Artificial Vision Report - Lab 3

Olivier Jais-Nielsen

April 7, 2009

Chapter 1

C++ Code

1.1 Heat Diffusion

The `HeatDiffusion` fucntion applies the heat diffusion equation to the image `_in` and stores the result in the image `_out` with the time step `_step` and for `_nbIteration` iterations. At each iteration, if x is a pixel it replace its intensity u with the following value : $u' = \delta t.\Delta u + u$ where δt is the time step. The function `LaplacianMask` from the previous lab is used to compute the Laplacian mask. Cf. algorithm 1.1.

Algorithm 1.1 HeatDiffusion

```
1 void HeatDiffusion(CImg<float> &_out, const CImg<float> &
      _in, float _step, int _nbIteration)
2 {
3         CImg<float> mask = LaplacianMask();
4         _out = _in;
5         for (int i = 0; i < _nbIteration; i++)
6         {
7                 _out += _step * _out.get_convolve(mask);
8         }
9 }
```

1.2 Anisotropic Smoothing

1.2.1 Gradient computation

The functions `HorGradientMask` and `VerGradientMask` respectively return mask for the horizontal gradient and for the vertical, according to the following for-

mulas : $\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$. Cf. algorithms 1.2 and 1.3.

Algorithm 1.2 HorGradientMask

```
1  CImg<float> HorGradientMask()
2  {
3          const char *const values = "-1,0,1";
4          CImg<float> mask(1, 3, 1, 1, values, 0);
5          return mask;
6  }
```

Algorithm 1.3 VerGradientMask

```
1  CImg<float> VerGradientMask()
2  {
3          const char *const values = "-1,0,1";
4          CImg<float> mask(3, 1, 1, 1, values, 0);
5          return mask;
6  }
```

1.2.2 The image dependent diffusion coeficient

The function `g` is the one that will be applied to the gradient magnitude. Cf. algorithm 1.4.

1.2.3 Main function

The main function first sets the global variable `paramK` with the input `_paramK`. Then, it applies, in the same way as the heat diffusion, the partial derivatives equation. It uses `HorGradientMask` and `VerGradientMask` to compute the gradient, its magnitude and the divergence. With the same notations as before,

Algorithm 1.4 g

```
1  float paramK;
2  float g(float _s)
3  {
4          return exp(-_s / paramK);
5  }
```

at each iteration it does : $u' = \delta t \left[\frac{\partial}{\partial x} \left(g \left(||\nabla u|| \frac{\partial u}{\partial x} \right) \right) + \frac{\partial}{\partial y} \left(g \left(||\nabla u|| \frac{\partial u}{\partial y} \right) \right) \right] + u$ where g is the function `g`. Cf. algorithm 1.5.

Algorithm 1.5 AnisotropicDiffusion

```
1  void AnisotropicDiffusion(CImg<float> &_out, const CImg<
       float> &_in, int _nbIt, float _paramK)
2  {
3          CImg<float> horMask = HorGradientMask();
4          CImg<float> verMask = VerGradientMask();
5          CImg<float> img_horGrad(_in);
6          CImg<float> img_verGrad(_in);
7          CImg<float> img_gradMagn(_in);
8          paramK = _paramK;
9          _out = _in;
10         for (int i = 0; i < _nbIt; i++)
11         {
12                 img_horGrad = _out.get_convolve(horMask);
13                 img_verGrad = _out.get_convolve(verMask);
14                 img_gradMagn = img_horGrad.get_sqr() +
                       img_verGrad.get_sqr();
15                 img_gradMagn.sqrt();
16                 img_gradMagn.apply(g);
17                 img_horGrad.mul(img_gradMagn);
18                 img_verGrad.mul(img_gradMagn);
19                 img_horGrad.convolve(horMask);
20                 img_verGrad.convolve(verMask);
21                 _out += 0.2 * (img_horGrad + img_verGrad)
                       ;
22         }
23 }
```

The time step is equal to 0.2.

1.3 Total Variation Flow

This algorithm is identical to the previous except for the definition of the function `g` which is replaced by the function `h`. Cf. algorithm 1.6 on the next page.

1.4 Bonus : Inpainting

To achieve the inpainting, the program smooths the erased region with the boundary parameters being the edge of the region, that is kept constant. First, we define the altered region : we assume that it is composed by all the perfectly

Algorithm 1.6 h

```
1  float h(float _s)
2  {
3          if (_s != 0)
4          {
5                  return paramK / _s;
6          }
7          else
8          {
9                  return 100000000000;
10         }
11 }
```

The function avoids the division by zero and logically replaces it with a very big integer representing infinity.

white pixels of the image (i.e. which value is equal to 255). We store the coordinates of these pixels in the vector `area`. At each iteration, we apply the total variation flow smoothing to a copy (`temp`) of the entire image and we put all the newly calculated values of the pixels of `area` in the original image with the function `put_vals` (Cf. algorithm 1.8 on page 6); finally the copy is updated for the next iteration. Cf. algorithm 1.7 on the next page.

Algorithm 1.7 Inpainting

```
1  void Inpainting(CImg<float> &_out, const CImg<float> &_in
       , int _nbIt, float _paramK)
2  {
3          CImg<float> horMask = HorGradientMask();
4          CImg<float> verMask = VerGradientMask();
5          CImg<float> img_horGrad(_in);
6          CImg<float> img_verGrad(_in);
7          CImg<float> img_gradMagn(_in);
8          paramK = _paramK;
9          int w = _in.dimx();
10         int h = _in.dimy();
11         std::vector<int> area;
12         CImg<float> temp(_in);
13
14         _out = _in;
15         for (int x = 0; x < w; x++)
16         {
17                 for (int y = 0; y < h; y++)
18                 {
19                         if (_in(x, y) == 255)
20                         {
21                                 area.push_back(x);
22                                 area.push_back(y);
23                         }
24                 }
25         }
26         for (int i = 0; i < _nbIt; i++)
27         {
28                 img_horGrad = _out.get_convolve(horMask);
29                 img_verGrad = _out.get_convolve(verMask);
30                 img_gradMagn = img_horGrad.get_sqr() +
                       img_verGrad.get_sqr();
31                 img_gradMagn.sqrt();
32                 img_gradMagn.apply(g);
33                 img_horGrad.mul(img_gradMagn);
34                 img_verGrad.mul(img_gradMagn);
35                 img_horGrad.convolve(horMask);
36                 img_verGrad.convolve(verMask);
37                 temp += 0.2 * (img_horGrad + img_verGrad)
                       ;
38                 put_vals(_out, temp, area);
39                 temp = _out;
40                 std::cout << "Itération : " << i << std::
                       endl;
41         }
42 }
```

Algorithm 1.8 put_vals

```
1  void put_vals(CImg<float> &_out, const CImg<float> &_in,
       std::vector<int> _area)
2  {
3          int x, y;
4          for (int i = 0; i < _area.size(); i += 2)
5          {
6                  x = _area[i];
7                  y = _area[i+1];
8                  _out(x, y) = _in(x, y);
9          }
10 }
```

Chapter 2

Experiments

2.1 Heat Diffusion

Using the algorithm, it appeared that to have a close result to the gaussian filter with a deviation of 2, the heat diffusion algorithm needs 10 iterations with a time step of 0.2. Cf. figure 2.1.

Figure 2.1: Comparison of a Gaussian filter and a heat diffusion algorithm

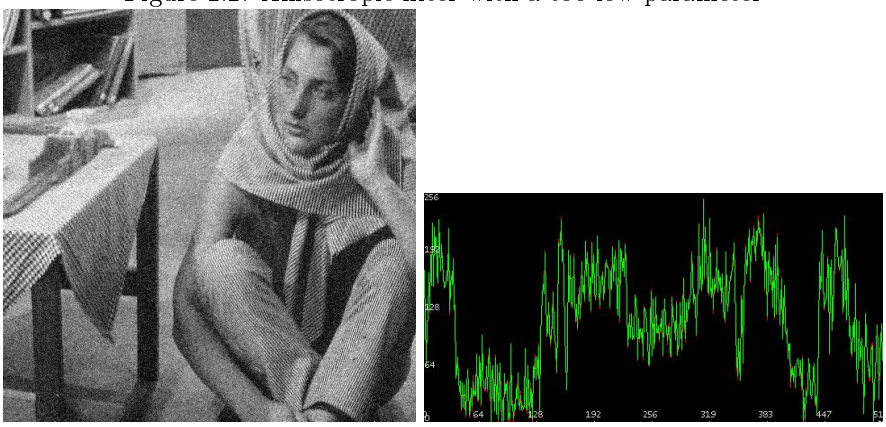
Left image : original. Right image : image convolved with a gaussian mask with a deviation of 2.

Left image : after 1 iteration of the heat diffusion (with a time step of 0.2). Center image : after 5 iterations. Right image : after 10 iterations.

2.2 Anisotropic Diffusion

For a too low value of K, the anisotropic filter is inefficient. Cf. figure 2.2.

Figure 2.2: Anisotropic filter with a too low parameter



K : 10
Iterations : 5
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

An appropriate value of K is 50; the anisotropic filter is more efficient with the iteration numbers. Cf. figure 2.3 on the following page.

A Gaussian filter smoothes better than an anisotropic filter but doesn't preserve the edges. Cf. figure 2.4 on page 10.

2.3 Total Variation Flow

For a too high value of K, the total variation flow filter is inefficient and adds artifacts. Cf. figure 2.5 on page 11.

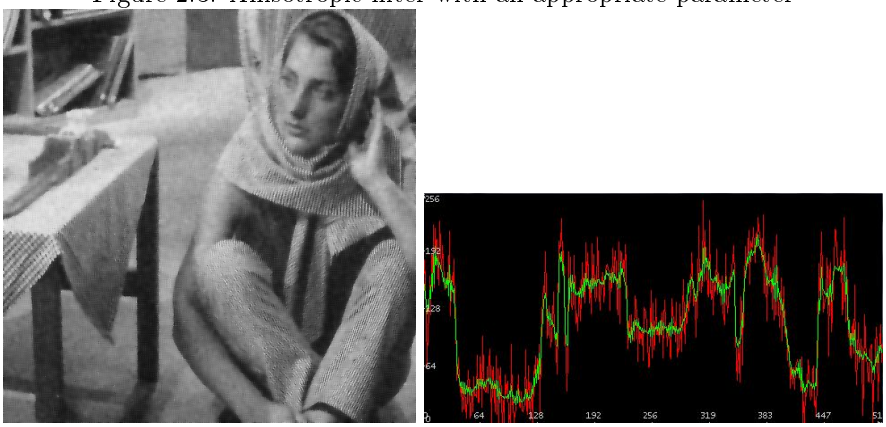
An appropriate value of K is 10; the total variation flow filter is more efficient with the iteration numbers. Cf. figure 2.6 on page 12.

Like for the anisotropic filter, a Gaussian filter smoothes better than an anisotropic filter but doesn't preserve the edges. Cf. figure 2.7 on page 13.

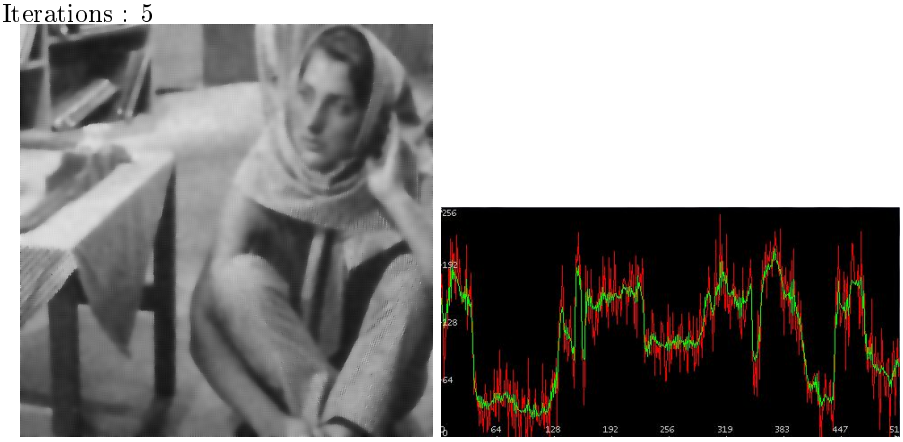
2.4 Bonus : Inpainting

For the inpainting, the results are pretty good but the algorithm converges slowly. Cf. figure 2.8 on page 14.

Figure 2.3: Anisotropic filter with an appropriate parameter



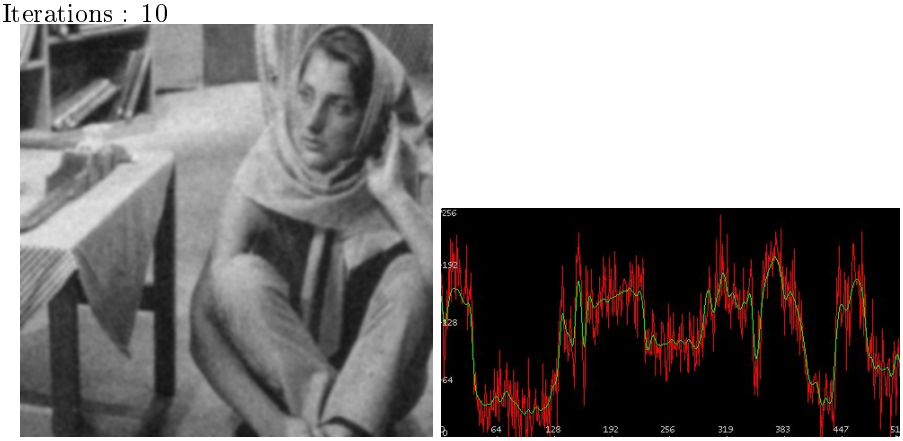
K : 50
Iterations : 5



K : 50
Iterations : 10
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

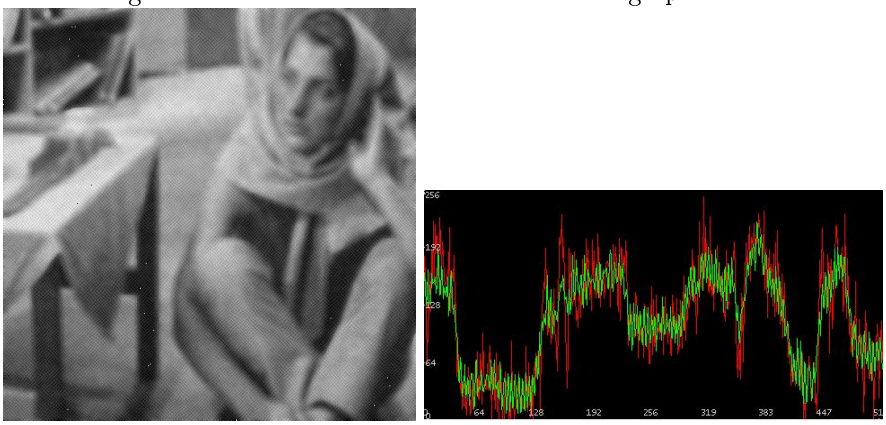
Figure 2.4: Anisotropic filter vs. Gaussian filter

Anisotropic filter.
K : 50
Iterations : 10



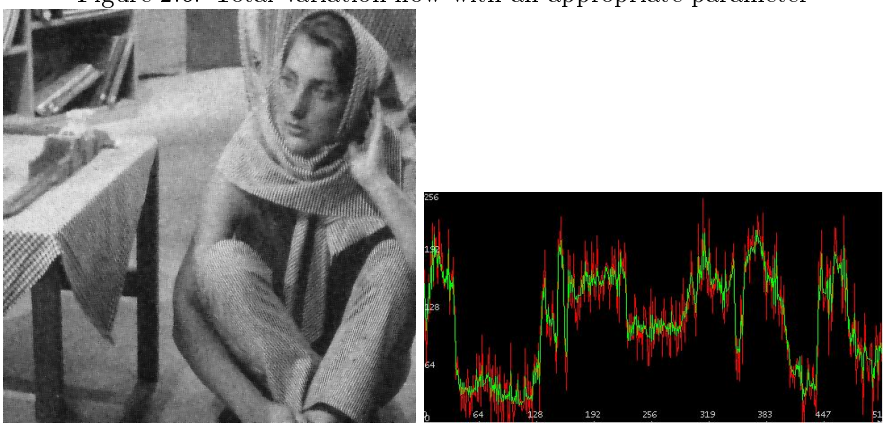
Gaussian filter.
Deviation : 2
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

Figure 2.5: Total variation flow with a too high parameter

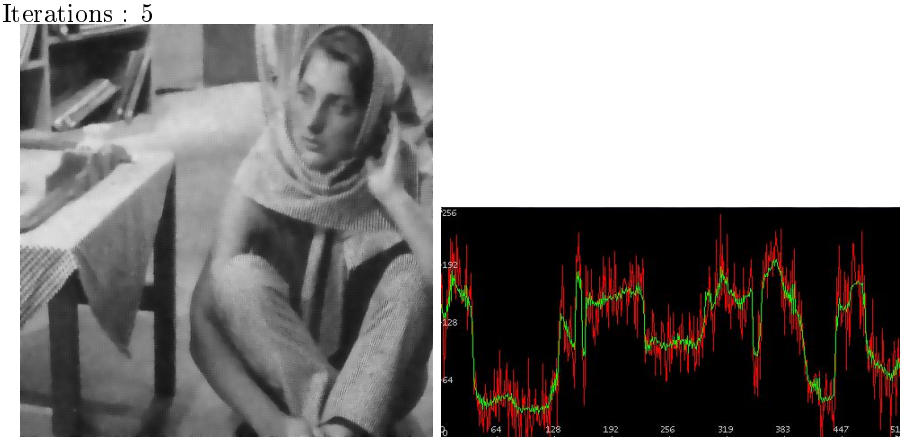


K : 50
Iterations : 5
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

Figure 2.6: Total variation flow with an appropriate parameter



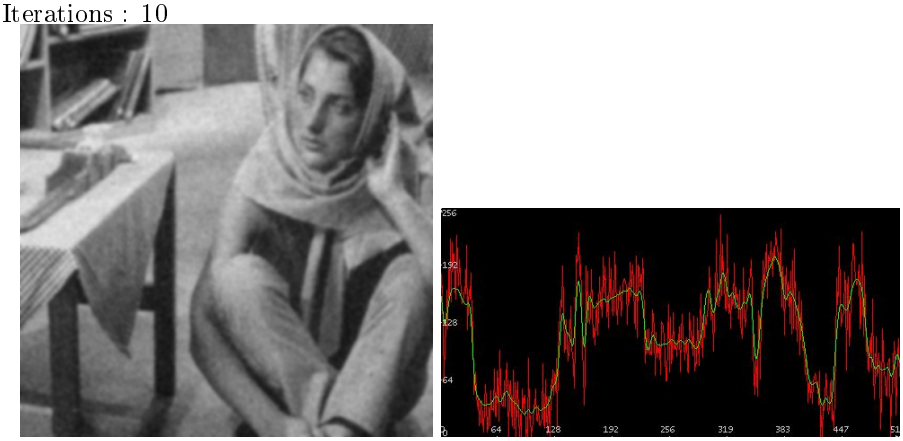
K : 10
Iterations : 5



K : 10
Iterations : 10
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

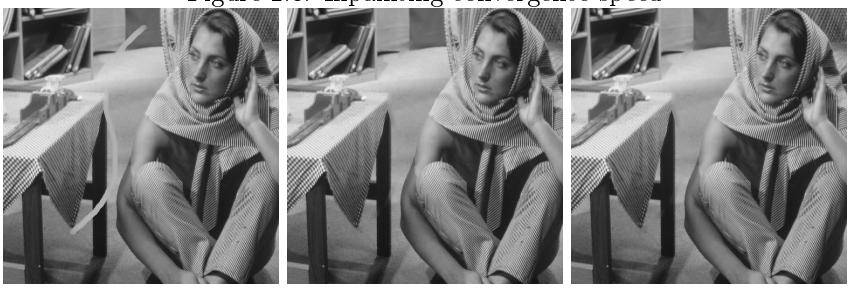
Figure 2.7: Total variation flow filter vs. Gaussian filter

Total variation flow filter.
K : 10
Iterations : 10



Gaussian filter.
Deviation : 2
Green plot : intensity value along a row on the filtered image.
Red plot : intensity value along the same row on the original image.

Figure 2.8: Inpainting convergence speed



From left to right : after 100, 500 and 1000 iterations