines the Matlab code used for the course on non-linear diffusion in image processing. In this report you will find algorithms for:

Gaussian convolutions. These image operators are the basic operators in any linear scale-space based approach to computer vision.

Scalar diffusion. A generic algorithm for scalar non linear diffusion is presented. As an example the classical Perona and Malik type of diffusion is implemented.

Tensor diffusion. A simple generic algorithm for tensor diffusion is presented. Examples using this scheme are *edge enhancing diffusion* and *structure enhancing diffusion.*

The algorithms presented in this report are largely based on the papers by Weickert [**?**]. The efficient AOS schemes that he presents in his papers are *not* implemented in this report; only the simple forward Euler schemes are given.

The code presented in this report is not meant as production quality code but merely to test the principles of non linear diffusion on real images. Matlab is used as programming language.

2 Gaussian derivatives

This section gives the code for the Gaussian (derivative) convolutions. The `gD` function provides the basic functionality for any scale-space based vision application. We make use of the build-in function `conv2` in Matlab to do the actual convolution. Only derivatives up to order 2 can be used in the `gD` function. The minimal scale to be used is 1 (although there is no check on the given scale).

The main structure of the program `gD` to calculate the Gaussian derivative of an image is like:

1a ⟨*gD.m* 1a⟩≡

```
function g = gD( f, scale, ox, oy )
% Gaussian (Derivative) Convolution
⟨Calculate sample points 1b⟩
⟨Sample Gaussian function and normalize 2a⟩
⟨Calculate the derivatives in x and y-direction 2b⟩
⟨Do the convolutions 3a⟩
```

This definition is continued in chunk 2c.
Root chunk (not used in this document).

We sample the 1D functions in the integer valued sample points $-K, -K+1, \ldots, -1, 0, 1, \ldots, K-1, K$, where we set $K = 3 \times \text{scale}$.

1b ⟨*Calculate sample points* 1b⟩≡

```
K = ceil( 3 * scale );
x = -K:K;
```

This code is used in chunk 1a.

Then we calculate the zero-order 1D kernel by sampling the continuous Gaussian function. This implies that we can only use scales larger then 1 (well. . . scales down to 0.7 are used in practice as well). For more accurate Gaussian convolutions at smaller scales we have to resort to a discrete convolution kernel based on an interpolation technique (see [**?**]).

2a ⟨*Sample Gaussian function and normalize* 2a⟩≡
```
Gs = exp( - x.^2 / (2*scale^2) );
Gs = Gs / sum(Gs);
```
This code is used in chunk 1a.

To calculate the required derivative of the Gaussian kernel we make use of the fact that:

$$\frac{dG^s}{dx} = -\frac{x}{s^2}G^s(x)$$

and

$$\frac{d^2G^s}{dx^2} = \frac{x^2 - s^2}{s^4}G^s(x)$$

Unfortunately the Hermite polynomials are not available in Matlab as a standard function (else it would be extremely simple to support *any* order of differentiation.

In Matlab code we have:

2b ⟨*Calculate the derivatives in x and y-direction* 2b⟩≡
```
Gsx = gDerivative( ox, x, Gs, scale );
Gsy = gDerivative( oy, x, Gs, scale );
```
This code is used in chunk 1a.

where the (local) function is defined as:

2c ⟨*gD.m* 1a⟩+≡
```
function r = gDerivative( order, x, Gs, scale )
switch order
case 0
    r = Gs;
case 1
    r = -x/(scale^2) .* Gs;
case 2
    r = (x.^2-scale^2)/(scale^4) .* Gs;
otherwise
    error('only derivatives up to second order are supported');
end
```

Finally the actual convolution is done using the build-in Matlab function `conv2`. This function can take two 1D kernel arguments and then does two consecutive convolutions (along the columns and along the rows).

2d ⟨*Do the convolutions (not used)* 2d⟩≡
```
g = conv2( Gsx, Gsy, f, 'same' );
```
Root chunk (not used in this document).

Although the above code chunk leads to running programs, the standard Matlab choice of padding the image with zeros leads to unwanted and unnecessary artifact at the border of the image. A better choice is to repeat the border. Instead of calling the function `conv2` we call a new function `convSepBrd`:

3a ⟨*Do the convolutions* 3a⟩≡

```
    g = convSepBrd( f, Gsx, Gsy );
```

This code is used in chunk 1a.

This function `convSepBrd` first makes a larger images such that a convolution using the buildin function `conv2` with the `'valid'` parameter used, returns an image of the original size again.

To build the correct border around the given image, consider the following 1D example:

f = [9 8 7 6 5];

If we want to convolve this image with repetition of the border using the kernel `1/25 * [1 1 1 1 1]` (i.e. a local average in a 1×5 neighborhood we first construct the larger 'image' **fw**ith**b**order:

fwb = [9 9 9 8 7 6 5 5 5];

This image is easily constructed from `f` using the indexing facilities of Matlab:

fwb = f([1 1 1 2 3 4 5 5 5]);

i.e. we put two starting indices (value 1) in front of the original index list `1 2 3 4 5` and append the last index twice. We need two extra 'pixels' in front and at the end of the list because of the size of the kernel that is used.

For a 2D image we can use this program construction for both axes. Here we assume that the kernel is off odd sizes and that the center of the kernel is in the center. This is indeed the case for the Gaussian (derivative) kernels.

3b ⟨*convSepBrd.m* 3b⟩≡

```
    function g = convSepBrd( f, w1, w2 )
    % convolve along colums + along rows with repetition of the border
    N = size(f,1);
    M = size(f,2);
    K = (size(w1(:),1)-1)/2;
    L = (size(w2(:),1)-1)/2;
    iind = min(max((1:(N+2*K))-K,1),N);
    jind = min(max((1:(M+2*L))-L,1),M);
    fwb = f(iind,jind);
    g=conv2(w1,w2,fwb,'valid');
```

Root chunk (not used in this document).

It should be noted that the Image Processing Toolbox version 3 introduces the `imfilter` function that has the border replication option build in. Unfortunately it doesn't have the option of separable kernels.

Based on the `gD` function we provide the functions to calculate the 1-jet and 2-jet of a given image at given scale. The image in the jet are returned separately (using the Matlab way of returning multiple results from a function[1].

4a ⟨*g1Jet.m* 4a⟩≡

```
function [fs, fsx, fsy] = g1Jet( f, scale )
% First order Gaussian jet
fs  = gD( f, scale, 0, 0 );
fsx = gD( f, scale, 1, 0 );
fsy = gD( f, scale, 0, 1 );
```

Root chunk (not used in this document).

4b ⟨*g2Jet.m* 4b⟩≡

```
function [fs, fsx, fsy, fsxx, fsxy, fsyy] = g2Jet( f, scale )
% Second order Gaussian jet
fs   = gD( f, scale, 0, 0 );
fsx  = gD( f, scale, 1, 0 );
fsy  = gD( f, scale, 0, 1 );
fsxx = gD( f, scale, 2, 0 );
fsxy = gD( f, scale, 1, 1 );
fsyy = gD( f, scale, 0, 2 );
```

Root chunk (not used in this document).

To test the `gD` and jet functions consider the following code (the result is shown in Fig. 1).

4c ⟨*gDtest.m* 4c⟩≡

```
a = imread( 'cameraman.tif' );
a = im2double(a);
[L, Lx, Ly, Lxx, Lxy, Lyy] = g2Jet( a, 3 );
subplot( 3,3,1 ); imshow(L, []);   title('L');
subplot( 3,3,4 ); imshow(Lx, []);  title('Lx');
subplot( 3,3,5 ); imshow(Ly, []);  title('Ly');
subplot( 3,3,7 ); imshow(Lxx, []); title('Lxx');
subplot( 3,3,8 ); imshow(Lxy, []); title('Lxy');
subplot( 3,3,9 ); imshow(Lyy, []); title('Lyy');
```

Root chunk (not used in this document).

3 Nonlinear Scalar Diffusion

In this section we consider the following non-linear diffusion scheme:

$$\partial_t L = \nabla \cdot (c \nabla L)$$

[1] It would be nice of course to use a more appropriate data structure to keep all image in the jet together (e.g. a Matlab structure)

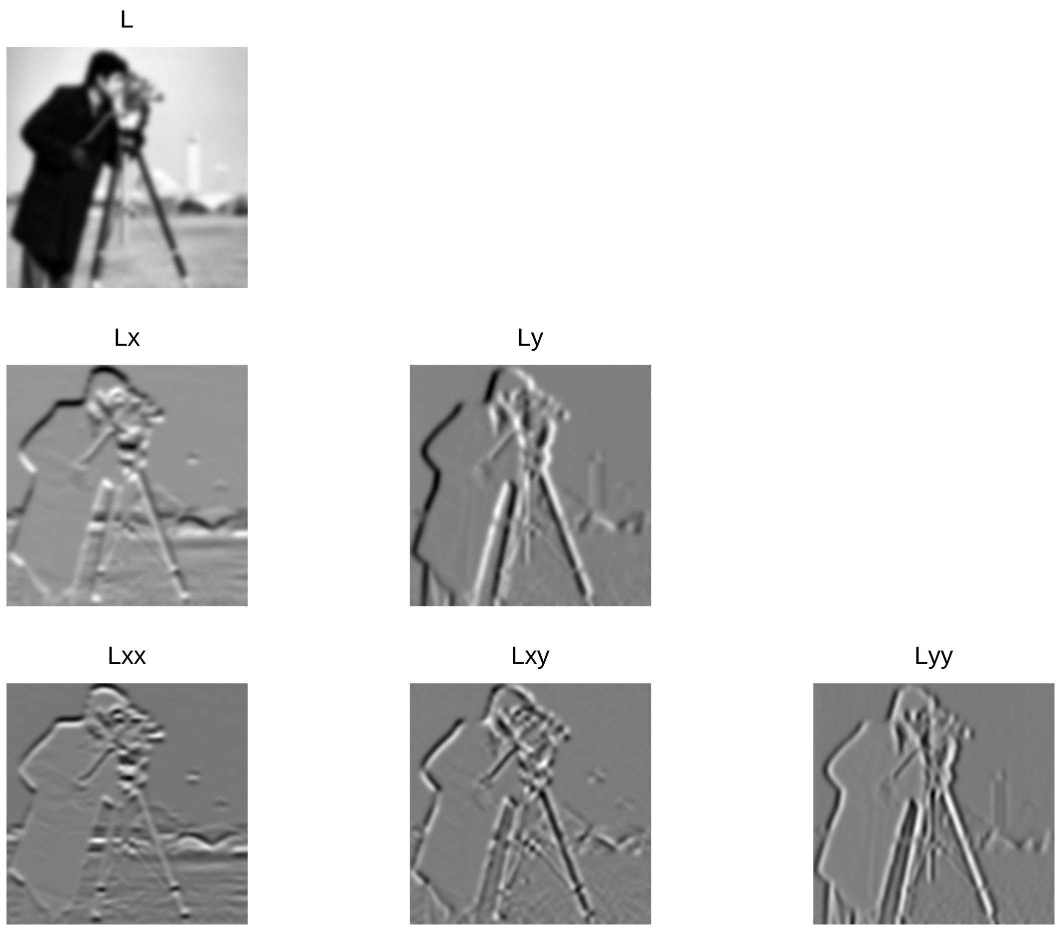


Figure 1: Test of `gD` function. The 2-jet of the `cameraman.tif` image is shown.

where c is a scalar function dependent on the gradient norm $\|\nabla L\|$. Here we only consider *forward Euler explicit numerical schemes.*

In our numerical solution schemes we use the notation $L^t_{i,j}$ where t is the time and i, j are the spatial indices[2]. Furthermore we write $c^t_{i,j}$ to denote the function c as a function of space and time (note that it *is* a function of time as well because c is dependent on the gradient norm $\|\nabla L\|$).

Writing the above PDE in its spatial components leads to:

$$\partial_t L = \partial_x(c\partial_x L) + \partial_y(c\partial_y L).$$

From this expression we see that in case we discretize this PDE using a symmetric scheme for the first order derivatives (in a 3×3 stencil) we end up with a numerical scheme in a 5×5 stencil. In this report we follow Weickert [] in the construction of a numerical scheme within a 3×3 stencil.

[2]Throughout this report we assume that the sampling grid has unit sampling distances along all spatial axes. Therefore the spatial indices directly correspond with spatial coordinates in the visual plane.

First we consider the term $\partial_x(c\partial_x L)$. The 'trick' is to use asymmetric schemes for ∂_x. Let F_i (we concentrate here on the x-dependence) be a function the we define the *left derivative*:

$$\partial_x^- F = F_i - F_{i-1}$$

and right derivative

$$\partial_x^+ F = F_{i+1} - F_i$$

Now we can restrict the stencil for the term $\partial_x(c\partial_x L)$ to a 3×3 stencil by using ∂_x^+ for the innermost derivative and ∂_x^- for the outer derivative or vice versa of course. The average of both possible choices makes the total scheme symmetrical.

$$\partial_x(c\partial_x L) \approx \frac{1}{2}\left(\partial_x^+(c\partial_x^- L) + \partial_x^-(c\partial_x^+ L)\right)$$

We have:

$$\begin{aligned}\partial_x^+(c\partial_x^- L) &= \partial_x^+(c_i(L_i - L_{i-1})) \\ &= c_{i+1}(L_{i+1} - L_i) - c_i(L_i - L_{i-1})\end{aligned}$$

and

$$\begin{aligned}\partial_x^-(c\partial_x^+ L) &= \partial_x^-(c_i(L_{i+1} - L_i)) \\ &= c_i(L_{i+1} - L_i) - c_{i-1}(L_i - L_{i-1})\end{aligned}$$

combining to

$$\partial_x(c\partial_x L) \approx \frac{1}{2}\left((c_i + c_{i+1})(L_{i+1} - L_i) - (c_{i-1} + c_i)(L_i - L_{i-1})\right)$$

It should be carefully noted that we have omitted the j-subscript in all above expressions for the discrete derivative. For all spatial terms the j-index is the same:

$$\partial_x(c\partial_x L) \approx \frac{1}{2}\left((c_{i,j} + c_{i+1,j})(L_{i+1,j} - L_{i,j}) - (c_{i-1,j} + c_{i,j})(L_{i,j} - L_{i-1,j})\right)$$

For the term $\partial_y(c\partial_y L$ the same analysis leads to

$$\partial_y(c\partial_y L) \approx \frac{1}{2}\left((c_{i,j} + c_{i,j+1})(L_{i,j+1} - L_{i,j}) - (c_{i,j-1} + c_{i,j})(L_{i,j} - L_{i,j-1})\right)$$

The discretization of the PDE then becomes

$$\begin{aligned}L_{i,j}^{t+dt} &= L_{i,j}^t + \frac{dt}{2}((c_{i,j}^t + c_{i+1,j}^t)(L_{i+1,j}^t - L_{i,j}^t) - (c_{i-1,j}^t + c_{i,j}^t)(L_{i,j}^t - L_{i-1,j}^t) + \\ &\quad (c_{i,j}^t + c_{i,j+1}^t)(L_{i,j+1}^t - L_{i,j}^t) - (c_{i,j-1}^t + c_{i,j}^t)(L_{i,j}^t - L_{i,j-1}^t))\end{aligned}$$

where the *stepsize* dt should be chosen less then 0.25 in order to result in a stable solution scheme (see Weickert []).

The function `snldStep` (**s**calar **n**on-**l**inear **d**iffusion) calculates the 'step value' in the right hand side of the above expression.

6 ⟨*snldStep.m* 6⟩≡
```
function r = snldStep( L, c )
% Discrete numerical scheme of dL/dt for scalar diffusion
N = size(L, 1);
M = size(L, 2);
⟨Translations of c image 7b⟩
⟨Translations of L image 8a⟩
⟨Calculate dL/dt 8b⟩
```
Root chunk (not used in this document).

Matlab is efficient in dealing with pixel wise image operators. Therefore we first construct images such that the expression to be implemented is a pixel wise combination of several images. For this we need several translated images. As we need so many translations we define a function to do so.

The translation function is based on Matlabs capability to use index vectors. As an example consider the 1D image (vector) `a=[6 5 4 3 2 1]`. Translating this 'image' two pixels to the left by using an index of the form `iind=[3 4 5 6 6 6]`. Then the translated image `[4 3 2 1 1 1]` is obtained as `a[iind]`.

7a ⟨*translateImage.m* 7a⟩≡
```
function r = translateImage( f, di, dj )
% Translation of an image
N = size( f, 1 );
M = size( f, 2 );
if di>0
    iind = [(di+1):N, N*ones(1,di)];
elseif di<0
    iind = [ ones(1,-di), 1:(N+di) ];
else
    iind = 1:N;
end
if dj>0
    jind = [(dj+1):M, M*ones(1,dj)];
elseif dj<0
    jind = [ ones(1,-dj), 1:(M+dj) ];
else
    jind = 1:M;
end
r = f( iind, jind );
```
Root chunk (not used in this document).

7b ⟨*Translations of c image* 7b⟩≡
```
cpc = translateImage( c, 1, 0 );
cmc = translateImage( c, -1 , 0);
ccp = translateImage( c, 0, 1 );
ccm = translateImage( c, 0, -1 );
```
This code is used in chunk 6.

Now `cpc` is the image such that `cpc(i,j)=c(i+1,j)`. Note that `cpc` stands for **c**-image with coordinate shifts i **p**lus 1 and no coordinate shift (translation) in the j coordinate (the **c** denotes the 'central' point).

We can do this for the `L` image as well:

8a ⟨*Translations of L image* 8a⟩≡
```
Lpc = translateImage( L, 1, 0 );
Lmc = translateImage( L, -1, 0 );
Lcp = translateImage( L, 0, 1 );
Lcm = translateImage( L, 0, -1 );
```
This code is used in chunk 6.

This makes the final calculation a simple pixel wise expression in Matlab:

8b ⟨*Calculate dL/dt* 8b⟩≡
```
r = ( (cpc+c).*(Lpc-L) - (c+cmc).*(L-Lmc) + ...
      (ccp+c).*(Lcp-L) - (c+ccm).*(L-Lcm) ) / 2;
```
This code is used in chunk 6.

Observe that for $c = 1$ (i.e. a constant function) the right hand side of the discretized PDE (what the `snldStep` function calculates) reduces to:

$$\texttt{r} = \texttt{Lpc} + \texttt{Lmc} + \texttt{Lcp} + \texttt{Lcm} - 4 * \texttt{L}$$

the classical approximation of the Laplacian. Indeed in this special case the PDE is the linear diffusion equation.

3.1 Perona and Malik Diffusion

Perona and Malik were the first to introduce non-linear diffusion within the image processing context. We consider only one of the *conductivity functions* that they introduced:

$$c(\|\nabla L\|) = \exp\left(-\frac{\|\nabla L\|^2}{k^2}\right)$$

The Matlab code for Perona and Malik diffusion is:

8c ⟨*pmc.m* 8c⟩≡
```
function g = pmc( f, k, stepsize, nosteps, verbose )
% Perona and Malik Diffusion
⟨If verbose then show original image 9b⟩
⟨Run diffusion step 'nosteps' times 9a⟩
```
Root chunk (not used in this document).

For every step the gradient norm and the c-function of the gradient norm should be calculated and the function `snldStep` is called to calculate the change in the image.

8d ⟨*Calculate the (square) of the gradient norm* 8d⟩≡
```
    gx = gD( g, 1, 1, 0 );
    gy = gD( g, 1, 0, 1 );
    grad2 = gx.*gx + gy.*gy;
```
This code is used in chunk 9a.

8e ⟨*Calculate the conductivity function* 8e⟩≡
```
    c = exp( -grad2 / (k^2) );
```
This code is used in chunk 9a.

9a ⟨*Run diffusion step 'nosteps' times* 9a⟩≡

```
g = f;
for i=1:nosteps
    ⟨Calculate the (square) of the gradient norm 8d⟩
    ⟨Calculate the conductivity function 8e⟩

    g = g + stepsize * snldStep( g, c );

    ⟨If verbose then show diffused image 9c⟩
end
```

This code is used in chunk 8c.

A `verbose` parameter equal to zero means that no images are shown of intermediate results. Other `verbose` values indicate the Matlab figure number in which intermediate results are depicted.

9b ⟨*If verbose then show original image* 9b⟩≡

```
if verbose
    figure(verbose);
    subplot(1,2,1); imshow(f); title('Original Image'); drawnow;
end
```

This code is used in chunk 8c.

9c ⟨*If verbose then show diffused image* 9c⟩≡

```
if verbose
    figure(verbose);
    subplot(1,2,2); imshow(g);
    title('Perona and Malik Diffusion'); drawnow;
end
```

This code is used in chunk 9a.

To test the Perona and Malik diffusion consider the following code (the results are shown in Fig. 2). Only a small part from the cameraman image is selected to speed up processing and to show the details.

9d ⟨*pmctest.m* 9d⟩≡

```
a = imread('cameraman.tif');
a = im2double(a);
ad = a(1:100,75:150);
b = pmc( ad, 0.1, .24, 10, 1 );
```

Root chunk (not used in this document).

Other types of conductivity functions can be easily incorporated into the code presented in this section. It would be interesting to look at:

- The Charbonnier conductivity function:

$$c(\|\nabla L\|) = \frac{1}{\sqrt{1 + \frac{\|\nabla L\|^2}{k^2}}}$$

  that results in an inherently stable PDE (compared to the P&M PDE that may be unstable when trying to do inverse diffusion).

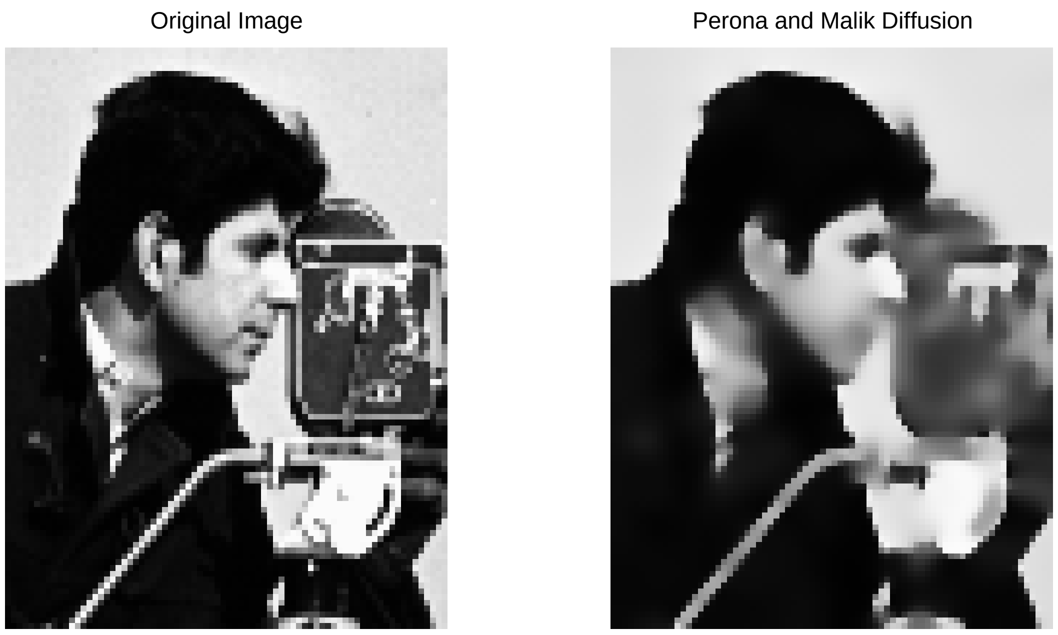


Figure 2: Test of `pmc` function. On the left the original image is shown. On the right the result of Perona and Malik diffusion. For the parameters see the `pmctest` code.

- The conductivity functions proposed by Black [?] that are based on a robust statistical interpretation of non linear diffusion.
- Constructing conductivity functions based on higher order image structure. One might argue that the diffusion should be linked to the isophote curvature.

4 Non-Linear Tensor Diffusion

In this section we consider anisotropic diffusion described with the PDE:

$$\partial_t L = \nabla \cdot (D \nabla L)$$

where D is a positive semi definite symmetric *diffusion tensor*. Here we only consider 2D images. The diffusion tensor is then assumed to be a the form:

$$D = \begin{pmatrix} a & b \\ b & c \end{pmatrix}.$$

All elements of the tensor are functions of the local image structure and hence functions of the spatial coordinates. In Cartesian coordinates we have:

$$\begin{aligned} \partial_t L &= (\partial_x \quad \partial_y) \begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} \partial_x L \\ \partial_y L \end{pmatrix} \\ &= (\partial_x \quad \partial_y) \begin{pmatrix} a\,\partial_x L + b\,\partial_y L \\ b\,\partial_x L + c\,\partial_y L \end{pmatrix} \end{aligned}$$

$$
\begin{aligned}
&= \partial_x(a\,\partial_x L + b\,\partial_y L) + \partial_y(b\,\partial_x L + c\,\partial_y L)\\
&= \partial_x(a\,\partial_x L) + \partial_x(b\,\partial_y L) + \partial_y(b\,\partial_x L) + \partial_y(c\,\partial_y L).
\end{aligned}
$$

If we compare this PDE with the case of scalar diffusion we see that two new terms arise: $\partial_x(b\,\partial_y L)$ and $\partial_y(b\,\partial_x L)$. For the other two terms the same numerical schemes can be used as we have developed in the previous section.

For the mixed terms we can use the symmetrical central differences and still come up with a scheme is a 3×3 neighborhood. Now we take

$$\partial_x^c F = \frac{1}{2}(F_{i+1,j} - F_{i-1,j})$$

and

$$\partial_y^c F = \frac{1}{2}(F_{i,j+1} - F_{i,j-1}).$$

This leads to:

$$
\begin{aligned}
\partial_x(b\,\partial_y L) &\approx \partial_x^c(b\,\partial_y^c L)\\
&= \partial_x^c\left(\frac{1}{2} b_{i,j}(L_{i,j+1} - L_{i,j-1})\right)\\
&= \frac{1}{2}\left(\frac{1}{2} b_{i+1,j}(L_{i+1,j+1} - L_{i+1,j-1}) - \frac{1}{2} b_{i-1,j}(L_{i-1,j+1} - L_{i-1,j-1})\right)\\
&= \frac{1}{4}\left(b_{i+1,j}(L_{i+1,j+1} - L_{i+1,j-1}) - b_{i-1,j}(L_{i-1,j+1} - L_{i-1,j-1})\right).
\end{aligned}
$$

For the other mixed term we obtain:

$$\partial_y(b\,\partial_x L) \approx \frac{1}{4}\left(b_{i,j+1}(L_{i+1,j+1} - L_{i-1,j+1}) - b_{i,j-1}(L_{i+1,j-1} - L_{i-1,j-1})\right).$$

The non linear tensor diffusion PDE now is:

$$
\begin{aligned}
L_{i,j}^{t+dt} = \;& L_{i,j}^{t} + dt\,(\\
& -\frac{b_{i-1,j} + b_{i,j+1}}{4} L_{i-1,j+1} + \frac{c_{i,j+1} + ci,j}{2} L_{i,j+1} + \frac{b_{i+1,j} + b_{i,j+1}}{4} L_{i+1,j+1} + \\
& \frac{a_{i-1,j} + a_{i,j}}{2} L_{i-1,j} - \frac{a_{i-1,j} + 2a_{i,j} + a_{i+1,j} + c_{i-1,j} + 2c_{i,j} + c_{i+1,j}}{2} L_{i,j} + \\
& \quad \frac{a_{i+1,j} + a_{i,j}}{2} L_{i+1,j}\\
& \frac{b_{i-1,j} + b_{i,j-1}}{4} L_{i-1,j-1} + \frac{c_{i,j-1} + ci,j}{2} L_{i,j-1} + -\frac{b_{i+1,j} + b_{i,j-1}}{4} L_{i+1,j-1}).
\end{aligned}
$$

Note that all terms in the rhs of the PDE are all linear in the values of $L_{i,j}$. This allows us to write the discretization of the PDE as a quasi convolution kernel (stencil). It is *not* a real convolution as the values in the 3×3 kernel are dependent on the diffusion tensor and thus dependent on the spatial position. The stencil is:

$-\frac{b_{i-1,j}+b_{i,j+1}}{4}$	$\frac{c_{i,j+1}+ci,j}{2}$	$\frac{b_{i+1,j}+b_{i,j+1}}{4}$
$\frac{a_{i-1,j}+a_{i,j}}{2}$	$\frac{a_{i-1,j}+2a_{i,j}+a_{i+1,j}+c_{i-1,j}+2c_{i,j}+c_{i+1,j}}{2}$	$\frac{a_{i+1,j}+a_{i,j}}{2}$
$\frac{b_{i-1,j}+b_{i,j-1}}{4}$	$\frac{c_{i,j-1}+ci,j}{2}$	$-\frac{b_{i+1,j}+b_{i,j-1}}{4}$

For implementing this discretization of the PDE in Matlab we again use the 'trick' first to construct translated versions of the images and then implement the scheme as a pixel wise combination of (a lot of) images. We need all 9 translations of L within a 3×3 neighborhood and 10 more translations of the images a, b and c, making a total of 19 images.

12a ⟨*tnldStep.m* 12a⟩≡

```
function r = tnldStep( L, a, b, c )
% Discrete numerical scheme of dL/dt for tensor diffusion
N = size( L, 1 );
M = size( L, 2 );
⟨Translations of L (tnldStep) 12b⟩
⟨Translations of a (tnldStep) 12c⟩
⟨Translations of b (tnldStep) 12d⟩
⟨Translations of c (tnldStep) 12e⟩
⟨Calculate dL/dt (tnldStep) 13⟩
```

Root chunk (not used in this document).

We need all translation of `L` within the 3×3 neighborhood.

12b ⟨*Translations of L (tnldStep)* 12b⟩≡

```
Lpc = translateImage( L, 1, 0 );
Lpp = translateImage( L, 1, 1 );
Lcp = translateImage( L, 0, 1 );
Lmp = translateImage( L, -1, 1 );
Lmc = translateImage( L, -1, 0 );
Lmm = translateImage( L, -1, -1 );
Lcm = translateImage( L, 0, -1 );
Lpm = translateImage( L, 1, -1 );
```

This code is used in chunk 12a.

We only need two translations of a besides the image a itself.

12c ⟨*Translations of a (tnldStep)* 12c⟩≡

```
amc = translateImage( a, -1, 0 );
apc = translateImage( a, +1, 0 );
```

This code is used in chunk 12a.

For b we need 4 translations:

12d ⟨*Translations of b (tnldStep)* 12d⟩≡

```
bmc = translateImage( b, -1, 0 );
bcm = translateImage( b, 0, -1 );
bpc = translateImage( b, +1, 0 );
bcp = translateImage( b, 0, +1 );
```

This code is used in chunk 12a.

And for c we need again only two translated versions:

12e ⟨*Translations of c (tnldStep)* 12e⟩≡

```
ccp = translateImage( c, 0, +1 );
ccm = translateImage( c, 0, -1 );
```

This code is used in chunk 12a.

This makes the final expression simple (although long...)

13 ⟨*Calculate dL/dt (tnldStep)* 13⟩≡

```
r = -1/4 * (bmc+bcp) .* Lmp + ...
     1/2 * (ccp+c)   .* Lcp + ...
     1/4 * (bpc+bcp) .* Lpp + ...
     1/2 * (amc+a)   .* Lmc - ...
     1/2 * (amc+2*a+apc+ccm+2*c+ccp) .* L + ...
     1/2 * (apc+a)   .* Lpc + ...
     1/4 * (bmc+bcm) .* Lmm + ...
     1/2 * (ccm+c)   .* Lcm - ...
     1/4 * (bpc+bcm) .* Lpm;
```

This code is used in chunk 12a.

The numerical scheme presented in this subsection is not the best possible one. It may lead in certain situations to behaviour that is not in accordance with the smoothing properties we have in mind while developing the (continuous) theory. Weickert gives a somewhat more complex numerical scheme on a 3×3 stencil that eliminates these problems (at the expense that the anisotropy should be somewhat limited). We leave the implementation of that scheme to the interested reader (and kindly ask to mail me the code).

4.1 Edge Enhancing Diffusion

In many situations the local gradient (measured at infinitesimally small scale) does not provide a useful indication of the local perceptual orientation in an image. Often the edges are quite noisy causing the gradient to fluctuate considerably, both in magnitude as well as in direction.

Edge enhancing diffusion is based on the idea that a better estimate of the perceptual significant orientation of the edge direction can be obtained by constructing the diffusion tensor based on an orientation estimate obtained from observing the image at a larger scale. We use the name 'edge enhancing diffusion' as was introduced by Weickert, although it might be better called 'edge preserving smoothing'.

Edge enhancing diffusion constructs the diffusion tensor D as follows:

$$D = R^T \begin{pmatrix} c_1 & 0 \\ 0 & c_2 \end{pmatrix} R$$

where R is the rotation matrix describing the local coordinate system aligned with the gradient vector observed at scale u.

$$R = \frac{1}{\sqrt{(L_x^u)^2 + (L_y^u)^2}} \begin{pmatrix} L_x^u & -L_y^u \\ L_y^u & L_x^u \end{pmatrix}$$

where L^u denotes the image observed at scale u. This leads to a diffusion tensor

$$D = \frac{1}{(L_x^u)^2 + (L_y^u)^2} \begin{pmatrix} c_1(L_x^u)^2 + c_2(L_y^u)^2 & (c_2 - c_1)L_x^u L_y^u \\ (c_2 - c_1)L_x^u L_y^u & c_1(L_y^u)^2 + c_2(L_x^u)^2 \end{pmatrix}.$$

Note that c_1 is the conductivity in the direction of the gradient (observed at scale u) and the c_2 is the conductivity along the isophote. In order to compare edge enhancing diffusion with scalar diffusion (Perona and Malik type) we set the diffusion along the edge to be equal to the isotropic diffusion in the Perona and Malik diffusion discussed earlier and set the conductivity across the edge to be one fifth of the conductivity along the edge.

$$\begin{aligned} c_2(L^u_w) &= e^{-\frac{(L^u_w)^2}{k^2}} \\ c_1(L^u_w) &= \frac{1}{5}c_2(L^u_w) \end{aligned}$$

Here $L^u_w = \sqrt{(L^u_x)^2 + (L^u_y)^2}$ is the gradient norm.

14a ⟨*eed.m* 14a⟩≡
```
function R = eed( L, k, uscale, stepsize, nosteps, verbose )
% eed: edge enhancing diffusion
⟨Show original image if verbose 15a⟩
⟨Run edge enhancing diffusion 'nosteps' times 14b⟩
```
Root chunk (not used in this document).

The edge enhancing diffusion is:

14b ⟨*Run edge enhancing diffusion 'nosteps' times* 14b⟩≡
```
R = L;
for i = 1:nosteps
    ⟨Calculate Rx, Ry and Rw2 at scale u 14c⟩
    ⟨Calculate c1 and c2 14d⟩
    ⟨Calculate diffusion tensor components 14e⟩
    R = R + stepsize * tnldStep( R, a, b, c );
    ⟨If verbose show edge enhanced diffusion result 15b⟩
end
```
This code is used in chunk 14a.

14c ⟨*Calculate Rx, Ry and Rw2 at scale u* 14c⟩≡
```
Rx = gD( R, uscale, 1, 0 );
Ry = gD( R, uscale, 0, 1 );
Rw2 = Rx.^2 + Ry.^2;
Rw = sqrt(Rw2);
```
This code is used in chunk 14b.

14d ⟨*Calculate c1 and c2* 14d⟩≡
```
c2 = exp( - (Rw / k).^2 );
c1 = 1/5 * c2;
```
This code is used in chunk 14b.

14e ⟨*Calculate diffusion tensor components* 14e⟩≡
```
a = (c1 .* Rx.^2 + c2 .* Ry.^2) ./ (Rw2+eps);
b = (c2-c1) .* Rx .* Ry ./ (Rw2+eps);
c = (c1 .* Ry.^2 + c2 .* Rx.^2) ./ (Rw2+eps);
```
This code is used in chunk 14b.

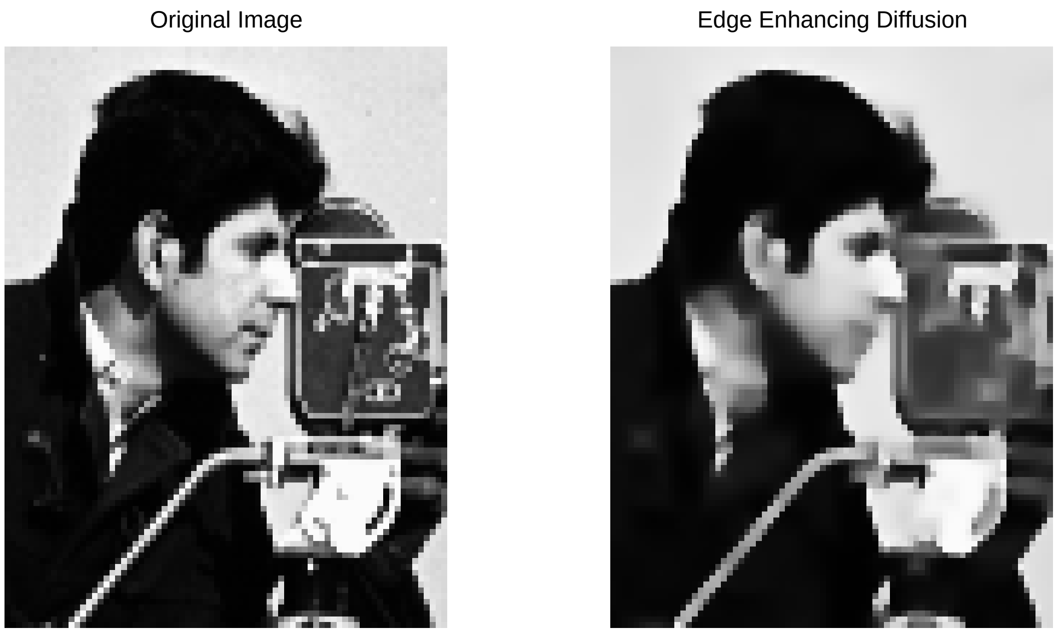


Figure 3: Test of `eed` function. On the left the original image is shown. On the right the result of the edge enhancing diffusion. For the parameters see the `eedtest` code.

Settting the `verbose` parameter to a positive integer value, the corresponding Matlab figure will show the original image and the final result.

15a ⟨*Show original image if verbose* 15a⟩≡

```
if verbose
    figure(verbose);
    subplot(1,2,1); imshow(L); title('Original Image'); drawnow;
end
```

This code is used in chunk 14a.

15b ⟨*If verbose show edge enhanced diffusion result* 15b⟩≡

```
if verbose
    figure(verbose);
    subplot(1,2,2); imshow(R);
    title('Edge Enhancing Diffusion'); drawnow;
end
```

This code is used in chunk 14b.

To test edge enhancing diffusion consider the following code.

15c ⟨*eedtest.m* 15c⟩≡

```
a = imread('cameraman.tif');
a = im2double(a);
ad = a(1:100,75:150);
b = eed( ad, 0.1, 1, .24, 10, 2 );
```

Root chunk (not used in this document).

The resultant images are depicted in Fig. 3. Comparing the results with the classical Perona and Malik diffusion we see that in the homogenuous regions the smoothing is comparable but that the edge enhancing diffusion preserves the edges much better.

4.2 Coherence Enhancing Diffusion

In some situations estimating the local orientation as the direction of the gradient vector is not possible. Consider the finger print image in Fig. ??(a). In figure (b), (c) and (d) the same image is shown observed at scales 0.7, 1.0 and 2.0. It is obvious that the local structure one is interested in, i.e. the orientation of the finger print pattern is totally wipes out when observing the image at scale 2. However observing it at smaller scales makes the orientation estimation very noise sensitive and renders it useless in practical applications.

One such practical application is image smoothing where we do like to smooth the image, without of course destroying the finger print lines. The isotropic Gaussian smoothing is of no great help here (as shown in Fig. ??). We would like to find the prominent (perceptual) local orientation and smooth along the finger print lines but not accross them.

The local orientation estimation is based on the *structure tensor*[3]:

$$S = \begin{pmatrix} s_{11} & s_{12} \\ s_{12} & s_{22} \end{pmatrix} = \begin{pmatrix} L_x L_x * G^u & L_x L_y * G^u \\ L_x L_y * G^u & L_y L_y * G^u \end{pmatrix}$$

whose eigenvectors indicate the most prominent orientation. The difference between the two eigenvalues is an indication of the anistropy in a local neighborhood in the image.

Coherence enhancing diffusion constructs the diffusion tensor D as follows:

$$D = \begin{pmatrix} d_{11} & d_{12} \\ d_{12} & d_{22} \end{pmatrix} = R^T \begin{pmatrix} c_1 & 0 \\ 0 & c_2 \end{pmatrix} R$$

Here is the rotation matrix whose columns are the eigenvectors of the *structure tensor* S (i.e. indicting the local orientation of an image patch) and c_1 and c_2 are the conductivity coefficients along the principal directions.

The eigenvectors of the structure tensor and thus the rotation matrix R can be calculated analytically (using for instance Mathematica). This leads to the diffusion tensor D with components:

$$\begin{aligned} d_{11} &= \frac{1}{2}\left(c_1 + c_2 + \frac{(c_2 - c_1)(s_{11} - s_{22})}{\alpha}\right) \\ d_{12} &= \frac{(c_2 - c_1)s_{12}}{\alpha} \\ d_{22} &= \frac{1}{2}\left(c_1 + c_2 - \frac{(c_2 - c_1)(s_{11} - s_{22})}{\alpha}\right). \end{aligned}$$

[3]In my opinion one should take $\tilde{L}_x = L_x - \langle L_x \rangle$ where $\langle L_x \rangle$ is the mean gradient (component) in the neighborhood. This way a correction for a constant gradient term is build in). It would make the calculation of the structure tensor somewhat more complex than a convolution. Maybe that is the reason that such a correction is not described in the literature.

where

$$\alpha = \sqrt{(s_{11} - s_{22})^2 + 4s_{12}^2}.$$

The eigenvalues of the structure tensor are given by:

$$\lambda_{1,2} = \frac{1}{2}\left(s_{11} + s_{22} \pm \alpha\right).$$

These eigenvalues determine the diffusion 'speeds' c_1 and c_2. We select:

$$c1 = \max(0.01, 1 - e^{-(\lambda_1-\lambda_2)^2/k^2}); c2 = 0.01;$$

17a ⟨*ced.m* 17a⟩≡

```
function R = ced( L, k, obsscale, intscale, stepsize, nosteps, verbose )
% ced: coherence enhancing diffusion
⟨Show original image if verbose (ced) 18a⟩
⟨Run coherence enhancing diffusion 'nosteps' times 17b⟩
```

Root chunk (not used in this document).

The coherence enhancing diffusion is:

17b ⟨*Run coherence enhancing diffusion 'nosteps' times* 17b⟩≡

```
R = L;
for i = 1:nosteps
    ⟨Calculate structure tensor components 17c⟩
    ⟨Calculate c1 and c2 (ced) 17d⟩
    ⟨Calculate diffusion tensor components (ced) 17e⟩
    R = R + stepsize * tnldStep( R, d11, d12, d22  );
    ⟨If verbose show coherence enhanced diffusion result 18b⟩
end
```

This code is used in chunk 17a.

The components of the structure tensor (S) are calculated as:

17c ⟨*Calculate structure tensor components* 17c⟩≡

```
Rx = gD( R, obsscale, 1, 0 );
Ry = gD( R, obsscale, 0, 1 );
s11 = gD( Rx.^2,  intscale, 0, 0 );
s12 = gD( Rx.*Ry, intscale, 0, 0 );
s22 = gD( Ry.^2,  intscale, 0, 0 );
alpha = sqrt( (s11-s22).^2 + 4*s12.^2 );
el1 = 1/2 * (s11 + s22 - alpha );
el2 = 1/2 * (s11 + s22 + alpha );
```

This code is used in chunk 17b.

17d ⟨*Calculate c1 and c2 (ced)* 17d⟩≡

```
c1 = max(0.01, 1-exp( -(el1-el2).^2 / k^2 ));
c2 = 0.01;
```

This code is used in chunk 17b.

17e ⟨*Calculate diffusion tensor components (ced)* 17e⟩≡

```
d11 = 1/2 * (c1+c2+(c2-c1).*(s11-s22)./(alpha+eps));
d12 = (c2-c1).*s12./(alpha+eps);
d22 = 1/2 * (c1+c2-(c2-c1).*(s11-s22)./(alpha+eps));
```

This code is used in chunk 17b.

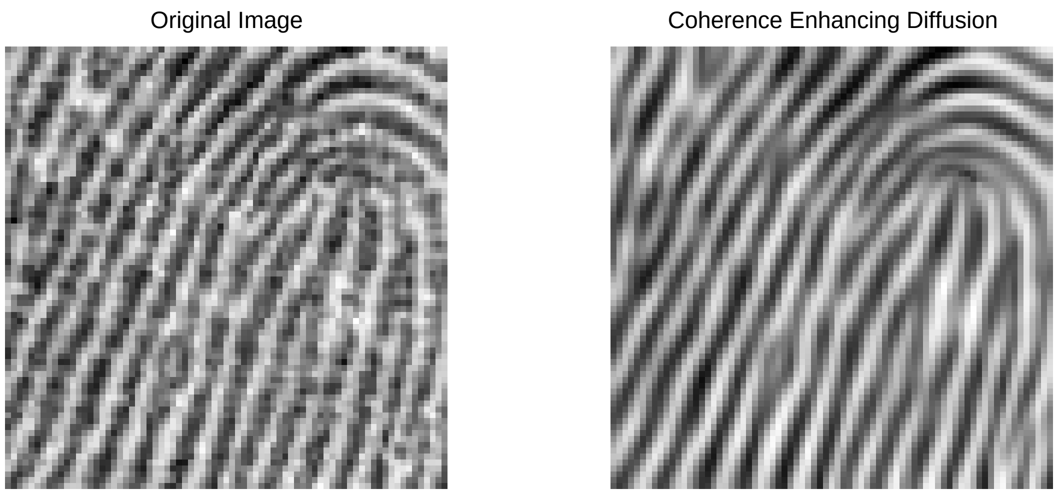


Figure 4: Test of `ced` (coherence enhancing diffusion) function. On the left the original image (showing a detail of a fingerprint) is shown. On the right the result of the coherence enhancing diffusion is shown. For the parameters see the `cedtest` code.

Settting the `verbose` parameter to a positive integer value, the corresponding Matlab figure will show the original image and the final result.

18a ⟨*Show original image if verbose (ced)* 18a⟩≡
```
if verbose
    figure(verbose);
    subplot(1,2,1); imshow(L,[]); title('Original Image'); drawnow;
end
```
This code is used in chunk 17a.

18b ⟨*If verbose show coherence enhanced diffusion result* 18b⟩≡
```
if verbose
    figure(verbose);
    subplot(1,2,2); imshow(R,[]);
    title('Coherence Enhancing Diffusion'); drawnow;
end
```
This code is used in chunk 17b.

To test the coherence enhancing diffusion algorithm consider the following code.

18c ⟨*cedtest.m* 18c⟩≡
```
a = imread( 'fp.jpg' );
a = im2double( a );
ad = a( 51:125, 76:150 );
bd = ced(ad, 0.001, 0.7, 5, 0.2, 10, 1);
```
Root chunk (not used in this document).

The resultant image is depicted in Fig. 4. The remarkable result is due to the combination of the local orientation measurement using the eigenvalues and eigenvectors of the structure tensor and the tensor diffusion where we direct the diffusion along the fingerprint lines.

5 Conclusion

In this report we have only presented the very simple forward Euler numerical schemes to solve the diffusion PDE's. Implicit schemes are more efficient (stable for arbitrary time step). It seems doable to implement these in Matlab as well. Any volunteers...???

A Literate programming using `noweb`

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

D. Knuth, 1984

This is not the place to explain what literate programming is about. If you couldn't understand this report (especially the way Matlab code should be 'constructed' from the chunks that are presented in the text) or in case you would like to read more about literate programming I suggest you take a look at

- `www.literateprogramming.com`: a site containing a lot of useful material.
- The literate programming FAQ. Also a source of a lot of information (`http://shelob.ce.ttu.edu/daves/lpfaq/faq.html`).

This report uses the `noweb` literate programming tool. Look at the website `http://www.eecs.harvard.edu/~nr/noweb/` for more information on `noweb`. If anyone dares to make a Windows version of the `noweb3` program, please send me an email...

B Generate the Matlab code

The `mltangle.bat` script (batch file) simplifies the calling sequence of the `notangle` script somewhat. It assumes the current working directory is the directory where the `noweb` file resides and that a `matlab` subdirectory exists.

19 ⟨*mltangle.bat* 19⟩≡
```
    call notangle -R%2 %1 > matlab\%2
```
Root chunk (not used in this document).

The `tangleAll.bat` file tangles all Matlab functions embedded in a `noweb` file and places them into the `matlab` directory. Please note that it should be doable to construct the `tangleAll` script automatically from all chunks ending with `.m`. For now the script below is constructed manually.

20a ⟨*tangleAll.bat* 20a⟩≡

```
call mltangle nldiffusioncode.w gD.m
call mltangle nldiffusioncode.w convSepBrd.m
call mltangle nldiffusioncode.w g1Jet.m
call mltangle nldiffusioncode.w g2Jet.m
call mltangle nldiffusioncode.w gDtest.m
call mltangle nldiffusioncode.w snldStep.m
call mltangle nldiffusioncode.w pmc.m
call mltangle nldiffusioncode.w pmctest.m
call mltangle nldiffusioncode.w tnldStep.m
call mltangle nldiffusioncode.w translateImage.m
call mltangle nldiffusioncode.w eed.m
call mltangle nldiffusioncode.w eedtest.m
call mltangle nldiffusioncode.w ced.m
call mltangle nldiffusioncode.w cedtest.m
call mltangle nldiffusioncode.w generateFigures.m
```

Root chunk (not used in this document).

C Generate the figures

This file assumes that a directory `..\figures` exists where all figures are stored.

20b ⟨*generateFigures.m* 20b⟩≡

```
gDtest;
print -depsc2 ..\figures\gDtest.eps

pmctest;
print -depsc2 ..\figures\pmctest.eps

eedtest;
print -depsc2 ..\figures\eedtest.eps

cedtest;
print -depsc2 ..\figures\cedtest.eps
```

Root chunk (not used in this document).

Acknowledgements

I would like to thank Joachim Weickert for patiently answering email questions concerning the nitty gritty details of nonlinear diffusion numerical schemes.