Université de Nice - Sophia Antipolis

Master IGMMV
Image et Géométrie pour le Multimédia
et la Modélisation du Vivant

Biologically plausible computation mechanisms in cortical areas

Sandrine Chemla
Encadrement: Thierry Viéville and Pierre Kornprobst

INRIA Sophia Antipolis, Odyssée project

Mars - Octobre 2006

Résumé

Nous nous intéressons à l'implémentation de mécanismes de calcul biologiquement plausibles au sein des aires corticales. Plus précisément, les cartes corticales codent des quantités vectorielles, calculées par les réseaux de neurones. En vision par ordinateur, des quantités similaires sont calculées efficacement en utilisant une implémentation d'équations aux dérivées partielles qui définissent des processus de régularisation, permettant d'obtenir une estimation bien définie de ces quantités. Ce mécanisme peut être relié au calcul effectué au sein d'une colonne corticale du cerveau et induit alors un modèle biologiquement plausible de calcul au sein des cartes corticales.
Ce travail s'inscrit dans le cadre du projet FACET et de ce fait sera rédigé en langue anglaise.

Abstract

We are interested in an implementation of biologically plausible computation mechanisms in cortical areas. More precisely, cortical maps code vectorial parametric quantities, computed by networks of neurons. In computer vision, similar quantities are efficiently calculated using partial differential equations implementations which define regularization process, allowing to obtain a well-defined estimation of these quantities. As such it may be linked to what is processed in a cortical column of the brain and provides a biological plausible model of cortical maps computation. This work is realized within the scope of the FACETS project, and then will be written in English language.

Contents

Introduction

The Odyssee research team is supported jointly by INRIA, the Ecole Normale Suprieure in Paris and the Ecole Nationale des Ponts et Chausses (CERTIS laboratory).

This group has three main areas of research:

- Variational methods and partial differential equations for vision
- Functional imaging for observing brain activity
- Modelling brain activity

The present project is part of the first and third areas with a main add-on, the introduction of **neural networks for computation mechanisms in cortical areas**. Indeed, in computer or biological vision [25, 5], computation of vectorial maps of parametric quantities (e.g. features parameters, 3D or motion cues) are of common use in perceptual processes. Defining them using continuous partial differential equations yields highly parallelizable regularization processes allowing to obtain well-defined estimations of these quantities [9, 13]. However, these equations have to be sampled on real data. Thus, an integral approximation of the diffusion operator used in regularization mechanisms have been introduced: it leads to a so-called particles methods [11, 33] which will be describe in this report. A step further, it appears that this method is also a biologically plausible approach when modelling networks of neural units.

We start in the first part by introducing artificial and biological neural networks. In part two, we propose a model of one cortical map which from a continuous variational formulation allows to pass to the related discrete neural network. Parts three and four describe the automatic implementation, using symbolic derivations (`Maple` specifications), of this above framework in an image processing application, with some illustrations. The key contribution, here, is to improve these derivations including non-linear constraints. Perspectives and future works are discussed in the conclusion.

Chapter 1

Neural Networks(NN)

1.1 Definitions

At a very general level, a neural network is an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on biological neuron. The processing ability of the network is stored in the inter-unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns.

Furthermore, neural networks are very powerful computational tools (see e.g. [28]). Neural networks are also very sophisticated modeling techniques capable of modeling extremely complex functions. In particular, neural networks can be considered as non-linear function approximating tool, where parameters of the networks should be found by applying optimization methods. Furthermore, neural networks are very easy to use.

1.2 Biological neural networks

The human brain contains about 10 billion nerve cells, or neurons (see Fig. 1.1), massively interconnected (with an average of several thousand interconnects per neuron). Please refer to [4] for details, not reviewed here.

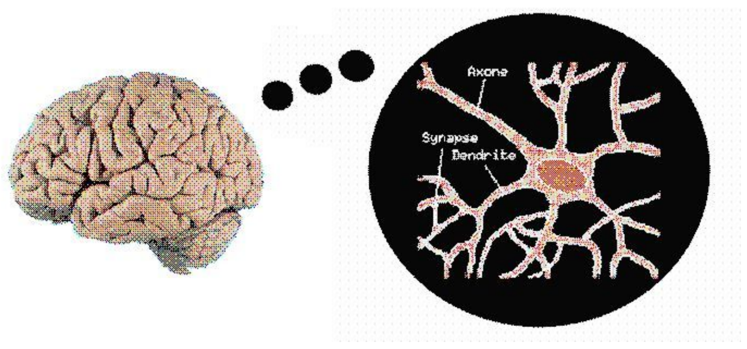


Figure 1.1: Brain and Neural Networks.

Each neuron is a specialized cell which can propagate an electrochemical signal. The neuron has a branching input structure (the dendrites), a cell body, and a branching output structure (the axon).

The axon of one cell connects to the dendrites of another via a synapse. When a neuron is activated, it fires an electrochemical signal (action potential or spike train) along the axon. This signal crosses the synapses to other neurons, which may in turn fire. A neuron fires only if the total signal received at the cell body from the dendrites exceeds a certain level (the firing threshold) (see [10] for a treatise of the subject). In this model, for each neuron, the main state variables are:

- its membrane potential
- its firing probability, spatially computed on a group of neurons, which function is identical (see the definition of a cortical hypercolumn in the next section).

1.3 Artificial neurons modelling

To capture the essence of biological neuron systems, an artificial neural network try to imitate the working mechanisms of their biological counterpart. McCulloch and Pitts first presented in 1943, a simple model of one artificial neuron (see Fig. 1.2) derived from the analysis of the biological reality. Neurons work by processing information. They receive and provide information in form of spikes [5].

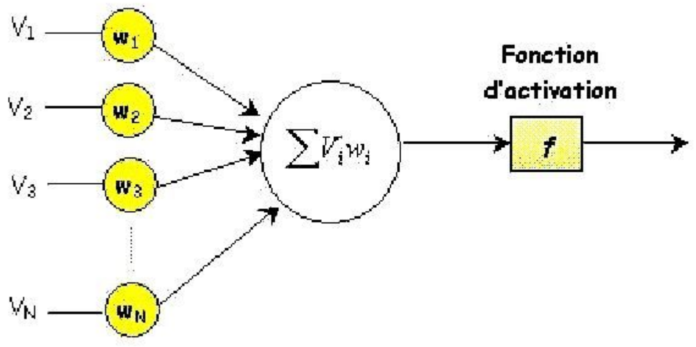


Figure 1.2: McCulloch and Pitts model of one artificial neuron(1943).

In this model, the spikes information is represented by spike rates and synaptic strength are translated as synaptic weights; these weights correspond to synaptic efficacy in a biological neuron. Each neuron also has a single threshold value. The weighted sum of the inputs is estimated, and the threshold subtracted, to compose the activation of the neuron. The activation signal is passed through an activation function to produce the output of the neuron. From this description of an individual neuron, at a higher scale, we can study an artificial neural network (see Fig. 1.3). This last one is composed of many artificial neurons that are linked together according to specific network architecture.

Let us consider aa more biologically plausible model.

1.4 Specification of a Hopfield neural network

A good compromise between the very different families of networks is the biological analog neural network model often referred to as the generalized Hopfield network [20]. Following, e.g. [13] (in which the context is reviewed), let us consider the following discrete dynamical system defined on a set of M signals which dynamics is of the form:

$$\dot{u}_i = -\xi_i \, u_i + \sum_j \sigma_{ij} \, v_j + \sum_k \kappa_{ik} \, w_k \text{ with } v_i = g(u_i - \theta_i) \tag{1.1}$$

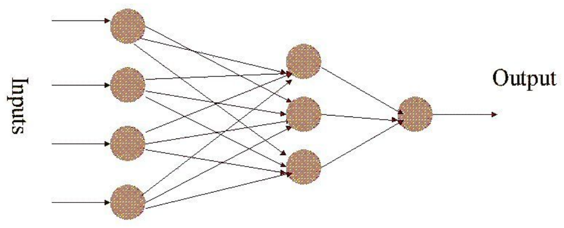


Figure 1.3: An artificial neural network.

where :

- $u_i = u(\mathbf{x}_i) \in \mathbb{R}$ represents the membrane potential of the neuron of index i at location $\mathbf{x}_i$ in the network,
- $\xi_i > 0$ is a "leakage" rate (also called resistive coefficient),
- σ_{ij} is the synaptic efficiency modulating the effect of the jth neuron on the ith neuron,
- $v_j \in [0, 1]$ represents the neuron average firing rate probability, the spatial average being taken over a set of neurons in a similar state [12] (originally, a neuronal network à-la Hopfield also called McCulloch-Pitts network is defined on a set of binary signals $v_j \in \{0, 1\}$),
- κ_{ik} is the synaptic efficiency modulating the effect the kth input on the ith neuron
- w_k represents the network kth input value
- $g : \mathbb{R} \to [0, 1]$ is a "sigmoid" profile, i.e. with $0 < g'(x) \leq g'(0) = \frac{1}{T}$ where T is the so-called stochastic temperature, as schematized in Fig.1.4.
- θ_i is the firing rate threshold. Without loss of generality we can state $\theta_i = 0$ using the following change of variables: $u_i \to u_i + \theta_i$ adding an input of the form $\kappa_{i0}\, w_0 = \xi_i\, \theta_i$.

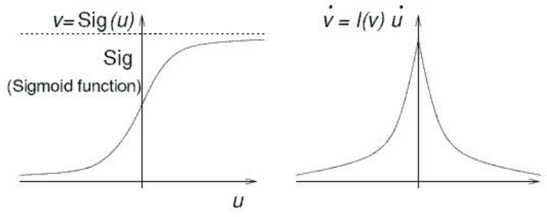


Figure 1.4: In analog networks a sigmoidal non-linearity (denoted by Sig) is introduced between the neuronal state $\mathbf{u} \in \mathbb{R}^N$ (usually related to the membrane potential) and the neuronal output $\mathbf{v} \in [0, 1]^N$ (usually related to the average firing rate probability).

This corresponds to a Hypercolumn simple model of one artificial neuron (see Fig. 1.3) derived from the analysis of the biological reality. Neurons work by processing information. They receive and provide information in form of spikes [5], [21], as detailed now.

Once all the parameters of such a network well-defined, we will show in the next part that neural network could be a natural base for building partial differential equations.

Chapter 2

Modelling Cortical Maps

2.1 Partial Differential Equation (PDE) and NN analogy

We revisit here the links between (i) high level specification of how the brain represents and categorizes the causes of its sensory input and (ii) related analog or spiking (will be studied later) neural networks [18, 21].

In computer vision, variational approaches yield to efficient computations (e.g. to resolve image processing problems) using implementation of regularization processes which allows to obtain well-defined and powerful estimations (see e.g. [33]). More precisely, a general class of cortical map computations can be specified representing what is to be done (perceptual task) as an optimization problem (variational formulation) with regularization term (implemented using so-called partial differential equations (PDE)). More precisely, the goal is to minimize an energy functional, via Euler-Lagrange equations:

$$\bar{\mathbf{v}} = \underset{\mathbf{v} \in H / \mathbf{c}(\mathbf{v})=0}{argmin} \; \mathcal{L}(\mathbf{v}), \tag{2.1}$$

with:

$$\mathcal{L}(\mathbf{v}) = \int_{\Omega} f(\mathbf{v}, \frac{\partial \mathbf{v}}{\partial}, \frac{\partial^2 \mathbf{v}}{\partial^2}). \tag{2.2}$$

Then, we want to link this formulation with neural network calculation (how to do it), in order to obtain a biologically plausible representation of such mechanisms. As such, this is a challenging issue. Indeed, even if the variational formulation is a well-founded framework, the necessity to discretize Euler equations is still an open issue in this context.
Thus, to build this link, we would like to implement the previous energy functional as a network of units, which correspond to cortex units called "Cortical Hypercolumns", and then derive from the well-known continuous formalism to a discrete one based on neural networks.

Let us show how to derive from this continuous variational formulation a discrete neural network of the previous form (1.1).

2.2 What is a Cortical Hypercolumn

By definition, a cortical hypercolumn (see Fig. 2.1) is a group of neurons in the brain cortex which related neuronal processing is homogenous. Such neural units are clearly identified by electrophysiology [5, 21].

Each column corresponds to a different sensori-motor or cognitive processing unit. They have a nearly identical structure everywhere within the cortex. Such cortical hypercolumn is composed of 6 layers; each layer has a specific property with respect to receiving and sending signals locally or to different brain regions [29].

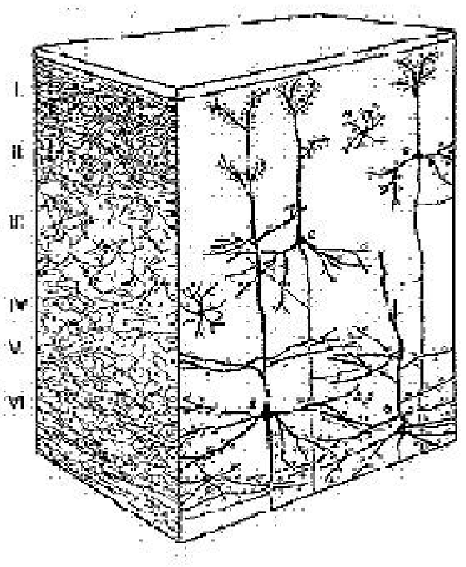


Figure 2.1: A cortical hypercolumn.

These columns, seen as neuronal unit, are assembled within areas, classified with respect to the nature of the treated information [20].

2.3 Cortical Hypercolumn modelling

2.3.1 Variational Formulation

The idea behind regularization with variational methods is as follows: suppress low image variations mainly due to noise, while preserving the high ones representing the image features. Regularizing images can be done by minimizing energy functionals measuring the global image variations (see [2] for a treatise on the subject) .

Moreover, according to generative approaches [25, 18, 10], a cortical hypercolumn can be also modeled as an optimization problem by a variational formulation (VF), more precisely:

$$\bar{\mathbf{v}} = \underset{\mathbf{v} \in H / \mathbf{c}(\mathbf{v})=0}{argmin} \; \mathcal{L}(\mathbf{v}), \quad \text{with} \tag{2.3}$$

$$\mathcal{L}(\mathbf{v}) = \int |\hat{\mathbf{w}} - \mathbf{w}|^2_{\mathbf{\Lambda}} + \int \phi(|\nabla \mathbf{v}|_{\mathbf{L}}) + 2 \int \psi(\mathbf{v}), \tag{2.4}$$

$$\text{and } \hat{\mathbf{w}} = \mathbf{P}\,\mathbf{v}, \tag{2.5}$$

where w and v respectively represents the input and the output vectors. The first term is a data fidelity term which specifies how the output is related to the input, the second term is a regularization term which defines the regularity of the output and the third term allows to control the form of the solution. The last equation shows the chosen relation between the estimation of the input given an output, while c(v) = 0 allows to constrain the solution. So the formulation specifies the cortical map computation (see Fig. 2.2) in the sense that it explains the goal (what is to be done), but without any reference to how it is done.

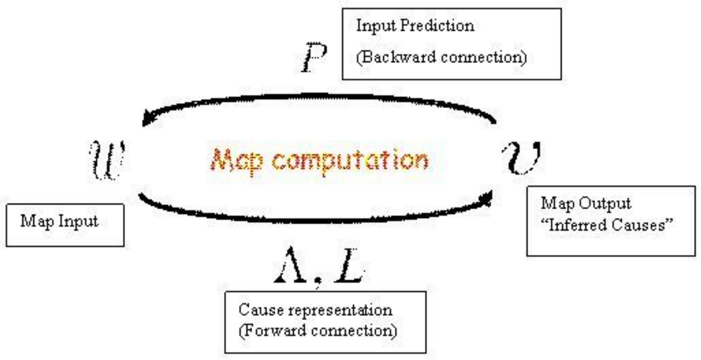


Figure 2.2: Cortical Map Computation

2.3.2 Neural Network formalism

The solution of (2.3)–(2.5) can be implemented using a discrete neural network (see Appendix A.2 for details [34]). More precisely, our optimization problem is, in the general case, locally minimized by the following linearized differential equation:

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\epsilon_i(\mathbf{v}_i) + \sum_j \sigma_{ij}(\mathbf{v}_i)\, \mathbf{v}_j + \kappa_i\, \mathbf{w}_i \tag{2.6}$$

with:

$$\left\{ \begin{array}{rcl} \epsilon_i(v) & = & \rho_i\, \mathbf{v} + \xi\, \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c} + \frac{\partial \psi}{\partial \mathbf{v}}^T, \\ \rho_i & = & \sum_j \sigma_{ij} + \mathbf{P}^T\, \mathbf{\Lambda}_i\, \mathbf{P}, \\ \kappa_i & = & \mathbf{P}^T\, \mathbf{\Lambda}_i, \end{array} \right. \tag{2.7}$$

and $\xi = (1-\lambda)\, |\frac{\partial \mathcal{L}}{\partial \mathbf{v}}| / |\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}|$ with $\lambda \ll 1$. The weights $\sigma = (\sigma_{ij})$ are given by considering the linearized optimal integral approximation up to order r ($r \geq 2$) of the non-linear diffusion operator (see section 2.3.3)

$$\bar{\mathbf{L}} = \phi'(|\nabla \mathbf{v}|_{\mathbf{L}})\, \mathbf{L}, \tag{2.8}$$

defined at M points, providing $M > \frac{(n+r)!}{n!\, r!} - \frac{n\,(n+1)}{2}$. They are given by solving the system (see Appendix A.3 [34]):

$$\begin{array}{rcl} \bar{\mathbf{L}}_{kl}(\mathbf{x}) & = & \frac{1}{2} \sum_j \sigma_{ij}\, \bar{\mu}_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}), \\ \mathbf{div}_k(\bar{\mathbf{L}}(\mathbf{x_i})) & = & \sum_j \sigma_{ij}\, \bar{\mu}_j^{\mathbf{e}_k}(\mathbf{x}), \end{array} \tag{2.9}$$

where [1]*:* $\bar{\mu}_j^{\alpha}(\mathbf{x}) = \int_{\mathcal{S}_j} (\mathbf{y}-\mathbf{x})^{\alpha}\, \mu_j(\mathbf{y})\, d\mathbf{y}$*, while:*

$$\forall i, \quad \sum_j \sigma_{ij}\, \bar{\mu}_j^{\alpha}(\mathbf{x}) = 0 \quad 2 < |\alpha| \le r \tag{2.10}$$

Among all σ_{ij} *verifying* (2.9) *and* (2.10) *we choose those which verify:*

$$min \sum_{ij} |\sigma_{ij}^2|. \tag{2.11}$$

2.3.3 Integral approximation of the diffusion operator

The main difficulty in this framework is the discrete implementation of the anisotropic diffusion operator L defining the regularization mechanism. Other terms derivations are straight forward. Indeed, in practice, it is not possible to implement a differential operator such as this 1st and 2nd order diffusion operator without choosing an approximation and a numerical scheme with the risk of loosing what was well-defined in the continuous case. Here biological assumptions allow to properly specify this process. Integral approximation of differential operators have been introduced in the field of neural networks by Cottet et al. [7, 8, 9, 13] and presented in the present form in [33], where the derivation of the present proposition is available. In particular, the present form provides an alternative to the use of so-called particles methods (e.g. [11]) which neural interpretation is weaker.

2.3.4 Temporal discretization

In this section, we do not consider the constraint c, thus the term $\xi \, \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}$ is null. See section 4 for the introduction of the constraint c. Let us rewrite (2.6) at time $t^n = n\,\Delta t$, i.e. at the iteration $n \in \{0..N\}$

$$\frac{\partial \mathbf{v}_i^n}{\partial t} = -\mathbf{A}(\mathbf{v}_i^n)\, \mathbf{v}_i^n + \mathbf{b}(\cdots \mathbf{v}_j^n \cdots, \mathbf{w}_i^n) \tag{2.12}$$

with

$$\left\{ \begin{array}{lclclcl} \mathbf{b}^n & = & \mathbf{b}(\cdots \mathbf{v}_j^n \cdots, \mathbf{w}_i^n) & = & \mathbf{P}^T\, \mathbf{\Lambda}_i\, \mathbf{w}_i^n & + & \sum_j \sigma_{ij}(\mathbf{v}_i^n)\, \mathbf{v}_j^n - \nabla\, \psi \\ \mathbf{A}^n & = & \mathbf{A}(\mathbf{v}_i^n) & = & \mathbf{P}^T\, \mathbf{\Lambda}_i\, \mathbf{P} & + & \sum_j \sigma_{ij} \end{array} \right. \tag{2.13}$$

while $\mathbf{A}^n > 0$ is symmetric.

The general form of the convergent discrete numerical Euler schema is of the form:

$$\mathbf{v}_i^{n+1} = \alpha^n\, \mathbf{v}_i^n + \beta^n\, \frac{\partial \mathbf{v}_i^n}{\partial t} \text{ with } \left\{ \begin{array}{lcl} \alpha^n & \rightarrow & 1 \\ \beta^n & > & 0 \end{array} \right.$$

while β^n is small enough in order

$$\epsilon = ||\mathbf{v}_i^{n+1} - \alpha^n\, \mathbf{v}_i^n|| = \beta^n || \frac{\partial \mathbf{v}_i^n}{\partial t} || \rightarrow 0$$

with iterations, yielding $\frac{\partial \mathbf{v}_i^n}{\partial t} = 0$ at the convergence. In our case, let us choose:

$$\alpha^n = 1 \text{ and } \beta^n = \upsilon\, (\mathbf{A}^n)^{-1} \text{ with } 1 > \upsilon \in \mathbb{R}^+$$

leading to the following numerical schema:

$$\mathbf{v}_i^{n+1} = (1-\upsilon)\, \mathbf{v}_i^n + \upsilon \left[(\mathbf{A}^n)^{-1}\, \mathbf{b}^n \right] \tag{2.14}$$

[1]We use the standart multi-index notation, for vector of integer indices $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathcal{N}^n$ and $|\alpha| = \alpha_1 + \ldots + \alpha_n$ and $\mathbf{x}^{\alpha} = x_1^{\alpha_1} \ldots x_n^{\alpha_n}$

where υ could be calculated by e.g. the following heuristic and :

$$\left\{ \begin{array}{l} \upsilon = 1/2 \\ \text{if } \epsilon^n < \epsilon^{n-1} \text{ then } \upsilon = \sqrt{\upsilon} \text{ else } \upsilon = \upsilon/2 \text{ and } \mathbf{v}^n = \mathbf{v}^{n-1} \end{array} \right. \tag{2.15}$$

i.e. :
(i) if υ is small enough to guaranty the convergence, it is increased (using, say, $\sqrt{}$) to attempt to speed up the process
(ii) else if convergence fails, υ is decreased (say by a factor of two) and the iteration is re-runned

Chapter 3

Implementation, first application and results

3.1 Implementation

Once this formalism has been stated, the next goal is then to automatically generate the linearized differential equation of a cortical map computation defined by a variational criterion, and to implement an automatic computation of the neural network parameters from the parameters of the energy functional, in order to experiment it, for instance, on image processing application. The general principle is represented in Figure 3.1.

Here we use the symbolic language `Maple` for our modified Edwards equations computation and for the PDE implementation. Thus, we obtain an automatic computation map generator independent of the application (Appendix B.1.1).

A step further, we can apply this mechanism on any image processing application, only by changing the different parameters (Appendix B.1.2 and B.1.3). Here, the critical part of the Java code (Appendix B.2.1 and Appendix B.2.2) is generated by the `Maple` computations (see Appendix B.3 describing the makefile).

3.2 Isotropic Filtering on colour image

We first chose to realize a simple isotropic filtering on colour images as a verification of the method. Isotropic regularization is a natural way to smooth and simplify data and has consequently been reached by several mathematical formulations. Here, we consider a vectorial signal.

In the present case, the energy functional to minimize is simplified. The input w and output v are three dimensional vectors, and we consider the linear relation between them as being the identity. Here, $\mathbf{P}$ is the identity function, so is L the kronecker four order double symmetric tensor and there is no constraint c and no control function Ψ. Thus the energy functional is reduced to the following well-known form:

$$\mathcal{L}(\mathbf{v}) = \int |\hat{\mathbf{w}} - \mathbf{w}|^2 + \int |\nabla \mathbf{v}|^2, \tag{3.1}$$

with an data attachment term and a regularization term. The minimum of such a functional is known: a constant image. Nevertheless, the important point we emphasize here is the behaviour of gradient de-

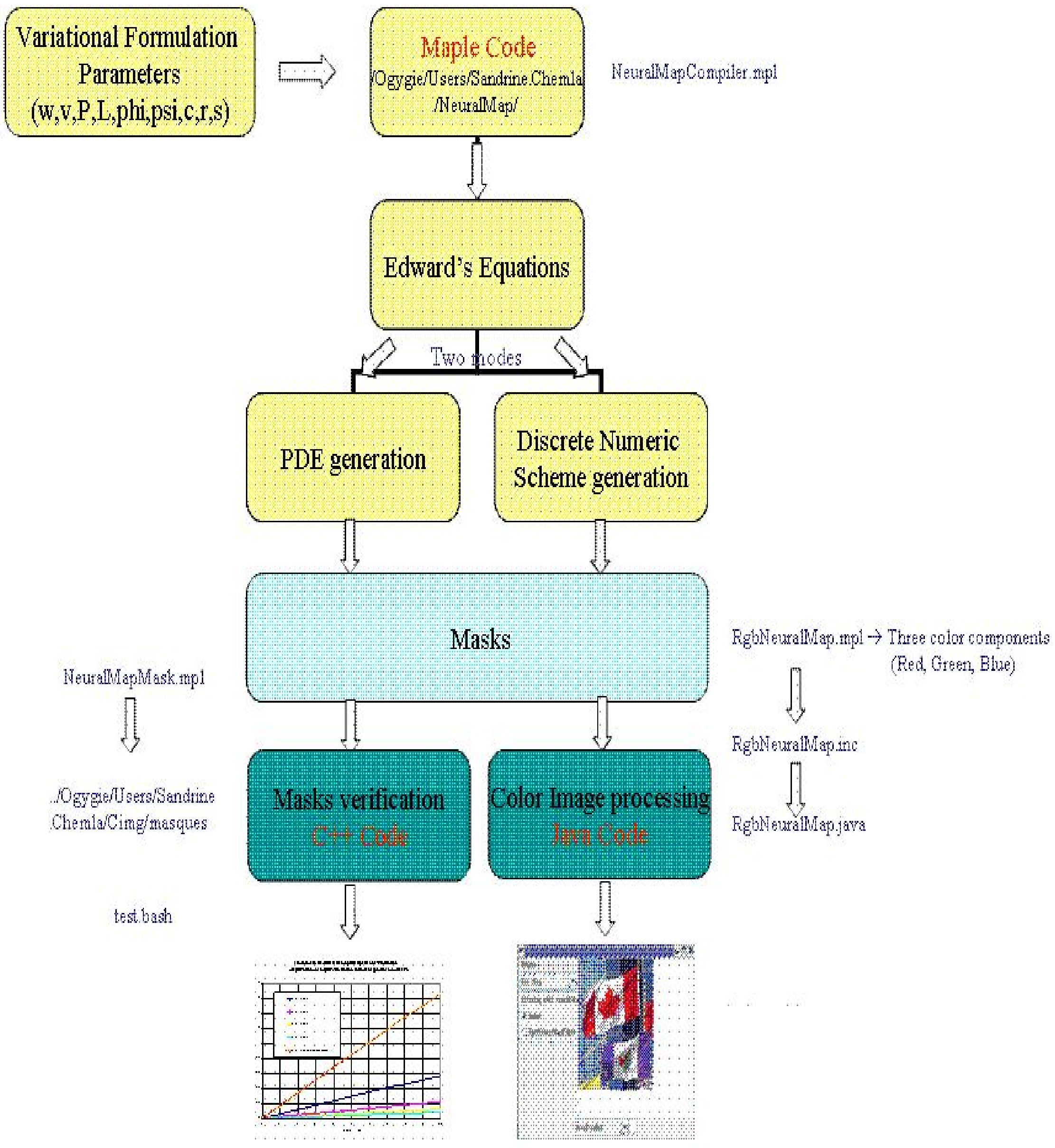

Variational Formulation Parameters (w,v,P,L,phi,psi,c,r,s)
Maple Code
/Ogygie/Users/Sandrine.Chemla/NeuralMap/
NeuralMapCompiler.mpl
Edward's Equations
Two modes
PDE generation
Discrete Numeric Scheme generation
Masks
RgbNeuralMap.mpl → Three color components (Red, Green, Blue)
NeuralMapMask.mpl
RgbNeuralMap.inc
../Ogygie/Users/Sandrine Chemla/Cimg/masques
Masks verification
C++ Code
Color Image processing
Java Code
RgbNeuralMap.java
test.bash

Figure 3.1: The present code generator general principle.

scent of the regularization term alone. Using the Euler-Lagrange equations gives the following gradient descent (PDE or diffusion equation):

$$\frac{\partial \mathbf{v}}{\partial \mathbf{t}} = \Delta \mathbf{v}.$$

Now, we can verify that our neural networks implementation give us the same result. The basic discretization of the Laplacian operator, as an approximate Dirac derivative convolution, is obtained by convolution with a mask of the following form:

$$\mathbf{M} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

while the rotation invariant operator, used for comparison in this section part since it is better than the previous one, is:

$$\mathbf{M} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

As expecting, after calculating the update linearized differential equation (r = 2 and s = 1), with the `maple` tool, we obtained the following mask, which roughly corresponds to the previous one:

$$\mathbf{M} = \begin{pmatrix} \frac{2}{5} & \frac{1}{5} & \frac{2}{5} \\ \frac{1}{5} & -\frac{12}{5} & \frac{1}{5} \\ \frac{2}{5} & \frac{1}{5} & \frac{2}{5} \end{pmatrix}$$

with somehow surprisingly different coefficients increasing at the periphery. This is coherent with what has been obtained previously (see [33] for details) and according to the formalism, it is expected to improves the signal/noise ratio.

3.3 Verification on synthetic images

In order to test our Laplacian masks and to verify their validity, we explore different values of both the neighbourhood s and the integral approximation order r, and we test the related masks on two profiles whose Laplacian is well known: the Gaussian function and the Heaviside function. Figure 3.2 shows the two input stimuli chosen.

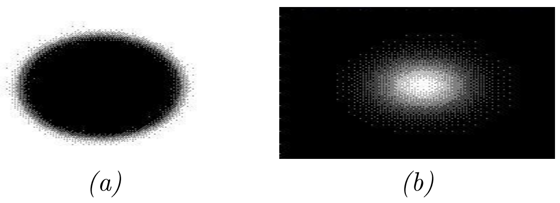


Figure 3.2: Input stimuli: (a) Heaviside, (b) Gaussian.

We only discuss the Gaussian stimuli, since the process is the same for the Heaviside one. The main scheme of our analysis is summed up in Figure 3.7, i.e. comparison between theoretical Laplacian of the Gaussian stimuli and the image obtained after applying our computed masks (through Maple generation) with Neural Networks. Note that we also add a Gaussian noise to the initial Gaussian kernel, in order to observe the expected the improvement obtained with the present formalism based on Neural Networks.

Figures 3.8, 3.9, 3.10 and 3.11, 3.12, 3.13, 3.14 show the different results. For a given neighbourhood s, the error curves between theoretical and computed Laplacian mask in function of a Gaussian noise while increasing the integral approximation order r, are plotted in Fig. 3.8, 3.9, 3.10. Next, for a given integral approximation order r, the curves while increasing s are shown in Fig. 3.11, 3.12, 3.13, 3.14.

Remark: We obtain very similar curves for the Heaviside stimuli. Thus, our masks do not significantly depend on the nature of the input stimuli For instance, the obtained results, for a given neighbourhood s = 2 while increasing the integral approximation order r, are illustrated Figure 3.6.

The interpretation of these results is the following:

- The masks computed with our formalism are always better than standard Laplacian masks.
- For a given integral approximation order r, the error decreases when the neighbourhood s increases, at any noise level and for both stimuli.
- For a given neighbourhood s, Figures 3.8 and 3.9 show that the error increases when r increases. However, Figure 3.13 shows that this result can be explained by a too restricted neighbourhood since with a higher one (when s = 4), the error starts to decrease. This variation with r has an interesting consequence: there seems to be an optimal order of approximation r given a neighbourhood size s.
- Furthermore, we can notice in Figures 3.8, 3.9, 3.10 that we obtain similar curves for (r=2 or r=3, s=2) and (r=4 or r=5, s=2). The explanation is trivial; the computed masks are symmetric and thus par (parity of the coefficients for the present approximation to be unbiased), so there are no anti-symmetric factor related to the odd degrees monomials in our polynomial development. The two masks are represented in Figures 3.5 and 3.6.

Figure 3.3 schematizes how the higher r, the less residual high-order derivatives bias in the operator diffusion. However, the higher r, the less degree of freedoms to generate an optimal mask.

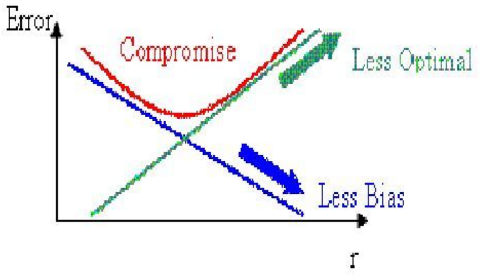


Figure 3.3: Relation between r, the residual bias and the number of freedom's degree.

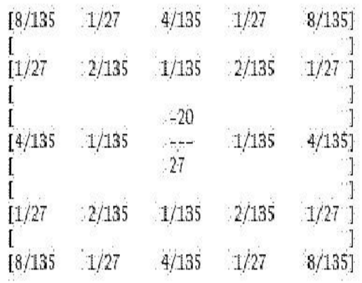
The key experimental result here is the regularity of the phenomenon: for any size s, for both Heaviside or Gaussian stimuli, if we collect all results we obtain the following table (see table 3.4). The maximal order r is calculated from the following inegality (3.2), with n = 2:

$$(2\,s+1)^n > \frac{(n+r)!}{n!\,r!} - \frac{n\,(n-1)}{2}, \tag{3.2}$$

It is clear in table 3.4 that, the maximal order is far from being optimal, and it is very likely that, given any other operator, deriving this optimal values is an open issue.

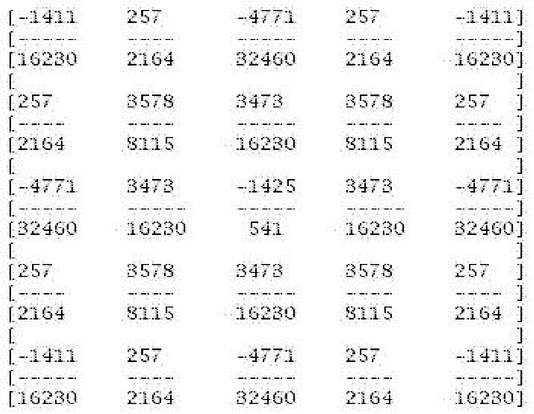
Smallest size s	1	2	3	4	5
Optimal order r	2	2	2	4	4
Maximal order r	3	5	8	11	14

Figure 3.4: Optimal and maximal order r computation in function of the smallest neighborhood size s.



```
[8/135   1/27    4/135   1/27    8/135]
[                                     ]
[1/27    2/135   1/135   2/135   1/27 ]
[                                     ]
[                -20                  ]
[4/135   1/135   ---     1/135   4/135]
[                27                   ]
[                                     ]
[1/27    2/135   1/135   2/135   1/27 ]
[                                     ]
[8/135   1/27    4/135   1/27    8/135]
```

Figure 3.5: Computed Masks:(r=2, s=2) or (r=3, s=2)



```
[-1411     257       -4771     257       -1411]
[-----     ----      -----     ----      -----]
[16230     2164      32460     2164      16230]
[                                             ]
[257       3578      3473      3578      257  ]
[----      ----      -----     ----      ---- ]
[2164      8115      16230     8115      2164 ]
[                                             ]
[-4771     3473      -1425     3473      -4771]
[-----     -----     -----     -----     -----]
[32460     16230      541      16230     32460]
[                                             ]
[257       3578      3473      3578      257  ]
[----      ----      -----     ----      ---- ]
[2164      8115      16230     8115      2164 ]
[                                             ]
[-1411     257       -4771     257       -1411]
[-----     ----      -----     ----      -----]
[16230     2164      32460     2164      16230]
```

Figure 3.6: Computed Masks:(r=4, s=2) or (r=5, s=2)

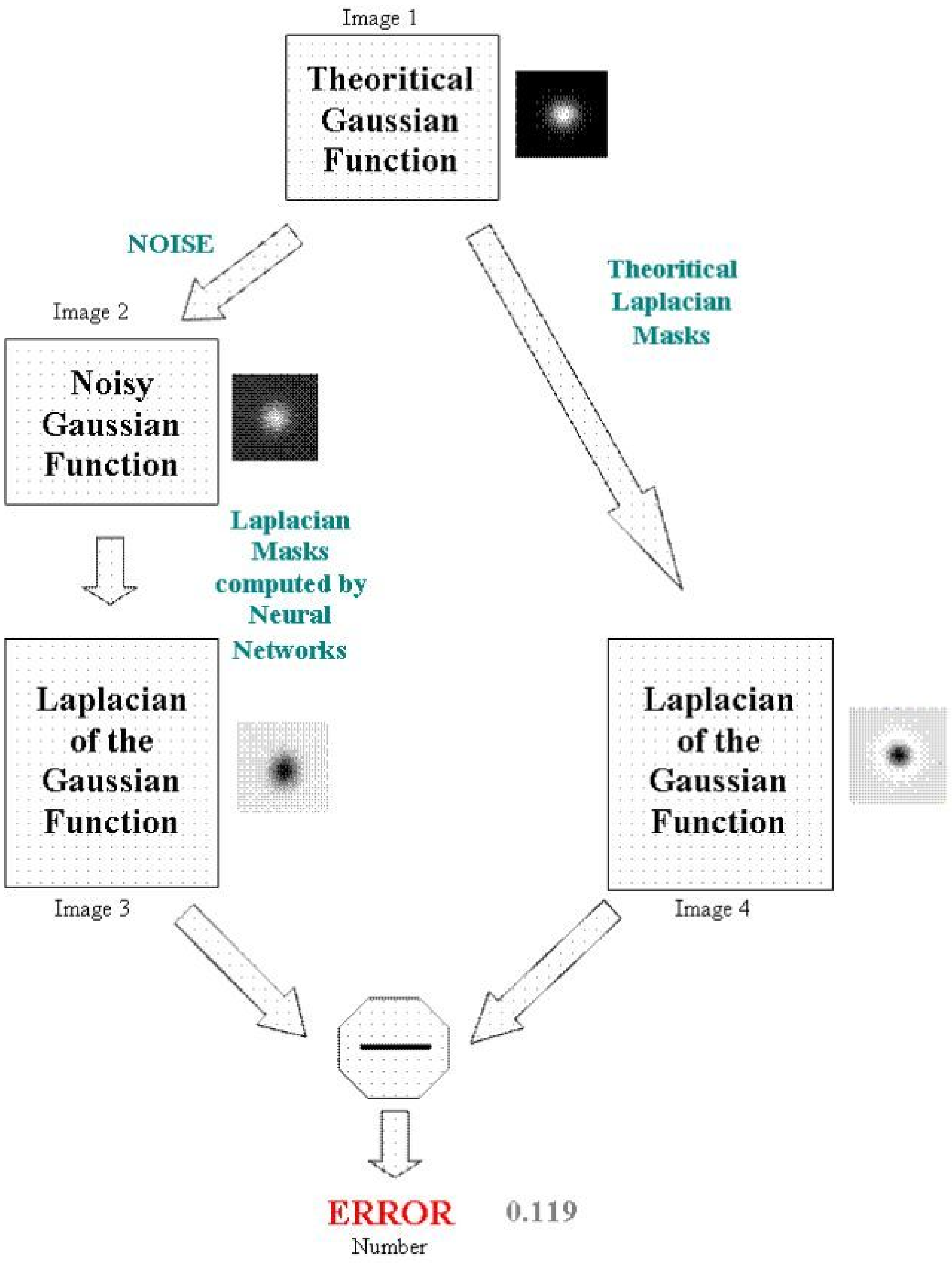


Image 1
Theoritical Gaussian Function
NOISE
Theoritical Laplacian Masks
Image 2
Noisy Gaussian Function
Laplacian Masks computed by Neural Networks
Laplacian of the Gaussian Function
Image 3
Laplacian of the Gaussian Function
Image 4
—
ERROR Number
0.119

Figure 3.7: Scheme of the comparison between theoretical Laplacian of the Gaussian function, while the result is obtained after applying the Maple generated masks using a NN implementation.

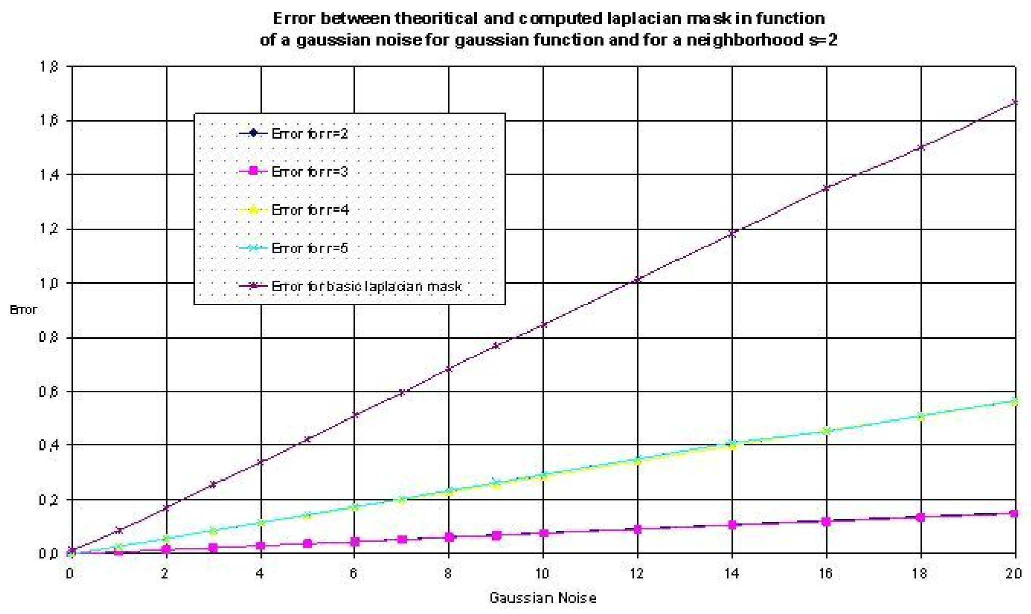


Figure 3.8: Variation of the integral approximation order r when s=2.

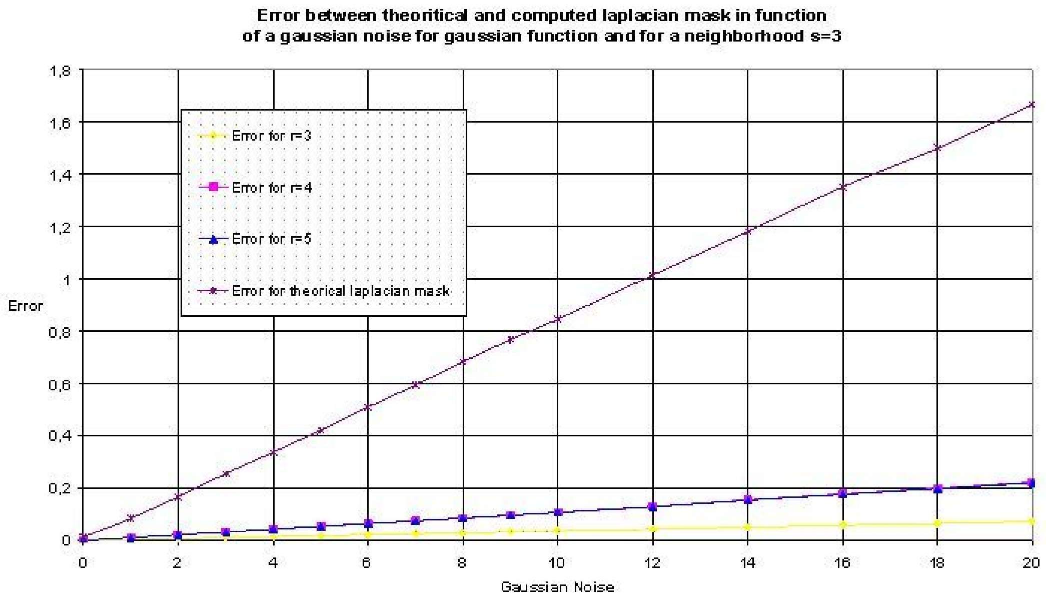


Figure 3.9: Variation of the integral approximation order r when s=3.

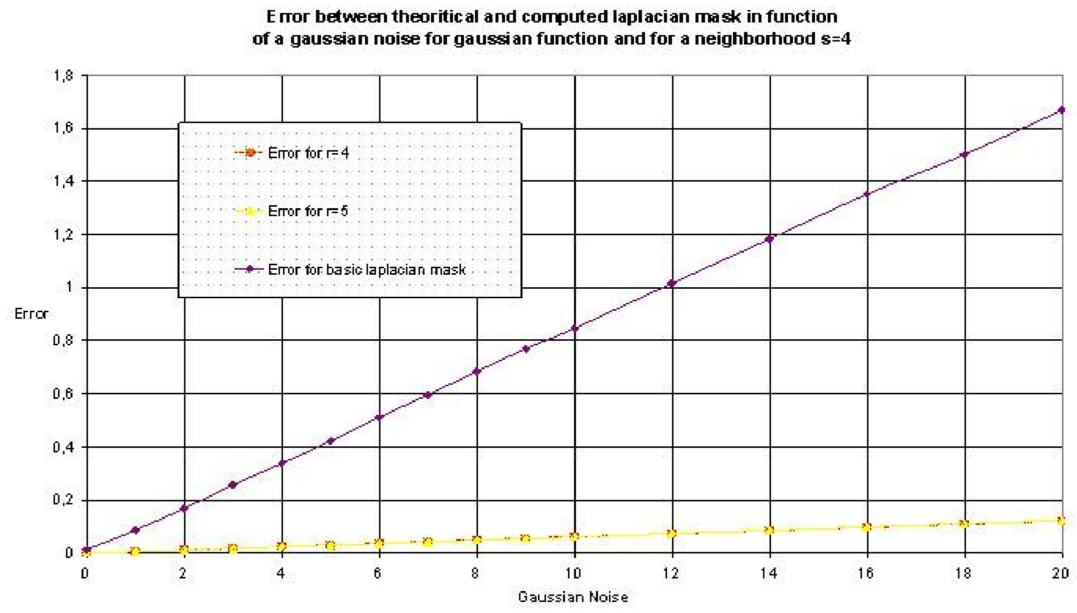


Figure 3.10: Variation of the integral approximation order r when s=4.

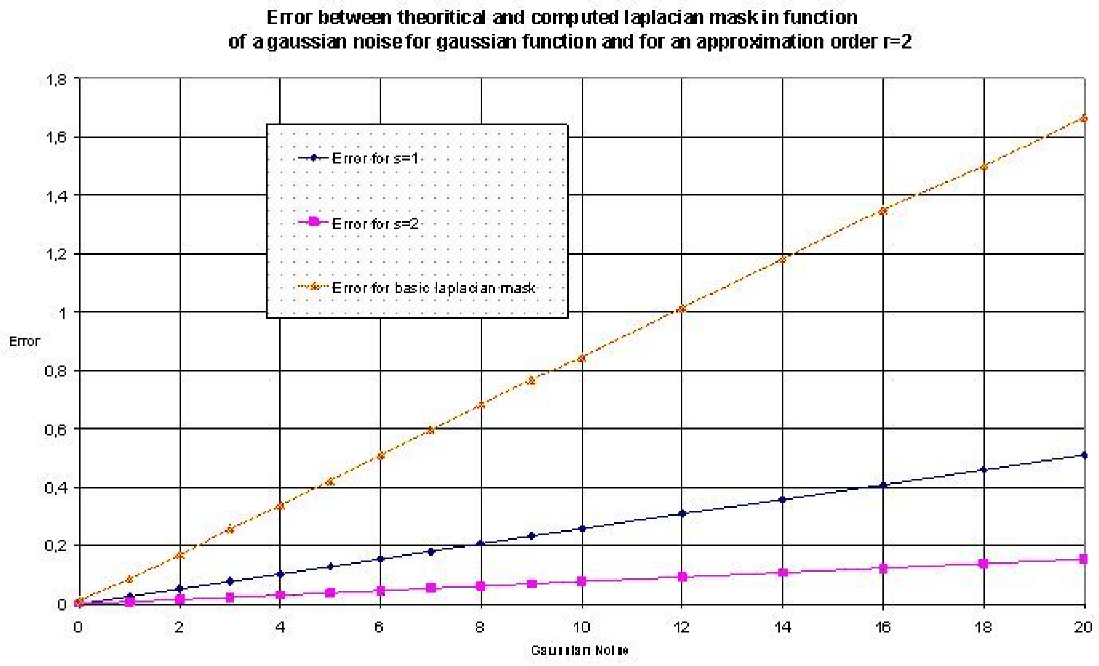


Figure 3.11: Variation of the neighbourhood s when r =2.

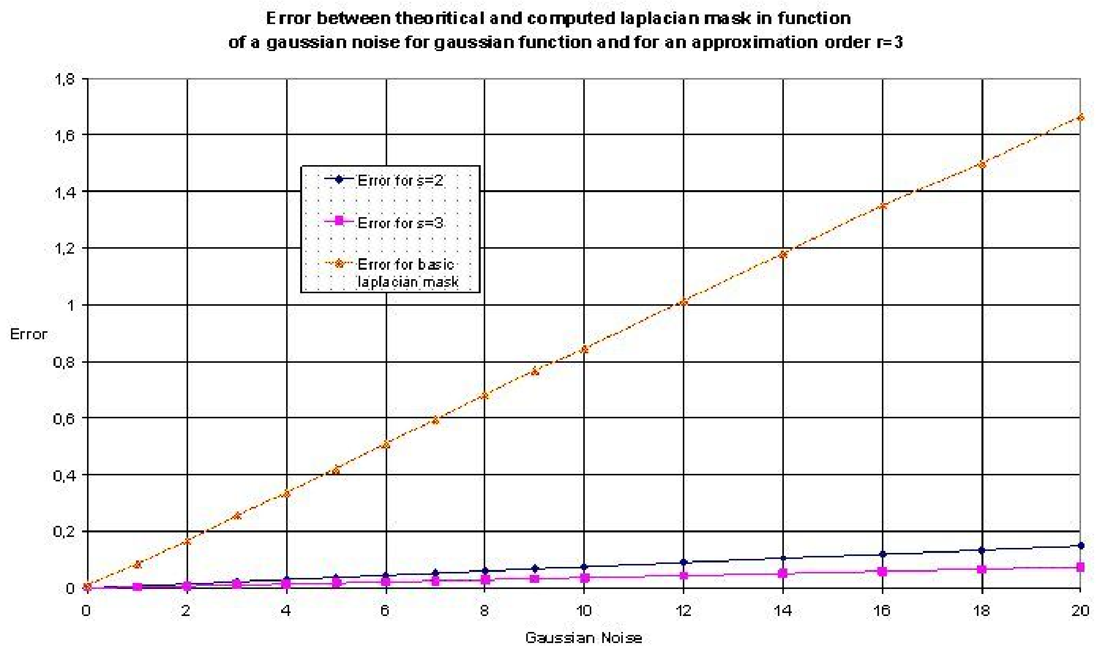


Figure 3.12: Variation of the neighbourhood s when r =3.

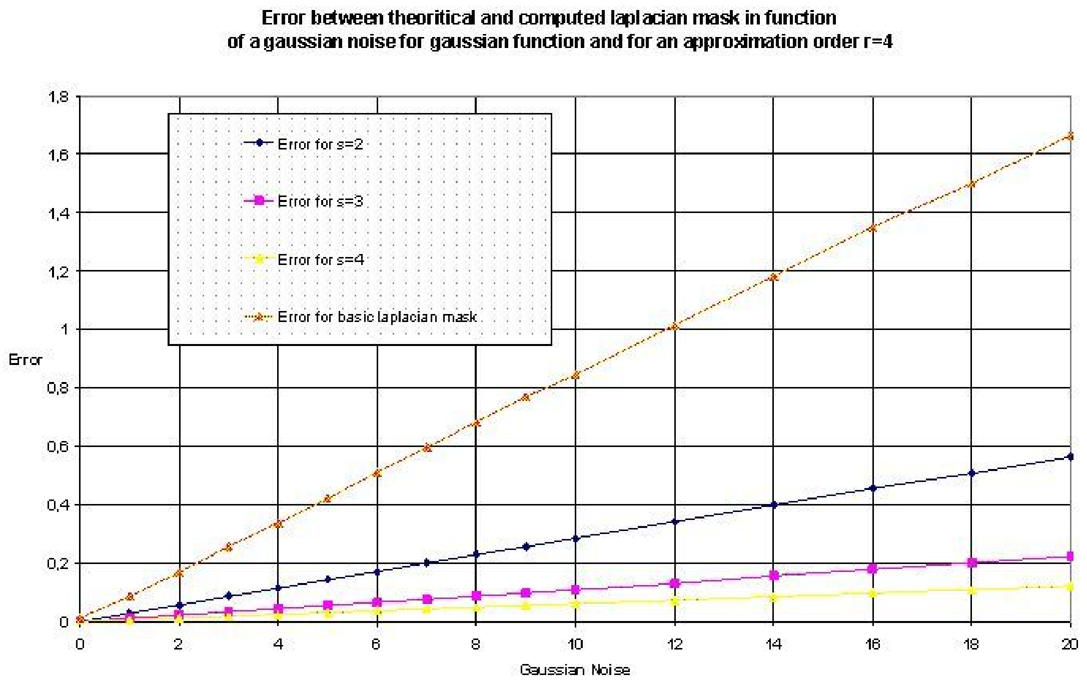


Figure 3.13: Variation of the neighbourhood s when r =4.

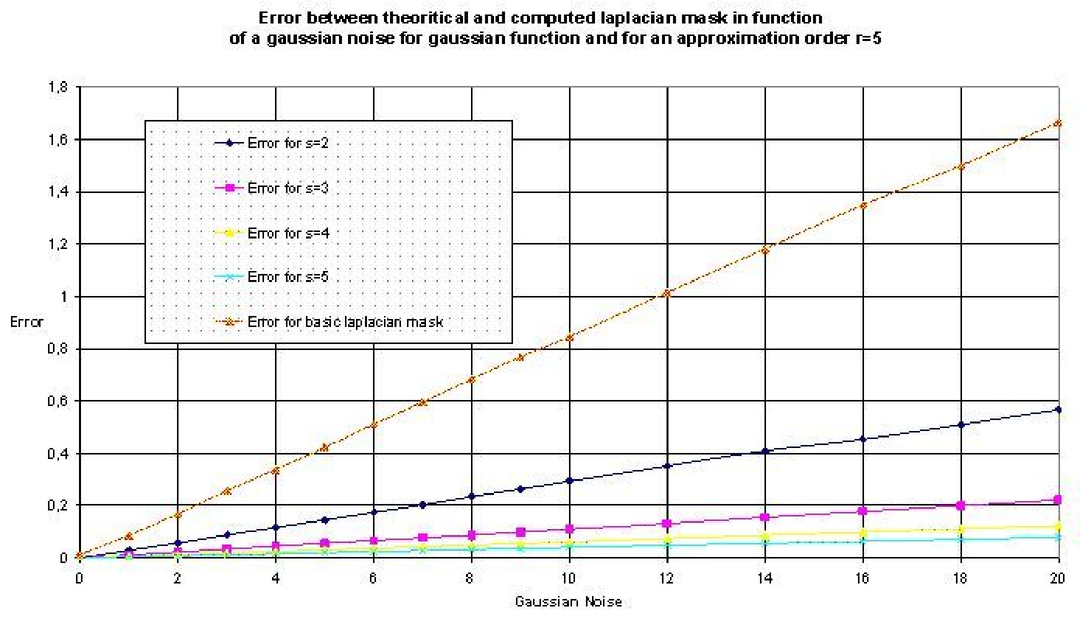


Figure 3.14: Variation of the neighbourhood s when r =5.

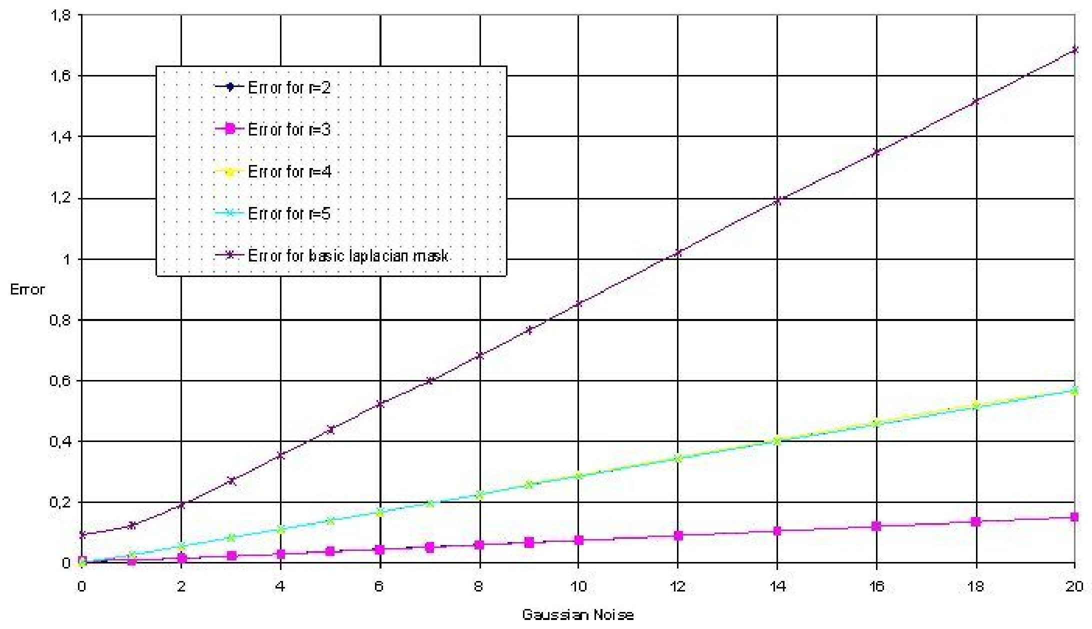


Figure 3.15: Error between theoretical and computed Laplacian masks in function of Gaussian noise for the Heaviside function and for a neighbourhood s=2.

3.4 Experimentation on real images

We also applied our isotropic filter (see Appendix B.2.3 for the filter implementation and Appendix B.2.4 for the applet) on a color image shown in Fig. 3.16, while Fig. 3.17 shows the different results in function of the iteration's number. The image is blurred step by step in an isotropic way during the PDE evolution (iterations), as expected. The isotropic regularization behaves as a low-pass filter suppressing high frequencies in the image, thus does not preserve contours.

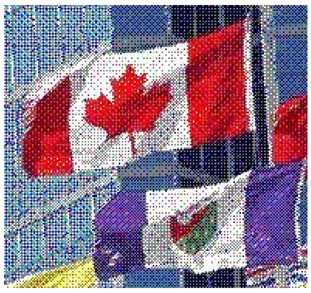


Figure 3.16: Initial colour image.

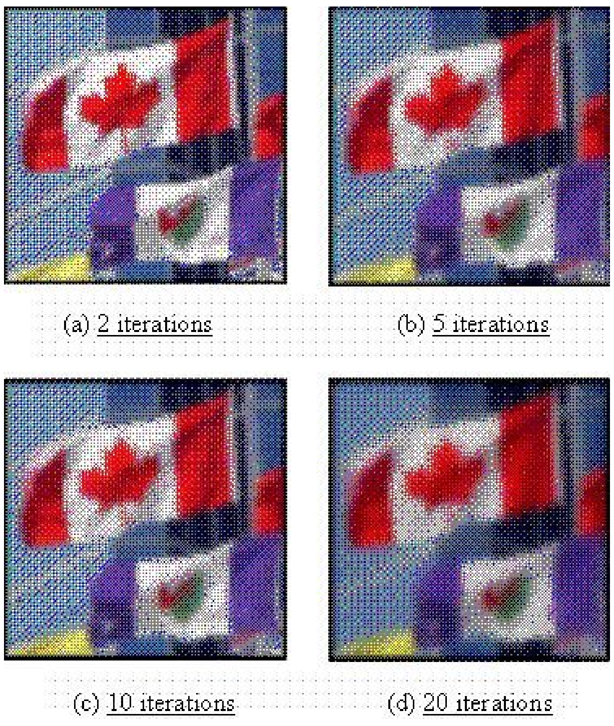


Figure 3.17: Isotropic filtering at different steps.

Chapter 4

Introduction of Constraints

4.1 Abstract

The next step is to generalize these previous derivations including linear and non-linear constraints. Let us rewrite our goal, i.e. the minimization of the following functional:

$$\bar{\mathbf{v}} = \underset{\mathbf{v} \in H / \mathbf{c}(\mathbf{v})=0}{argmin} \; \mathcal{L}(\mathbf{v}), \quad \text{with} \tag{4.1}$$

$$\mathcal{L}(\mathbf{v}) = \int |\mathbf{P}\,\mathbf{v} - \mathbf{w}|^2_{\mathbf{\Lambda}} + \int \phi(|\nabla \mathbf{v}|_{\mathbf{L}}) + 2 \int \psi(\mathbf{v}), \tag{4.2}$$

while:

$$\begin{array}{rcll} \frac{1}{2}\,\frac{\partial \mathcal{L}}{\partial \mathbf{v}}^T & = & \mathbf{P}^T \mathbf{\Lambda}\, [\mathbf{P}\,\mathbf{v} - \mathbf{w}] - \mathbf{div}(\phi'(|\nabla \mathbf{v}|_{\mathbf{L}})\, \mathbf{L} \nabla \mathbf{v}) & + \frac{\partial \psi}{\partial \mathbf{v}} \\ & \simeq & \mathbf{P}^T \mathbf{\Lambda}\, [\mathbf{P}\,\mathbf{v} - \mathbf{w}] - \left[\sum_j \sigma_{ij}(\mathbf{v}_i)\, \mathbf{v}_j - \sum_j \sigma_{ij}(\mathbf{v}_i)\right] & + \frac{\partial \psi}{\partial \mathbf{v}} \end{array}$$

We are going to develop the temporal discretization, for the minimization on the one hand, and for the constraint verification on the other hand.

4.2 Temporal discretization for the minimization

Let us write at time $t^n = n\,\Delta t$, i.e. at the iteration $n \in \{0..N\}$

$$\mathbf{l}^n = \frac{1}{2}\,\frac{\partial \mathcal{L}}{\partial \mathbf{v}}^T = \mathbf{A}^n\,\mathbf{v}^n - \mathbf{b}^n, \tag{4.3}$$

with:

$$\left\{ \begin{array}{rclclcl} \mathbf{b}^n & = & \mathbf{b}(\cdots \mathbf{v}^n_j \cdots, \mathbf{w}^n_i) & = & \mathbf{P}^T \mathbf{\Lambda}_i\, \mathbf{w}^n_i & + & \sum_j \sigma_{ij}\, \mathbf{v}^n_j - \frac{\partial \psi}{\partial \mathbf{v}}, \\ \mathbf{A}^n & = & \mathbf{A}(\mathbf{v}^n_i) & = & \mathbf{P}^T \mathbf{\Lambda}_i\, \mathbf{P} & + & \sum_j \sigma_{ij}, \end{array} \right. \tag{4.4}$$

while $\mathbf{A}^n > 0$ is symmetric.

In order to have up to the 1st order

$$\mathcal{L}(\mathbf{v}^{n+1}) = \mathcal{L}(\mathbf{v}^n) + \frac{\partial \mathcal{L}}{\partial \mathbf{v}}(\mathbf{v}^n)\,[\mathbf{v}^{n+1} - \mathbf{v}^n] + o(|\mathbf{v}^{n+1} - \mathbf{v}^n|^2) < \mathcal{L}(\mathbf{v}^n)$$

we must verify:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}}(\mathbf{v}^n)\,[\mathbf{v}^{n+1} - \mathbf{v}^n] < 0 \tag{4.5}$$

or,

$$-\frac{\partial \mathcal{L}}{\partial \mathbf{v}}(\mathbf{v}^n)\,[\mathbf{v}^{n+1} - \mathbf{v}^n] > 0 \tag{4.6}$$

$$-\mathbf{l}^{nT}\left[\mathbf{v}^{n+1}-\mathbf{v}^{n}\right]=\left[\mathbf{v}^{n+1}-\mathbf{v}^{n}\right]\mathbf{A}^{n}\left[(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}\right]>0 \tag{4.7}$$

thus $\mathbf{v}^{n+1}-\mathbf{v}^{n}$ to be as much as possible aligned with $(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}$ in order the left-hand side (which is a scalar product for the $\mathbf{A}^{n}$ metric) to be maximal,
for a $\mathbf{v}^{n+1}$ sufficiently close to $\mathbf{v}^{n}$ in order the 1st order approximation to be valid.

4.3 Temporal discretization for the constraint verification

In order to verify the constraint (also up to the 1st order) we write:

$$\begin{array}{rcl} 0 & = & \mathbf{c}(\mathbf{v}^{n+1}) \\ & = & \mathbf{c}(\mathbf{v}^{n})+\frac{\partial \mathbf{c}}{\partial \mathbf{v}}(\mathbf{v}^{n})\left[\mathbf{v}^{n+1}-\mathbf{v}^{n}\right]+o(|\mathbf{v}^{n+1}-\mathbf{v}^{n}|^2) \end{array}$$

and writing

$$\mathbf{c}^{n}=\mathbf{c}(\mathbf{v}^{n}) \quad \mathbf{C}^{n}=\frac{\partial \mathbf{c}}{\partial \mathbf{v}}(\mathbf{v}^{n}) \quad \mathbf{Q}^{n}=\mathbf{C}^{nT}\left(\mathbf{C}^{n}\,\mathbf{C}^{nT}\right)^{-1} \quad \mathbf{P}^{n}=\mathbf{Q}^{n}\,\mathbf{C}^{n}$$

as the reader can easily verified, we can choose without loss of generality:

$$\mathbf{v}^{n+1}=\mathbf{v}^{n}-\mathbf{Q}^{n}\,\mathbf{c}^{n}+\left[\mathbf{I}-\mathbf{P}^{n}\right]\mathbf{z} \tag{4.8}$$

for any $\mathbf{z}$, since $\mathbf{c}^{n}+\mathbf{C}^{n}\left[\mathbf{v}^{n+1}-\mathbf{v}^{n}\right]=0$ because $\mathbf{C}^{n}\,\mathbf{Q}^{n}=\mathbf{I}$ and $\mathbf{C}^{n}\left[\mathbf{I}-\mathbf{P}^{n}\right]=0$,
for a $\mathbf{v}^{n+1}$ sufficiently close to $\mathbf{v}^{n}$ in order the 1st order approximation to be valid.

Here $[\mathbf{I}-\mathbf{P}^{n}]$ is the projection onto the $\mathbf{c}(\mathbf{v}^{n})$ level-set tangent space.

In order (4.8) (related to the constraint verification) to be compatible with (4.7) (related to the criterion minimization)[1] we align $\mathbf{v}^{n+1}-\mathbf{v}^{n}$ as much as possible aligned with $(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}$ and choose $\mathbf{z}\equiv(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}$.

4.4 Algorithm

Finally we come up with the following scheme, with $0<\epsilon<1$ an d$0<\upsilon<1$:

$$\mathbf{v}^{n+1}=\mathbf{v}^{n}-\epsilon\,\underbrace{\left[\mathbf{Q}^{n}\,\mathbf{c}^{n}\right]}_{\beta}+\upsilon\,\underbrace{\left[\mathbf{I}-\mathbf{P}^{n}\right]\left[(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}\right]}_{\alpha} \tag{4.9}$$

so that :

- $|\mathbf{c}(\mathbf{v}^{n+1})|<|\mathbf{c}(\mathbf{v}^{n})|$ for some sufficiently small ϵ and $\upsilon=0$ while
- $\mathcal{L}(\mathbf{v}^{n+1})<\mathcal{L}(\mathbf{v}^{n})$ for some sufficiently small ϵ and υ if $|\mathbf{c}(\mathbf{v}^{n+1})|$ is small enough

Without any constraint: $\mathbf{c}^{n}=0$ and $\mathbf{P}^{n}=0$ so that we can choose $\epsilon=0$ and obtain:

$$\mathbf{v}^{n+1}=\mathbf{v}^{n}+\upsilon\left[(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}\right] \tag{4.10}$$

as obtained by another specific method (see 2.14 in 2.3.4).

[1] We must choose $\mathbf{z}$ in a suitable direction with a magnitude:
- small enough for the 1st order approximation to be valid,
- sufficiently high for (4.7) (here $\mathbf{l}^{nT}\,\mathbf{Q}^{n}\,\mathbf{c}^{n}-\mathbf{l}^{nT}\left[\mathbf{I}-\mathbf{P}^{n}\right]\mathbf{z}>0$) to be verified.
In the case where (4.7) is not verified, equation (4.8) allows only to solve the constraint (up to 1st order at each step and iteratively in reasonable cases, see below) without minimizing the criterion. But after some iterations, the constraint is likely solved, i.e. $\mathbf{c}^{n}=\mathbf{c}(\mathbf{v}^{n})\rightarrow 0$ and (4.7) is verified as soon as $\mathbf{l}^{nT}\left[\mathbf{I}-\mathbf{P}^{n}\right]\mathbf{z}<0$ even for small $\mathbf{z}$.
This is exactly what is required here: minimize the criterion in (or close to) the $\mathbf{c}(\mathbf{v})=0$ set.
When $\mathbf{c}^{n}\simeq 0$, $\mathbf{v}^{n+1}-\mathbf{v}^{n}\simeq\left[\mathbf{I}-\mathbf{P}^{n}\right]\mathbf{z}$, i.e. vanishing in the $\mathbf{c}(\mathbf{v})=0$ normal space and unconstrained in the related tangent space. Choosing, as discussed previously, $\mathbf{v}^{n+1}-\mathbf{v}^{n}$ as much as possible aligned with $(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}$ in the tangent space, we come up with $\mathbf{z}\equiv(\mathbf{A}^{n})^{-1}\mathbf{b}^{n}-\mathbf{v}^{n}$.

This algorithm is implemented on the existing `Maple` code with several modifications:

- the previous code is simplified, especially by using a general structure (see "contains" and "append" procedures in Appendix B.1.1 for details):
- the two modes: the unconstrained and the constraint filtering are integrated in the same mechanism of the code generator.
- here, the code generates two java files 'RgbNeuralMap.java' and IsoNeuralMap.java' (Appendix B.2.1 and B.2.2) from the two `Maple` files 'RgbNeuralMap.mpl' and 'IsoNeuralMap.mpl' (Appendix B.1.2 and B.1.3).
- the choice between this different modes is done in the applet file 'RgbNeuralMapApplet.java' (Appendix B.2.4).

4.5 Illustrations

Let us revisit isotropic filtering. However, we add this time a structural constraint $\mathbf{c}(\mathbf{v}) = 0$. More precisely, we choose to introduce the following constraint:

$$\mathbf{c}(\mathbf{v}) = \mathbf{R} + \mathbf{G} + \mathbf{B} - \mathit{constant} = 0 \tag{4.11}$$

Figures 4.1 and 4.2 show the results on our previous images.
In figure 4.1, the result is very clear; the image do not stop to be filtered, but under the constraint R + G + B - constant = 0, which is first verified. Thus, in the case where constant = 255, the blue part remains almost perfectly blue (0, 0, 254) instead of (0, 0, 255) on the initial image, and the dark black part becomes a gray-level part with iso-luminance.

Furthermore, the convergence is very fast (only one iteration to verify the constraint and few iterations for the iso-luminance smoothing) since the constraint is linear. A step further, we will apply non-linear constraint, like $R^2 + G^2 + B^2 - 255^2 = 0$, and compare the convergence speed.

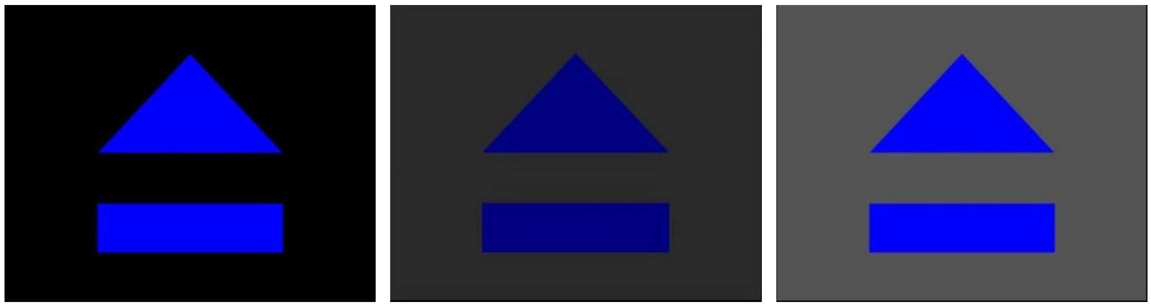


Figure 4.1: Filtering with iso-luminance : R + G + B = constant.

The Java interface is represented in Figures 4.3 and 4.4. The basic previous filtering is activated by clicking on the "RgbNeuralMapFilter" button, while the constraint mode, called "IsoNeuralMapFilter", depends on the choice of the constant (128 or 255).

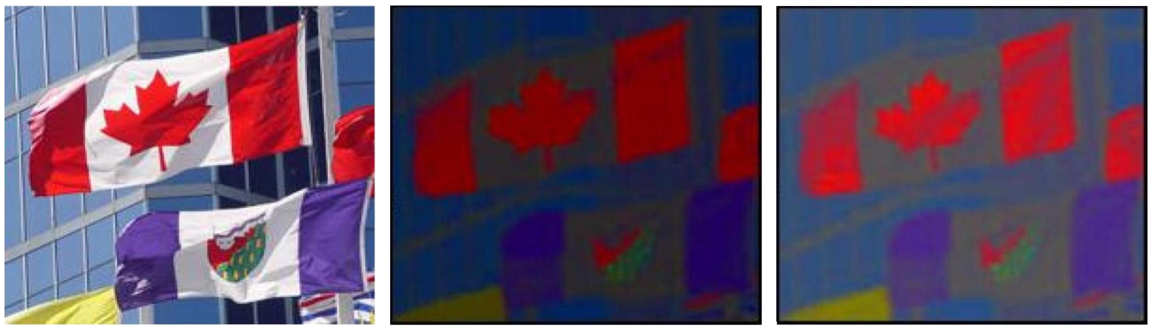


Figure 4.2: Color image filtering with iso-luminance : R + G + B = constant.

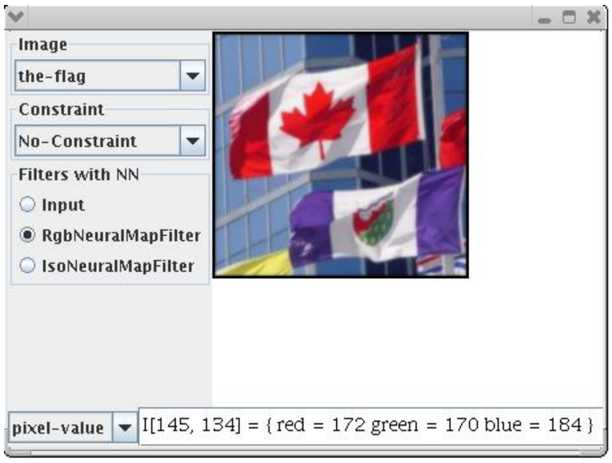


Image: the-flag

Constraint: No-Constraint

Filters with NN: Input [✓ RgbNeuralMapFilter] IsoNeuralMapFilter

pixel-value: I[145, 134] = { red = 172 green = 170 blue = 184 }

Figure 4.3: Java Interface: Color inage filtering

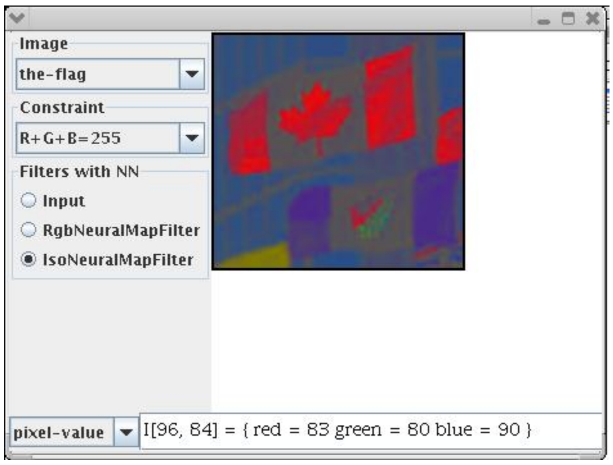


Image: the-flag

Constraint: R+G+B=255

Filters with NN: Input RgbNeuralMapFilter [✓ IsoNeuralMapFilter]

pixel-value: I[96, 84] = { red = 83 green = 80 blue = 90 }

Figure 4.4: Java Interface: Constraint introduction

Chapter 5

Others Results

Our generator is independent of the application. Thus we can also apply our formalism and verify it for anisotropic diffusion and also for some WinnerTakeAll mechanisms. For the moment, our formalism has been verified by hand on these applications. The next step is to use the automatic generator for double verification and improvement.

5.1 Edge-preserving smoothing

Let us revisit an edge-preserving smoothing approach proposed by Cottet and Ayyadi [9] which corresponds to the framework presented in this paper. In [9], given an initial image $w : \mathbb{R}^2 \to \mathbb{R}$, the authors proposed a diffusion processes of the form :

$$\frac{\partial \mathbf{v}}{\partial t} = l(\mathbf{v})\, \Delta_{\mathbf{L}(\mathbf{v})} \mathbf{v}$$

where $l = 1/Sig^{-1}$, which is in fact related to the minimisation of the criterion

$$\bar{v} = \underset{v}{argmin}\ \mathcal{L}(v) = \lambda \int (w - v)^2 + \int |\nabla v|^2_{\mathbf{L}}, \tag{5.1}$$

where λ is a small constant and $\mathbf{L}$ is defined by

$$\mathbf{L} = \left[\rho^2\, \mathbf{P}_{\mathbf{g}^\perp} + \frac{3}{2}\,(1 - \rho^2)\, \mathbf{I} \right] \tag{5.2}$$

$$\text{with} \quad \left\{ \begin{array}{l} \mathbf{g} = S \star \nabla v, \\ \rho = \min\left(1, \frac{|\mathbf{g}|^2}{s^2}\right), \\ \mathbf{P}_{g^\perp} = \left(\begin{array}{cc} g_2^2 & -g_1 g_2 \\ -g_1 g_2 & g_1^2 \end{array} \right), \end{array} \right.$$

where s is the contrast threshold, τ is an adaptation time constant and S is a spatial smooting kernel. $\mathbf{P}_{\mathbf{g}^\perp}$ is the 2D projection onto $\mathbf{g}^\perp$, thus on the edge tangent, $\mathbf{g}$ being aligned with the edge normal direction. Depending on the norm of the gradient of the intensity, the smoothing term will infer two kinds of behaviors:

- For low contrasts, when ρ is close to zero, we have $\mathbf{L} \equiv \mathbf{I}$: the smoothing term is quadratic which corresponds to an isotropic smoothing in the Euler-Lagrange equation.

- For high contrasts, when ρ is close to one, we have $\mathbf{L} \equiv \mathbf{P}_{\mathbf{g}^\perp}$: the smoothing term will perform anisotropic diffusion only in the normal direction to the edges.

Fig. 5.1 shows some comparison of this adaptive linear diffusion process compared with classical linear diffusion. Thanks to the short-term adaptation of the diffusion tensor $\mathbf{L}$ discontinuities are preserved. The adaptive rule (5.2) corresponds to a Hebbian rule at the implementation level [9], and it can interpreted as a feedback link from previous estimation of v onto the forward diffusion process.

It has been formally shown [9] that combining short-term adaptation with the diffusion process is a convergent process: the key point is that the feedback from v to $\mathbf{L}$ is smooth in space and time.

Note that contrarily to [9],we have not introduced here a non-linearity, we have implemented a linear neural-network as in (2.6). We verified experimentaly that this non-linearity is not determinant and does not influence significantly the resulting image.

At step ahead, in [9], a temporal filtering is introduced in the feedback. Thus, it is not directly $\mathbf{L}$ but an exponential temporal filtering of $\mathbf{L}$ which is taken into account. Proposition **??** prediction is that such a low-pass filtering is not required and we have been able to verify this fact in this context. More precisely, we have experimented that a small-delay (0 .. 10 times the sampling period) low-pass filter does not significantly influence the result, whereas higher delays inhibit the feedback, inducing a convergence with only a poor edge-preserving smoothing.

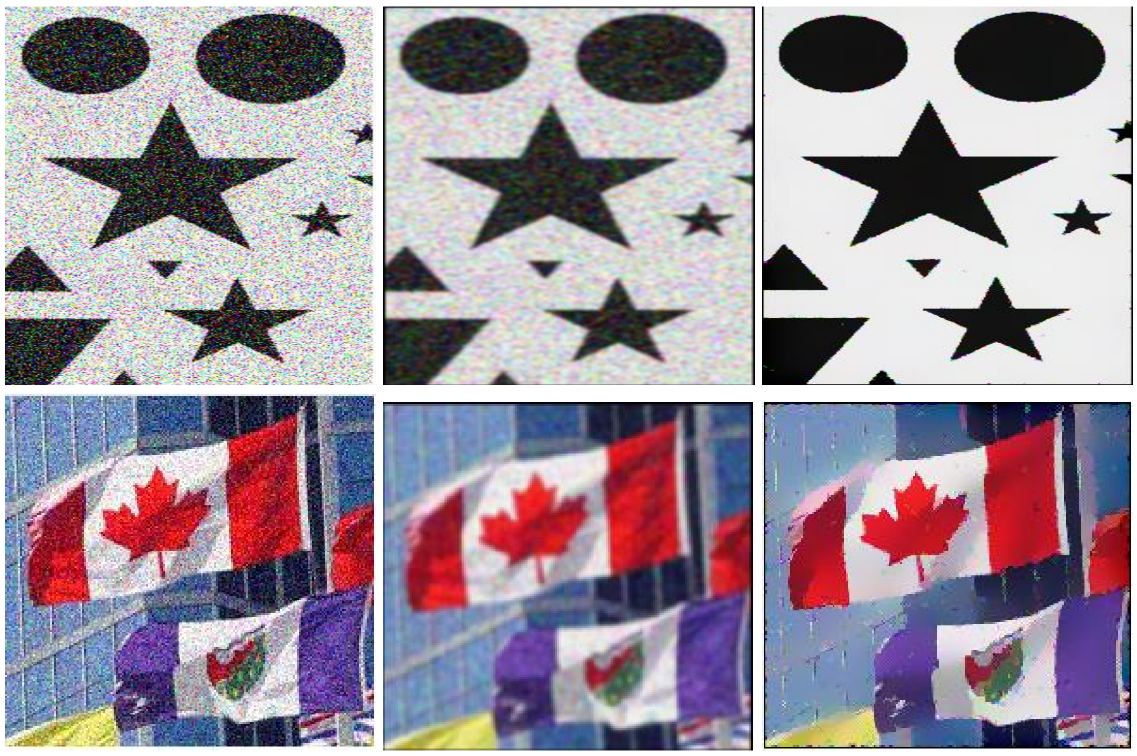


Figure 5.1: Two examples of results using anisotropic diffusion (right image), the 1st example being the same as in [9] to validate the present method. The original image is on the left. As a comparison, a Gaussian filtering (isotropic diffusion) is shown in the middle. The synthetic image contains a huge (80%) amount of noise. The real image contains features at several scales. In both cases edges are preserved, while an important smoothing has been introduced (from [33]).

5.2 The Winner-Take-All (WTA)

Let us now describe how WTA mechanism can be written in this framework.

WTA mechanisms are usually realized (e.g. [35, 17, 21]) using an ad-hoc mechanism with an explicit definition of inter-neuron inhibition in order to allow one neuron to maintain its activity whereas all other activites vanish. They are used in may neuronal computations (see the review in [35]) and the way they could be implemented is still an issue. It is thus a important test for the present method to verify if such a mechanism is easily formalized.

Given an initial condition w, one look for a solution $\bar{v}$ verifying

$$\bar{v} = \underset{v}{argmin}\ \mathcal{L}(v) = \int (w - v)^2 + \int |\nabla v|^2 + \int \psi(v) \tag{5.3}$$

where $\psi : [0, 1] \rightarrow \mathbb{R}$ is a bi-modal function, for example

$$\psi(v) = v^{2\,t/(1-t)} (1 - v)^2,$$

with $\psi(0) = \psi(1) = 0$ and $\phi'(t) = 0$ in fact maximal at $t \in]0, 1[> 1/2$. This previous expression is the simplest polynomial profile with the suitable craracteristics: this non-linear term will force the values of the network to be zero or one, with a pivildege to the zero value.

Formulation (5.3) is directly related to the general framework (2.3–2.5) where no constraint is used and $P \equiv I$.

In Fig. 5.2 an example of result is shown, with an adaptive profile $\psi'(\cdot)$ (the threshold t is initialized to the distribution mean and incremented/decremented during the process to maintain a small binarization with respect to diffusion). The iteration is stopped when the output has a predefined small size.

This very simple mechanism shows how the present formalism may provide a complementary view with respect to other analog network approaches [17, 21].

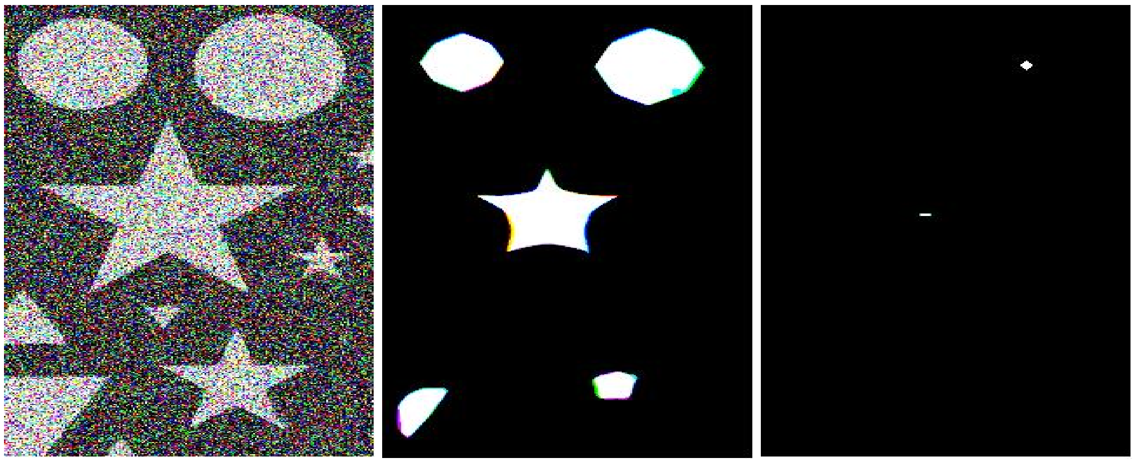


Figure 5.2: Two examples of result for the winner-take-all mechanism implemented using the proposed method. The very noisy (more than 80%) original image is on the left; the intermediate result shows how diffusion is combined with erosion yielding the final result, shown also with a zoom. Clearly the focus is given on the main structures of the image, We have experimented a correct behavior on many different inputs.

Appendix A

Main theoritical demonstrations

A.1 Derivation of proposition 1

The solution of

$$\bar{\mathbf{v}} = \underset{\mathbf{v}\in H/\mathbf{c}(\mathbf{v})=0}{argmin} \; \mathcal{L}(\mathbf{v}), \quad \text{with} \tag{A.1}$$

$$\mathcal{L}(\mathbf{v}) = \int |\hat{\mathbf{w}} - \mathbf{w}|^2_{\mathbf{\Lambda}} + \int \phi(|\nabla \mathbf{v}|_{\mathbf{L}}) + 2\int \psi(\mathbf{v}), \tag{A.2}$$

$$\text{and } \hat{\mathbf{w}} = \mathbf{P}\,\mathbf{v}, \tag{A.3}$$

can be implemented using a discrete neural network (see Appendix A.2 for details or [34]). More precisely, our optimization problem is, in the general case, locally minimized by the following linearized differential equation:

$$\frac{\partial \mathbf{v}_i}{\partial t} = -\epsilon_i(\mathbf{v}_i) + \sum_j \sigma_{ij}(\mathbf{v}_i)\,\mathbf{v}_j + \kappa_i\,\mathbf{w}_i \tag{A.4}$$

with:

$$\left\{ \begin{array}{rcl} \epsilon_i(v) & = & \rho_i\,\mathbf{v} + \xi\,\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c} + \frac{\partial \psi}{\partial \mathbf{v}}^T, \\ \rho_i & = & \sum_j \sigma_{ij} + \mathbf{P}^T\,\mathbf{\Lambda}_i\,\mathbf{P}, \\ \kappa_i & = & \mathbf{P}^T\,\mathbf{\Lambda}_i, \end{array} \right. \tag{A.5}$$

and $\xi = (1-\lambda)\,|\frac{\partial \mathcal{L}}{\partial \mathbf{v}}|/|\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T\mathbf{c}|$ with $\lambda \ll 1$. The weights $\sigma = (\sigma_{ij})$ are given by considering the linearized optimal integral approximation up to order r ($r \geq 2$) of the non-linear diffusion operator (see section 2.3.3)

$$\bar{\mathbf{L}} = \phi'(|\nabla \mathbf{v}|_{\mathbf{L}})\,\mathbf{L}, \tag{A.6}$$

defined at M points, providing $M > \frac{(n+r)!}{n!\,r!} - \frac{n\,(n+1)}{2}$. They are given by solving the system (see Appendix A.3 or [34]):

$$\begin{array}{rcl} \bar{\mathbf{L}}_{kl}(\mathbf{x}) & = & \frac{1}{2}\sum_j \sigma_{ij}\,\bar{\mu}_j^{\mathbf{e}_k+\mathbf{e}_l}(\mathbf{x}), \\ \mathbf{div}_k(\bar{\mathbf{L}}(\mathbf{x_i})) & = & \sum_j \sigma_{ij}\,\bar{\mu}_j^{\mathbf{e}_k}(\mathbf{x}), \end{array} \tag{A.7}$$

where [1]: $\bar{\mu}_j^{\alpha}(\mathbf{x}) = \int_{\mathcal{S}_j} (\mathbf{y}-\mathbf{x})^{\alpha}\,\mu_j(\mathbf{y})\,d\mathbf{y}$, while:

$$\forall i, \quad \sum_j \sigma_{ij}\,\bar{\mu}_j^{\alpha}(\mathbf{x}) = 0 \quad 2 < |\alpha| \leq r \tag{A.8}$$

Among all σ_{ij} verifying (2.9) and (2.10) we choose those which verify:

$$\min \sum_{ij} |\sigma_{ij}^2|. \tag{A.9}$$

A.1.1 Deriving the Euler-Lagrange equations

Let us first derive a differential equation, in order to converge to the minimum of $\mathcal{L}(\mathbf{v})$ with $\mathbf{c}(\mathbf{v}) = 0$.

$$\mathcal{L}(\mathbf{v}) = \frac{1}{2} \int \underbrace{|\hat{\mathbf{w}} - \mathbf{w}|^2_{\mathbf{\Lambda}}}_{\text{input}} + \underbrace{\phi(|\nabla \mathbf{v}|^2_{\mathbf{L}})}_{\text{regularization}} + 2\,\psi(\mathbf{v}) \text{ with } \left\{ \begin{array}{l} \hat{\mathbf{w}} = \mathbf{P}\,\mathbf{v} \\ \mathbf{c}(\mathbf{v}) = 0 \end{array} \right. \tag{A.10}$$

If, writing $\mathcal{L} = \mathcal{L}(\mathbf{v}, \hat{\mathbf{w}}), \mathbf{c} = \mathbf{c}(\mathbf{v}, \hat{\mathbf{w}})$ and $\Xi = |\mathbf{c}|^2$ we must obtain:

$$\dot{\mathcal{L}} = \mathcal{L}_{\mathbf{v}}\ \dot{\mathbf{v}} < 0 \text{ with } \mathcal{L}_{\mathbf{v}} = \frac{\partial \mathcal{L}}{\partial \mathbf{v}} + \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}} \frac{\partial \mathbf{p}}{\partial \mathbf{v}} \text{ and } \dot{\Xi} = \mathbf{c}^T \frac{\partial \mathbf{c}}{\partial \mathbf{v}}\, \dot{\mathbf{v}} < -\epsilon < 0$$

for some ϵ, since both $\mathcal{L}$ and Ξ are positive strictly decreasing quantities thus converging to their minimum. Since $\dot{\Xi} < -\epsilon < 0$, Ξ can not converge towards a strictly positive minimum, thus converge to 0, as required.

In order $\dot{\mathbf{v}}$ to verify both $\dot{\mathcal{L}} < 0$ and $\dot{\Xi} < -\epsilon$, let us consider a scheme of the form, for some $\lambda > 0$:

$$\dot{\mathbf{v}} = -\mathcal{L}^T_{\mathbf{v}} - \lambda \frac{|\mathcal{L}^T_{\mathbf{v}}|}{|\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}|} \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}$$

yielding, writing $\theta = \widehat{\mathcal{L}^T_{\mathbf{v}}, \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T} \mathbf{c}$:

$$-\dot{\mathcal{L}} = |\mathcal{L}^T_{\mathbf{v}}|^2\,(1 + \lambda\, \cos(\theta)) \text{ and } -\dot{\Xi} = |\mathcal{L}^T_{\mathbf{v}}|\,|\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}|\,(\cos(\theta) + \lambda)$$

the conditions reducing to:

$$1 + \lambda\, \cos(\theta) > 0 \text{ and } \cos(\theta) + \lambda > \varepsilon$$

since we write ε the "infinitesimal" value under which $|\mathcal{L}^T_{\mathbf{v}}|$ and $|\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}|$ are negligible (which stops the algorithm) and we also choose $\epsilon = \varepsilon^3$. Now, as soon as $\cos(\theta) > -1$, numerically as soon as $\cos(\theta) \geq -1 + \varepsilon$ (the case where $\cos(\theta) = -1$ is a singular case, the present solution being only valid in the general case) if we choose $\lambda = 1 + \varepsilon$, we obtain:

$$1 + \lambda\, \cos(\theta) \geq \varepsilon^2 > 0 \text{ and } \cos(\theta) + \lambda \geq 2\,\varepsilon > \varepsilon$$

the conditions being verified.

Now considering the integral approximation derived in A.1.2, we can write:

$$\begin{array}{rcl} -\frac{\partial \mathcal{L}}{\partial \mathbf{v}}^T & = & \mathbf{div}(\phi'(|\nabla \mathbf{v}|^2_{\mathbf{L}})\, \mathbf{L} \nabla \mathbf{v}) - \frac{\partial \psi}{\partial \mathbf{v}}^T \\ & \simeq & \sum_j \sigma_{.j}\, \mathbf{v}_j - \nu\, \mathbf{v} - \frac{\partial \psi}{\partial \mathbf{v}}^T \end{array} \text{ and } \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}}^T = \mathbf{\Lambda}\, [\hat{\mathbf{w}} - \mathbf{w}]$$

and obtain, as expected:

$$\begin{array}{rcl} \dot{\mathbf{v}} & = & -\frac{\partial \mathcal{L}}{\partial \mathbf{v}}^T - \frac{\partial \mathbf{p}}{\partial \mathbf{v}}^T \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{w}}}^T - \lambda \frac{||-\nu\, \mathbf{v} + \sum_j \sigma_{.j}\, \mathbf{v}_j + \kappa\, [\mathbf{w} - \mathbf{p}(\mathbf{v})]||}{||\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}||} \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c} \\ & = & -\nu\, \mathbf{v} + \sum_j \sigma_{.j}\, \mathbf{v}_j + \kappa\, \mathbf{w} - \epsilon \end{array}$$

with:

$$\left\{ \begin{array}{rcl} \kappa & = & \frac{\partial \mathbf{p}}{\partial \mathbf{v}}^T \mathbf{\Lambda} \\ \epsilon & = & \kappa\, \mathbf{P} \mathbf{v} + \lambda \frac{||\dot{\mathbf{v}} + \epsilon - \kappa\, \mathbf{p}(\mathbf{v})]||}{||\frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c}||} \frac{\partial \mathbf{c}}{\partial \mathbf{v}}^T \mathbf{c} + \frac{\partial \psi}{\partial \mathbf{v}}^T \end{array} \right. \tag{A.11}$$

for some λ infinitesimally higher than 1, corresponding to A.4 and A.5.

A.1.2 Unbiased integral approximation of the differential operator

Clearly, the only difficult point is the regularization term derivation because the Euler Lagrange equations lead to the following well-known expression, which need to be implemented in a discrete way:

$$\mathbf{div}(\phi'(||\nabla \mathbf{v}||^2_{\mathbf{L}})\, \mathbf{L}\, \nabla \mathbf{v}), \tag{A.12}$$

If we consider the specifications of the diffusion mechanism in Appendix A.3, then we are able to derive equations (A.4)-(A.9) defining:

$$\Delta^*_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) = \int_{\mathcal{S}} \sigma(\mathbf{x}, \mathbf{y})\, \mathbf{f}(\mathbf{y})\, d\mathbf{y} - \nu(\mathbf{x})\, \mathbf{f}(\mathbf{x})$$

for a kernel $\sigma(\mathbf{x}, \mathbf{y})$ defined in $\mathbb{R}^n \times \mathbb{R}^n$.

This integral, in the discrete case where only $\mathbf{f}(\mathbf{y}_j), l = 1..M$ is available, can not be estimated for a general kernel $\sigma(\mathbf{x}, \mathbf{y})$, but must be written with kernels of the form:

$$\sigma(\mathbf{x}, \mathbf{y}) = \left\{ \begin{array}{cl} \sigma(\mathbf{x}, \mathbf{y}_j)\, \mu_j(\mathbf{y}) & \mathbf{y} \in \mathcal{S}_j \\ 0 & \mathbf{y} \in \mathcal{S} - \cup_j \mathcal{S}_j \end{array} \right.$$

which appears to be the more general kernel compatible with the summation property, yielding:

$$\begin{array}{rcl} \Delta^*_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) & = & \sum_j \int_{\mathcal{S}_j} \sigma(\mathbf{x}, \mathbf{y})\, \mathbf{f}(\mathbf{y})\, d\mathbf{y} - \nu(\mathbf{x})\, \mathbf{f}(\mathbf{x}) \\ & = & \sum_j \int_{\mathcal{S}_j} \sigma(\mathbf{x}, \mathbf{y}_j)\, \mu_j(\mathbf{y})\, \mathbf{f}(\mathbf{y})\, d\mathbf{y} - \nu(\mathbf{x})\, \mathbf{f}(\mathbf{x}) \\ & = & \sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \int_{\mathcal{S}_j} \mu_j(\mathbf{y})\, \mathbf{f}(\mathbf{y})\, d\mathbf{y} - \nu(\mathbf{x})\, \mathbf{f}(\mathbf{x}) \\ & = & \sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \mathbf{f}(\mathbf{y}_j) - \nu(\mathbf{x})\, \mathbf{f}(\mathbf{x}) \end{array}$$

.

Let us now consider the Taylor expansion of $\mathbf{g}(\mathbf{y}, \mathbf{x}) = \mathbf{f}(\mathbf{y}) - \mathbf{f}(\mathbf{x})$ with respect to $\mathbf{d} = \mathbf{y} - \mathbf{x}$:

$$\begin{array}{rcl} \mathbf{g}(\mathbf{y}, \mathbf{x}) & = & \sum_{|\alpha|=1}^{r} \frac{\partial^\alpha \mathbf{f}}{\alpha!}(\mathbf{x})\Big|_{\mathbf{y}=\mathbf{x}} (\mathbf{y} - \mathbf{x})^\alpha + o(||\mathbf{y} - \mathbf{x}||^r) \\ & = & \left[\sum_{|\alpha|=1}^{r} (\mathbf{y} - \mathbf{x})^\alpha \frac{\partial^\alpha}{\alpha!}\right] \mathbf{f}(\mathbf{x}) + o(||\mathbf{y} - \mathbf{x}||^r) \end{array}$$

considering here that

$$\nu(\mathbf{x}) = \int_{\mathcal{S}} \sigma(\mathbf{x}, \mathbf{y})\, d\mathbf{y} = \sum_j \int_{\mathcal{S}_j} \sigma(\mathbf{x}, \mathbf{y}_j)\, \mu(\mathbf{y})\, d\mathbf{y} = \sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \bar{\mu}_j^0$$

we obtain:

$$\begin{array}{rcl} \Delta^*_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) & = & \int_{\mathcal{S}} \sigma(\mathbf{x}, \mathbf{y})\, [\mathbf{f}(\mathbf{y}) - \mathbf{f}(\mathbf{x})]\, d\mathbf{y} \\ & = & \int_{\mathcal{S}} \sigma(\mathbf{x}, \mathbf{y})\, \mathbf{g}(\mathbf{y}, \mathbf{x})\, d\mathbf{y} \\ & = & \sum_j \int_{\mathcal{S}_j} \sigma(\mathbf{x}, \mathbf{y}_j)\, \mu_j(\mathbf{y}) \left[\sum_{|\alpha|=1}^{r} (\mathbf{y} - \mathbf{x})^\alpha \frac{\partial^\alpha}{\alpha!} \mathbf{f}(\mathbf{x})\right] d\mathbf{y} + \mathbf{R}^r \mathbf{f} \\ & = & \left[\sum_{|\alpha|=1}^{r} \sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \int_{\mathcal{S}_j} (\mathbf{y} - \mathbf{x})^\alpha\, \mu_j(\mathbf{y})\, d\mathbf{y}\, \frac{\partial^\alpha}{\alpha!}\right] \mathbf{f}(\mathbf{x}) + \mathbf{R}^r \mathbf{f} \\ & = & \left[\sum_{|\alpha|=1}^{r} \sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \bar{\mu}_j^\alpha(\mathbf{x})\, \frac{\partial^\alpha}{\alpha!}\right] \mathbf{f}(\mathbf{x}) + \mathbf{R}^r \mathbf{f} \end{array}$$

where the remainder $R^\epsilon_{kl}\mathbf{f}$ of this expansion may be written using an integral form:

$$\mathbf{R}^r \mathbf{f} = \sum_{|\alpha|=r+1} \frac{r+1}{\alpha!} \int_{\mathcal{S} \times [0,1]} \sigma(\mathbf{x}, \mathbf{y})\, (\mathbf{y} - \mathbf{x})^\alpha\, (1-u)^r\, \partial^\alpha \mathbf{f}^j (\mathbf{x} + u\, (\mathbf{y} - \mathbf{x}))\, d\mathbf{y}\, du,$$

and because the support is included in a ball of radius ϵ, the remainder is bounded by the standard condition:

$$||R^r \mathbf{f}||_{0,\infty} < C\, \epsilon^{r-1} ||\mathbf{f}||_{r+1,\infty}$$

where C is a fixed quantity [11]. This would have been also true for an unbounded support providing $\sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \bar{\mu}_j^\alpha(\mathbf{x}) < +\infty, |\alpha| = r+1$. (see e.g. [11]).

If we expand the diffusion operator with the same notations:

$$\begin{array}{rcl} \Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) & = & \mathbf{div}(\mathbf{L}(\mathbf{x})\, (D\mathbf{f})(\mathbf{x})) \\ & = & \mathbf{div}(\mathbf{L}(\mathbf{x}))\, (D\mathbf{f})(\mathbf{x}) + \mathrm{trace}(\mathbf{L}(\mathbf{x})\, (D^2\mathbf{f})(\mathbf{x})) \\ & = & \left[\sum_k \mathbf{div}^k(\mathbf{L}(\mathbf{x}))\partial^{\mathbf{e}_k} + \sum_{kl} \mathbf{L}^{kl}(\mathbf{x})\partial^{\mathbf{e}_k + \mathbf{e}_l}\right] \mathbf{f}(\mathbf{x}) \end{array}$$

and identify with the previous expression of $\Delta^*_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x}))$ up the rth order we obtain the conditions (A.7) and (A.8) for $|\alpha| > 0$

We also finally obtain:

$$\Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) - \Delta^*_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) = \mathbf{R}^r \mathbf{f}$$

leading to the proposed approximation.

A.1.3 Generality of the integral approximation

On the reverse, from equations (A.7)-(A.9) to the related diffusion operator $\mathbf{L}$, let us analyze to which extend a family of values $\sigma_{ij} = \sigma(\mathbf{x}_i, \mathbf{y}_j)$ can be extended to an operator $\sigma(\mathbf{x}, \mathbf{y}_j)$ in relation with some $\mathbf{L}$.

Without loss of generality we can write:

$$\sigma(\mathbf{x}, \mathbf{y}_j) = \sum_i \sigma(\mathbf{x}_i, \mathbf{y}_j)\, K_i(\mathbf{x})$$

for some smooth positive kernel $K_i()$ around $\mathbf{x}_i$ providing :

$$K_i(\mathbf{x}_{i'}) = \delta_{ii'} \text{ with } \delta_{ii'} = \left\{ \begin{array}{ll} 1 & i = i' \\ 0 & i \neq i' \end{array} \right.$$

in order the formula to be coherent. This is the more general formula writable to extends $\sigma(\mathbf{x}_i, \mathbf{y}_j)$ linearly to $\sigma(\mathbf{x}, \mathbf{y}_j)$ (see e.g. [16, 15, 3]).

In what follows we are going to restrain to a linear subspace of co-dimension 1 in the previous kernel space.

Since $\sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \bar\mu_j^\alpha(\mathbf{x}) = \sum_{ij} \sigma_{ij}\, K_i(\mathbf{x})\, \bar\mu_j^\alpha(\mathbf{x})$ the identification in (??) writes

$$\mathbf{L}_{kl}(\mathbf{x}) = \frac{1}{2} \sum_{ij} \sigma_{ij}\, K_i(\mathbf{x})\, \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x})$$

thus:

$$\mathbf{L}_{kl}(\mathbf{x}_i) = \frac{1}{2} \sum_j \sigma_{ij}\, \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}_i)$$

A step further:

$$\begin{array}{rcl} \mathbf{div}_k(\mathbf{L}(\mathbf{x})) & = & \sum_l \frac{\partial}{\partial x^l} \mathbf{L}_{kl}(\mathbf{x}) \\ & = & \frac{1}{2} \sum_{ij} \sigma_{ij} \left[\sum_l \mathbf{grad}^l(K_i(\mathbf{x}))\, \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}) - n\, K_i(\mathbf{x})\, \bar\mu_j^{\mathbf{e}_k}(\mathbf{x}) \right] \end{array}$$

because:

$$\begin{array}{rcl} \mathbf{div}_k(\bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x})) & = & \sum_l \frac{\partial}{\partial x^l} \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}) \\ & = & \sum_l \frac{\partial}{\partial x^l} \int_{\mathcal{S}_j} (\mathbf{y} - \mathbf{x})^{\mathbf{e}_k}\, (\mathbf{y} - \mathbf{x})^{\mathbf{e}_l}\, \mu_j(\mathbf{y})\, d\mathbf{y} \\ & = & \sum_l \int_{\mathcal{S}_j} (\mathbf{y} - \mathbf{x})^{\mathbf{e}_k}\, (-1)\, \mu_j(\mathbf{y})\, d\mathbf{y} \\ & = & -n \int_{\mathcal{S}_j} (\mathbf{y} - \mathbf{x})^{\mathbf{e}_k}\, \mu_j(\mathbf{y})\, d\mathbf{y} = -n\, \bar\mu_j^{\mathbf{e}_k}(\mathbf{x}) \end{array}$$

Thanks to these formula, the constraints in (??) write:

$$\left\{ \begin{array}{l} \sum_{ij} \sigma_{ij} \left[\sum_l \mathbf{grad}_l(K_i(\mathbf{x}))\, \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}) - (n+2)\, K_i(\mathbf{x})\, \bar\mu_j^{\mathbf{e}_k}(\mathbf{x}) \right] = 0 \\ \sum_{ij} \sigma_{ij}\, K_i(\mathbf{x})\, \bar\mu_j^\alpha(\mathbf{x}) = 0 \quad 2 < |\alpha| \leq r \end{array} \right.$$

Using the functional notation $\mathbf{K} = [\cdots K_i \cdots]$ for the M unknowns, the equations write $\mathbf{H}\,\mathbf{K} = 0$ at each $\mathbf{x}$, where $\mathbf{H}$ is the linear differential operator defined by the previous $\xi_{n,r} - n\,(n+1)/2$ linear equations (we have equations for α exponents up to r, thus $\xi_{n,r}$ but not for $r = 2$ thus without $n\,(n+1)/2$). A generic solution now writes $\mathbf{K} = \mathbf{H}^\perp\, \mathbf{Z}$ for some function $\mathbf{Z}$. There are non-zero solutions, in the general case, in the continuous dimensional space of kernels or in a linear sub-space of finite co-dimension, as soon as there are more unknowns than equations, as written in the statement.

One drawback with the previous general equations is the fact there are not local since they link $\sigma_{i.}$ for different point $\mathbf{x}_i$. In order to restrain as much as possible to local constraints, let us constraint the support of $K_i()$ to be $\mathcal{S}_i - \cup_{j \neq i} \mathcal{S}_j$, i.e.:

$$\forall \mathbf{x} \in \mathcal{S}_j, j \neq i, K_i(\mathbf{x}) = 0 \Leftrightarrow \int_{\mathcal{S} - [\mathcal{S}_i - \cup_{j \neq i} \mathcal{S}_j]} K_i(\mathbf{x})\, d\mathbf{x} = 0$$

The equivalence is due to the fact that $K_i()$ is positive. Choosing such a particular class of kernels still provide solutions as described previously, but in a smaller functional subspace of co-dimension 1. This restrictive condition may over-constraint σ_{ij} but has a very interesting consequence since everything vanishes in $\mathcal{S} - \cup_i \mathcal{S}_i$ while $\forall \mathbf{x} \in \mathcal{S}_h$:

$$\left\{ \begin{array}{l} \sum_j \sigma_{hj} \left[\sum_l \mathbf{grad}_l(K_h(\mathbf{x}))\, \bar\mu_j^{\mathbf{e}_k + \mathbf{e}_l}(\mathbf{x}) - (n+2)\, K_h(\mathbf{x})\, \bar\mu_j^{\mathbf{e}_k}(\mathbf{x}) \right] = 0 \\ \sum_j \sigma_{hj}\, K_h(\mathbf{x})\, \bar\mu_j^\alpha(\mathbf{x}) = 0 \quad 2 < |\alpha| \leq r \end{array} \right.$$

i.e. the unbiasness conditions are now only *local* which seems a reasonable requirement in this context.

A step further, since we must consider solutions with $K_i(\mathbf{x}_{i'}) = \delta_{ii'}$, writing $\mathbf{grad}(K_h(\mathbf{x}_h)) = \gamma_h$, we obtain for $\mathbf{x}_h \in \mathcal{S}_h$:

$$\left\{ \begin{array}{l} \sum_j \sigma_{hj} \left[\sum_l \gamma_h^l \, \bar{\mu}_j^{\mathbf{e}_k+\mathbf{e}_l}(\mathbf{x}_h) - (n+2)\, \bar{\mu}_j^{\mathbf{e}_k}(\mathbf{x}_h)\right] = 0 \\ \sum_j \sigma_{hj}\, \bar{\mu}_j^{\alpha}(\mathbf{x}_h) = 0 \quad 2 < |\alpha| \leq r \end{array} \right.$$

Since we also obtain:

$$\mathbf{div}_k(\mathbf{L}(\mathbf{x}_h)) = \frac{1}{2} \sum_j \sigma_{hj} \left[\sum_l \sum_l \gamma_h^l \, \bar{\mu}_j^{\mathbf{e}_k+\mathbf{e}_l}(\mathbf{x}_h) - n\, K_i(\mathbf{x}_h)\, \bar{\mu}_j^{\mathbf{e}_k}(\mathbf{x}_h)\right]$$

combining this with the second line of the previous constraints yields the second equality in (A.7).

Now, given σ_{ij} verifying (A.8), for any kernels $K_i()$ in the functional space for which $K_i(\mathbf{x}_{i'}) = \delta_{ii'}$, with $\int_{\mathcal{S}-[\mathcal{S}_i - \cup_{j\neq i}\mathcal{S}_j]} K_i(\mathbf{x})\, d\mathbf{x} = 0$, while $\int_{\mathcal{S}_i} K_i(\mathbf{y})\, d\mathbf{y} = 1$, verifying $\mathbf{H}\,\mathbf{K} = 0$ (and there are such kernels as soon as $M > \xi_{n,r} - n\,(n+1)/2$ as explained previously) we can define the related operator $\mathbf{L}_{kl}(\mathbf{x}) = \frac{1}{2} \sum_{ij} \sigma_{ij}\, K_i(\mathbf{x})\, \bar{\mu}_j^{\mathbf{e}_k+\mathbf{e}_l}(\mathbf{x})$, verifying (A.7) by construction.

A.2 Results about neural network formalisms

An abstract analog Hopfield network locally minimizes -in the general case- the following functional:

$$\frac{1}{2} \int \underbrace{||v - \kappa\, w||^2}_{\textbf{input}} + \underbrace{\phi(||\nabla v||_L^2)}_{\textbf{regularization}} + 2\,\psi(v)$$

with $\psi(v) = \xi_i\,(\int g^{-1}(v) + \theta v) + \frac{1}{2}\,(\nu - 1)\, v^2$
considering, from (A.3), an integral approximation of the diffusion operator $\phi'(||\nabla v||_L^2)\, L$ **written** $\sigma_{ij} = \sigma(\mathbf{x}_i, \mathbf{y}_j)$ **and** $\nu_i = \nu(\mathbf{x}_i)$
as soon as σ_{ij} **is unbiased according to (A.8).**

Let us derive this fact. The $\mathcal{L}$ functional Euler-Lagrange equation writes:

$$\frac{\partial \mathcal{L}}{\partial v} = [v - \kappa\, w] - div(\phi'(||\nabla v||_L^2)\, L \nabla v) + \psi'(v)$$

thus from (A.3) is approximated up to some small error $\tilde{\varepsilon}$:

$$\begin{array}{l} \frac{\partial \mathcal{L}}{\partial v}(\mathbf{x}_i) \\ = [v_i - \sum_k \kappa_{ik}\, w_k] - \left[\sum_j \sigma_{ij}\, v_j - \nu_i\, v_i - \tilde{\varepsilon}_i\right] + [\xi_i\,(g^{-1}(v_i) + \theta_i) + (\nu_i - 1)\, v_i] \\ = - \left[-\xi_i\,(g^{-1}(v_i) + \theta_i) + \sum_j \sigma_{ij}\, v_j + \sum_k \kappa_{ik}\, w_k\right] + \tilde{\varepsilon}_i \end{array}$$

because $\psi'(v) = \xi\,(g^{-1}(v) + \theta) + (\nu - 1)\, v$.

Since $u_i = g^{-1}(v_i) + \theta_i$ and $\dot{u}_i = g^{-1'}(v_i)\, \dot{v}_i$, with respect to v_i, equation (1.1) writes:

$$\begin{array}{rcl} \dot{v}_i & = & \frac{1}{g^{-1'}(v_i)} \left[-\xi_i\,(g^{-1}(v_i) + \theta_i) + \sum_j \sigma_{ij}\, v_j + \sum_k \kappa_{ik}\, w_k\right] \\ & = & -a(v_i)\, \left[\frac{\partial \mathcal{L}}{\partial v}(\mathbf{x}_i) - \tilde{\varepsilon}_i\right] \end{array}$$

with $a(v_i) = \frac{1}{g^{-1'}(v_i)} > 0$. As a consequence (1.1), as soon as $\tilde{\varepsilon}$ is small enough, this defines a *gradient descent* of the convex criterion leading to the solution.

More precisely, we consider a set of admissible functions $\mathcal{F}$ in a dense linear subset of a Hilbert space H, the scalar product of which is denoted by $(\cdot,\cdot)_H$. The criterion $\mathcal{L}$ being regular, its first variation (also called its Gâteaux derivative) at $v \in \mathcal{F}$ in the direction $\mathbf{k} \in H$ is defined by:

$$\delta_{\mathbf{k}} \mathcal{L}(v) = \lim_{\epsilon \to 0} \frac{\mathcal{L}(v + \epsilon \mathbf{k}) - \mathcal{L}(v)}{\epsilon}$$

If the mapping $\mathbf{k} \to \delta_{\mathbf{k}} \mathcal{L}(v)$ is linear and continuous, the Riesz representation theorem [14] guarantees the existence of a unique vector, denoted by $\frac{\partial \mathcal{L}}{\partial v}$, and called the gradient of $\mathcal{L}$, which satisfies the equality

$$\delta_{\mathbf{k}} \mathcal{L}(v) = (\frac{\partial \mathcal{L}}{\partial v}, \mathbf{k})_H$$

for every $\mathbf{k} \in H$. The gradient depends on the choice of the scalar product $(\cdot,\cdot)_H$. If a minimizer v^* of $\mathcal{L}$ exists, then the set of equations $\delta_{\mathbf{k}} \mathcal{L}(v) = 0$ must hold for every $\mathbf{k} \in H$, which is equivalent to $\frac{\partial \mathcal{L}}{\partial v} = 0$. These equations are called the Euler-Lagrange equations associated with the energy functional $\mathcal{L}$.

A step further, the partial differential rule writes: $\dot{\mathcal{L}} = \frac{\partial \mathcal{L}}{\partial v}\, \dot{v}$. Considering $\dot{v} = -a(v)\,[\frac{\partial \mathcal{L}}{\partial v} - \tilde{\varepsilon}]$ yields $\dot{\mathcal{L}} = -a(v)\,[||\frac{\partial \mathcal{L}}{\partial v}||^2 - (\tilde{\varepsilon}^T\, \frac{\partial \mathcal{L}}{\partial v})] < 0$ as soon as $\tilde{\varepsilon}$ is small enough, so that the criterion is strictly decreasing. Since $\mathcal{L} \geq 0$ it is also bounded and thus converges towards a minimal value. At this local or global minimum $||\frac{\partial \mathcal{L}}{\partial v}|| = 0$ so that $\mathcal{L}$ is stationary.

In fact, the gradient magnitude smoothly decreases with time and convergence is detected as soon as the gradient is below a given numerical threshold, which always occurs in finite time.

In the sequel, we re-use the same mechanism without re-detailing these technical details.

- If $\phi()$ is convex, the criterion is convex since (i) the input term is quadratic thus convex, (ii) the regularization term is convex as a combination of convex functionals, while (iii) $\psi()$ is convex as the sum of convex (linear, quadratic) functions since $h = \int g^{-1}(v)$ is convex because

$$h'' = g^{-1'} = 1/(g' \circ g^{-1}) > 0$$

It thus has a unique optimum, given an input.

- Otherwise, the mechanism only drives towards a local minimum of the criterion. This is an interesting property in practice since it means finding the optimum close to the initial à-priory value, i.e. choose a pertinent solution.

- Furthermore, at the optimum, when $\dot{u} = 0$ we obtain:

$$\xi\,(g^{-1}(v) + \theta) = div(\phi'(||\nabla v||^2_L)\, L\, \nabla v) + \kappa\, w + \nu\, v$$

and v is the solution of an elliptic partial differential equation: clearly less easy to study than the minimum of the related criterion.

- Obviously, minimizing a functional is not the only possible characterization of such an analog network. Such networks may have very different behaviors, including chaotic modes, which can not be related to such a criterion. However, as illustrated in the introduction, a large class of neural network computations can be formalized that way.

About the specification of Continuous Neural Fields.

Considering, following [1], the specification of continuous neural fields defined by the following equation:

$$\begin{array}{rcl} \dot{u}(\mathbf{x}, t) & = & -\xi(\mathbf{x})\, u(\mathbf{x}, t) + \int \sigma(\mathbf{x}, \mathbf{x}')\, v(\mathbf{x}', t)\, d\mathbf{x}' + \int \kappa(\mathbf{x}, \mathbf{x}')\, w(\mathbf{x}', t)\, d\mathbf{x}' \\ & & \text{with } v(\mathbf{x}, t) = g(u(\mathbf{x}, t) - \theta(\mathbf{x})) \end{array}$$

which is a continuous form of (1.1) the correspondent of the previous result allows to consider continuous neural fields for which $\sigma(\mathbf{x}, \mathbf{x}')$ is the integral approximation of a differential operator as a direct application of [30, 11, 9, 13]. However, this introduces a bias since this continuous equation has to be approximated by a discrete formula. Such problem is avoided using the present approach as made explicit in (A.13). In other words our approach has to be considered as an alternative to continuous neural fields.

However, it must be noted that the present approach accounts only for *local kernel*, i.e. *without remote connections*, since the integral is an integral approximation of a differential operator, thus a local operator by construction.

Generalization to Grossberg dynamical systems.

Let us now apply the same result to a more powerful class of models, such as what is proposed by Grossberg and collaborators (e.g. [21, 29]). A Cohen and Grossberg dynamical system is a system of the form:

$$\dot{u}_i = a_i(u_i)\left[b_i(u_i) - \sum_j c_{ij}\, d_j(u_j)\right]$$

with $a_i() > 0$ and $d'_j() > 0$ and convergence is demonstrated for the case where $c_{ij} = c_{ji}$ for which the system minimizes a criterion of the form:

$$\sum_{i=1} \int^{u_i} b_i(v_i) d'(v_i) dv_i + \frac{1}{2} \sum_{ij} c_{ij} d_i(u_i) d_j(u_j)$$

as shown in [21].

This result is in fact rather profound as made explicit by the Benaim[2] theorem [6] if we consider a general dynamical system:

$$\dot{u}_i = F_i(\mathbf{u})$$

as soon it can be decomposed in the form $F_i(\mathbf{u}) = h_i(\mathbf{u})\, G_i(\mathbf{u})$ where $h_i(\mathbf{u})$ are $\mathcal{C}^1$ strictly positive functions and

$$\forall ij \; \partial_j\, G_i(\mathbf{u}) = \partial_i\, G_j(\mathbf{u})$$

(detailled balance condition) are verified, the differential form $\omega = \sum_{i=1}^N G_i(\mathbf{u})\, du_i$ is exact ($d\omega = 0$) thus derives from a potential $V = -\int \omega$ (i.e. $\omega = -dV$) with

$$\dot{V} = \frac{dV}{dt} = -\sum_{i=1}^N G_i(\mathbf{u})\, \frac{du_i}{dt} = -\sum_{i=1}^N G_i(\mathbf{u})^2\, h_i(\mathbf{u}) \left\{ \begin{array}{lll} = & 0 & \text{iff} \quad \dot{u} = 0 \\ < & 0 & \text{iff} \quad \dot{u} \neq 0 \end{array} \right.$$

thus defining a Lyapounov function for the system (see [6] for a detailled discussion), which is convergent towards stable fixed points, $\mathbf{u}, \mathbf{F}(\mathbf{u}) = 0$.

Considering a Cohen and Grossberg dynamical system, if we choose:

$$h_i(\mathbf{u}) = \frac{a_i(u_i)}{d_i'(u_i)} > 0 \text{ with } G_i(\mathbf{u}) = d_i'(u_i) \, \left[b_i(u_i) - \sum_j c_{ij}\, d_j(u_j) \right]$$

thus $\partial_j\, G_i(\mathbf{u}) = c_{ij}\, d_i'(u_i)\, d_j'(u_j)$ the detailled balance conditions are obviously equivalent to $c_{ij} = c_{ji}$

More generaly the $n(n-1)/2$ detailled balance conditions (writing equality between functions) take the form:

$$\partial_j\, G_i = \partial_i\, G_j \text{ with } \partial_j\, G_i = [\partial_j F_i - F_i\, \partial_j h_i / h_i] / h_i \text{ while } h_i = F_i / [k_i + \int \partial_j\, G_i du_j]$$

for a set of n suitable function h_i, and a given constant k_i. In the general case $n(n-1)/2 - n = n(n-3)/2$ conditions must be verified between F_i for such solution to exist.

In the cases where assuming the symmetry of c_{ij} is too restrictive, the following complementary result may help:

A Cohen and Grossberg dynamical system locally minimizes -in the general case- the following functional:

$$\frac{1}{2} \int \phi(||\nabla v||_L^2) + 2\, \psi(v) \textbf{ with } v = -d(u)$$

with $\psi(v) = \int b(d^{-1}(v)) + \frac{1}{2}\, \nu\, v^2$
considering, from (A.3), an integral approximation of the diffusion operator $\phi'(||\nabla v||_L^2)\, L$ **written** $d_{ij} = c(\mathbf{x}_i, \mathbf{y}_j)$ **and** $\nu_i = \nu(\mathbf{x}_i)$
as soon as c_{ij} **is unbiased according to (A.8).**

- This technical result does not bring any highlight regarding the cognitive models based on Cohen and Grossberg dynamical systems but is worthwhile to mention because it enlarges the convergence of such dynamical systems considering complementary assumptions about c_{ij}.

- It also provides an abstract view of “what is done” by such a mechanism. In [23], a particular case of such representation is considered.

Ley us show the algebra to derive this fact, the derivation is the same as the previous one. The $\mathcal{L}$ functional Euler-Lagrange equation writes in this case:

$$\frac{\partial \mathcal{L}}{\partial v} = -div(\phi'(||\nabla v||_L^2)\, L \nabla v) + \psi'(v) \simeq -\sum_j c_{ij}\, v_j + b(d^{-1}(v))$$

yielding:

$$\dot{v} \simeq -\frac{a(d^{-1}(v))}{d^{-1\prime}(v)} [b(d^{-1}(v)) - \sum_j c_{ij}\, v_j] = -k(v)\, \frac{\partial \mathcal{L}}{\partial v}$$

for some positive function $k(v)$, i.e. a criterion gradient descent as made explicit for the previous result.

Here, the fact that $d_j'() > 0$ allows to consider $d()$ as a bijection between two bounded intervals and use $d^{-1}()$.

[2]Here we propose an obviously equivalent form of the Benaim theorem but with simpler notations.

A.3 Integral approximation of the non-linear diffusion operator

Position of the problem.

Computation maps (which inputs are previous computation maps or sensors) are well specified using a general diffusion operator.

However, in practice, it is not possible to implement such a 1st and 2nd differential operator without choosing an approximation and a numerical scheme, etc.. with the risk of loosing what was well-defined in the continuous case.

Why is it not possible in practice to implement such "punctual" operator? Because what is given, in the real world, is a set of "samples". More precisely, we have to consider a set of "measures", defined as integral values of the continuous function over the measurement sensor receptive field or integral values calculated by a previous computation map. Furthermore, since numerical values contains uncertainties, it is a reasonable choice to "average" several values in order to smooth these uncertainties.

At the implementation level, a differential operator is implemented using a derivative filter, i.e. performing a convolution with some suitable kernel. In other words a differential operator is implemented as an $\underline{\text{integral}}$ operator. Let us make this basic fact explicit: this will guide us towards a very general, simple and nice solution.

Following [30, 11, 9, 13, 33] we may formally write, using the Dirac distribution $\delta(\mathbf{x})$ (see [31] for an introduction), with $\bar\sigma(\mathbf{x},\mathbf{y}) = \Delta_{\mathbf{L}(\mathbf{x})}(\delta(\mathbf{y}-\mathbf{x}))$:

$$\Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) = \int_{\mathcal{S}} \bar\sigma(\mathbf{x},\mathbf{y})\,\mathbf{f}(\mathbf{y})\,d\mathbf{y}$$

(in words a differential operator is equivalent to a convolution with the combination of Dirac derivatives).

A differential operator is "punctual" in the sense its magnitude is zero except at $\mathbf{x}$:

$$\int_{\mathcal{S}-B(\mathbf{x},\varepsilon)} ||\bar\sigma(\mathbf{x},\mathbf{y})||\,d\mathbf{y} = 0$$

where $B(\mathbf{x},\varepsilon)$ is a ball around $\mathbf{x}$ of radius ε, which can be as small as possible, but must be excluded.

Here the road-map is to define an implementable *kernel* $\sigma(\mathbf{x},\mathbf{y})$ for a diffusion operator, with a maximal amount of information close to the point: the "sharpest" the operator, the best.

Let us verify the previous expressions:

$$\begin{array}{ll}
\Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) & \\
\quad = \delta(\mathbf{x}) * \Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) & \text{since } \delta \text{ is the convolution unary element} \\
\quad = \Delta_{\mathbf{L}(\mathbf{x})}(\delta(\mathbf{x})) * \mathbf{f}(\mathbf{x}) & \text{since derivation/convolution commutes} \\
\quad = \int_{\mathbb{R}^n} \bar\sigma(\mathbf{x},\mathbf{y})\,\mathbf{f}(\mathbf{y})\,d\mathbf{y} & \text{expanding the proposed definition} \\
\quad = \int_{\mathcal{S}} \bar\sigma(\mathbf{x},\mathbf{y})\,\mathbf{f}(\mathbf{y})\,d\mathbf{y} & \text{since } \bar\sigma(\mathbf{x},\mathbf{y}) \text{ vanishes in } \mathbb{R}^n - \mathcal{S} \\
 & \delta \text{ and derivatives vanish except at } \mathbf{x}
\end{array}$$

Furthermore in $\mathcal{S} - B(\mathbf{x},\varepsilon)$ the Dirac and all its derivatives vanish so that $||\bar\sigma(\mathbf{x}-\mathbf{y})||$ also vanishes.

Specification of the diffusion mechanism over a set of samples.

We consider:

- a vectorial function $\mathbf{f} : \Omega \subset \mathbb{R}^n \rightarrow \mathbb{R}^m$ from a bounded open set Ω of $\mathbb{R}^n$ into $\mathbb{R}^m$; here we consider $\mathbf{f} \in \mathcal{F}$ a set of admissible functions, a dense linear subset of a Hilbert space H, presently the Sobolev function space $W^{s,\infty}(\mathbb{R}^n)$ provided with the norm:

$$||\mathbf{f}||_{s,\infty} = \sup_{0 \geq k \geq s} \left[\sup \, \text{ess}_{|\alpha|=k,\, \mathbf{x} \in \mathbb{R}^n} |\partial^\alpha \mathbf{f}(\mathbf{x})|\right]$$

where sup $\text{ess}_{\mathbf{x}} u(\mathbf{x})$ is the smallest constant, if any, for which $u(\mathbf{x})$ is bounded except in a negligible set of measure zero,

- a diffusion operator $\Delta_{\mathbf{L}(\mathbf{x})}(\mathbf{f}(\mathbf{x})) = \mathbf{div}(\mathbf{L}(\mathbf{x})\,(\nabla \mathbf{f})(\mathbf{x}))$; here $\nabla \mathbf{f}$ is the first order derivative of the function $\mathbf{f}$, while $\mathbf{L} : \mathbb{R}^n \rightarrow S_n \in W^{s,\infty}(\mathbb{R}^n)$, where S_n is the set of square symmetric matrices,

  It verifies the conservation property:

  $$\int_{\mathbb{R}^n} \Delta_{\mathbf{L}} \mathbf{h}(\mathbf{x})\, d\mathbf{x} = 0$$

  in coherence with the physical law of conservation property for diffusion processes: it guaranties that if it reduces the value at one location it will increase elsewhere accordingly, as in a fluid which particles are neither created nor deleted. This guarantees the stability of the numerical computations.

- a finite set of M samples $\{\mathbf{y}_1..\mathbf{y}_j..\mathbf{y}_M\} = \mathcal{M}_M \subset \mathcal{S} = \mathcal{B}(\mathbf{x}, \epsilon)$ in a ball of radius ϵ around $\mathbf{x}$, where the function $\mathbf{f}$ is sampled in a small neighborhood $\mathbf{y}_j \in \mathcal{S}_j \subset \mathcal{B}(\mathbf{x}, \epsilon)$ of each point, i.e.

  $$\mathbf{f}(\mathbf{y}_j) = \int_{\mathcal{S}_j} \mathbf{f}(\mathbf{y})\, \mu_j(\mathbf{y})\, d\mathbf{y}$$

  most often:
  - either [11] a punctual sample $\mu_j(\mathbf{y}) = \delta(\mathbf{y} - \mathbf{y}_j)$ where δ is the Dirac distribution
  - or [33] an average measure (over a pixel or voxel, given a 2D or 3D image) $\mu_j(\mathbf{y}) = 1$ is chosen

  while the assumption is true for any linear measurement.

Considering a biological neural network it is important, due to the huge complexity of the underlying mechanisms, to consider the weakest assumption about how the continuous function is sampled at each point (e.g. allow (i) overlapped neighborhoods $\mathcal{S}_j \cap \mathcal{S}_i \neq \emptyset$ or (ii) partial partitioning $\cup_j \mathcal{S}_j \subset\neq \mathcal{S}$). Here, we propose to rely on the following *summation property*, for some measure $\mu(\mathbf{y})$:

$$\int_{\mathcal{S}} \mathbf{f}(\mathbf{y})\, \mu(\mathbf{y})\, d\mathbf{y} = \sum_j \int_{\mathcal{S}_j} \mathbf{f}(\mathbf{y})\, \mu_j(\mathbf{y})\, d\mathbf{y} + \int_{\mathcal{S} - \cup_j \mathcal{S}_j} \mathbf{f}(\mathbf{y})\, \mu_\bullet(\mathbf{y})\, d\mathbf{y} \tag{A.13}$$

defining $\mu_\bullet(\mathbf{y})$ as the measure density where no sample is available. This formula is verified by any (often implicit) sampling model [30, 11, 9, 13, 33] (e.g. if $\mu_j(\mathbf{y}) = \delta(\mathbf{y} - \mathbf{y}_j)$ with $\mu_\bullet(\mathbf{y}) = 0$ and any small neighborhood $\mathcal{S}_j, \mathbf{y}_j \in \mathcal{S}_j$ or if $\mu_j(\mathbf{y}) = 1$ and $\cup_j \mathcal{S}_j = \mathcal{S}$) and is proposed here to obtain the largest possible representation.

The generalized Degond, Raviat and Mas-Gallic result.

Following Degond, Raviat and Mas-Gallic [30, 11, 33], Proposition 1 (Appendix A.1) characterize unbiased bounded integral approximations of a diffusion operator.

- These approximations are *unbiased* in the sense that they provide a formula directly considering the discrete samples set without any further approximation. This is an improvement with respect to [30, 13, 11] where a "discretization" step is introduced. Here, no additional quadrature of the integral is indeed, since the proposed summation property (A.13) makes the link.

- These approximation are *bounded* in the sense that (contrary to [30, 13, 11]) we assume that the sample set is bounded. This is a reasonable choice which really simplifies the derivations, but -in fact- this is not a limitation and, as made explicit in [30, 13, 11], the reader can easily verify that the same result is obviously true for any $\mathcal{S} \subset \mathbb{R}^n$ providing

$$\textstyle\sum_j \sigma(\mathbf{x}, \mathbf{y}_j)\, \bar{\mu}_j^\alpha(\mathbf{x}) < +\infty, |\alpha| = r + 1$$

- Regarding the fact we limit the approximation up to the rth order, since from a few algebra:

$$\left|\int_{\mathcal{S}} \sigma(\mathbf{x},\mathbf{y})\,(\mathbf{y}-\mathbf{x})^\alpha\,d\mathbf{y}\right| \leq [||\sigma^\epsilon_{kl}||_{0,\infty} \int_{B(0,1)} /(|\alpha|+1)]\,\epsilon^{|\alpha|+1}$$

(here $\int_{B(0,1)} = \frac{\pi^{n/2}}{\Gamma(n/2+1)}$ is the volume of the unit ball) thus exponentially decreases with $|\alpha|$, for high values of $|\alpha|$ *unbiasness* constraints are automatically verified up to a negligible quantity.

- Since there are $\xi_{n,r} = \frac{(n+r)!}{n!\,r!}$ monomial of degree less or equal r with n variables, $\xi_{n,r}$ is also the number of constraints in (**??**). At a given location $\mathbf{x}$, there thus must be at least $M > \xi_{n,r}$ samples for (**??**) to have a solution.

- On one hand, the linear constraints allow the *identification* between the continuous diffusion operator parameter $\mathbf{L}$ and the integral kernel σ. Given a kernel $\sigma()$ the 2nd order linear constraint entirely defines $\mathbf{L}$. On the other hand, not all kernel $\sigma()$ correspond to a diffusion operator $\mathbf{L}$ only those which verify the *unbiasness* constraints. In fact, we do not fix a given solution at this stage but try to characterize a *whole family of kernel implementing a diffusion operator.*

- As demonstrated in [11] in a general case, the convergence of the operator approximation (**??**) yields the convergence of approximate solutions of partial differential equations using the related diffusion operator because, in the present case, $\mathbf{L}$ is a positive operator.

- Here, the *unbiasness* constraints are local for each $\mathbf{x}_i$. Biased kernels correspond to other differential operators. If unbiasness is not verified for $|\alpha| > 2$, higher order differential terms are included. As mentioned already high-order terms become anyway negligible. This also means that the present approach is easily generalizable to other differential operators as sometimes considered in cognitive processes [22]. Furthermore, general 1st and 2nd order differential operator are easily generated using this approach (see [33] for an explicit development).

- As made explicit in the following derivation, conditions (A.8) are not the weakest constraints for σ_{ij} to be average symmetric and unbiased. We however make the choice to state unbiasness constraints which are local.

The optimal unbiased integral approximation of a diffusion operator.

A step further, following [33], among all unbiased integral approximations of a diffusion operator, it is very easy to define the "sharpest" operator. i.e. as close as possible to the "exact" diffusion operator, which writes:

$$min_\sigma \int_{\mathcal{S}} ||\sigma(\mathbf{x},\mathbf{y})||\,d\mathbf{x}\,d\mathbf{y}$$

The induced distance writes:

$$\begin{array}{lll} & \int_{\mathcal{S}-B(\mathbf{x},\varepsilon)} ||\bar\sigma(\mathbf{x},\mathbf{y}) - \sigma(\mathbf{x},\mathbf{y})||\,d\mathbf{x}\,d\mathbf{y} & \text{for } \varepsilon > 0 \\ = & \int_{\mathcal{S}-B(\mathbf{x},\varepsilon)} ||\sigma(\mathbf{x},\mathbf{y})||\,d\mathbf{x}\,d\mathbf{y} & \text{since } \bar\sigma(\mathbf{x},\mathbf{y}) \text{ vanishes in } \mathcal{S}-B(\mathbf{x},\varepsilon) \\ \rightarrow & \int_{\mathcal{S}} ||\sigma(\mathbf{x},\mathbf{y})||\,d\mathbf{x}\,d\mathbf{y} & \text{since } \sigma(\mathbf{x},\mathbf{y}) \text{ is well defined at } \mathbf{x}=\mathbf{y} \\ & & \text{we can take } \varepsilon = 0 \end{array}$$

For instance, if we assume that we compute the diffusion at N samples points $\mathbf{x}_i$ so that we need to define the $N \times M$ values $\sigma(\mathbf{x}_i, \mathbf{y}_j)$ the canonical choice is to choose $\sigma(\mathbf{x}, \mathbf{y}_j) = \sum_i \sigma(\mathbf{x}_i, \mathbf{y}_j)\,\delta(\mathbf{x} - \mathbf{x}_i)$ (i.e. punctual values of the kernel) the problem being thus to minimize $\sum_{ij} ||\sigma(\mathbf{x}_i, \mathbf{y}_j)||$ with the constraints (**??**) in which $\int_{\mathcal{S}} \sigma(\mathbf{y},\mathbf{x})\,d\mathbf{y} = \sum_i \sigma(\mathbf{x}_i, \mathbf{x})$.

If we consider a quadratic norm (i.e. minimize the operator variance or 2nd order inertia) this is a quadratic minimization problem with linear constraints with a closed-form solution (automatically generated using a piece of symbolic calculator code, e.g. [33]).

Appendix B

Coding: Maple and Java sources

B.1 Maple specifications

B.1.1 Maple specifications for the automatic cortical map generator

```
************************************************************************************************************************************
************************************************NeuralMapCompiler.mpl***************************************************************
************************************************************************************************************************************
with(linalg): with(CodeGeneration):

'NeuralMapCompiler/help' := TEXT(
"",
" NeuralMapCompiler - Generates the linearized differential equation of a cortical map computation defined by a variational criterion.",
" Calling Sequence",
" NeuralMapCompiler(definition..)",
" Parameters",
" definition: equations of the form name = value defining the cortical map computation, with:",
"",
" p = Cortical map location vector, e.g. Vector(2) for 2D images.",
" w = Input vector at each point p.",
" v = Output vector p -> v[p], vectorial real vectorial function R^(dim(p)) -> R^(dim(v)).",
" these 3 elements being mandatory, the other being optional:",
"",
" P = Matrix(dim(w), dim(v))",
"     default to Matrix(dim_w, dim_v, shape = identity),",
"     is the linear relation between the map input w and the map output v.",
" Lambda = Matrix(dim(w), dim(w))",
"          default to Matrix(dim_w, dim_w, shape = identity),",
"          is the input attach semi-definite symmetric positive metric.",
" L = Array(d2met, 1..dim(v), 1..dim(v), 1..dim(p), 1..dim(p)),",
"     default to Array(d2met, 1..dim_v, 1..dim_v, 1..dim_p, 1..dim_p, (x,y,i,j) -> if x = y and i = j then 1 else 0 fi),",
"     is the output definite symmetric positive diffusion tensor at each neural locus,",
" with |grad(v)|_L = >_x,y,i,j grad(v)^x_i L_xy^ij grad(v)^y_j while grad(v) is a Matrix(dim_v, dim_p).",
" Phi = Real function R -> R (do NOT use symbol p, w, v to define it)",
"       default to x -> x,",
"       defines non-linear diffusion and control the smoothness modulation of the regularization term.",
" Psi = Vectorial real function R^(dim(v)) -> R (do NOT use symbol p, w, v to define it)",
"       default to x -> 0,"
"       defines a weakly constraint of the map output to be closer to a given set of solutions.",
" c = Vectorial real vectorial function R^(dim(v)) -> R^(dim(c)) (do NOT use symbol p, w, v to define it),",
"     default to x -> Vector(0),",
"     defines structural constraints, enforcing the solution to belong to a manifold defined by such implicit equations.",
"",
" r = integer,"
"     default to 2,"
"     diffusion integral approximation order, r >= 2 since diffusion is a 2nd order operator.",
" s = integer,"
"     default to the minimum compatible value,"
"     neural site size, here hyper-cubes of size {-s .. s}^n are considered.",
"",
" on output, default values and the following elements are added to the initial definitions",
" dim_p = dim(p).",
" dim_w = dim(w).",
" dim_v = dim(v).",
" M = (2 * s + 1)^dim_p, the neighborhood size.",
"",
" sigma  = Vector(1..dim_s, k -> Matrix(dim_v, dim_v)),"
"          the integral approximation of the diffusion operator at each neighborhood point."
" lambda = Real expression, the numerical expression of the |grad(v)|_L."
" zeta = Vector(dim_v), the symbolic expression of grad(Psi)."
"",
" A = P^T Lambda P + >_k sigma[k].",
" b = P^T Lambda w + >_k sigma[k] v(p[k]) - grad(Psi).",
" C = grad(c(v), v), (only defined if c() is defined).",
" Q = C^T (C C^T)^(-1), (only defined if c() is defined).",
"",
" beta = Q c(v).",
" alpha = [I - Q c] [A^(-1) b - v]."
" v1 = v - epsilon beta + upsilon alpha.",
" d = A v - b = d l(v(t)) / d t."
" l = l(v), criterion value at point p.",
" constr = |c(v)|, constraint magnitude at point p.",
"",
" V = vector(dim_p) (inert version of v[p]).",
"",
" in addition code generation is derived if the following definition is given:",
" file = The file name.",
" code = Either Java (default) or C.",
" what = list(symbol),",
"        default to [] empty code,",
"        the list of symbol to generate in the code."
"",
" Description",
" The cortical map computation can be modeled as a variational problem (more precisely the minimization of a criterion to optimize), ",
" considering regularization mechanisms and introducing non-linear constraints in the specification.",
" Local solution can be implemented-in the general case using an analog dynamic neural network.",
" This module derives the local update equation of this network in order to solve the optimization problem.",
" It returns the augmented definition list.",
"",
""):

NeuralMapCompiler := proc()
  local def, contains, append: def := [args]:
  # Returns true if name is defined in def
  contains := proc(def :: list(symbol = anything), name :: symbol)
    local ivalue: ivalue := subs(def, name): ivalue <> name
  end:
  # Appends in def the name = value definition if not yet defined, else verifies the condition
  append := proc(def :: list(symbol = anything), name :: symbol, value, condition)
    if   not contains(def, name) then [op(def), name = eval(subs(def, eval(subs(def, value))))]
    elif not evalb(eval(subs(def, condition))) then ERROR(cat("When defining ",name,": ",convert(''condition'',string)," is not verified"))
    else def
    fi
  end:

  # Adds the problem dimension
  def := append(def, 'dim_p',  'vectdim(p)', 'dim_p = vectdim(p)'):
  def := append(def, 'dim_w',  'vectdim(w)', 'dim_w = vectdim(w)'):
  def := append(def, 'dim_v',  'vectdim(v(p))', 'dim_v = vectdim(v(p))'):
  def := append(def, 'V',      'vector(dim_v, i -> V[i])', true):

  # Adds default definition
  def := append(def, 'P',      'Matrix(dim_w, dim_v, shape = identity)', 'rowdim(P) = dim_w and coldim(P) = dim_v'):
  def := append(def, 'Lambda', 'Matrix(dim_w, dim_w, shape = identity)', 'rowdim(Lambda) = dim_w or coldim(Lambda) = dim_w'):
  def := append(def, 'L',      'Array(d2met, 1..dim_v, 1..dim_v, 1..dim_p, 1..dim_p, (x,y,i,j) -> if x = y and i = j then 1 else 0 fi)',
                               '[op(2,L)] = [1..dim_v, 1..dim_v, 1..dim_p, 1..dim_p]'):
  def := append(def, 'Phi',    'x -> x', true):
  def := append(def, 'Psi',    'x -> 0', true):
  def := append(def, 'r',      2, true):
  def := append(def, 's',      '`NeuralMapCompiler/definition2s_min`(dim_p, r)', 's >= `NeuralMapCompiler/definition2s_min`(dim_p, r)'):
  def := append(def, 'M',      '(2 * s + 1)^dim_p', true):

  # Adds integral approximation and related definition
  def := append(def, 'sigma',  '`NeuralMapCompiler/definition2sigma`(dim_p, dim_v, r, s, M, p, L, Phi, lambda)', true):
```

```
  def := append(def, 'lambda', '`NeuralMapCompiler/definition2lambda`(dim_p, dim_v, p, L)', true):
  def := append(def, 'zeta',   'subs(V=v(p), grad(Psi(V), V))', true):

  # Computes the iteration elements
  def := append(def, 'A',      'evalm(transpose(P) &* Lambda &* P + sum('sigma[k]', k = 1 .. M))', true):
  def := append(def, 'b',      'evalm(transpose(P) &* Lambda &* w + sum('sigma[k] &*
                                v(p + Vector(dim_p, `NeuralMapCompiler/induxes`(k, s, dim_p)))', k = 1 .. M) - zeta)', true):
  def := append(def, 'd',      'evalm(A &* v(p) - b)', true):
  def := append(def, 'l',      'evalm(transpose(convert(evalm(P &* v(p) - w), vector)) &* Lambda &* (P &* v(p) - w)) + Phi(lambda)
                                + 2 * Psi(v(p))', true):
  if contains(def, c) then
    def := append(def, 'C',      'subs(V=v(p), evalm(jacobian(c(V), V)))', true):
    def := append(def, 'Q',      'evalm(transpose(C) &* inverse(C &* transpose(C)))', true):

    def := append(def, 'beta',   'evalm(Q &* c(v(p)))', true):
    def := append(def, 'alpha',  'evalm((1 - Q &* C) &* (A^(-1) &* b - v(p)))', true):
    def := append(def, 'constr', 'evalm(transpose(convert(c(v(p)), vector)) &* c(v(p)))', true):
  else
    def := append(def, 'beta',   'vector(dim_v, i -> 0)', true):
    def := append(def, 'alpha',  'evalm(A^(-1) &* b - v(p))', true):
    def := append(def, 'constr',      '0', true):
  fi:
  def := append(def, 'v1',       'evalm(v(p) - epsilon * beta + upsilon * alpha)'):


  # Generates the code if required
  if contains(def, 'file') then
    def := append(def, 'code', 'Java', 'code = Java or code = C'):
    def := append(def, 'what', '[]', true):
    `NeuralMapCompiler/generate`(subs(def, code), subs(def, file), map(name -> name = subs(def, name), subs(def, what))):

  fi:

  # In order to return only a subset (for debug)
  map(name -> name = subs(def, name), [v1])

end proc:

########################################################################################################################################

 # Returns the minimal s given r and dim_p

`NeuralMapCompiler/definition2s_min` := proc(dim_p :: integer, r :: integer)
  local M, s:
  M := (dim_p + r)! / dim_p! / r! - dim_p * (dim_p + 1) / 2:
  s := (M^(1/dim_p) - 1) / 2: # since M = (2 * s + 1)^dim_p
  ceil(s)
end proc:

 # Returns the numerical approximation of the gradient norm |grad(v)|_L

`NeuralMapCompiler/definition2lambda` := proc(dim_p :: integer, dim_v ::integer, p :: Vector, L :: Array)
  local Dv, i, j, x, y:

  # Computes numerically Dv = grad(v)
  Dv := Matrix(dim_v, dim_p, (x, i) -> (v(p + Vector(dim_p, j -> if i = j then 1 else 0 fi))[x]
              - v(p + Vector(dim_p, j -> if i = j then -1 else 0 fi))[x]) / 2):

  # Computes lambda = |grad(v)|_L = >_x,y,i,j grad(v)^x_i L_xy^ij grad(v)^y_j
  add(add(add(add(Dv[x, i] * L[x, y, i, j] * Dv[y, j], x = 1..dim_v), y = 1..dim_v), i = 1..dim_p), j = 1..dim_p)
end proc:

 # Returns the integral approximation of the diffusion operator sigma

`NeuralMapCompiler/definition2sigma` := proc(dim_p :: integer, dim_v ::integer, r :: integer, s :: integer,
                                              M :: integer, p :: Vector, L :: Array, Phi :: operator, lambda)
  local D0, D1, D2, eq, sigma, i, j, k, x, y:

  # Calculates D0 = Phi'(lambda) L
  D0 :=  Array(d2met, 1..dim_v, 1..dim_v, 1..dim_p, 1..dim_p, (x, y, i, j) -> D(Phi)(lambda) * L[x, y, i, j]):

  # Derives the equations for sigma
  eq := {}: for x to dim_v do for y to dim_v do
    D2 := Matrix(dim_p, dim_p, (i, j) -> D0[x, y, i, j]):
```

```
    D1 := Vector(dim_p, i -> diverge(convert(Vector(dim_p, j -> D2[i, j]), vector), convert(p, vector))):
    eq := eq union `NeuralMapCompiler/invedwards`(D1, D2, sigma(x, y), r, dim_p, s):
  od od:

  # Reorganize sigma as an array of matrix
  Vector(M, k -> Matrix(dim_v, dim_v, (i, j) ->subs(eq, sigma(i, j) (k))))
end proc:

####################################################################################################################################

 # Returns the Edwards linear equations linking a 2nd order differential operator D1 D2 and its weights sigma
 #  up to r-order for a n-dimensional problem in an hypercube of size +- s

`NeuralMapCompiler/edwards` := proc(D1, D2, sigma, r :: integer, n :: integer, s :: integer)
  local pow, ind, lhs, rhs, mu;
  # calculates the power of vector x with respect to multi-indices
  pow := (x :: vector, d :: list(integer)) -> convert(map((i, x, d) -> x[i]^d[i], [$1..min(vectdim(x),nops(d))], x, d), `*`):
  # generates the momentum over a hyper-pixel: here the measure model is punctual (mu = Dirac)
  mu  := (x, d) -> pow(vector(x), d):
  # returns the indexes of non-zero values of a multi-index [d_1 .. d_n]
  ind := proc(i :: list(integer)) local j, l, k: l := NULL: for j to nops(i) do l := l,seq(j, k = 1..i[j]) od: end:
  # defines equation lhs and rhs
  lhs := (i, r) -> if r = 1 then D1[ind(i)] elif r = 2 then 2 * D2[ind(i)] else 0 fi:
  rhs := (i, n, s) -> sum('sigma(j) * mu(`NeuralMapCompiler/induxes`(j, s, n), i)', j = 1 .. (2*s+1)^n):
  # returns the Edwars equations
  {seq(op(map((i, r, n, s) -> lhs(i, r) = rhs(i, n, s), `NeuralMapCompiler/indexes`(k, n), k, n, s)), k = 1 .. r)}
end proc:

 # Returns the inverse Edwards linear equations linking a 2nd order differential operator D1 D2 and its optimal weights sigma
 #  up to r-order for a n-dimensional problem in an hypercube of size +- s

`NeuralMapCompiler/invedwards` := proc(D1, D2, sigma, r :: integer, n :: integer, s :: integer)
  local c: c:= sum(sigma(j)^2, j = 1 .. (2*s+1)^n): `NeuralMapCompiler/leastsquare`(c, `NeuralMapCompiler/edwards`(D1, D2, sigma, r, n, s),
                                                                                  indets(c, function))
end proc:

 # Returns the n-dimensional set of multi-indexes d = [d_1 .. d_n] of degree r, i.e. with r = sum(d_i, i=1..n)

`NeuralMapCompiler/indexes` := proc(r :: integer, n :: integer)
   option remember:
   if r = 0 then
    {[0$i=1..n]}
   else
    map((d, n) -> op(map((i, n, d) -> map((j, i, d) -> if i = j then d[j] + 1 else d[j] fi, [$1..n], i, d), {$1..n}, n, d)),
                {op(procname(r-1, n))}, n)
   fi
 end proc:

 # Converts a unique index 1..(2s+1)^n to indexes d = [d_1 .. d_n] with -s <= d_i <= s

`NeuralMapCompiler/induxes` := proc(i :: integer, s :: integer, n :: integer)
   map((k, i, s) -> (trunc((i-1) / (2*s+1)^(k-1)) mod (2*s+1)) - s, [$1..n], i, s)
 end proc:

####################################################################################################################################

 # Generates the code into file from the definition

`NeuralMapCompiler/generate` := proc(code, file :: string, def :: list(symbol = anything))
  # previous file is cleared avoiding spurious append
  system(cat("/bin/rm -f ",file)):
  # double vector, matrix and scalar declaration is added (this is a fragible action)
  if code = eval(Java) then
    system(StringTools[RegSubs](",'" = ";'", cat("echo 'double",
          op(map(e -> if   type(op(2, e), vector) then cat(" ",op(1, e),"[] = new double[",vectdim(op(2,e)),"],")
                      elif type(op(2, e), matrix) then cat(" ",op(1, e),"[][] = new double[",coldim(op(2, e)),"][",rowdim(op(2, e)),"],")
                                                  else cat(" ",op(1, e),",")
                      fi, def)), "' > ",file)))
  fi:
  # code generation is mapped on the the definition list
  map(e -> code(map(evalf, op(2,e)),  resultname=op(1,e), output=file), def):
end proc:

 # Returns the extremum of a quadratic criterion cri with linear constraints ctr with respect to variables vars
```

```
`NeuralMapCompiler/leastsquare` := proc(cri, ctr :: set, vars :: set)
   local v, s, l, e, c, r:
   v := {'cat(_Z_v_,i)'$i=1..nops(vars)}: s := {'vars[i]=v[i]'$i=1..nops(v)}: l := {'cat(_Z_l_,i)'$i=1..nops(ctr)}: v := v union l:
   e := map(u -> if type(u, `=`) then op(1, u) - op(2, u) else u fi, ctr):  c := subs(s, cri + sum(l[i] * e[i], i = 1 .. nops(e))):
   r := solve(convert(linalg[grad](c, convert(v, list)), set), v): if r <> NULL then subs(r, s) fi
  end:
```

B.1.2 Parameters specifications for a color image filtering

```
*******************************************************************************************************************************
**************************************************RgbNeuralMap.mpl*************************************************************
*******************************************************************************************************************************

read "NeuralMapCompiler.mpl":

ll := NeuralMapCompiler(
        p = Vector([x, y]),
        w = Vector([red0[ij + 1], green0[ij + 1], blue0[ij + 1]]),
        v = (proc(p) local k: global ij: k := subs({x = 0, y =0}, ij + 1 + p[1] + p[2] * width): Vector([red[k], green[k], blue[k]]) end),
        file = "RgbNeuralMap.inc",
        what = [alpha, beta, l , constr],
        NULL);
```

B.1.3 Parameters specifications for a color image filtering with constraint

```
*******************************************************************************************************************************
**************************************************IsoNeuralMap.mpl*************************************************************
*******************************************************************************************************************************

read "NeuralMapCompiler.mpl":

ll := NeuralMapCompiler(
        p = Vector([x, y]),
        w = Vector([red0[ij + 1], green0[ij + 1], blue0[ij + 1]]),
        v = (proc(p) local k: global ij: k := subs({x = 0, y =0}, ij + 1 + p[1] + p[2] * width): Vector([red[k], green[k], blue[k]]) end),
        c = (x -> Vector(1, i -> x[1] + x[2] + x[3] - 255)),
        file = "IsoNeuralMap.inc",
        what = [alpha, beta, l, constr],
        NULL);
```

B.2 Java implementation

B.2.1 Java code generated from the Maple specification for the color image filtering

```
*******************************************************************************************************************************
**************************************************RgbNeuralMap.mjava***********************************************************
*******************************************************************************************************************************

public class RgbNeuralMap extends RgbDiffusionFilter {
  /**/public RgbNeuralMap() { super(2, 0); }

  /**/public myParam getRGB(int ij, int width, double eps, double upsilon, int red0[], int green0[], int blue0[], int red[],
                          int green[], int blue[]){
      cat RgbNeuralMap.inc`;
      return new myParam(alpha, beta, constr, l);
  }
}

*******************************************************************************************************************************
**************************************************RgbNeuralMap.java************************************************************
*******************************************************************************************************************************

public class RgbNeuralMap extends RgbDiffusionFilter {
  /**/public RgbNeuralMap() { super(2, 0); }

  /**/public myParam getRGB(int ij, int width, double eps, double upsilon, int red0[], int green0[], int blue0[], int red[],
                            int green[], int blue[]){
  double alpha[] = new double[3], beta[] = new double[3], l, constr;
alpha[0] = 0.2941176471e0 * red0[ij] + 0.1176470588e0 * red[ij - width - 1] + 0.5882352941e-1 * red[ij - width]
  + 0.1176470588e0 * red[ij + 1 - width] + 0.5882352941e-1 * red[ij - 1] + 0.5882352941e-1 * red[ij + 1] + 0.1176470588e0
  * red[ij + width - 1] + 0.5882352941e-1 * red[ij + width] + 0.1176470588e0 * red[ij + 1 + width] - 0.1e1 * red[ij];
alpha[1] = 0.2941176471e0 * green0[ij] + 0.1176470588e0 * green[ij - width - 1] + 0.5882352941e-1 * green[ij - width] + 0.1176470588e0
  * green[ij + 1 - width] + 0.5882352941e-1 * green[ij - 1] + 0.5882352941e-1 * green[ij + 1] + 0.1176470588e0 * green[ij + width - 1]
  + 0.5882352941e-1 * green[ij + width] + 0.1176470588e0 * green[ij + 1 + width] - 0.1e1 * green[ij];
alpha[2] = 0.2941176471e0 * blue0[ij] + 0.1176470588e0 * blue[ij - width - 1] + 0.5882352941e-1 * blue[ij - width] + 0.1176470588e0
  * blue[ij + 1 - width] + 0.5882352941e-1 * blue[ij - 1] + 0.5882352941e-1 * blue[ij + 1] + 0.1176470588e0 * blue[ij + width - 1]
  + 0.5882352941e-1 * blue[ij + width] + 0.1176470588e0 * blue[ij + 1 + width] - 0.1e1 * blue[ij];
l = Math.pow(red[ij] - 0.1e1 * red0[ij], 0.2e1) + Math.pow(green[ij] - 0.1e1 * green0[ij], 0.2e1) + Math.pow(blue[ij] - 0.1e1
  * blue0[ij], 0.2e1) + Math.pow(0.5000000000e0 * red[ij + 1] - 0.5000000000e0 * red[ij - 1], 0.2e1) + Math.pow(0.5000000000e0
* green[ij + 1] - 0.5000000000e0 * green[ij - 1], 0.2e1) + Math.pow(0.5000000000e0 * blue[ij + 1] - 0.5000000000e0 * blue[ij - 1], 0.2e1)
+ Math.pow(0.5000000000e0 * red[ij + width] - 0.5000000000e0 * red[ij - width], 0.2e1) + Math.pow(0.5000000000e0 * green[ij + width]
- 0.5000000000e0 * green[ij - width], 0.2e1) + Math.pow(0.5000000000e0 * blue[ij + width] - 0.5000000000e0 * blue[ij - width], 0.2e1);
constr = 0.0e0;;
    return new myParam(alpha, beta, constr, l);
  }
}
```

B.2.2 Java code generated from the `Maple` specification for the color image filtering with constraint

```
****************************************************************************************************************************************
*******************************************************IsoNeuralMap.mjava***************************************************************
****************************************************************************************************************************************

public class IsoNeuralMap extends RgbDiffusionFilter {
  /**/public IsoNeuralMap() { super(2, 0); }

  /**/public myParam getRGB(int ij, int width, double eps, double upsilon, int red0[], int green0[], int blue0[], int red[],
                            int green[], int blue[]){
  `cat IsoNeuralMap.inc`;
    return new myParam(alpha, beta, constr, l);
  }
}

****************************************************************************************************************************************
*******************************************************IsoNeuralMap.java****************************************************************
****************************************************************************************************************************************

public class IsoNeuralMap extends RgbDiffusionFilter {
  /**/public IsoNeuralMap() { super(2, 0); }

  /**/public myParam getRGB(int ij, int width, double eps, double upsilon, int red0[], int green0[], int blue0[], int red[],
                            int green[], int blue[]){
  double alpha[] = new double[3], beta[] = new double[3], l, constr;
alpha[0] = 0.7843137255e-1 * red[ij - width - 1] - 0.3921568627e-1 * blue[ij - width - 1] - 0.3921568627e-1 * green[ij - width - 1]
  - 0.3921568627e-1 * blue[ij + width - 1] - 0.3921568627e-1 * green[ij + 1 - width] + 0.1960784314e0 * red0[ij] - 0.9803921569e-1
  * green0[ij] - 0.9803921569e-1 * blue0[ij] + 0.3333333333e0 * green[ij] - 0.3921568627e-1 * green[ij + 1 + width] + 0.3333333333e0
  * blue[ij] + 0.7843137255e-1 * red[ij + width - 1] + 0.7843137255e-1 * red[ij + 1 + width] - 0.6666666667e0 * red[ij] - 0.1960784314e-1
  * green[ij - 1] + 0.3921568627e-1 * red[ij - 1] - 0.1960784314e-1 * green[ij + 1] - 0.1960784314e-1 * blue[ij + 1] - 0.1960784314e-1
  * blue[ij - 1] - 0.3921568627e-1 * blue[ij + 1 + width] - 0.3921568627e-1 * green[ij + width - 1] - 0.3921568627e-1 * blue[ij + 1 - width]
  + 0.3921568627e-1 * red[ij + 1] + 0.3921568627e-1 * red[ij + width] - 0.1960784314e-1 * green[ij + width] - 0.1960784314e-1
  * blue[ij + width] + 0.3921568627e-1 * red[ij - width] - 0.1960784314e-1 * green[ij - width] - 0.1960784314e-1 * blue[ij - width]
  + 0.7843137255e-1 * red[ij + 1 - width];
alpha[1] = -0.3921568627e-1 * red[ij - width - 1] - 0.3921568627e-1 * blue[ij - width - 1] + 0.7843137255e-1 * green[ij - width - 1]
  - 0.3921568627e-1 * blue[ij + width - 1] + 0.7843137255e-1 * green[ij + 1 - width] - 0.9803921569e-1 * red0[ij] + 0.1960784314e0
  * green0[ij] - 0.9803921569e-1 * blue0[ij] - 0.6666666667e0 * green[ij] + 0.7843137255e-1 * green[ij + 1 + width] + 0.3333333333e0
  * blue[ij] - 0.3921568627e-1 * red[ij + width - 1] - 0.3921568627e-1 * red[ij + 1 + width] + 0.3333333333e0 * red[ij]
  + 0.3921568627e-1 * green[ij - 1] - 0.1960784314e-1 * red[ij - 1] + 0.3921568627e-1 * green[ij + 1] - 0.1960784314e-1
  * blue[ij + 1] - 0.1960784314e-1 * blue[ij - 1] - 0.3921568627e-1 * blue[ij + 1 + width] + 0.7843137255e-1 * green[ij + width - 1]
  - 0.3921568627e-1 * blue[ij + 1 - width] - 0.1960784314e-1 * red[ij + 1] - 0.1960784314e-1 * red[ij + width] + 0.3921568627e-1
  * green[ij + width] - 0.1960784314e-1 * blue[ij + width] - 0.1960784314e-1 * red[ij - width] + 0.3921568627e-1 * green[ij - width]
  - 0.1960784314e-1 * blue[ij - width] - 0.3921568627e-1 * red[ij + 1 - width];
alpha[2] = -0.3921568627e-1 * red[ij - width - 1] + 0.7843137255e-1 * blue[ij - width - 1] - 0.3921568627e-1 * green[ij - width - 1]
  + 0.7843137255e-1 * blue[ij + width - 1] - 0.3921568627e-1 * green[ij + 1 - width] - 0.9803921569e-1 * red0[ij] - 0.9803921569e-1
  * green0[ij] + 0.1960784314e0 * blue0[ij] + 0.3333333333e0 * green[ij] - 0.3921568627e-1 * green[ij + 1 + width] - 0.6666666667e0
  * blue[ij] - 0.3921568627e-1 * red[ij + width - 1] - 0.3921568627e-1 * red[ij + 1 + width] + 0.3333333333e0 * red[ij]
  - 0.1960784314e-1 * green[ij - 1] - 0.1960784314e-1 * red[ij - 1] - 0.1960784314e-1 * green[ij + 1] + 0.3921568627e-1
  * blue[ij + 1] + 0.3921568627e-1 * blue[ij - 1] + 0.7843137255e-1 * blue[ij + 1 + width] - 0.3921568627e-1 * green[ij + width - 1]
  + 0.7843137255e-1 * blue[ij + 1 - width] - 0.1960784314e-1 * red[ij + 1] - 0.1960784314e-1 * red[ij + width] - 0.1960784314e-1
  * green[ij + width] + 0.3921568627e-1 * blue[ij + width] - 0.1960784314e-1 * red[ij - width] - 0.1960784314e-1 * green[ij - width]
  + 0.3921568627e-1 * blue[ij - width] - 0.3921568627e-1 * red[ij + 1 - width];
beta[0] = 0.3333333333e0 * red[ij] + 0.3333333333e0 * green[ij] + 0.3333333333e0 * blue[ij] - 0.85e2;
beta[1] = 0.3333333333e0 * red[ij] + 0.3333333333e0 * green[ij] + 0.3333333333e0 * blue[ij] - 0.85e2;
beta[2] = 0.3333333333e0 * red[ij] + 0.3333333333e0 * green[ij] + 0.3333333333e0 * blue[ij] - 0.85e2;
l = Math.pow(red[ij] - 0.1e1 * red0[ij], 0.2e1) + Math.pow(green[ij] - 0.1e1 * green0[ij], 0.2e1) + Math.pow(blue[ij] - 0.1e1
  * blue0[ij], 0.2e1) + Math.pow(0.5000000000e0 * red[ij + 1] - 0.5000000000e0 * red[ij - 1], 0.2e1) + Math.pow(0.5000000000e0
  * green[ij + 1] - 0.5000000000e0 * green[ij - 1], 0.2e1) + Math.pow(0.5000000000e0 * blue[ij + 1] - 0.5000000000e0 * blue[ij - 1], 0.2e1)
  + Math.pow(0.5000000000e0 * red[ij + width] - 0.5000000000e0 * red[ij - width], 0.2e1) + Math.pow(0.5000000000e0 * green[ij + width]
  - 0.5000000000e0 * green[ij - width], 0.2e1) + Math.pow(0.5000000000e0 * blue[ij + width] - 0.5000000000e0 * blue[ij - width], 0.2e1);
constr = Math.pow(red[ij] + green[ij] + blue[ij] - 0.255e3, 0.2e1);;
    return new myParam(alpha, beta, constr, l);
  }
}
```

B.2.3 Java implementation of the RGB Filter

```
*****************************************************************************************************************************************
*************************************************** RgbDiffusionFilter.java*************************************************************
*****************************************************************************************************************************************

/***************************************************************************
 *   schemla@sophia.inria.fr, Copyright (C) 2006.  All rights reserved.    *
 ***************************************************************************/

import imp.ima.RgbImage;
import imp.ima.util.RgbFilter;

/** Encapsulates a RGB color image diffusion filter. */
public class RgbDiffusionFilter extends RgbFilter {

  /** Creates a RgbDiffusionFilter.
   * @param borderSize @optional<0> Border size: width/height in pixel on the image border where the filter must not be computed.
   * @param borderValue @optional<0> Border RGB value: value to be set on border where the filter is not computed.
   */
  public RgbDiffusionFilter(int borderSize, int borderValue) { this.borderSize = borderSize; this.borderValue = borderValue; }
  /**/public RgbDiffusionFilter(int borderSize) { this(borderSize, 0); }
  /**/public RgbDiffusionFilter() { this(0, 0); }
  private int borderSize, borderValue;

  /**/public void filterRGB(int src[], int dst[], int width, int height) {
    int size = width * height;

    // Creates the source initial values
    int red0[] = new int[size], green0[] = new int[size], blue0[] = new int[size];
    for (int ij = 0; ij < size; ij++) { int rgb = src[ij]; red0[ij] = (rgb >> 16) & 0xff;
                                        green0[ij] = (rgb >>  8) & 0xff;
                                        blue0[ij] = (rgb      ) & 0xff; }

    // Initializes the loop parameters and buffers
    double epsilon = 0, constr, cn_1 = Double.MAX_VALUE;
    double upsilon = 0, en, en_1 = Double.MAX_VALUE;
    int tmp[] = new int[size];
    for (int ij = 0; ij < dst.length; ij++) { dst[ij] = src[ij]; }

    // Iterative estimation loop
    for(int iteration = 0; iteration < 30; iteration++) {
      System.out.println(iteration);

      // Calculates v_n+1 (in tmp[]), c(v_n+1) (in cn) <- from- v_n (in dst[]) and w (in src[]) and en =|| v_n+1 - v_n||
      {
// Initializes v_n buffers, ln and cn
int red[] = new int[size], green[] = new int[size], blue[] = new int[size];
for (int ij = 0; ij < size; ij++) { int rgb = dst[ij]; red[ij] = (rgb >> 16) & 0xff;
                                                       green[ij] = (rgb >>  8) & 0xff;
                                                       blue[ij] = (rgb      ) & 0xff; }
```

```
constr = en = 0;

// Loop on the image
int ij0 = borderSize * (1 + width), i0 = borderSize, i1 = width - borderSize, j0 = borderSize, j1 = height - borderSize,
        ij2 = 2 * borderSize;
for (int ij = ij0, j = j0; j < j1; ij+=ij2, j++)
  for(int i = i0; i < i1; ij++, i++) {
    myParam param_ij = getRGB(ij, width, epsilon, upsilon, red0, green0, blue0, red, green, blue);

    // Updates ln and e and tmp[]
    double red_beta  = param_ij.getBeta()[0],    green_beta = param_ij.getBeta()[1],    blue_beta = param_ij.getBeta()[2];
    double red_alpha = param_ij.getAlpha()[0],  green_alpha = param_ij.getAlpha()[1],  blue_alpha = param_ij.getAlpha()[2];

    constr += Math.abs(param_ij.getConstraint());

    double red_tmp   = red[ij]   - epsilon * red_beta   + upsilon * red_alpha;
    double green_tmp = green[ij] - epsilon * green_beta + upsilon * green_alpha;
    double blue_tmp  = blue[ij]  - epsilon * blue_beta  + upsilon * blue_alpha;

    tmp[ij] = sat(red_tmp) << 16 | sat(green_tmp) << 8 | sat(blue_tmp);

    en += sqr(red_tmp - red[ij]) + sqr(green_tmp - green[ij]) + sqr(blue_tmp - blue[ij]);
  }

en /= dst.length; en = Math.sqrt(en);
constr /= dst.length;
      }

      System.out.println(" en = " +en+" upsilon = " +upsilon);
      System.out.println(" c = " +constr+" epsilon = " +epsilon);

      // Manage the estimation update
      if (iteration == 0) {

// Since epsilon = upsilon = 0 a the 0th iteration: we simply obtain initial values for
en_1 = en; cn_1 = constr;
// Let us start with some contsraint minimization only
epsilon = 0.5; upsilon = 0;

      } else {
// If true we initiate a new iteration, else we backtrack to the previous value
boolean update = true; int state = 0; // (state is just for debug)

if (constr < cn_1) { state = 1;
  // If the constraint decreases we carry on, accelerating on epsilon
  epsilon = Math.sqrt(epsilon); upsilon = 0;

} else if (epsilon > 1e-3) { state = 2;
  // If the constraint does not decrease and is higher than expected, we reduce the rate
  epsilon /= 2; upsilon = 0; update = false;

  // If epsilon is too small we stop the process
  if (epsilon < 1e-6) { state = 3;
    iteration = 1000000;
  }
} else {
  // The constraint is satisfied we decrease the criterion
  if (upsilon == 0) { state = 4;
    // Let us start with some criterion minimization
    upsilon = 0.5;
  } else if (en < en_1) { state = 5;
    // If the criterion decreases we carry on, accelerating on epsilon
    upsilon = Math.sqrt(upsilon);
  } else { state = 6;
    // If the criterion does not decrease, we reduce the rate
    upsilon /= 2; epsilon /= 2; update = false;

    // If upsilon is too small we stop the process
    if (upsilon < 1e-6) { state = 7;
      iteration = 1000000;
    }
  }
}
```

```
System.out.println(" update = " +update+" state : "+state);

// Here we update buffer and indicator
en_1 = en; cn_1 = constr;
if (update) {
  for (int ij = 0; ij < dst.length; ij++) { dst[ij] = tmp[ij];}
}

      }
    }
  }

    private static int sat(double x) { return (int) (x < 0 ? 0 : x > 255 ? 255 : x); }
    private static double sqr(double x) { return x * x; }

  /** Defines the filter iterative operation at the RGB pixel neighborhood level.
   *
   * <div>Filters with a border defined on each RGB pixel overwrite this method.</div>
   *
   * @param ij Buffer index with pixels[i,j] = pixels[ij = i + j * width]
   * @param width Buffer width.
   * @param red0 Initial red channel buffer.
   * @param green0 Initial green channel buffer.
   * @param blue0 Initial blue channel buffer.
   * @param red Current (at iteration n) red channel buffer.
   * @param green Current (at iteration n) green channel buffer.
   * @param blue Current (at iteration n) blue channel buffer.
   */
  public myParam getRGB(int ij, int width, double epsilon, double upsilon, int red0[], int green0[], int blue0[],
                      int red[], int green[], int blue[]) {
    throw new IllegalStateException("Filter iteration not implemented at the pixel neighborhood level");
  }
}
```

B.2.4 Java Applet implementation

```
*****************************************************************************************************************************************
**************************************************RgbNeuralMapApplet.java****************************************************************
*****************************************************************************************************************************************

import imp.ima.RgbImage;
import imp.ima.util.*;

import imp.gui.*;
import javax.swing.JComponent;

/** Displays an applet to <a href="../doc-files/rgb-neural-map-applet.html">experiment</a> with <a href="RgbNeuralMap.html"
    >Rgb Neural Map Filter</a>. */

public class RgbNeuralMapApplet extends IApplet {
  /** Constructs the applet. */public RgbNeuralMapApplet() { }
  private OptionInput option; private StringInput items; private RgbImage image = null;

  public JComponent getComponent() {
    component = (ImageViewer) new ImageViewer();
    component.setWest(new TabularContainer().setColumns(1)
      .add((items = new StringInput() { public void outputValue(String value){
setImage(value);
      }}).setEditable(false).setItems("the-triangle the-noisy-triangle corsica the-flag the-raw-flag the-forms
                      the-image-1 the-image-3").setValue("the-triangle").setTitle("Image"))
      .add((items = new StringInput() { public void outputValue(String value){
setImage(value);
      }}).setEditable(false).setItems("No-Constraint R+G+B=128 R+G+B=255").setValue("No-Constraint")
                      .setTitle("Constraint"))
              .add((option = new OptionInput() { public void outputValue(String value) {
showImage(value);
      }}).setItems("Input RgbNeuralMapFilter IsoNeuralMapFilter").setValue("Input RgbNeuralMapFilter")
                      .setTitle("Filters with NN"))
              .addSpace());
    if ("true".equals(get("french"))) {
      option.setItems("Filtre Rgb").setValue("Entree").setTitle("Filtrage avec des reseaux de neurones");
      items.setItems("le-triangle le-triangle-bruite la-corse le-drapeau le-drapeau-bruite les-formes l-image-1 l-image-3");
      items.setItems("Pas-de-Contrainte R+G+B=128 R+G+B=255");
    }
    return component;
  }

  private ImageViewer component;

  private void setImage(String image) {
     try {
if ("the-triangle".equals(image)) {
   component.setImage(the_triangle);
        } else if ("the-noisy-triangle".equals(image)) {
   component.setImage(the_noisy_triangle);
} else if ("corsica".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/image.jpg")));
} else if ("the-flag".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/synthe24.jpg")));
} else if ("the-raw-flag".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/drap-raw.jpg")));
} else if ("the-forms".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/synthe22.jpg")));
} else if ("the-image-1".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/image1.jpg")));
} else if ("the-image-3".equals(image)) {
component.setImage(new RgbImage(resolve("../NeuralMap/images/image3.jpg")));
} else {
   component.setImage(image);
        }
      this.image = component.getImage();
    } catch(Exception e) { echo(e.toString()); }
  }

  private static final RgbImage the_triangle = RgbImages.getTriangle(400, 300);
  private static final RgbImage the_noisy_triangle = RgbImages.getTriangle(400, 300).getNoisy(300).getChannel("blue");


  private void showImage(String mode) {
    if (image == null) return;
    if ("RgbNeuralMapFilter".equals(mode) || "Entree".equals(mode)) {
component.setImage(image.getFiltered(new RgbNeuralMap()));
    } else {
if ("IsoNeuralMapFilter".equals(mode) || "Entree".equals(mode)){
    component.setImage(image.getFiltered(new IsoNeuralMap()));
} else {
    component.setImage(image);
}
    }
  }
}
```

B.3 Makefile

```
#<pre>
#*************************************************************
#* Sandrine Chemla, Copyright (C) 2006.  All rights reserved. *
#*************************************************************

default : start

test2 :
rsh tasmanie "cd $(patsubst /home/vthierry/%,/user/vthierry/laptop/home/%,$(PWD)) ;
        /net/lib/maple9.5/linux-v4/bin/maple -q YaRgbNeuralMap.mpl"


# Usage de ce makefile

usage :
@echo 'make start Lancement du programme java'
@echo 'make compile Compilation des classes java'
@echo 'make javadoc Genere la documentation java'
@echo 'make jarfile Genere l archive des classes java'
```

```
@echo ’make imp-jar Recompile les classes imp’
@echo ’make clean Nettoie tous les fichiers generes’

# Scan la Jdk1.5 : choisir l’une de ces locations ou ajouter la votre a cette liste

JAVAS = $(wildcard /usr/java/jdk1.5.*) $(wildcard $(HOME)/.jdk1.5.*)

JAVA = $(shell for d in $(JAVAS) ; do if [ -d $$d ] ; then echo "$$d" ; exit ; fi ; done)
ifeq ($(JAVA), )
$(error Jdk1.5 location not found)
endif

# Scan le JAR des Imp : l’ajouter a la liste si besoin

IMPS = $(JAVA)/jre/lib/ext/imp.jar /usr/java/imp-4/imp.jar /net/servers/www-sop/odyssee/imp/imp.jar $(PWD)/imp.jar

IMP = $(shell for d in $(IMPS) ; do if [ -f $$d ] ; then echo "$$d" ; exit ; fi ; done)
ifeq ($(IMP), )
$(error Imp location not found)
endif

# Scan maple : choisir l’une de ces locations ou ajouter la votre a cette liste

MAPLES = /usr/bin/maple /net/lib/maple9.5/linux-v4/bin/maple /usr/local/maple9.5/bin/maple

MAPLE =  $(shell for f in $(MAPLES) ; do if [ -f $$f ] ; then echo "$$f" ; exit ; fi ; done)
ifeq ($(MAPLE), )
$(error Maple location not found)
endif

#
# Lancement d’un des programmes
#

ifndef MAIN
MAIN = RgbNeuralMapApplet
endif

start : compile
@echo start      ; $(JAVA)/bin/java -server -Xmx384m -classpath .:$(IMP) imp.gui.IApplet -code $(MAIN)

#
# Generation automatique du code %.mpl + %.mjava -> %.java
#

%.java : %.mjava %.inc
@sed ’s/"/\\"/g’ < $*.mjava | sed ’s/\(.*\)/echo "\1"/’ > mjava.sh ; sh mjava.sh > $@ ; rm mjava.sh

%.inc : %.mpl
@$(MAPLE) -q $*.mpl

#
# Compile tous les fichiers java
#

compile : $(patsubst %.mjava,%.java,$(shell find . -name ’*.mjava’)) $(patsubst %.java,%.class,$(shell find . -name ’*.java’))

%.class : %.java
@echo "javac $ˆ" ; $(JAVA)/bin/javac -classpath .:$(IMP) $ˆ

# Nettoie tous les fichiers generes par ce makefile

clean :
@echo clean ; rm -rf ‘find . -name ’*.class’ -o -name ’*˜’ -o -name ’*.inc’‘ ‘find -name ’*.mjava’ | sed ’s/\.mjava/.java/’‘

#</pre>
```

Conclusion & Future work

This work is realized within the scope of the FACETS project which goal is to create a theoretical and experimental foundation for the practical implementation of novel computing paradigms, which exploit the concepts experimentally observed in the brain.

In this work, we first developed a model of cortical hypercolumns, specific groups of neurons in the brain. This model allows to link computer-vision continuous formalism with neural networks and the related discrete formulation.

Additionally, we developed a VF local mechanism generator including non-linear vectorial state. This generator, based on neural networks and implemented in Maple, is independent of the application. Furthermore, constraint estimation in the color space has been developped. For instance, isotropic filtering with iso-luminance (R+G+B=128 and R+G+B=255) have been tested and validated.

My personal contributions were several corrections of the initial T. Vieville Research report [33] and also the implementation of the existing `Maple` code (see Appendix B.1) in order to obtain a first automatic map computation generator in Java (see Appendix B.2). Some aspects of the present work has been published in ECVP ' 06 [24].

As a future work, a validation on edge-preserving smoothing, and some WinnerTakeAll mechanism (see [35] for details) is going to be developed, and we also want to test non-linear constraint like $R^2 + G^2 + B^2 = 255^2$.

On the other hand, we will analyse what can be state about several maps interacting together (as used in [32] and [17] and starting an application to V1 optical imaging diffusion estimation. The compilation of spiking-network parameters [19] from a variational formulation is another perspective of this work, following [26, 27].

Bibliography

[1] S.-I. Amari. Dynamical study of formation of cortical maps. *Biological Cybernetics*, 27:77–87, 1977.

[2] G. Aubert and P. Kornprobst. *Mathematical Problems in Image Processing: Partial Differential Equations and the Calculus of Variations*, volume 147 of *Applied Mathematical Sciences*. Springer-Verlag, January 2002.

[3] G. Bouchitté and M. Valadier. Integral representation of convex functions on a space of measures. *J. Funct. Anal.*, 80(2):398–420, 1988.

[4] J. Bullier. Integrated model of visual processing. *Brain Res. Reviews*, 36:96–107, 2001.

[5] Y. Burnod. *An adaptive neural network: the cerebral cortex*. Masson, Paris, 1993. 2nd edition.

[6] B. Cessac and M. Samuelides. *From Neuron to Neural Networks Dynamics*, chapter X. t.b.d., 2006.

[7] G. Cottet and S. Mas-Gallic:. A particle method to solve the navier-stokes system. *Numer. Math.*, 57, 1990.

[8] G.-H. Cottet. Neural networks: continuous approach and applications to image processing. *J. Biological Systems*, 3, 1995.

[9] G.-H. Cottet and M. El Ayyadi. A Volterra type model for image processing. 7(3), March 1998.

[10] P. Dayan and L. F. Abbott. *Theoretical Neuroscience : Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001.

[11] P. Degond and S. Mas-Gallic. The weighted particle method for convection-diffusion equations. *Mathematics of Computation*, 53(188):485–525, 1989.

[12] R.J. Douglas and K. A. C. Martin. Neuronal circuit of the neocortex. *Ann. Rev. Neuroscience*, 27:419, 2004.

[13] R. Edwards. Approximation of neural network dynamics by reaction-difusion equations. *Mathematical Methods in the Applied Sciences*, 19:651–677, 1996.

[14] L.C. Evans. *Partial Differential Equations*, volume 19 of *Graduate Studies in Mathematics*. 1998.

[15] L.C. Evans and R.F. Gariepy. *Measure Theory and Fine Properties of Functions*. CRC Press, 1992.

[16] H. Federer. *Geometric Measure Theory*. Classics in Mathematics. Springer–Verlag, Berlin . Heidelberg . New York, 1969.

[17] Hervé Frezza-Buet, Nicolas Rougier, and Frédéric Alexandre. Integration of biologically inspired temporal mechanisms into a cortical framework for sequence processing. In *Neural, Symbolic and Reinforcement methods for sequence learning*, Lecture Notes in Computer Science. Springer, 2001.

[18] Karl Friston. Functional integration and inference in the brain. *Prog Neurobiol*, 68:113–143, 2002.

[19] W. Gerstner and W. M. Kistler. Mathematical formulations of hebbian learning. *Biol Cybern*, 87:404–415, 2002.

[20] T. Gisiger, S. Dehaene, and J. P. Changeux. Computational models of association cortex. *Curr. Opin. Neurobiol.*, 10:250–259, 2000.

[21] Stephen Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1(1):1–97, 1988.

[22] Jan J. Koenderink. *Solid Shape.* 1990.

[23] I. Kokkinos, R. Deriche, P. Maragos, and O. Faugeras. A biologically motivated and computationally tractable model of low and mid-level vision tasks. In *Proceedings Eighth European Conference on Computer Vision,*, Prague, May 2004.

[24] P. Kornprobst, T. Viéville, S. Chemla, and O. Rochel. Modeling cortical maps with feed-backs. Master's thesis, 2006.

[25] T. Lee and D. Mumford. Hierarchical bayesian inference in the visual cortex. *J. Opt. Soc. Am. A*, 20(7), 2003.

[26] W. Maass. Fast sigmoidal networks via spiking neurons. *Neural Computation*, 9:279–304, 1997.

[27] W. Maass and T. Natschler. Networks of spiking neurons can emulate arbitrary hopfield nets in temporal coding. *Neural Systems*, 8(4):355–372, 1997.

[28] J. Petitot and Y. Tondut. Vers une neuro-géométrie. fibrations corticales, structures de contact et contours subjectifs modaux. *Mathématiques, Informatique et Sciences Humaines*, 145:5–101, 1999.

[29] R.D.S. Raizada and S. Grossberg. Towards a theory of the laminar architecture of the cerebral cortex: Computational clues from the visual system. *Cerebral Cortex*, 13:100–113, 2003.

[30] P.A. Raviat. An analysis of the particle methods. In F. Brezzi, editor, *Numerical Methods in Fluid Dynamics*, volume 1127 of *Lecture Notes in Math.*, pages 243–324. Springer Verlag, Berlin, 1985.

[31] L. Schwartz. *Théorie des distributions.* Hermann, 1957.

[32] S.J. Thorpe and M. Fabre-Thorpe. Seeking categories in the brain. *Science*, 291:260–263, 2001.

[33] T. Viéville. An unbiased implementation of regularization mechanisms. *Image and Vision Computing*, 23(11):981–998, 2005.

[34] Thierry Viéville. An abstract view of biological neural networks. Technical Report RR-5657, INRIA, August 2005.

[35] A. J. Yu, M.A. Giese, and T. Poggio. Biophysiologically plausible implementations of maximum operation. *Neural Computation*, 14(12), 2003.