Practice 1
Image processing with Partial Differential Equations

Jean-François Aujol & Nicolas Papadakis

IMB, Université Bordeaux
351 Cours de la libération, 33405 Talence Cedex, FRANCE

Email : jean-francois.aujol@math.u-bordeaux.fr

1. Introduction

All along this practice, we will consider gray scale images and we will use MATLAB®.

1.1 General advices when using MATLAB

MATLAB has been first developed for solving matrix problems and its intern functions are optimized for such purpose. Hence, for optimizing computational runtime, it is recommended to write your algorithms in a vectorial/matrix form and to avoid the use of *for* loops when possible. Even if not necessary, it is recommended to initialize variables (with functions such as *zeros*, *ones*). To comment a line, use the character %. All created figures can be closed with the command *close all*. All created variables can be cleared with the command *clear all*. Do not hesitate to look at the documentation to correctly use the predefined functions (command: *help*).

1.2 MATLAB and images

In MATLAB, a gray scale image I of size (m, n) is a matrix I of size (m, n), and the value $I(i, j)$ corresponds to the gray value at pixel location (i, j), $i = 1 \cdots m$, $j = 1 \cdots n$.

MATLAB can read images written in any standard format with the command *imread*. In the same way, results can be written with the command *imwrite*. When reading an image with *imread*, the loaded values are integers and the gray values are in the range $[0; 255]$, from black to white.

Remark: You can copy/paste from this pdf the following scripts and commands.

1.3 Basic examples

Example 1: Reading and displaying an image

```
%load the image camerama.tif that is included in MATLAB's own dataset:

Im_data=imread('cameraman.tif');

%VERY IMPORTANT: do not forget to convert integer values
%in real ones for future manipulations:

Im_data=double(Im_data);

%Display:

imagesc(Im_data);
colormap gray;

%To open a second figure:

figure;

%Other command for image display:

imshow(Im_data/255.);

%If the image has real values, imshow require these values to be
%in the range [0;1] for a correct display. A normalization is thus
%required if working in the range [0;255]
```

The previous script should display:

Display of image Cameraman with *imagesc*

Display of image Cameraman with *imshow*

Example 2: Avoid "for loops"

```
N=2000;
A=ones(N,N);   %Initialize A with values 1
B=randn(N,N);  %Initialize B with random values in [0;1]

%Compute product between matrices elements.
tic
for i=1:N,
     for j=1:N,
          A(i,j)=A(i,j)*B(i,j);
     end
end
toc    %Display runtime since tic command

tic
A=A.*B;  % .* for an element wise operation between matrices
%(different from A*B that realizes a matrix multiplication)
toc
```

The previous script should give you as output something (depending on the computer) like:

```
Result after execution:
Elapsed time is 0.129287 seconds.
Elapsed time is 0.005119 seconds.
```

Exemple 3: Adding gaussian noise to an image with a function

Create the file add_gaussian_noise.m containing:

```
function out = add_gaussian_noise(I,s)
%Add Gaussian noise of standard deviation s to an image I

[m,n]=size(I);
%creation and addition of gaussian noise
out=I+s*randn(m,n);
```

The script:

```
Im_noised=add_gaussian_noise(Im_data,30);
imagesc(Im_noised)
colormap gray;
```

should give you:

2 Denoising with PDEs

Let Ω be a bounded open set of $\mathbb{R}^2$. We will typically consider Ω as a rectangle representing the image domain. An image is a function defined on Ω such as $u : \Omega \to \mathbb{R}$ ($u : \Omega \to \mathbb{R}^3$ for color images), i.e. for pixels $x \in \Omega$.

We now consider that we have an image f that corresponds to an unknown ground truth image u^* perturbed with an additional Gaussian noise ω. From the previous examples we can take:

u^* $\qquad$ $f = u^* + \omega$

and the objective is to denoise the available image f and recover an image as close as possible to u^*.

2.1 Heat equation

The first PDE that have been used in image processing is the Heat equation that realizes a spatial diffusion of the gray values of a given image. It is a parabolic equation that reads:

$$\begin{cases} \frac{\partial u(t,x)}{\partial t} &= \Delta u(t,x) \quad \text{for } t \geq 0 \text{ and } x \in \Omega \\ u(0,x) &= f(x) \quad \text{for } x \in \Omega \\ \frac{\partial u(t,x)}{\partial N} &= 0 \quad \text{for } t > 0 \text{ and } x \in \partial\Omega, \end{cases} \tag{2.1}$$

where Δ is the Laplacian operator, f is the initial temporal condition and $\frac{\partial u}{\partial N} = 0$ are Neumann boundary conditions. This model diffuses the initial condition f (that can be seen as an initial temperature) along time.

The introduction of this equation comes from the following remark. If f, the initial condition, is smooth enough, then the explicit solution of (2.1) is given by:

$$u(t,x) = \int_{\mathbb{R}^2} G_{\sqrt{2t}}(x-y) f(y)\, dy = \left(G_{\sqrt{2t}} * u_0\right)(x) \tag{2.2}$$

where G_σ is a Gaussian kernel of dimension 2:

$$G_\sigma(x) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{|x|^2}{2\sigma^2}\right) \tag{2.3}$$

The convolution of a data f with a positive kernel is a basic operation in image processing that corresponds to a low-pass filter that will "kill" high frequencies of f and thus remove its noise.

2.2 From continuous to discrete

In image processing, we only have access to discrete values $u(i,j)$ that are located at pixels $x = (i,j)$, $i = 1 \cdots m$, $j = 1 \cdots n$, on the discrete grid describing the image domain Ω.

Discrete spatial operators A discrete image is thus a matrix of dimension $m \times n$. We denote as X the euclidean space $\mathbb{R}^{m \times n}$, and $Y = X \times X$. The space X is equipped with the scalar product:

$$\langle u, v \rangle_X = \sum_{1 \leq i,j \leq N} u_{i,j} v_{i,j} \tag{2.4}$$

and the norm:

$$\|u\|_X = \sqrt{\langle u, u \rangle_X}. \tag{2.5}$$

If $u \in X$, the gradient $\nabla u = [\partial_x u; \partial_y u]^T$ is a vector of Y given by :

$$(\nabla u)_{i,j} = ((\nabla u)^1_{i,j}, (\nabla u)^2_{i,j}) \tag{2.6}$$

where the horizontal gradient can be discretized as

$$(\nabla u)^1_{i,j} = \begin{cases} u_{i+1,j} - u_{i,j} & \text{si } i < M \\ 0 & \text{si } i = M \end{cases} \tag{2.7}$$

and the vertical gradient as:

$$(\nabla u)^2_{i,j} = \begin{cases} u_{i,j+1} - u_{i,j} & \text{si } j < N \\ 0 & \text{si } j = N \end{cases} \tag{2.8}$$

These schemes correspond to the general forward discretization $\partial_x u(i,j) = \frac{u(i+h,j) - u(i,j)}{h}$ with a spatial step of $h = 1$.

We also introduce a discrete version of the divergence operator $\operatorname{div}(p) = \partial_x p_1 + \partial_y p_2$ for a vector $p = [p^1, p^2] \in Y$. It is defined in analogy with the continuous case as:

$$\operatorname{div} = -\nabla^* \tag{2.9}$$

where ∇^* is the adjoint operator of ∇ : i.e., for all $p \in Y$ and $u \in X$, $\langle -\operatorname{div} p, u \rangle_X = \langle p, \nabla u \rangle_Y$. One can then show that:

$$(\operatorname{div}(p))_{i,j} = \begin{cases} p^1_{i,j} - p^1_{i-1,j} & \text{si } 1 < i < M \\ p^1_{i,j} & \text{si i=1} \\ -p^1_{i-1,j} & \text{si i=M} \end{cases} + \begin{cases} p^2_{i,j} - p^2_{i,j-1} & \text{si } 1 < j < N \\ p^2_{i,j} & \text{si j=1} \\ -p^2_{i,j-1} & \text{si j=N} \end{cases} \tag{2.10}$$

Hence, the discretization of the Laplacian operator $\Delta u = \partial_{xx} u + \partial_{yy} u$ for $u \in X$ will be given as:

$$\Delta u = \operatorname{div} \nabla u. \tag{2.11}$$

One can check that combining (2.7), (2.8) and (2.10), we recover the standard Laplacian discretization for $1 < i < m$ and $1 < j < n$:

$$\Delta u^n_{i,j} = u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} - 4u^n_{i,j} \tag{2.12}$$

The functions gradx.m , grady.m and div.m computing these operators are available here: gradx.m | grady.m | div.m.

Boundary conditions As we consider a bounded open set Ω, adequate boundary conditions are required. Neumann conditions are the natural conditions that arise in image processing.

Nevertheless from the above discretizations of the gradient and divergence operators that ensures that $\langle p, \nabla u \rangle_Y = \langle -\mathrm{div}\, p, u \rangle_X$ for all $p \in Y$ and $u \in X$, we have the Neumann conditions for free from the Theorem of Stokes. ***As a consequence, we will not have to deal with the conditions*** $\frac{\partial u(t,x)}{\partial N} = 0$ ***in the implementation.***

Temporal scheme Let us now detail how discretizating in time the PDE (2.1). We denote as δt the numerical time step and $u_{i,j}^k$ represents the value of the image at time $t = k\delta_t$. We consider an explicit Euler scheme of order 1 in time, which leads to the following approximation of the temporal partial derivative:

$$\frac{\partial u_{i,j}^{k+1}}{\partial t} = \frac{u_{i,j}^{k+1} - u_{i,j}^k}{\delta t}. \tag{2.13}$$

Gathering all previous information, the discretization of problem (2.1) finally reads:

$$\begin{cases} u_{i,j}^{k+1} &= u^k + \delta_t (\Delta u^k)_{ij} \\ u_{ij}^0 &= f_{ij}, \end{cases} \tag{2.14}$$

where, from relation (2.11), the discretization of the Laplacian is obtained with schemes (2.7), (2.8) and (2.10).

2.3 Time to work:

Heat equation: The objective is now to solve numerically the problem (2.14).

1 Write a function *heat_ equation* that

   - takes as argument the noisy image f, a time step δ_t and a number of iterations K.
   - solves problem (2.14) for iterations $k = 1 \cdots K$ (the values u^{k+1} should erase the temporary values u^k to save memory space).
   - returns u^K.

2 Write a script that test this function on a noisy image f for different evolution times K. The time step can be set to $\delta_t = 1/8$ to ensure the stability of the scheme.

   How does the optimal evolution time K_{opt} evolves with respect to the amount of noise ?

One can display u^k along iterations to visualize the diffusion process (i.e. the evolution of the solution):

```
imagesc(u);
colormap gray;
drawnow;  %to have a real time display
```

Depending on the evolution time K, we observe images u^K that are more or less smooth:


$K = 0$ $(u^0 = f)$


$K = 20$


$K = 100$

Convolution with a Gaussan kernel: We now check that (2.2) is indeed a solution of (2.1).

3 Define a matrix G of size $(2P-1, 2P-1)$ representing a Gaussian kernel of standard deviation $\sigma^2 = 2K\delta_t$. One can take for instance $P = \lfloor K\delta_t + 1 \rfloor$. Two options are possible to define this kernel:

- Hard one: using (2.3) (in which the index $x = (0,0)$ will correspond to the position $G(P,P)$).
- Esay one: using *fspecial* with options *'gaussian'*, size $(2P-1, 2P-1)$ and standard deviation $\sqrt{2K\delta_t}$.

4 Compute a convolution of f with G with the function *imfilter* and the option *'replicate'* to mimic Neumann conditions.

5 Compare the result with the one obtained with the Heat equation.

Application to contour detection: PDEs can be used as pre-processing tasks for other applications, such as contour detection that consists in finding the significant contours present in an image. A main ingredient of contour detectors is to look for pixels (i,j) for which the gradient norm $||\nabla u(i,j)||$ is large. The detection of significant contours of an image it then more robust when the image is smooth enough.

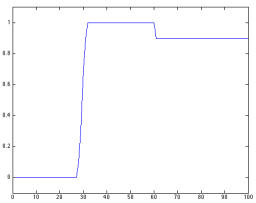
We illustrate this fact with the Marr-Hildreth contour detector that estimates that a pixel belongs to a significant contour if 2 conditions are met:

(1) $||\nabla u(i,j)|| > \eta > 0$, where η is a threshold that selects pixels of high gradient. A small parameters involves the selection of too many pixel (and bold contours), whereas a large one will only select a few one. Considering only this criteria does not give accurate detection as the threshold is hard to tune.

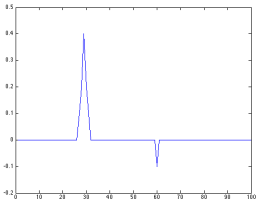
(2) $\Delta u(i,j)$ changes its sign at location (i,j). This means that the pixel (i,j) is a local extrema of $\nabla u(i,j)$. This criteria will detect thin contours but will be sensible to noise.

We now given a $1D$ illustration of this algorithm. In the next Figure, we consider a line of the image (a) denoted as u and shown in (b). The gradient ∇u and the Laplacian Δu are illustrated respectively in (c) and (d).

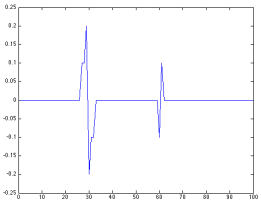
(a) *Image*



(b) u

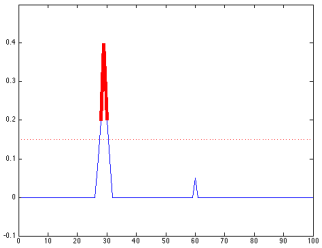


(c) ∇u

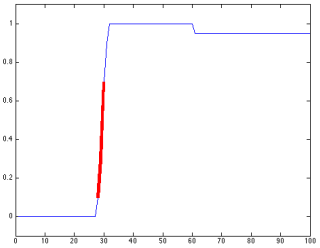


(d) Δu

Let us now observe the effect of condition (1) on ∇u. When taking (a) a threshold of $\eta = 0.15$, we see that (b) the condition $||\nabla u(i,j)|| > \eta$ find the main contour of the image but it involves a bold contour in the final detection (c).



(a) $||\nabla u|| > \eta = 0.15$

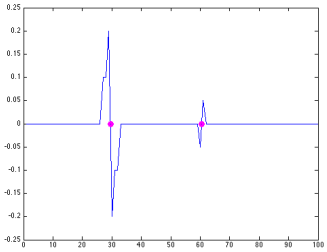


(b) Contours in u

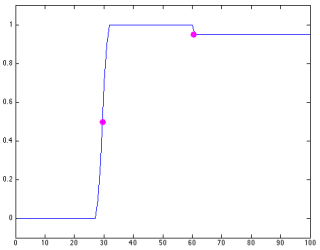
(b) Contour in the image

Illustration of condition (1)

We now detail the effect of condition (2) on Δu. When looking (a) at pixels (i,j) where Δu changes its sign, we see that (b) the local extrema of ∇u are found but it involves the detection of non significant contours in the final detection (c).



(a) Δu changes its sign



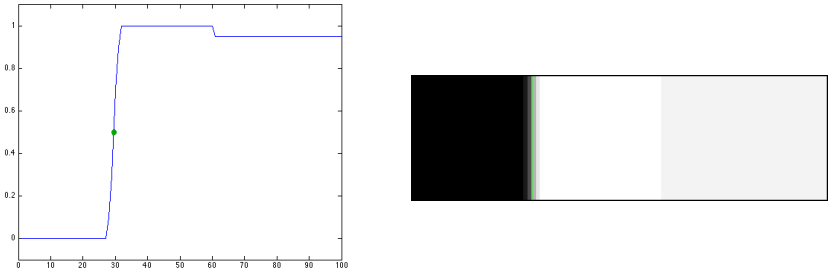
(b) Contours in u

(b) Contour in the image

Illustration of condition (2)

Combining the two criteria, we see below that the algorithm only selects the pixel with a large enough gradient norm that is also a local extrema. This gives an accurate contour detection (b).

6 Write a function *grad_edge* that takes as input an image u and a threshold $\eta > 0$ and returns the contour image that contains the value 1 for pixels checking condition (1) and 0 otherwise.



(a) Contours in u (b) Contour in the image

Illustration of conditions (1) + (2)

Write a function *lap_edge* that takes as input an image u and returns the contour image that contains the value 1 for pixels checking condition (2) and 0 otherwise.

Write a function *Marr_Hildreth* that takes as input an image u and a threshold $\eta > 0$ and returns the contour image that contains the value 1 for pixels checking conditions (1) and (2) and 0 otherwise. The function detecting change of sign can be found here: change_sign.m.

7 Apply these functions to the noisy image f and to the denoised one u^K. Comments ?

We illustrate the influence of the threshold η and the smoothness of the image on the contour detection in the following Figure. When the image is noisy (first line), the detector finds a lot of pixels with a large gradient that are local extrema. When smoothing the image (lines 2 and 3), the noise is removed. However when the smoothing of the image is too important (line 3), the contours are also smoothed in the detection.

Notice that this algorithm is basic and more evolved detectors exist, see for instance the MATLAB's function *edge* and its options.

The Heat equation which is equivalent to the convolution with a Gaussian Kernel, is isotropic, meaning that no particular direction are considered during the diffusion. This is an issue for denoising and contour detection, since one would like to preserve the significant image discontinuities present in the original image.

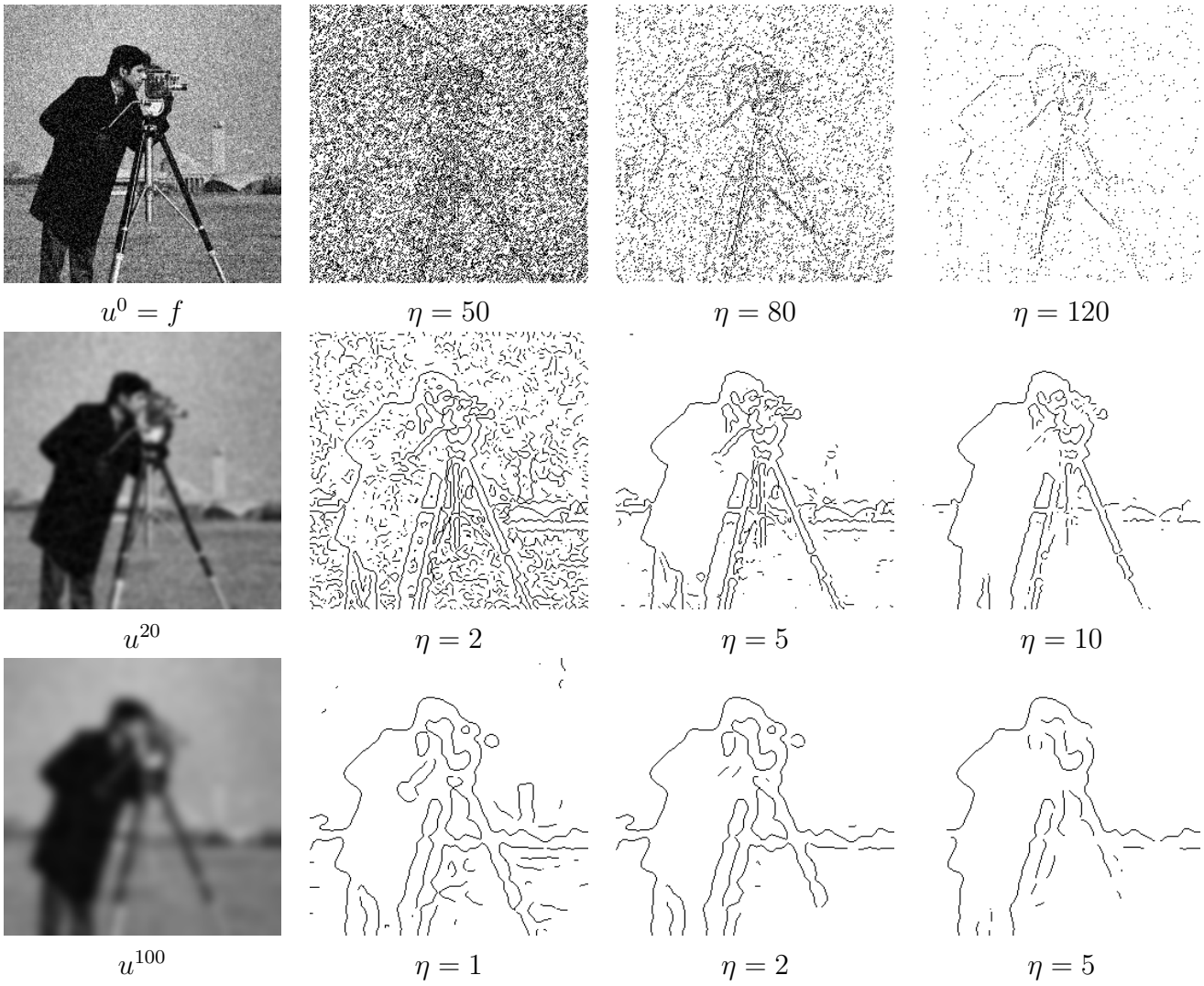
2.4 Perona-Malik

In order to enhance the results obtained with the Heat equation, Perona and Malik proposed to modify the equation by integrating information related to the presence of boundaries:

$$\begin{cases} \frac{\partial u(t,x)}{\partial t} & = \text{div}\left(g\big(||\nabla u(t,x)||\big)\nabla u(t,x)\right) & \text{for } t \geq 0 \text{ and } x \in \Omega \\ u(0,x) & = f(x) & \text{for } x \in \Omega \\ \frac{\partial u(t,x)}{\partial N} & = 0 & \text{for } t > 0 \text{ and } x \in \partial\Omega, \end{cases} \tag{2.15}$$

where g is a decreasing function from $\mathbb{R}_+$ to $\mathbb{R}_+$.

Remark: If taking a constant function $g = 1$, we recover the ***isotropic*** Heat equation.



The function g is classically taken such that $g(0) = 1$ and $\lim_{\xi \to +\infty} g(\xi) = 0$, with for instance $g(\xi) = exp(-\xi^2/\alpha^2)$. Hence, in an uniform (i.e constant) area where the gradient of a pixel (i, j) is small ($||\nabla u(i, j)|| \approx 0$), the Perona-Malik model acts like an isotropic diffusion with $g = 1$ at this pixel. On the other hand, when the gradient is large ($||\nabla u(i, j)|| >> 0$), the diffusion is stopped with $g = 0$ at this location of high gradient, which allows a better preservation of contours. The Perona-Malik PDE is thus an ***anisotropic diffusion*** since specific directions are discouraged. In the following we will consider

$$g(\xi) = \frac{1}{\sqrt{(\xi/\alpha)^2 + 1}}. \tag{2.16}$$

8 Write a function *Perona_ Malik* that

- takes as argument the noisy image f, a time step δ_t, a number of iterations K and a parameter α
- solves problem (2.15), with the temporal derivative (2.13) and the function g defined in (2.16), for iterations $k = 1 \cdots K$
- returns u^K.

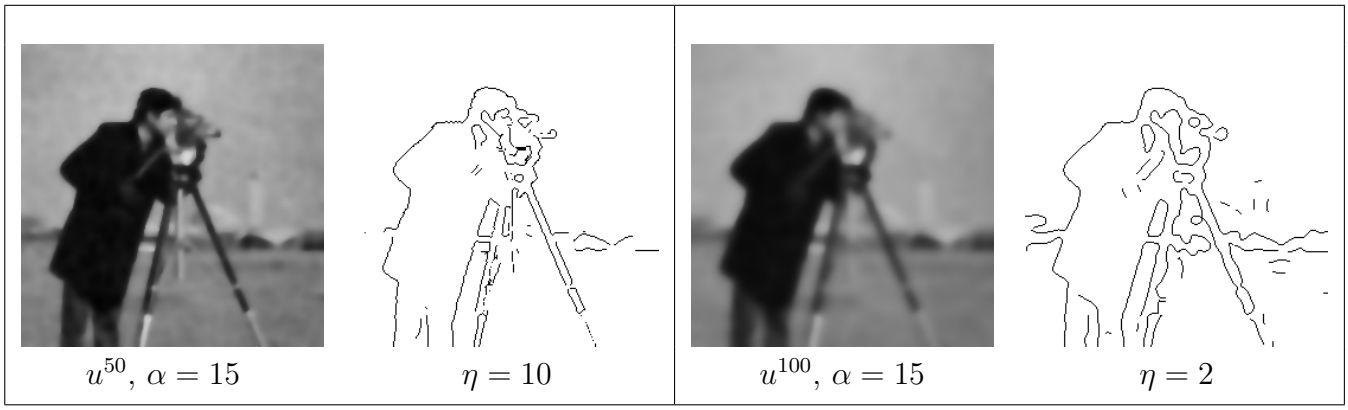
9 Write a script that test this function on a noisy image f for different evolution times K and parameters α.

How does the optimal evolution time K_{opt} evolves with respect to the amount of noise ?

How the optimal parameter α_{opt} evolves with respect to the amount of noise ?

10 Apply the Marr Hildreth contour detector on the obtained results. Commets ?

As shown in the next figure, the Perona-Malik PDE better preserves the discontinuities in the final images u^K, which involves more accurate contour detections.



u^{50}, $\alpha = 15$ $\eta = 10$ u^{100}, $\alpha = 15$ $\eta = 2$

Enhacement with a convolution of the gradient with a Gaussian As previously observed, whene the original image f is very noisy, its gradients present strong oscillations that perturb the diffusion. The noise may indeed be considered as a significant contour. E A simple approach to circumvent this issue is to smooth the gradient that is send to the function c. The improved model then reads:

$$\begin{cases} \frac{\partial u(t,x)}{\partial t} & = \text{div}\left(g(||\nabla(G_\sigma * u(t,x))||)\nabla u(t,x)\right) & \text{for } t \geq 0 \text{ and } x \in \Omega \\ u(0,x) & = f(x) & \text{for } x \in \Omega \\ \frac{\partial u(t,x)}{\partial N} & = 0 & \text{for } t > 0 \text{ and } x \in \partial\Omega. \end{cases} \tag{2.17}$$

11 Implement this model and compare it with the previous one. Comments ? How it the σ parameter chosen ?

With this convolution, the main drawbacks of the previously studied PDEs are still present. Namely, tuning the number of iterations is a hard task. Doing few iterations does not denoise the image, whereas iterating a lot oversmooth the image. The main problem comes from the fact that the initial condition f is "lost" during the diffusion. We will see in the next practice how to re-inject such information into the process.