Cellular Sensory and Wave Computing Laboratory of the Computer and Automation Research Inst., Hungarian Academy of Sciences and the Jedlik Laboratories of the Pázmány P. Catholic University

Software Library for Cellular Wave Computing Engines

in an era of kilo-processor chips

Version 3.1

Budapest, 2010

EDITED BY, K. KARACS , GY, CSEREY, Á. ZARÁNDY, P. SZOLGAY, CS. REKECZKY, L- KÉK, V. SZABÓ, G. PAZIENZA AND T. ROSKA
BUDAPEST, HUNGARY

TABLE OF CONTENTS

INTRODUCTION

We are witnessing a proliferation of cellular topographic processor arrays: the Blue-Gene and the CELL based IBM supercomputer, the "heart" of the Playstation 3, the CELL Multiprocessor (Sony-IBM-Toshiba), as well as the kilo-processor FPGA chips and the new 48-core chip of Intel, ets. In addition, the Eye-RIS system of AnaFocus, and the Xenon chip of Eutecus (in collaboration with MTA SZTAKI and the Pazmany University), as well as "the proliferation of GPU" graphics chips.

Indeed, a new scenario is emerging: Cellular Wave Computing[*], cellular means the procedure of geometrical locality in a see of processor and memory arrays[†].

This new edition of the old CNN SOFTWARE LIBRARY contains some new elements and some editing. The style of the description follows the first textbook[+].

What is brand new: the introduction of the Virtual and Physical Cellular Machine framework (Appendix 2.) and the first steps of its use (Chapter3.)

As for the templates/instructions, they are classified according to their functions: basic image processing, mathematical morphology, spatial logic, texture segmentation and detection, motion, color, depth, optimization, game of life and combinatorics. Some useful template sequences are defined as subroutines, and a few complete programs are shown as well. Some templates for neuromorphic modeling and solving partial differential equations (PDE) are also included.

Unless otherwise noted, the normalized first order CNN equation with linear delay-less templates is

$$\dot{x}_{ij} = -x_{ij} + z + \sum A_{ij;kl*} y_{kl} + \sum B_{ij;kl*} u_{kl} + \sum C_{ij;kl*} x_{kl} + \sum \hat{D}_{ij;kl}(u_{kl}, x_{kl}, y_{kl})$$

Without the last two terms we call it "standard" CNN dynamics.

Time is scaled in $\tau = \tau_{CNN}$, the time constant of the first order CNN cell. As a default, $\tau_{CNN}=1$. Observe that local template operators might have different forms (e.g. the D operator)

This library is result of a continuous development. It contains results published by dozens of researchers all over the world.

The library is not complete. New templates, operators and subroutines can be added. Moreover, the emergence of a new world of algorithms is foreseen. Completey new algorithms are evolving for a given task if it is implemented in a virtual cellular machine on kilo-processor chips. We encourage designers all over the world to send their templates, subroutines, programs to be included in this library, with proper reference to the original publication. Initiating this action, you can e-mail to zarandy@sztaki.hu or roska@ppke.hu.

---

[*] T. Roska, Cellular wave computers for nano-tera-scale technology – beyond Boolean, spatial-temporal logic in million processor devices, *Electronics Letters,* Vol. 43., No.8., April 2007.
C. Baatar, W. Porod, T. Roska, (Eds.), Cellular Nanoscale Sensory Wave Computing, Springer, 2009, ISBN 978-1-4419-1010-3

[+] L. O. Chua and T. Roska, Cellular Neural Networks and visual computing: Foundations and applications, Cambridge University Press, 2002 (paperback: 2005)

Chapter 1. Templates/Instructions

1.1. BASIC IMAGE PROCESSING

GradientIntensityEstimation: Estimation of the gradient intensity in a local neighborhood

Old names: AVERGRAD

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \\ \boldsymbol{b} & 0 & \boldsymbol{b} \\ \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \end{bmatrix} \qquad z = \boxed{0}$$

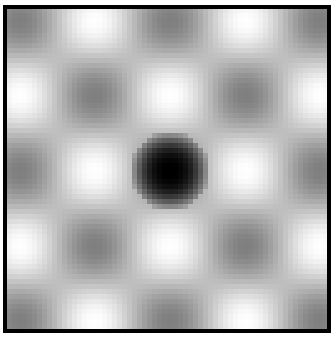
where $\boldsymbol{b} = |v_{u_{ij}} - v_{u_{kl}}| / 8$.

I. Global Task

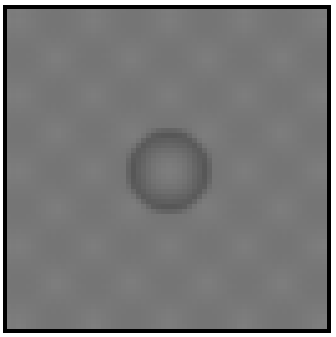
Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Grayscale image representing the estimated average gradient intensity in a local neighborhood in **P**.

II. Examples

Example 1: image name: avergra1.bmp, image size: 64x64; template name: avergrad.tem .

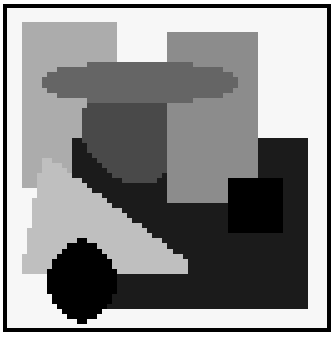


input



output

Example 2: image name: avergra2.bmp, image size: 64x64; template name: avergrad.tem.



input

output

Smoothing: ***Smoothing with binary output*** **[1]**

Old names: AVERTRSH, AVERAGE

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \begin{bmatrix} 0 \end{bmatrix}$$

I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black (white) pixels correspond to the locations in **P** where the average of pixel intensities over the r=1 feedback convolution window is positive (negative).

II. Example: image name: madonna.bmp, image size: 59x59; template name: avertrsh.tem .

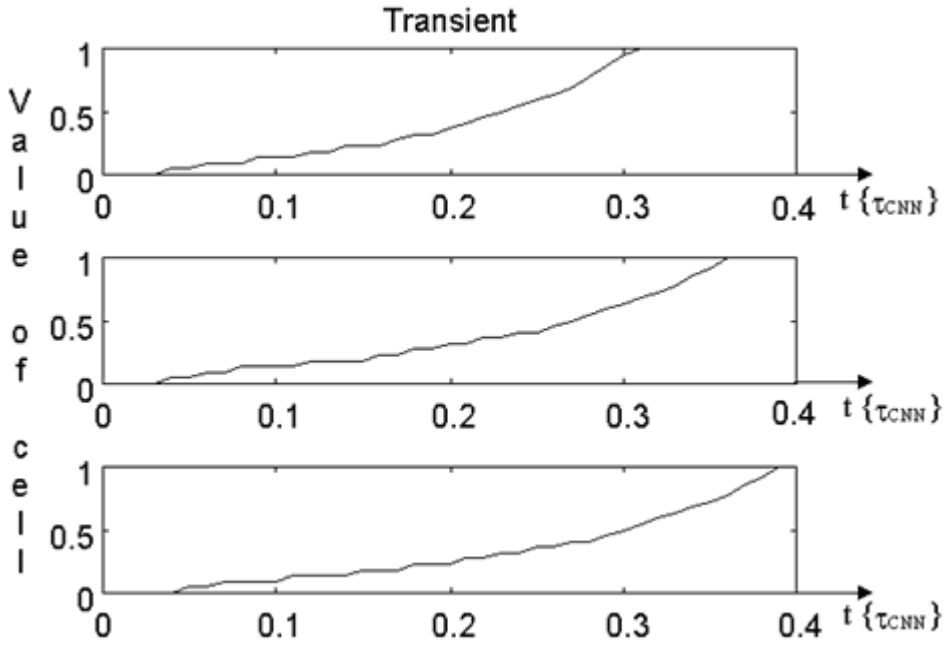
input

output

The transient of the cell C(38,10) has been examined in 3 cases:
1. applying the original **A** template shown before;
2. applying the following $\mathbf{A}_1$ template (template name: avertrs1.tem);
3. applying the following $\mathbf{A}_2$ template(template name: avertrs2.tem).

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 1.2 & 0 \\ 1.2 & 1.8 & 1.2 \\ 0 & 1.2 & 0 \end{bmatrix} \qquad \mathbf{A}_2 = \begin{bmatrix} 0 & 0.9 & 0 \\ 0.9 & 1.8 & 0.9 \\ 0 & 0.9 & 0 \end{bmatrix}$$

The transients of the examined cell are presented in the following figure corresponding to the templates **A**, $\mathbf{A}_1$ and $\mathbf{A}_2$.


Transient
Value of cell
1
0.5
0
0
0.1
0.2
0.3
0.4
t {τCNN}

DiagonalHoleDetection: ***Detects the number of diagonal holes from each diagonal line*** **[6]**

Old names: CCD_DIAG

A = | 1 | 0 | 0 |
| --- | --- | --- |
| 0 | 2 | 0 |
| 0 | 0 | -1 |

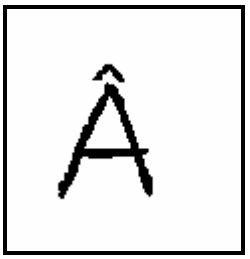
B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |
| --- |

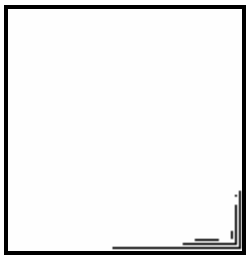
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image that shows the number of diagonal holes in each diagonal line of image **P**.

II. Example: image name: a_letter.bmp, image size: 117x121; template name: ccd_diag.tem .



input



output

HorizontalHoleDetection: ***Detects the number of horizontal holes from each horizontal row*** **[6]**

Old names: HorizontalCCD ,CCD_HOR

A = | 0 | 0 | 0 |
| 1 | 2 | -1 |
| 0 | 0 | 0 |

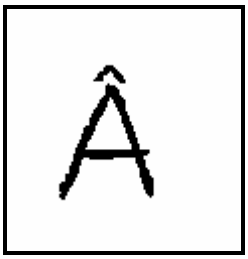
B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = 0

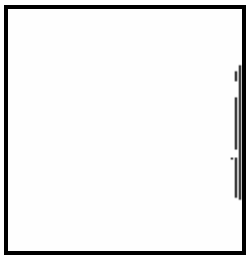
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image that shows the number of horizontal holes in each horizontal row of image **P**.

II. Example: image name: a_letter.bmp, image size: 117x121; template name: ccd_hor.tem .



input



output

VerticalHoleDetection: ***Detects the number of vertical holes from each vertical column*** **[6]**

Old names: VerticalCCD, CCD_VERT

A = | 0 | 1 | 0 |
| 0 | 2 | 0 |
| 0 | -1 | 0 |

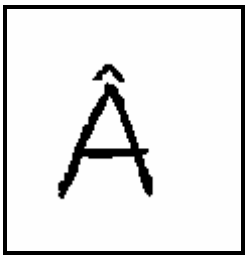
B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |

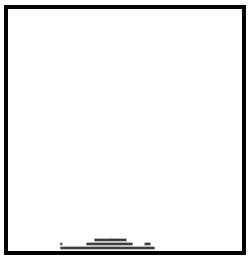
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image that shows the number of vertical holes in each vertical column of image **P**.

II. Example: image name: a_letter.bmp, image size: 117x121; template name: ccd_vert.tem .



input



output

***MaskedCCD: Masked connected component detector* [24]**

Old names: CCDMASK

Left-to-right

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & -1 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{-3}$$

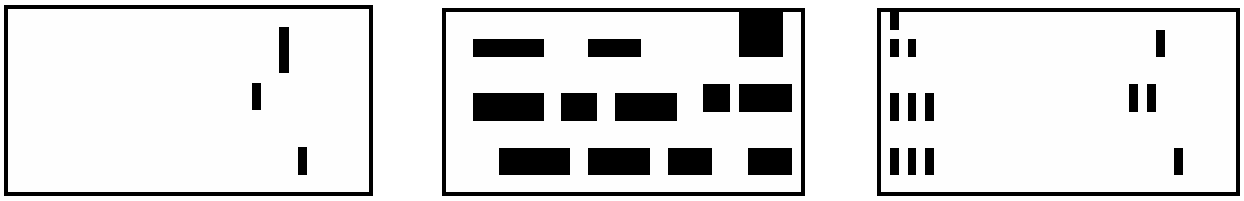
I. Global Task

Given:	static binary images $\mathbf{P_1}$ (mask) and $\mathbf{P_2}$
Input:	$\mathbf{U(t) = P_1}$
Initial State:	$\mathbf{X}(0) = \mathbf{P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image that is the result of CCD type shifting $\mathbf{P_2}$ from left to right. Shifting is controlled by the mask $\mathbf{P_1}$.

Remark:

This is a CCD operation, but the black trains stop in front of a black wall of the mask ($\mathbf{P_1}$) instead of the boundary of the image. The direction of shift is from the positive to the negative non-zero off-center feedback template entry. By rotating **A** the template can be sensitized to other directions as well.

II. Example: Right-to-left shifting. Image names: ccdmsk1.bmp, ccdmsk2.bmp; image size: 40x20; template name: ccdmaskr.tem .



input initial state output

CenterPointDetector: ***Center point detection*** **[21]**

Old names: CENTER

$\mathbf{A}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix}$ $\mathbf{B}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_1 = \boxed{-1}$

$\mathbf{A}_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 6 & 0 \\ 1 & 0 & -1 \end{bmatrix}$ $\mathbf{B}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_2 = \boxed{-1}$

$\mathbf{A}_3 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 4 & 0 \\ 0 & -1 & 0 \end{bmatrix}$ $\mathbf{B}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_3 = \boxed{-1}$

$\mathbf{A}_4 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 6 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ $\mathbf{B}_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_4 = \boxed{-1}$

. . .

$\mathbf{A}_8 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 6 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ $\mathbf{B}_8 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_8 = \boxed{-1}$

I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where a black pixel indicates the center point of the object in **P**.

Remark:

The algorithm identifies the center point of the black-and-white input object. This is always a point of the object, halfway between the furthermost points of it. Here a DTCNN template sequence is given, each element of it should be used for a single step. It can easily be transformed to a continuous-time network:

CENTER1:

$\mathbf{A}_1 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_1 =$ | 1 | 0 | 0 |
|---|---|---|
| 1 | 4 | -1 |
| 1 | 0 | 0 |

$z_1 =$ -1

CENTER2:

$\mathbf{A}_2 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_2 =$ | 1 | 1 | 1 |
|---|---|---|
| 1 | 6 | 0 |
| 1 | 0 | -1 |

$z_2 =$ -1

CENTER3:

$\mathbf{A}_3 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_3 =$ | 1 | 1 | 1 |
|---|---|---|
| 0 | 4 | 0 |
| 0 | -1 | 0 |

$z_3 =$ -1

CENTER4:

$\mathbf{A}_4 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_4 =$ | 1 | 1 | 1 |
|---|---|---|
| 0 | 6 | 1 |
| -1 | 0 | 1 |

$z_4 =$ -1

. . .

CENTER8:

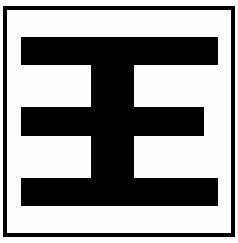
$\mathbf{A}_8 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_8 =$ | 1 | 0 | -1 |
|---|---|---|
| 1 | 6 | 0 |
| 1 | 1 | 1 |

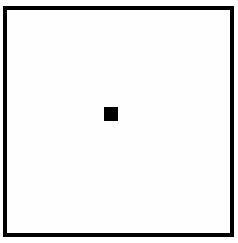
$z_8 =$ -1

The robustness of templates CENTER1 and CENTER2 are ρ(CENTER1) = 0.22 and ρ(CENTER2) = 0.15, respectively. Other templates are the rotated versions of CENTER1 and CENTER2, thus their robustness values are equal to the mentioned ones.

II. Example: image name: chineese.bmp, image size: 16x16; template name: center.tem .



input



output

ConcentricContourDetector: ***Concentric contour detection (DTCNN)*** **[16]**

Old names: CONCCONT

A =

0	-1	0
-1	3.5	-1
0	-1	0

B =

0	0	0
0	4	0
0	0	0

z = -4

I. Global Task

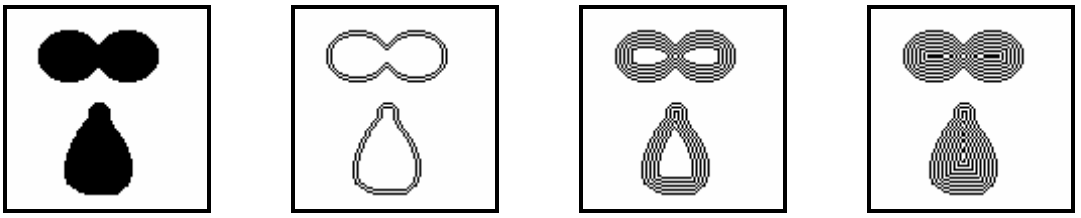
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the concentric black and white rings obtained from **P**.

Examples

Example 1: image name: conc1.bmp, image size: 16x16; template name: concont.tem .

input output

Example 2: image name: conc2.bmp, image size: 100x100; template name: concont.tem .



input output, 3. step output, 9. step output, t = ∞

GlobalConnectivityDetection: ***Deletes marked objects*** **[36]**

Old names: Connectivity

$$\mathbf{A} = \begin{bmatrix} 0 & 0.5 & 0 \\ 0.5 & 3 & 0.5 \\ 0 & 0.5 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & -0.5 & 0 \\ -0.5 & 3 & -0.5 \\ 0 & -0.5 & 0 \end{bmatrix} \qquad z = \boxed{-4.5}$$

I. Global Task

Given: two static binary images $\mathbf{P_1}$ (mask) and $\mathbf{P_2}$ (marker). The mask contains some black objects against the white background. The marker contains the same objects, except for some objects being marked. An object is considered to be marked, if some of its black pixels are changed into white.

Input: $\mathbf{U(t) = P_1}$

Initial State: $\mathbf{X(0) = P_2}$

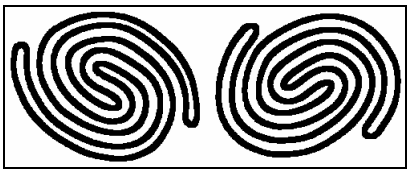
Boundary Conditions: Fixed type, $u_{ij} = -1$, $y_{ij} = -1$ for all virtual cells, denoted by [U]=[Y]=[-1]

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image containing the unmarked objects only.

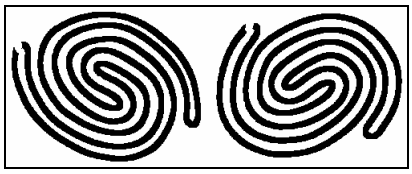
Remark:

The template determines whether a given geometric pattern is "globally" connected in one contiguous piece, or is it composed of two or more disconnected components.

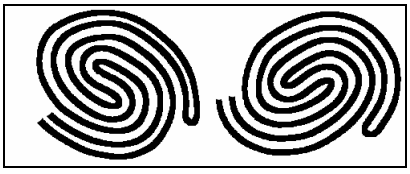
II. Example: image names: connect1.bmp, connect2.bmp; image size: 500x200; template name: connecti.tem .



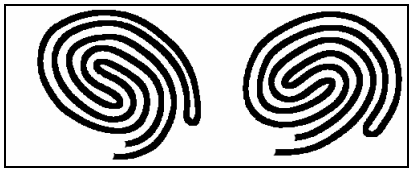
INPUT



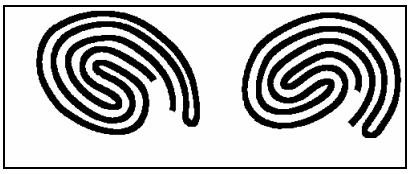
INITIAL STATE



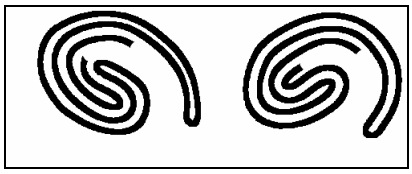
t=125τ



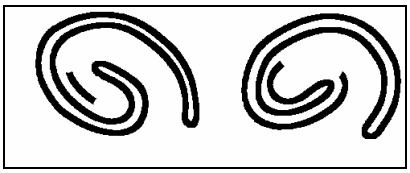
t=250τ



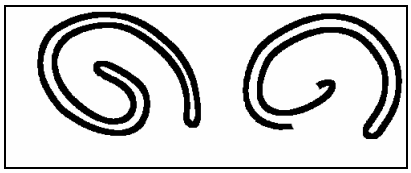
t=375τ



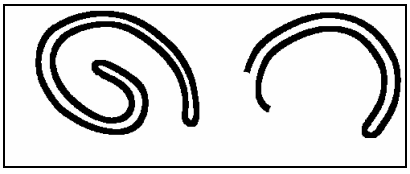
t=500τ



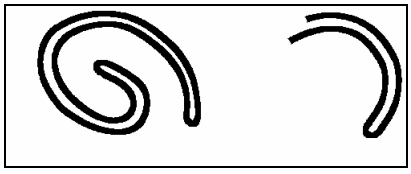
t=625τ



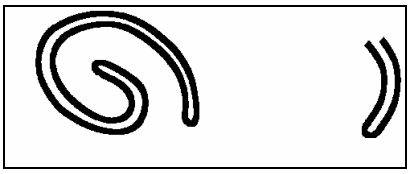
t=750τ



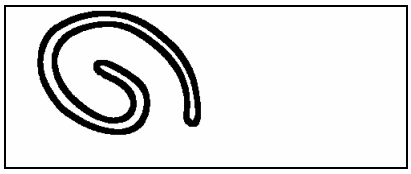
t=875τ



t=1000τ



t=1125τ



t=1250τ

***GlobalConncetivityDetection1: Detects the one-pixel thick closed curves and deletes the open curves from a binary image* [61]**

$\mathbf{A} =$ | 6 | 6 | 6 |
|---|---|---|
| 6 | 9 | 6 |
| 6 | 6 | 6 |

$\mathbf{B} =$ | -3 | -3 | -3 |
|---|---|---|
| -3 | 9 | -3 |
| -3 | -3 | -3 |

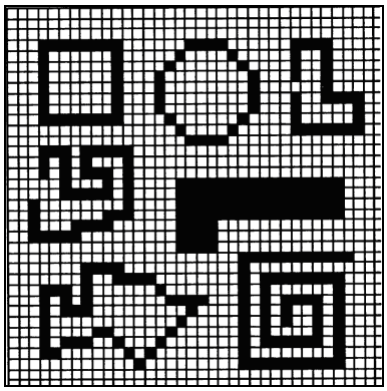
$z =$ -4.5

I. Global Task

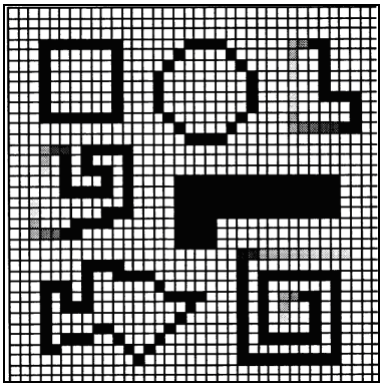
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image which contains all closed curves present in the initial image **P**

II. Example: image size: 36x36.

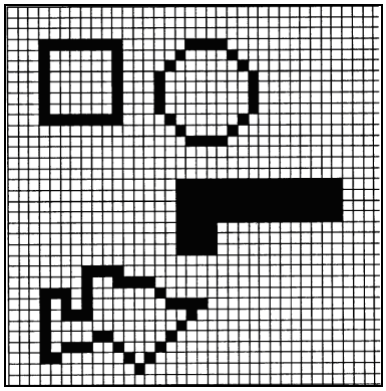
Remarks: *The binary image* **P** *containing closed and open curves (one-pixel thick) is applied both at the input and loaded as initial state. If one pixel is removed from a closed curve, it becomes an open curve and is deleted, as shown in the second image. The compact (solid) objects from the image are not modified.*



Input



Intermediate result



Output

ContourExtraction: ***Grayscale contour detector*** **[8]**

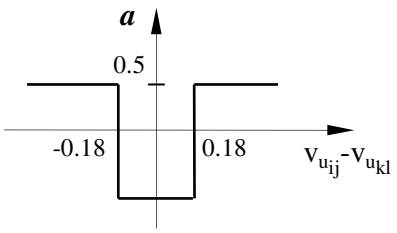
Old names: ContourDetector, CONTOUR

A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | ***a*** | ***a*** | ***a*** |
| ***a*** | 0 | ***a*** |
| ***a*** | ***a*** | ***a*** |

z = 0.7

where ***a*** is defined by the following nonlinear function:



I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels represent the contours of the objects in **P**.

Remark:

The template extracts contours which resemble edges (resulting from big changes in gray level intensities) from grayscale images.

II. Example: image name: madonna.bmp, image size: 59x59; template name: contour.tem .

input

output

CornerDetection: ***Convex corner detection template*** **[1]**

Old names: CornerDetector, CORNER

A = | 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | -1 | -1 | -1 |
| -1 | 4 | -1 |
| -1 | -1 | -1 |

z = -5

I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels represent the convex corners of objects in **P**.
Template robustness:	ρ = 0.2 .

Remark:

Black pixels having at least 5 white neighbors are considered to be convex corners of the objects.

II. Example: image name: chineese.bmp, image size: 16x16; template name: corner.tem .

input output

The transient of the cell C(6,3) has been examined in 3 cases:

1. applying the original *CornerDetector* template shown before;
2. applying the following CORNCH_1 template;
3. applying the following CORNCH_2 template.

CORNCH_1

A = | 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | -1 | -1 | -1 |
| -1 | 3.9 | -1 |
| -1 | -1 | -1 |

z = -5

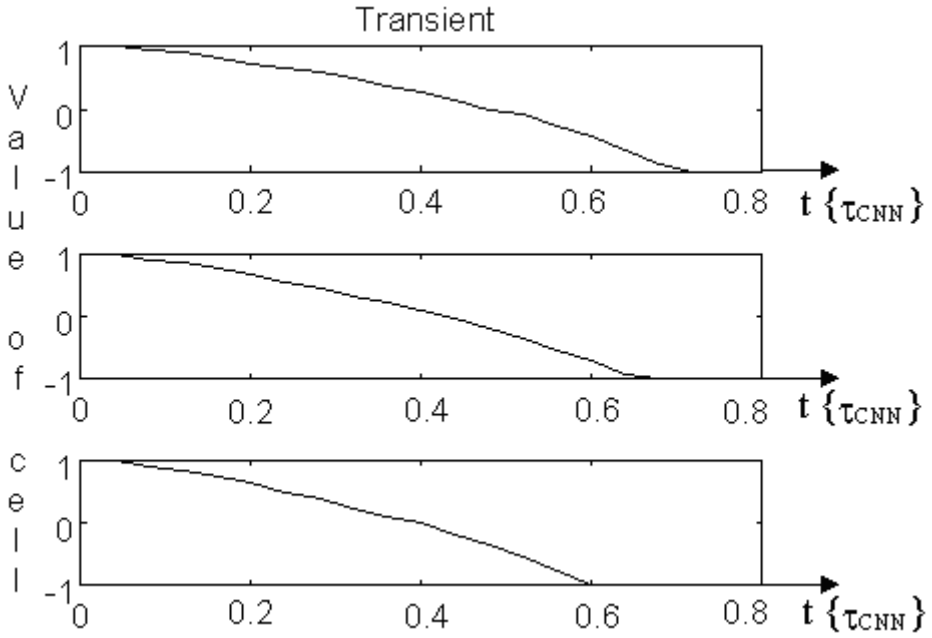
CORNCH_2

A = | 0 | 0 | 0 |
| 0 | 0.5 | 0 |
| 0 | 0 | 0 |

B = | -1 | -1 | -1 |
| -1 | 3.6 | -1 |
| -1 | -1 | -1 |

z = -5

The transients of the examined cell are presented in the following figure corresponding to the templates *CornerDetector*, CORNCH_1 and CORNCH_2.



DiagonalLineRemover: ***Deletes one pixel wide diagonal lines*** **[8]**

Old names: DELDIAG1

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | -1 | 0 | -1 |
|---|---|---|
| 0 | 1 | 0 |
| -1 | 0 | -1 |

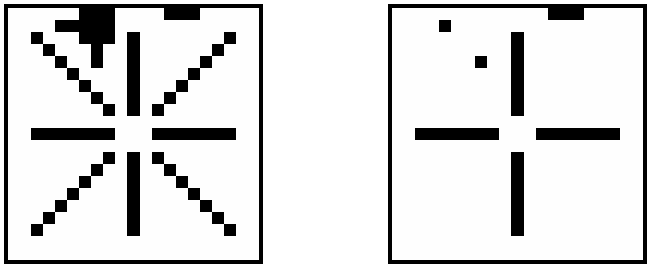
$z =$ -4

I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, u_{ij} = -1, for all virtual cells, denoted by [U] = -1
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels have no black neighbors in diagonal directions in **P**.
Template robustness:	ρ = 0.45 .

Remark:
The template may be used for deleting one pixel wide diagonal lines.

II. Example: image name: deldiag1.bmp, image size: 21x21; template name: deldiag1.tem .



input output

VerticalLineRemover: ***Deletes vertical lines* [8]**

Old names: DELVERT1

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \qquad z = \boxed{-2}$$

I. Global Task

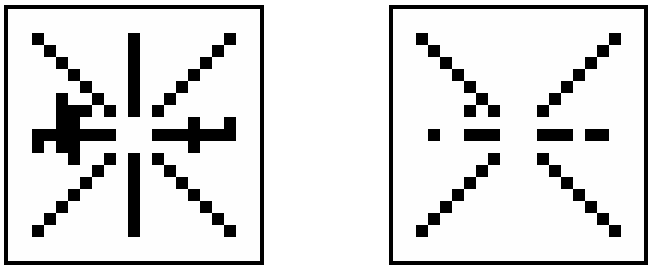
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, u_{ij} = -1, for all virtual cells, denoted by [U] = -1
Output:	**Y(t)⇒Y(∞)** = Binary image representing **P** without vertical lines. Those parts of the objects that could be interpreted as vertical lines will also be deleted.
Template robustness:	ρ = 0.58 .

Remark:

The template deletes every black pixel having either a northern or southern black neighbor.

The *HorizontalLineRemover* template, that deletes one pixel wide horizontal lines, can be obtained by rotating the *VerticalLineRemover* by 90°. The functionality of the WIREHOR and WIREVER templates that were published in earlier versions of this library, is identical to the functionality of the *HorizontalLineRemover* and *VerticalLineRemover* templates.

II. Example: image name: delvert1.bmp, image size: 21x21; template name: delvert1.tem .



input output

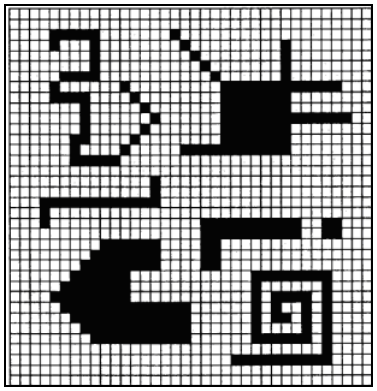
ThinLineRemover: ***Removes thin (one-pixel thick) lines from a binary image***

$$\mathbf{A} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 8 & 2 \\ 2 & 2 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{-2}$$

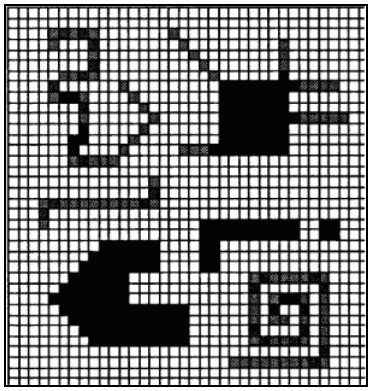
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image containing compact black objects (without any thin lines) against a white background

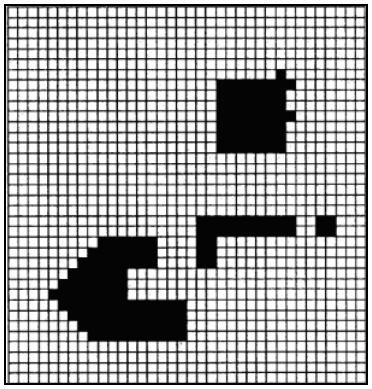
II. Example: image size: 36x36.



Initial state



Intermediate state



Output

ApproxDiagonalLineDetector: ***Detects approximately diagonal lines***

Old names: DIAG

A =

0	0	0	0	0
0	0	0	0	0
0	0	2	0	0
0	0	0	0	0
0	0	0	0	0

B =

-1	-1	-1	0.5	1
-1	-1	1	1	0.5
-1	1	5	1	-1
0.5	1	1	-1	-1
1	0.5	-1	-1	-1

z = -13

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= P**

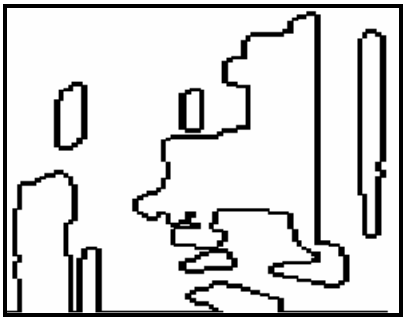
Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

Output: **Y(t)⇒Y(∞)** = Binary image representing the locations of approximately diagonal lines in **P**.

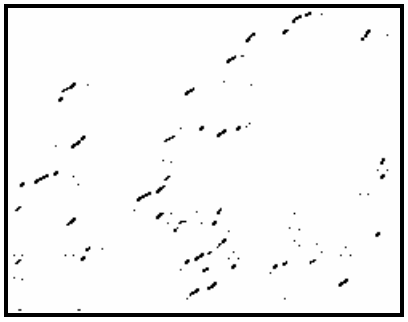
Remark:

Detects approximately diagonal lines being in the SW-NE direction. By modifying the positions of the elements of the **B** template, namely rotating **B**, the template can be sensitized to other directions as well (vertical, horizontal or NW-SE diagonal).

II. Example: image name: diag.bmp, image size: 246x191; template name: diag.tem .



input



output

DiagonalLineDetector: ***Diagonal-line-detector template***

Old names: DIAG1LIU

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \quad z = \boxed{-4}$$

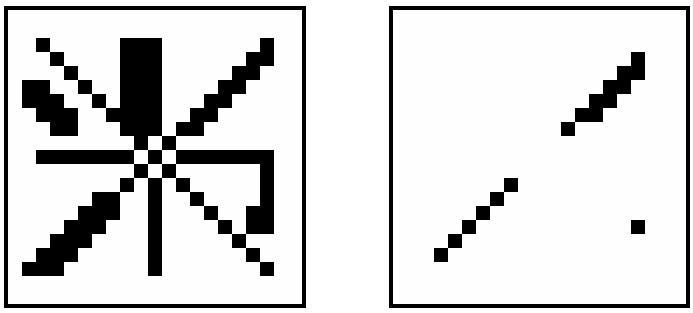
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the locations of diagonal lines in **P**.
Template robustness:	ρ = 0.45 .

Remark:

Detects every black pixel having black north-eastern, black south-western, white north-western, and white south-eastern neighbors. It may be used for detecting diagonal lines being in the SW-NE direction (like /). By modifying the position of the ±1 values of the **B** template, the template can be sensitized to other directions as well (vertical, horizontal or NW-SE diagonal).

II. Example: image name: diag1liu.bmp, image size: 21x21; template name: diag1liu.tem .



input output

GrayscaleDiagonalLineDetector: ***Grayscale diagonal line detector***

Old names: DIAGGRAY

A =

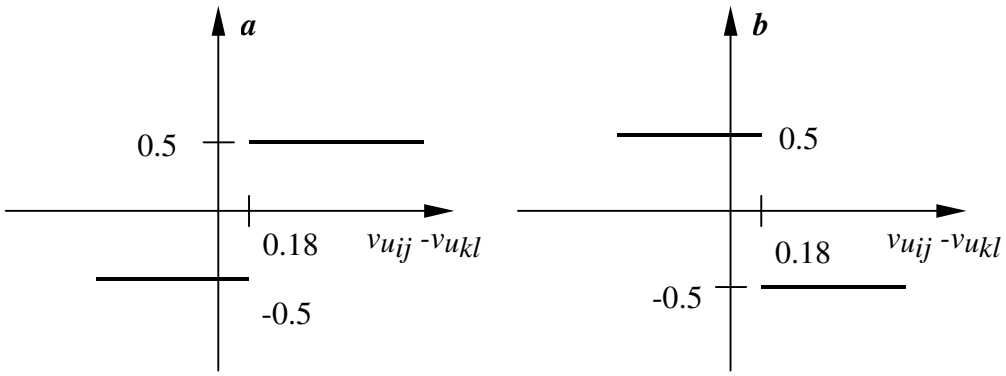
0	0	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	0	0	0
0	0	0	0	0

B =

b	***b***	***a***	***a***	***a***
b	***b***	***b***	***a***	***a***
a	***b***	0	***b***	***a***
a	***a***	***b***	***b***	***b***
a	***a***	***a***	***b***	***b***

z = -1.8

where ***a*** and ***b*** are defined by the following nonlinear functions:



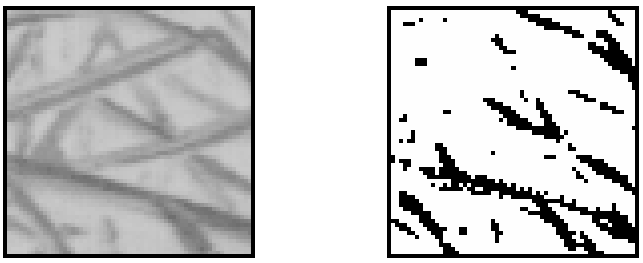
I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels identify the diagonal lines of north-west, south-eastern direction in **P**.

Remark:

If the nonlinear **B** template is rotated by 90°, the grayscale line detector of the north-east, south-west direction will be obtained.

II. Example: image name: diaggray.bmp, image size: 61x61; template name: diaggray.tem .



input output

RotationDetector: ***Detects the rotation of compact objects in a binary image, having only horizontal and vertical edges; removes all inclined objects or objects having at least one inclined edge*** **[61]**

$$\mathbf{A} = \begin{bmatrix} -0.8 & 5 & -0.8 \\ 5 & 5 & 5 \\ -0.8 & 5 & -0.8 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -0.4 & -2.5 & -0.4 \\ -2.5 & 5 & -2.5 \\ -0.4 & -2.5 & -0.4 \end{bmatrix} \quad z = \boxed{-11.2}$$

I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image which retains from the initial state **P** only the compact objects with horizontal or vertical edges

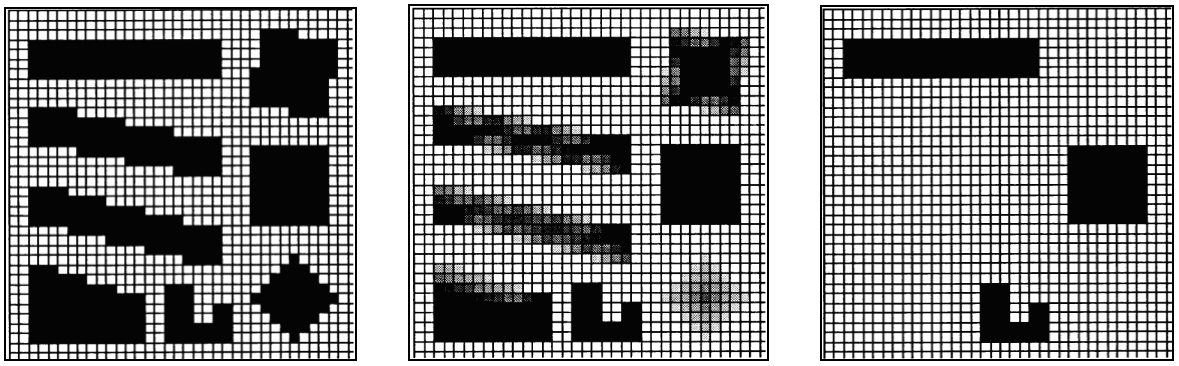
Remark:

The binary image is loaded as initial state and also applied at the input.

These templates can be used as a rotation detector, but only for a specific class of objects, namely with horizontal and vertical edges. Its robustness depends on the resolution of the CNN (number of cells) related to the object size. For larger objects, or a CNN with larger resolution, smaller rotation angles can be detected.

Every object having at least an inclined edge will be gradually deleted, as seen in the second image. Also the one pixel thick lines or curves will be removed. Binary noise can affect the operation. If a black or white parasitic pixel representing noise appears on the edge of an object, the object will be deleted, even if it has no inclined edge.

II. Example: image name: binary image; image size: 36x36



Initial state Intermediate state Output

HeatDiffusion: Heat-diffusion

Old names: DIFFUS

A = | 0.1 | 0.15 | 0.1 |
| 0.15 | 0 | 0.15 |
| 0.1 | 0.15 | 0.1 |

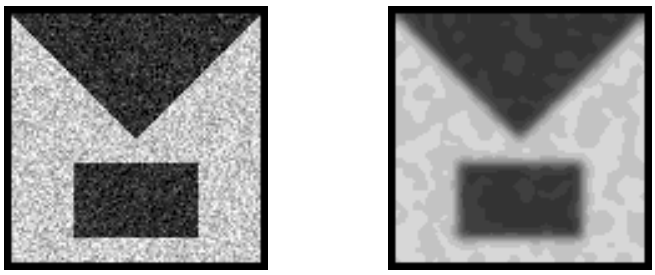
B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |

I. Global Task

Given:	static noisy grayscale image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(T)** = Grayscale image representing the result of the heat diffusion operation.

II. Example: image name: diffus.bmp, image size: 106x106; template name: diffus.tem .



input output

EdgeDetection: Binary edge detection template

Old names: EdgeDetector, EDGE

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | -1 | -1 | -1 |
|---|---|---|
| -1 | 8 | -1 |
| -1 | -1 | -1 |

$z =$ -1

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X(0) =** Arbitrary (in the examples we choose $x_{ij}(0)=0$)

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

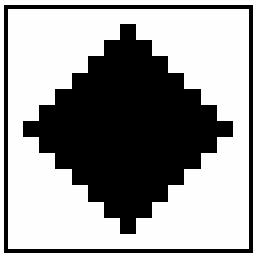
Output: **Y(t)⇒Y(∞)** = Binary image showing all edges of **P** in black

Template robustness: ρ = 0.12 .

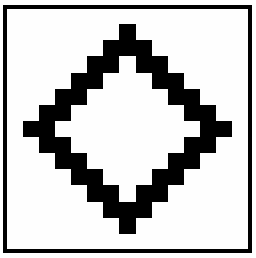
Remark:
Black pixels having at least one white neighbor compose the edge of the object.

II. Examples

Example 1: image name: logic05.bmp, image size: 44x44; template name: edge.tem .



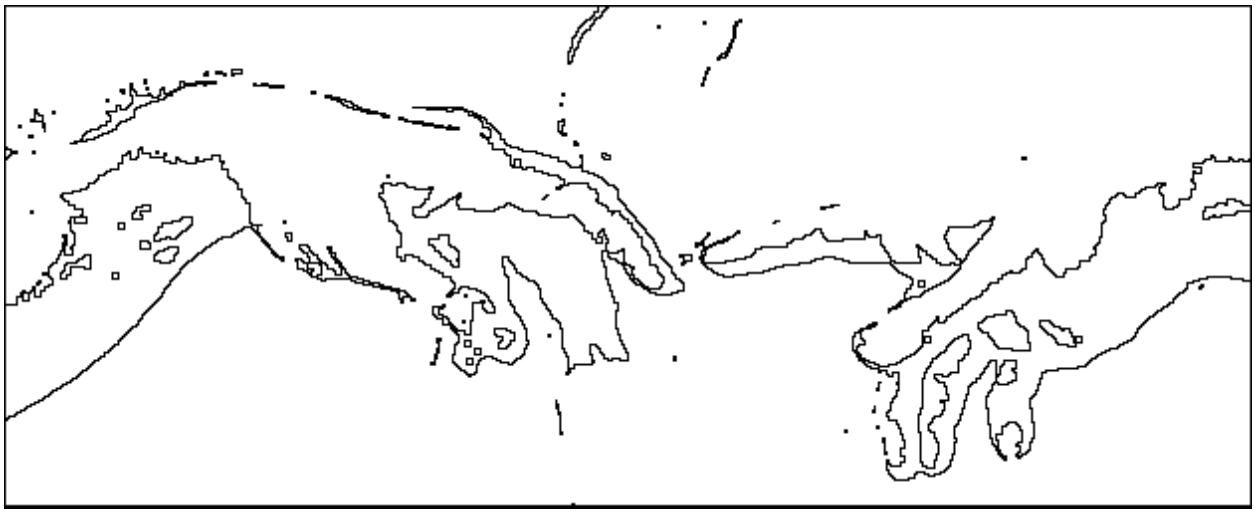
input



output

Example 2: image name: michelan.bmp, image size: 627x253; template name: edge.tem .

input



output

OptimalEdgeDetector: ***Optimal edge detector [43]***

A =

0	0	0
0	0	0
0	0	0

B =

-0.11	0	0.11
-0.28	0	0.28
-0.11	0	0.11

z = 0

I. Global Task

Given: static grayscale image **P**

Input: **U(t) = P**

Initial State: **X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)

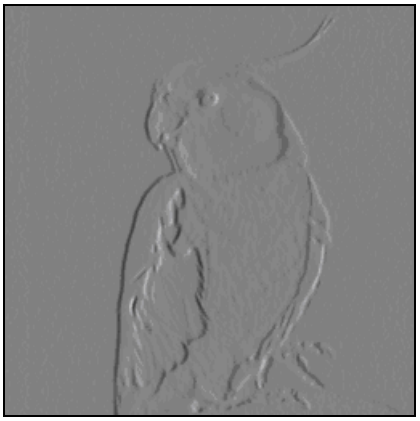
Boundary Conditions: Zero-flux boundary condition (duplicate)

Output: **Y(t)⇒Y(∞)** = Grayscale image representing edges calculated in horizontal direction.

Remark:
The B template represents the optimal edge detector operator.

II. Example: image name: bird.bmp, image size: 256x256; template name: optimedge.tem .

input



output

MaskedObjectExtractor: ***Masked erase*** **[24]**

Old names: ERASMASK

Left-to-right

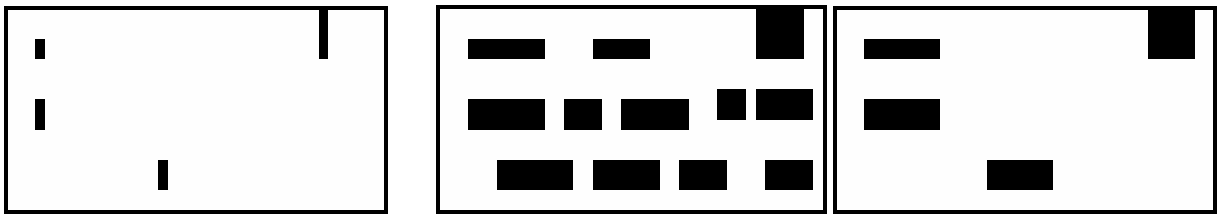
$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1.5 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{-1.5}$$

I. Global Task

Given:	static binary images $\mathbf{P_1}$ (mask) and $\mathbf{P_2}$
Input:	$\mathbf{U(t) = P_1}$
Initial State:	$\mathbf{X}(0) = \mathbf{P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image that is the result of erasing $\mathbf{P_2}$ from left to right. Erasure is stopped by the black walls on the mask ($\mathbf{P_1}$) image.

Remark:
By rotating **A** the template can be sensitized to other directions as well.

II. Example: Left-to-right erase. Image names: ccdmsk3.bmp, ccdmsk2.bmp; image size: 40x20; template name: erasmask.tem .

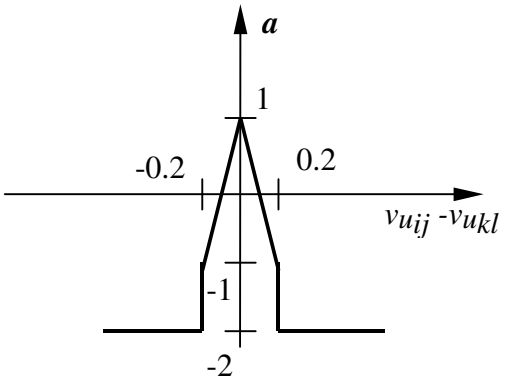


GradientDetection: ***Finds the locations where the gradient of the field is smaller than a given threshold value*** **[9]**

Old names: EXTREME

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{a} & \boldsymbol{a} & \boldsymbol{a} \\ \boldsymbol{a} & 0 & \boldsymbol{a} \\ \boldsymbol{a} & \boldsymbol{a} & \boldsymbol{a} \end{bmatrix} \qquad z = \boxed{z^*}$$

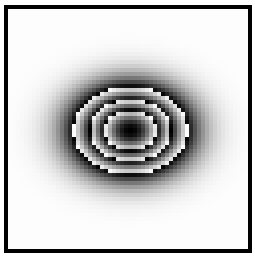
where z^* is a given threshold value, and ***a*** is defined by the following nonlinear function:



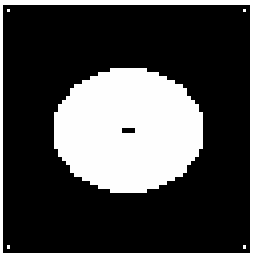
I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the example we choose **X**(0) **= P**)
Boundary Conditions:	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels represent the locations in **P** where the gradient of the field is smaller than a given threshold value.

II. Example: image name: circles.bmp, image size: 60x60; template name: extreme.tem . Threshold value z^* = 3.9 .



input



output

PointExtraction: ***Extracts isolated black pixels***

Old names: FigureRemover, FIGDEL

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

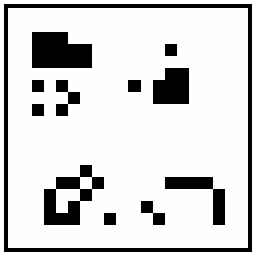
B = | -1 | -1 | -1 |
| --- | --- | --- |
| -1 | 1 | -1 |
| -1 | -1 | -1 |

z = -8

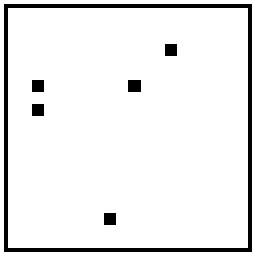
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X(0) =** Arbitrary
Boundary Conditions:	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing all isolated black pixels in **P**.
Template robustness:	ρ = 0.33 .

II. Example: image name: figdel.bmp, image size: 20x20; template name: figdel.tem .



input



output

PointRemoval: Deletes isolated black pixels

Old names: FigureExtractor, FIGEXTR

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad z = \boxed{-1}$$

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X(0) =** Arbitrary

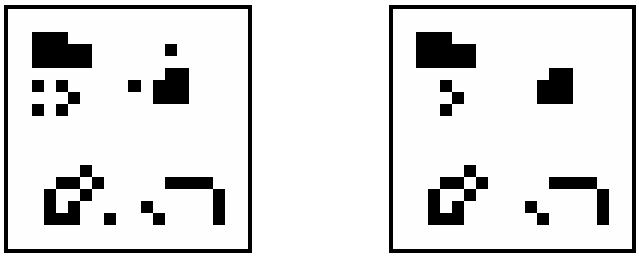
Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

Output: **Y(t)⇒Y(∞)** = Binary image showing all connected components in **P**.

Remark:

Black pixels having no black neighbors are deleted. This template is the opposite of *FigureRemover*.

II. Example: image name: figdel.bmp, image size: 20x20; template name: figextr.tem .



input output

SelectedObjectsExtraction: ***Extracts marked objects***

Old names: FigureReconstructor, FIGREC, RECALL

$\mathbf{A} =$ | 0.5 | 0.5 | 0.5 |
|---|---|---|
| 0.5 | 4 | 0.5 |
| 0.5 | 0.5 | 0.5 |

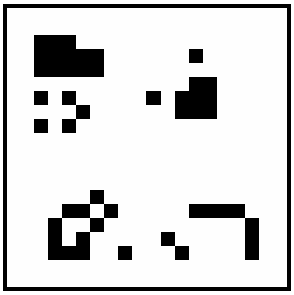
$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 4 | 0 |
| 0 | 0 | 0 |

$z =$ 3

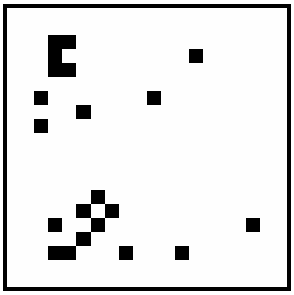
I. Global Task

Given:	two static binary images $\mathbf{P_1}$ (mask) and $\mathbf{P_2}$ (marker). $\mathbf{P_2}$ contains just a part of $\mathbf{P_1}$ ($\mathbf{P_2} \subset \mathbf{P_1}$).
Input:	$\mathbf{U(t) = P_1}$
Initial State:	$\mathbf{X(0) = P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image representing those objects of $\mathbf{P_1}$ which are marked by $\mathbf{P_2}$.
Template robustness:	$\rho = 0.12$.

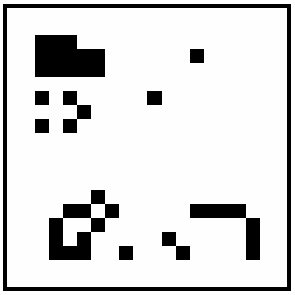
II. Example: image names: figdel.bmp, figrec.bmp; image size: 20x20; template name: figrec.tem



input



initial state



output

FilledContourExtraction: ***Finds solid black framed areas***

Old names: FramedAreasFinder, FINDAREA

$\mathbf{A} =$ | 0 | 1 | 0 |
|---|---|---|
| 1 | 5 | 1 |
| 0 | 1 | 0 |

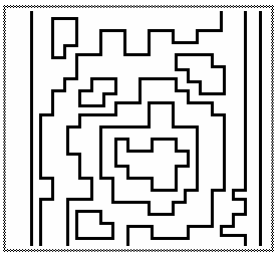
$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 2 | 0 |
| 0 | 0 | 0 |

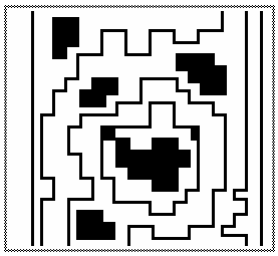
$z =$ -5.25

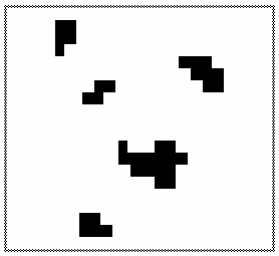
I. Global Task

Given:	two static binary images $\mathbf{P_1}$ and $\mathbf{P_2}$
Input:	$\mathbf{U(t) = P_1}$
Initial State:	$\mathbf{X(0) = P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image representing objects of $\mathbf{P_2}$ which totally fit/fill in closed curves of $\mathbf{P_1}$.

II. Example: image names: findare1.bmp, findare2.bmp; image size: 270x246; template name: findarea.tem .


input


initial state

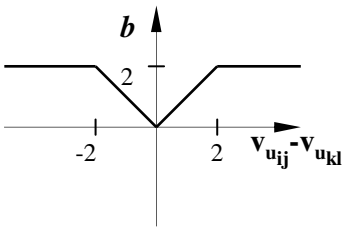

output

ThresholdedGradient: ***Finds the locations where the gradient of the field is higher than a given threshold value*** **[9]**

Old names: GRADIENT

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \\ \boldsymbol{b} & 0 & \boldsymbol{b} \\ \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \end{bmatrix} \qquad z = \boxed{z^*}$$

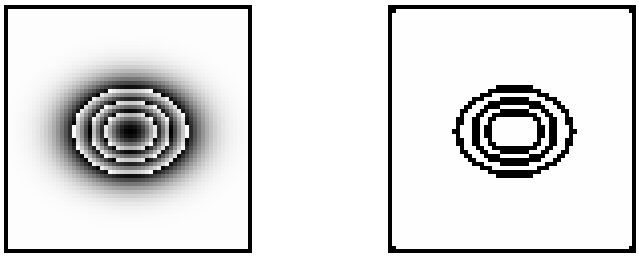
where z^* is a given threshold value, and $\boldsymbol{b}$ is defined by the following nonlinear function:



I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the example we choose **X**(0) **= P**)
Boundary Conditions:	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels represent the locations in **P** where the gradient of the field is higher than a given threshold value.

II. Example: image name: circles.bmp, image size: 60x60; template name: gradient.tem . Threshold value z^* = -4.8 .



input output

3x3Halftoning: 3x3 image halftoning

Old names: HLF3, HLF33

$$\mathbf{A} = \begin{bmatrix} -0.07 & -0.1 & -0.07 \\ -0.1 & 1+\varepsilon & -0.1 \\ -0.07 & -0.1 & -0.07 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0.07 & 0.1 & 0.07 \\ 0.1 & 0.32 & 0.1 \\ 0.07 & 0.1 & 0.07 \end{bmatrix} \quad z = \boxed{0}$$

I. Global Task

Given: static grayscale image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= P**

Boundary Conditions: Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0

Output: **Y(t)⇒Y(∞)** = Binary image preserving the main features of **P**.

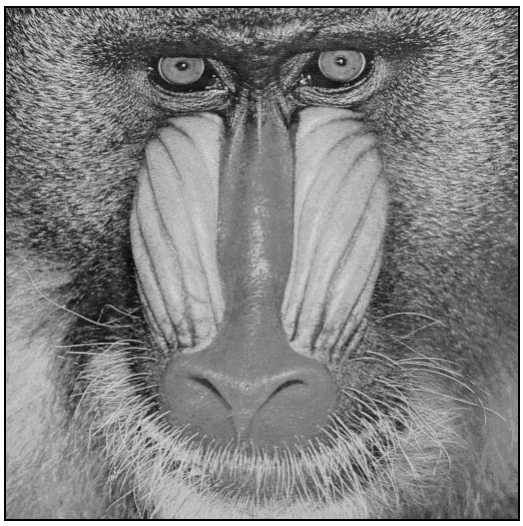
Remark:

The speed of convergence is controlled by ε≈[0.1...1]. The greater the ε is, the faster the process and the rougher the result will be. The inverse of the template is *3x3InverseHalftoning*. The result is acceptable in the Square Error measure [17,35].

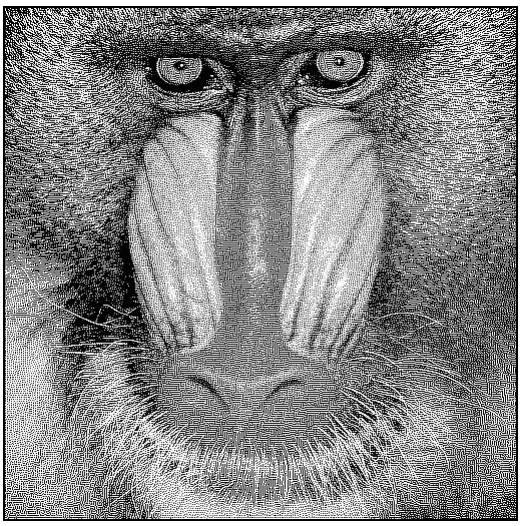
This template is called "Half-Toning" in [44].

II. Examples

Example 1: image name: baboon.bmp, image size: 512x512; template name: hlf3.tem .



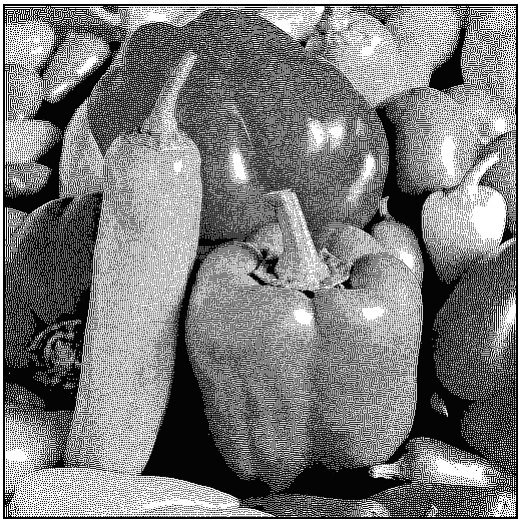
input



output

Example 2: image name: peppers.bmp, image size: 512x512; template name: hlf3.tem .

input



output

5x5Halftoning1: ***5x5 image halftoning*** **[15]**

Old names: HLF5KC, HLF55_KC

A =

-0.03	-0.09	-0.13	-0.09	-0.03
-0.09	-0.36	-0.60	-0.36	-0.09
-0.13	-0.60	1.05	-0.60	-0.13
-0.09	-0.36	-0.60	-0.36	-0.09
-0.03	-0.09	-0.13	-0.09	-0.03

B =

0	0	0.07	0	0
0	0.36	0.76	0.36	0
0.07	0.76	2.12	0.76	0.07
0	0.36	0.76	0.36	0
0	0	0.07	0	0

z = 0

I. Global Task

Given: static grayscale image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= P**

Boundary Conditions: Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0

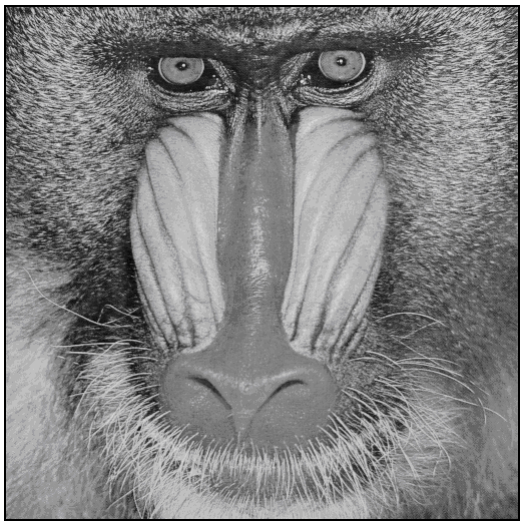
Output: **Y(t)⇒Y(∞)** = Binary image preserving the main features of **P**.

Remark:

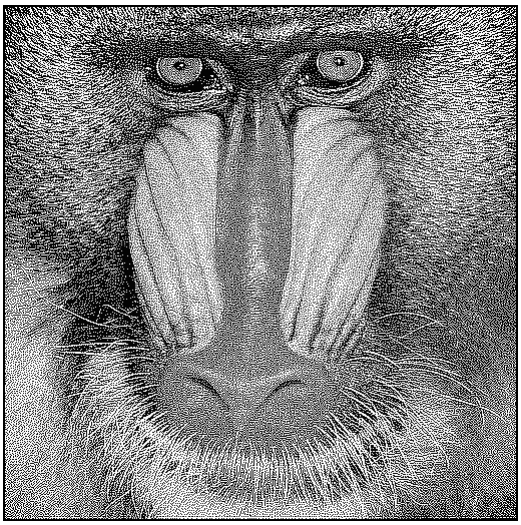
The output image quality is optimized by considering human visualization. When simulating the behavior of the CNN by using the forward Euler integration form, the time step should be less than 0.4.

II. Examples

Example 1: image name: baboon.bmp, image size: 512x512; template name: hlf5kc.tem .



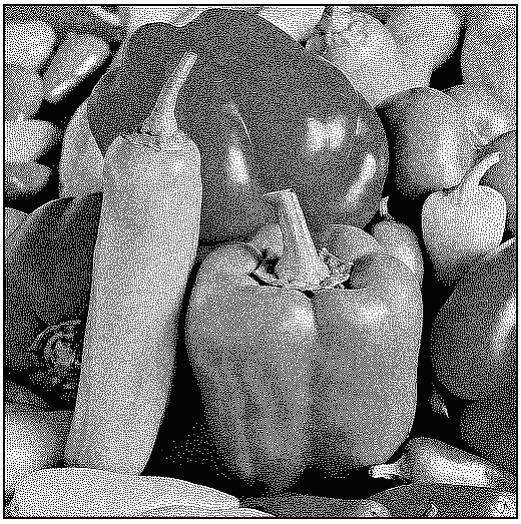
input



output

Example 2: image name: peppers.bmp, image size: 512x512; template name: hlf5kc.tem .

input



output

5x5Halftoning2: 5x5 image halftoning

Old names: HLF5, HLF55

A =

-0.02	-0.07	-0.10	-0.07	-0.02
-0.07	-0.32	-0.46	-0.32	-0.07
-0.10	-0.46	1.05	-0.46	-0.10
-0.07	-0.32	-0.46	-0.32	-0.07
-0.02	-0.07	-0.10	-0.07	-0.02

B =

0.02	0.07	0.10	0.07	0.02
0.07	0.32	0.46	0.32	0.07
0.10	0.46	0.81	0.46	0.10
0.07	0.32	0.46	0.32	0.07
0.02	0.07	0.10	0.07	0.02

z = 0

I. Global Task

Given: static grayscale image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= P**

Boundary Conditions: Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0

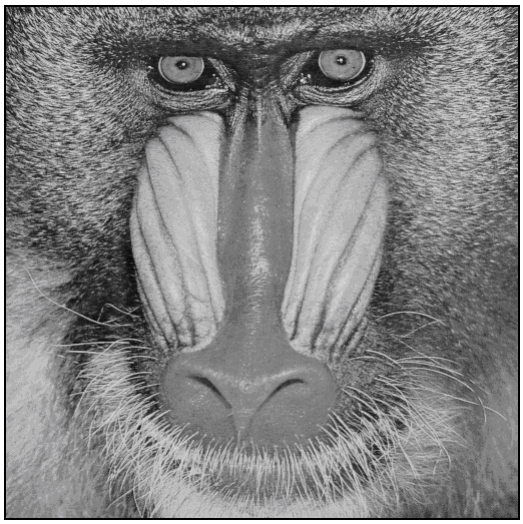
Output: **Y(t)⇒Y(∞)** = Binary image preserving the main features of **P**.

Remark:

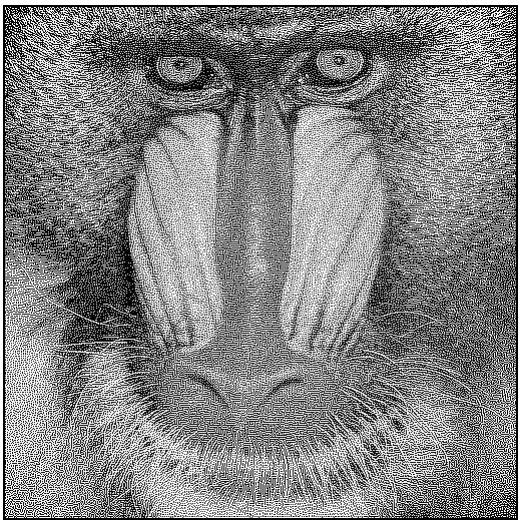
The inverse of the template is *5x5InverseHalftoning*. The result is optimal in the Square Error measure [17,35].

II. Examples

Example 1: image name: baboon.bmp, image size: 512x512; template name: hlf5.tem .



input



output

Example 2: image name: peppers.bmp, image size: 512x512; template name: hlf5.tem .

input

output

Hole-Filling: ***Fills the interior of all closed contours*** **[6]**

Old names: HoleFiller, HOLE

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{-1}$$

I. Global Task

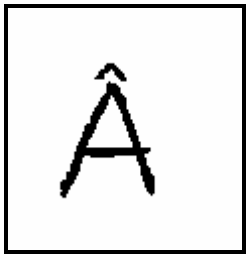
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= 1**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing **P** with holes filled.

Remark:

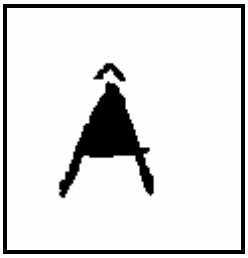
(i) this is a propagating template, the computing time is proportional to the length of the image

(ii) a more powerful template is the *ConcaveLocationFiller* template in this library.

II. Example: image name: a_letter.bmp, image size: 117x121; template name: hole.tem .



input



output

ObjectIncreasing: ***Increases the object by one pixel (DTCNN)*** **[16]**

Old names: INCREASE

A = | 0.5 | 0.5 | 0.5 |
|---|---|---|
| 0.5 | 0.5 | 0.5 |
| 0.5 | 0.5 | 0.5 |

B = | 0 | 0 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = 4

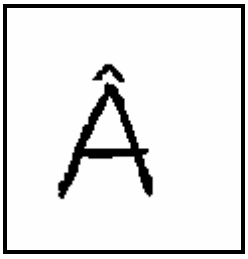
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	**Y(1)** = Binary image representing the objects of **P** increased by 1 pixel in all direction.

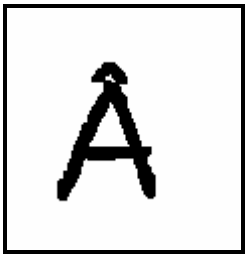
Remark:

Increasing the size of an object by *N* pixels in all directions can be achieved by *N* iteration steps of a DTCNN.

II. Example: image name: a_letter.bmp, image size: 117x121; template name: increase.tem . One iteration step of a DTCNN is performed.



input



output

3x3InverseHalftoning: ***Inverts the halftoned image by a 3x3 template***

Old names: INVHLF3, INVHLF33

$\mathbf{A} =$

0	0	0
0	0	0
0	0	0

$\mathbf{B} =$

0.07	0.1	0.07
0.1	0.32	0.1
0.07	0.1	0.07

$z =$ 0

I. Global Task

Given: static binary image **P** obtained by using the 3x3Halftoning template

Input: **U(t) = P**

Initial State: **X**(0) **=** Arbitrary

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

Output: **Y(t)⇒Y(∞)** = Grayscale image representing **P**.

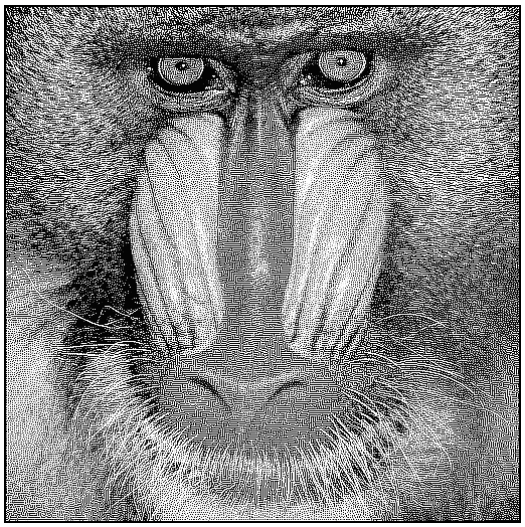
Remark:

Inverts the 3*3 halftoned image created by the *3x3Halftoning* template. The result lacks fine edges as the control of *3x3Halftoning* smoothes the input [17,35].

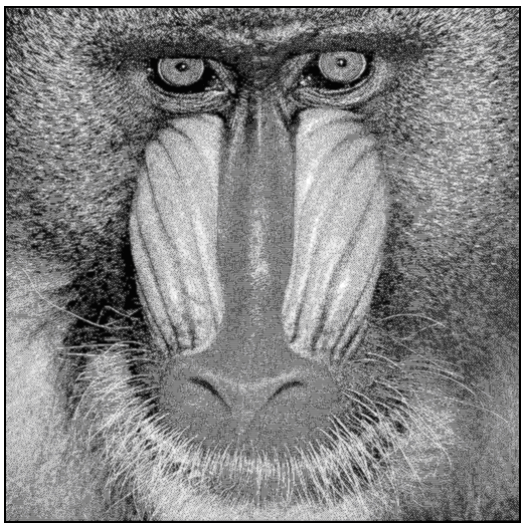
This template is called "Inverse Half-Toning" in [44].

II. Examples

Example 1: image name: invhlf3_1.bmp, image size: 512x512; template name: invhlf3.tem .

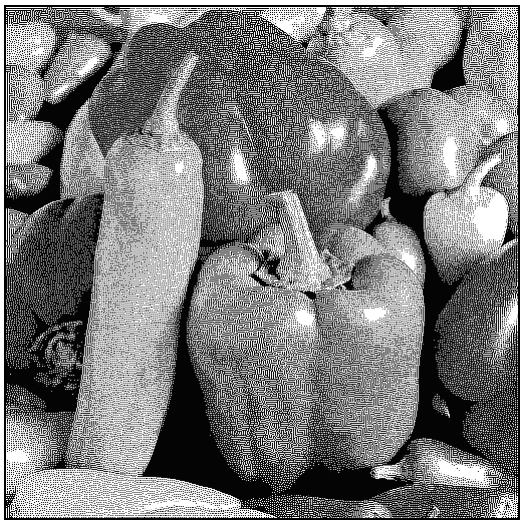


input

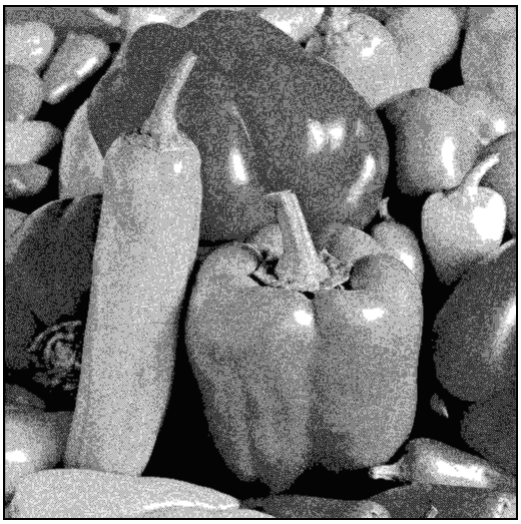


output

Example 2: image name: invhlf3_2.bmp, image size: 512x512; template name: invhlf3.tem .



input



output

5x5InverseHalftoning: ***Inverts the halftoned image by a 5x5 template***

Old names: INVHLF5, INVHLF55

A =

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

B =

0	0.01	0.02	0.01	0
0.01	0.06	0.09	0.06	0.01
0.02	0.09	0.16	0.09	0.02
0.01	0.06	0.09	0.06	0.01
0	0.01	0.02	0.01	0

z = 0

I. Global Task

Given: static binary image **P** obtained by using the 5x5Halftoning2 template

Input: **U(t) = P**

Initial State: **X**(0) **=** Arbitrary

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

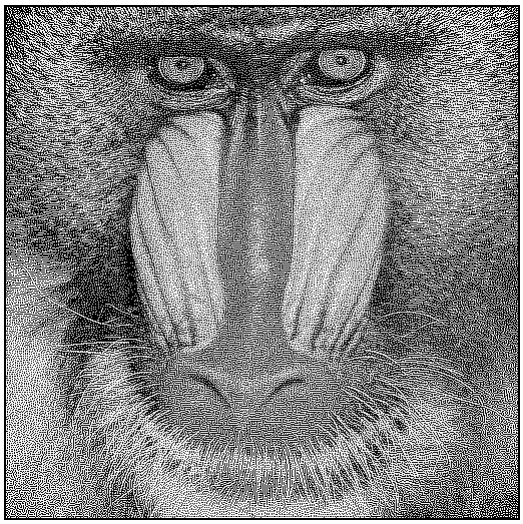
Output: **Y(t)⇒Y(∞)** = Grayscale image representing **P**.

Remark:

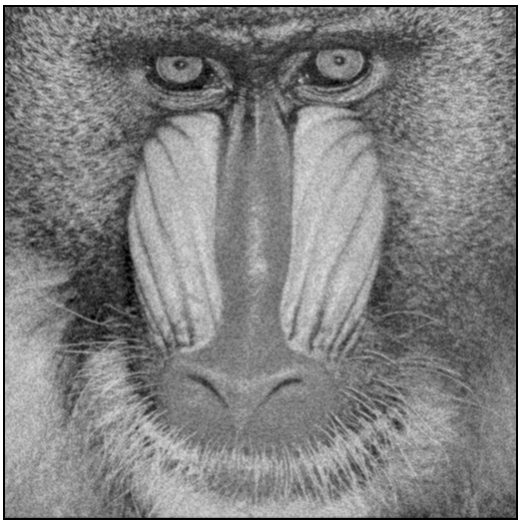
Inverts the 5*5 halftoned image created by the *5x5Halftoning2* template. The result lacks fine edges as the control of *5x5Halftoning2* smoothes the input.

II. Examples

Example 1: image name: invhlf5_1.bmp, image size: 512x512; template name: invhlf5.tem .



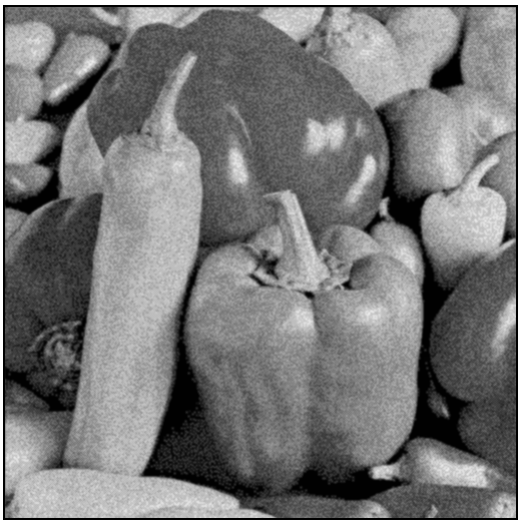
input



output

Example 2: image name: invhlf5_2.bmp, image size: 512x512; template name: invhlf5.tem .

input



output

LocalSouthernElementDetector: ***Local southern element detector*** **[11]**

Old names: LSE

A = | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| -1 | -1 | -1 |

z = -3

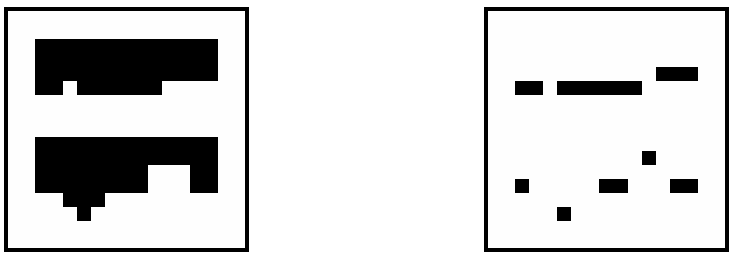
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing local southern elements of objects in **P**.

Remark:

Local southern elements are pixels having neither south-western, nor southern or south-eastern neighbors.

II. Example: image name: lcp_lse.bmp, image size: 17x17; template name: lse.tem .



input output

PatternMatchingFinder: ***Finds matching patterns***

Old names: MATCH

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \\ \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \\ \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \end{bmatrix} \qquad z = \boxed{-N+0.5}$$

where

$$\boldsymbol{b} = \begin{cases} 1, & \text{if corresponding pixel is required to be black} \\ 0, & \text{if corresponding pixel is do not care} \\ -1, & \text{if corresponding pixel is required to be white} \end{cases}$$

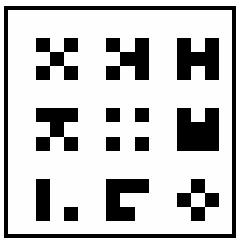
N = number of pixels required to be either black or white, i.e. the number of non-zero values in the **B** template

I. Global Task

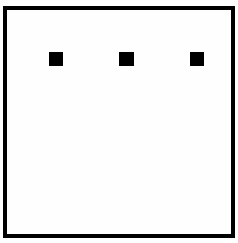
Given:	static binary image **P** possessing the 3x3 pattern prescribed by the template
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the locations of the 3x3 pattern prescribed by the template. The pattern having a black/white pixel where the template value is +1/-1, respectively, is detected.

II. Example: image name: match.bmp, image size: 16x16; template name: match.tem .

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & 0 \\ 1 & -1 & 1 \end{bmatrix} \qquad z = \boxed{-6.5}$$



input



output

LocalMaximaDetector: ***Local maxima detector template*** **[33]**

Old names: MAXLOC

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \\ \boldsymbol{b} & 0 & \boldsymbol{b} \\ \boldsymbol{b} & \boldsymbol{b} & \boldsymbol{b} \end{bmatrix} \qquad z = \boxed{-3.5}$$

where

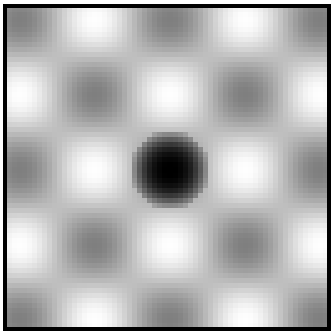
$$\boldsymbol{b} = \begin{cases} 0.5 & \text{if } v_{u_{ij}} - v_{u_{kl}} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

I. Global Task

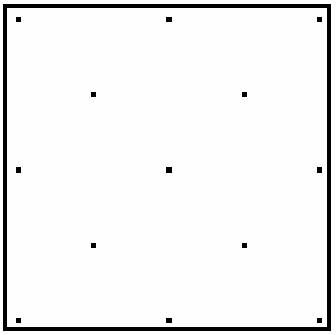
Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= 0**
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the locations of all local maxima in the specified local neighborhood.

Remark:
All local minima can also be detected if the input image is inverted.

II. Example: image name: avergra1.bmp, image size: 64x64; template name: maxloc.tem .



input



output

MedianFilter: ***Removes impulse noise from a grayscale image*** **[34]**

Old names: MEDIAN

CNN base model of the median filter (linear region of f):

$$C\frac{d}{dt}v_{x_{ij}}(t) = -R^{-1}v_{x_{ij}}(t) + Af(v_{x_{ij}}) + \sum_{kl\in N_r}\hat{D}^{M}{}_{ij,kl}(v_{x_{ij}}(t) - v_{u_{kl}}(t))$$

The corresponding CNN template (T_M) ($R=1, C=1, Q=1$):

$$\mathbf{A} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \qquad \hat{D}^{M} = \begin{array}{|c|c|c|}\hline \boldsymbol{d} & \boldsymbol{d} & \boldsymbol{d} \\ \hline \boldsymbol{d} & 0 & \boldsymbol{d} \\ \hline \boldsymbol{d} & \boldsymbol{d} & \boldsymbol{d} \\ \hline \end{array} \qquad z = \begin{array}{|c|}\hline 0 \\ \hline \end{array}$$

where $\boldsymbol{d}$ = -Q sign($v_{x_{ij}}$ - $v_{u_{kl}}$).

Special types derived from the base model:

- **(i)** **Rank Order Class** **-** simply varying the bias value
  / e.g. **Min filter** : I=+8Q, **Max filter** :I=-8Q /
- **(ii)** **Weighted Median** **-** locally space variant weighting
  / e.g. Plus-shape Median Filter (Q=0 in the corners) /
- **(iii)** **Cascade Median** **-** consecutive steps with T_M,
  intermediate result is stored in LAMs of the CNNUM
- **(iv)** **Selective Median** **-** filtering is made only at the locations
  of the local extremities with T_M - fixed-state CNN model
- **(v)** **MinMax, MaxMin, Pseudo Median -** combining (i)-(iv)

I. Global Task

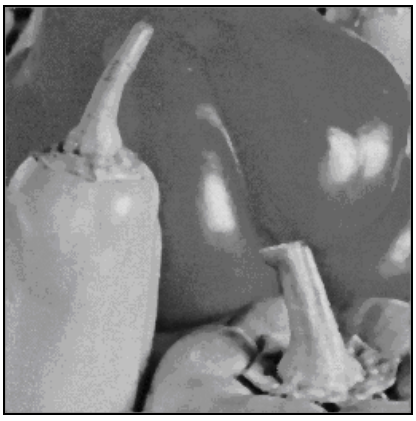
Given:	static grayscale image **P** with impulse noise
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, u_{ij} = 0 for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Grayscale image representing **P** filtered from impulse noise.

Remark:

Impulse noise is removed only if the impulses are placed further from each other than half of the window width (r+1).

II. Example: image name: median.bmp, image size: 256x256; template name: median.tem .

input



output

LeftPeeler: ***Peels one pixel from the left*** **[14]**

Old names: PEELHOR

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

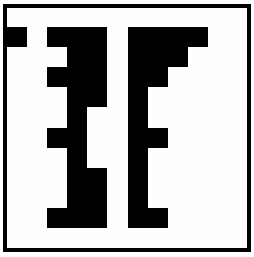
B = | 0 | 0 | 0 |
| --- | --- | --- |
| 1 | 1 | 0 |
| 0 | 0 | 0 |

z = | -1 |
| --- |

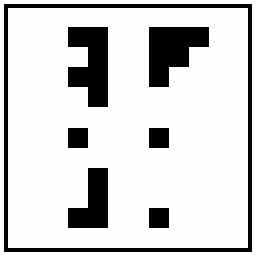
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, u_{ij} = -1 for all virtual cells, denoted by [U]= -1
Output:	**Y(t)⇒Y(∞)** = Binary image representing the objects of **P** peeled with one pixel from the left.
Template robustness:	ρ = 0.71 .

II. Example: image name: peelhor.bmp, image size: 12x12; template name: peelhor.tem .



input



output

RightEdgeDetection: Extracts right edges of objects

Old names: RightContourDetector, RIGHTCON

$\mathbf{A} =$

0	0	0
0	1	0
0	0	0

$\mathbf{B} =$

0	0	0
1	1	-1
0	0	0

$z =$ -2

I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the right edges of objects in **P**.
Template robustness:	ρ = 0.58 .

Remark:
By rotating **B** the template can be sensitized to other directions as well.

II. Example: image name: chineese.bmp, image size: 16x16; template name: rightcon.tem .

input output

***MaskedShadow: Masked shadow* [24]**

Old names: SHADMASK, MASKSHAD

Left-to-right

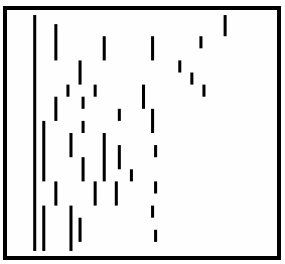
$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 1.5 & 1.8 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1.2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{0}$$

I. Global Task

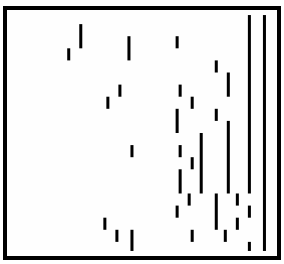
Given:	static binary images $\mathbf{P_1}$ (mask) and $\mathbf{P_2}$
Input:	$\mathbf{U(t) = P_1}$
Initial State:	$\mathbf{X}(0) = \mathbf{P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = -1$ for all virtual cells, denoted by [Y]= -1
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image representing the result of pattern propagation of $\mathbf{P_2}$ in a particular direction. The propagation goes from the direction of the non-zero off-center feedback template entry $\mathbf{A}_{ij}$ and is halted by the mask $\mathbf{P_1}$.

Remark:
By rotating **A** the template can be sensitized to other directions as well.

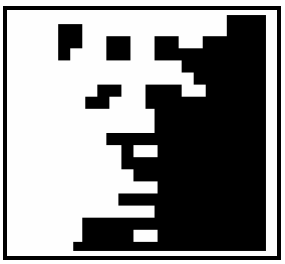
II. Example: Right-to-left horizontal shadow. Image names: shdmsk1.bmp, shdmsk2.bmp; image size: 270x246; template name: shadmask.tem .



input



initial state



output

***ShadowProjection: Projects onto the left the shadow of all objects illuminated from the right* [6]**

Old names: LeftShadow, SHADOW

A = | 0 | 0 | 0 |
| 0 | 2 | 2 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

z = 0

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= 1**

Boundary Conditions: Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0

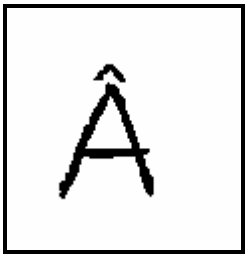
Output: **Y(t)⇒Y(∞)** = Binary image representing the left shadow of the objects in **P**.

Remark:

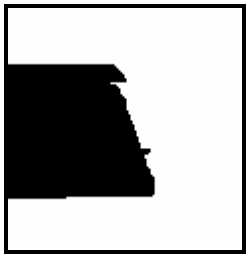
By modifying the position of the off-center **A** template element the template can be sensitized to other directions as well.

II. Examples

Example 1: Left shadow. Image name: a_letter.bmp, image size: 117x121; template name: shadow.tem .



input



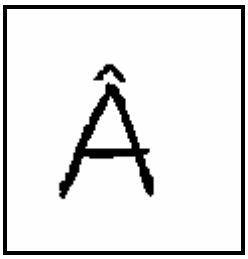
output

Example 2: Shadow in the east-western direction. Image name: a_letter.bmp, image size: 117x121.

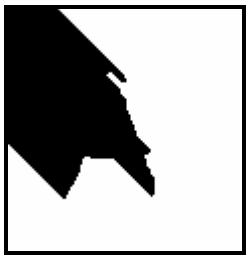
A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 2 |

B = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

z = 0



input



output

VerticalShadow: ***Vertical shadow template***

Old names: SHADSIM, SUPSHAD

A = | 0 | 1 | 0 |
| 0 | 2 | 0 |
| 0 | 1 | 0 |

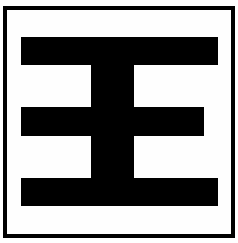
B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = 2

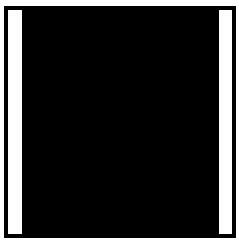
I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	**Y(t)⇒Y(∞)** = Binary image representing the vertical shadow of the objects in **P** taken upward and downward simultaneously.

II. Example: image name: chineese.bmp, image size: 16x16; template name: shadsim.tem .



input



output

DirectedGrowingShadow: ***Generate growing shadows starting from black points***

SHADOW0:

$$\mathbf{A} = \begin{bmatrix} 0.4 & 0.3 & 0 \\ 1 & 2 & -1 \\ 0.4 & 0.3 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1.4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{2.5}$$

SHADOW45:

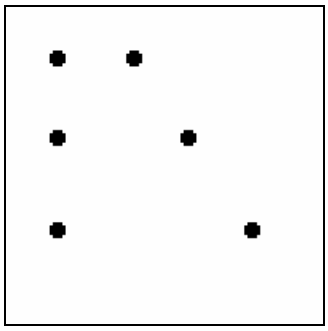
$$\mathbf{A} = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1.4 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{2.5}$$

I. Global Task

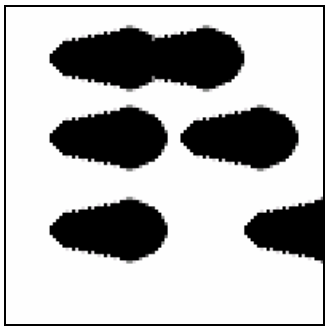
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = -1 for all virtual cells, denoted by [Y]=-1
Output:	**Y(t)⇒Y(∞)** = Binary image in which shadows are generated starting from black pixels. During the transient shadows become wider and wider.

Remark:
Other directions can be gained by appropriate rotation and flipping of the templates.

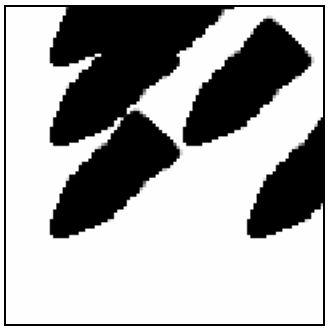
II. Example: image name: points.bmp; image size: 100x100; template names: shadow0.tem, shadow45.tem.



input



output of shadow0 template ($t=35\tau_{CNN}$)



output of shadow45 template ($t=45\tau_{CNN}$)

Threshold: ***Grayscale to binary threshold template***

Old names: TRESHOLD

$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z = \boxed{-z^*}$, $-1 < z^* < 1$

I. Global Task

Given:	static grayscale image **P** and threshold z^*
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Output:	**Y(t)⇒Y(∞)** = Binary image where black pixels correspond to pixels in **P** with grayscale intensity $p_{ij} > z^*$.

II. Example: Threshold value $z^* = 0.4$. Image name: madonna.bmp, image size: 59x59; template name: treshold.tem .

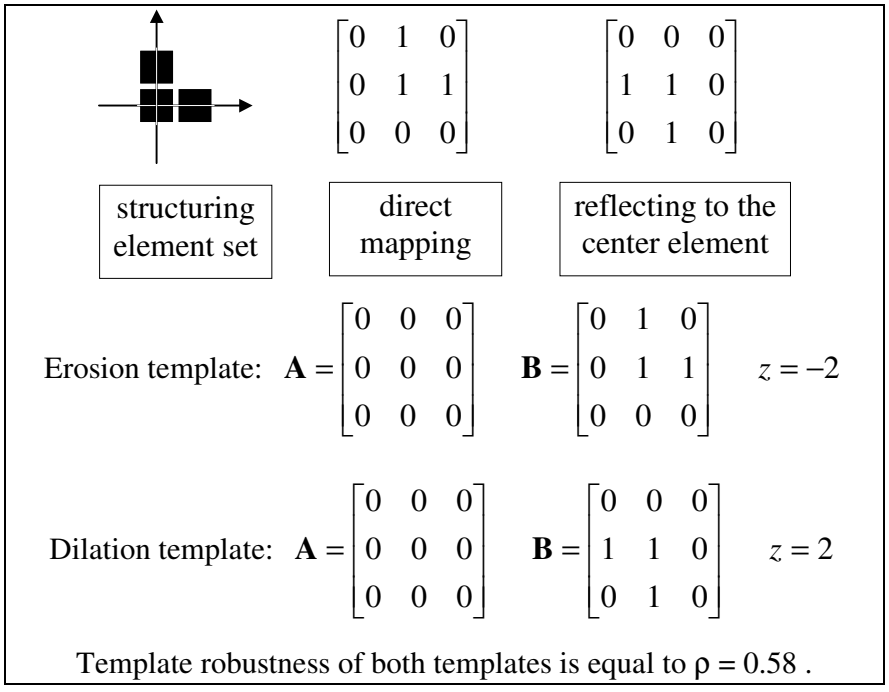
input

output

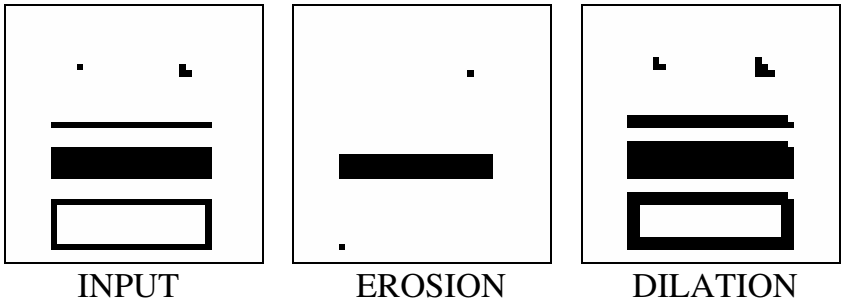
1.2. MATHEMATICAL MORPHOLOGY

BINARY MATHEMATICAL MORPHOLOGY

The basic operations of binary mathematical morphology [32] are erosion and dilation. These operations are defined by two binary images, one being the operand, the other the structuring element. In the CNN implementation, the former image is the input, while the function (the templates) itself depends on the latter image. If the structuring element set does not exceed the size of the CNN template the dilation and erosion operators can be implemented with a single CNN template. The implementation method is the following: The **A** template matrix is zero in every position. The structuring element set should be directly mapped to the **B** template (See Figure). If it is an erosion operator, the z value is equal to (1-n), where n is the number of 1s in the **B** template matrix. If it is a dilation operator, the **B** template must be reflected to the center element, and the z value is equal to (n-1), where n is the number of 1s in the **B** template matrix. When calculating the operator, the image should be put to the input of the CNN, and the initial condition is zero everywhere. The next Figure shows an example for template syntetization.



Example: Erosion and dilation with the given structuring element set. Image name: binmorph.bmp; image size: 40x40; template names: eros_bin.tem, dilat_bin.tem .



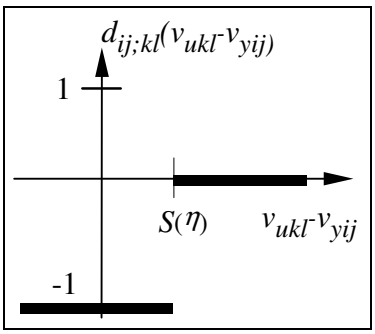
GRAYSCALE MATHEMATICAL MORPHOLOGY

The basic operations of grayscale mathematical morphology [32] are erosion and dilation. These operations are defined by two grayscale images, one being the operand, the other the structuring element set (S). In the CNN implementation, the former image is the input, while the function (the templates) itself depends on the latter image. If the structuring element set does not exceed the size of the CNN template the dilation and erosion operators can be implemented with a single CNN template. The implementation method is the following: single template grayscale mathematical morphology is implemented on a slightly modified CNN. The state equation of the modified CNN is the following:

$$\dot{v}_{xij}(t) = -v_{xij}(t) + v_{yij}(t) + \sum_{kl \in N_r(ij)} \hat{D}_{ij;kl}(v_{ukl}(t) - v_{yij}(t)) + z$$

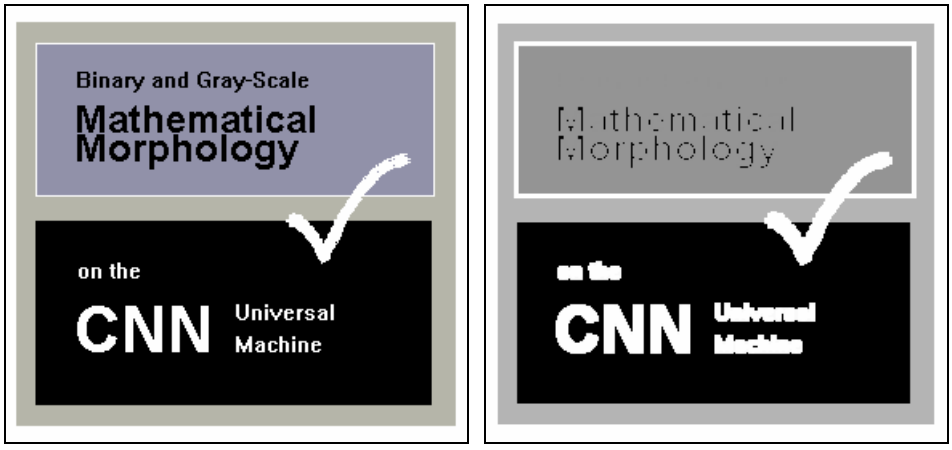
It means that **A**=[1], and the inputs of the nonlinear functions of the **D** template are the difference between the input values and the appropriate neighborhood positions, and the center output value. The morphological operation can be implemented with a single template on this CNN structure.
The erosion template is the following:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} d_{-1,1} & d_{0,1} & d_{1,1} \\ d_{-1,0} & d_{0,0} & d_{1,0} \\ d_{-1,-1} & d_{0,-1} & d_{1,-1} \end{bmatrix}, \quad z = 1$$



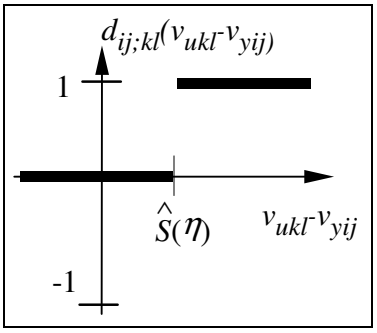
where: $S(\eta)$ is the real value of the structuring element set (S) at point $\eta=(k,l)$. If it is not defined $d_{ij;kl} \equiv 0$.

Example for grayscale erosion with a 3x3 square shaped zero value structuring element set. The black areas have shrunk. Image name: grmorph.bmp; image size: 288x272.



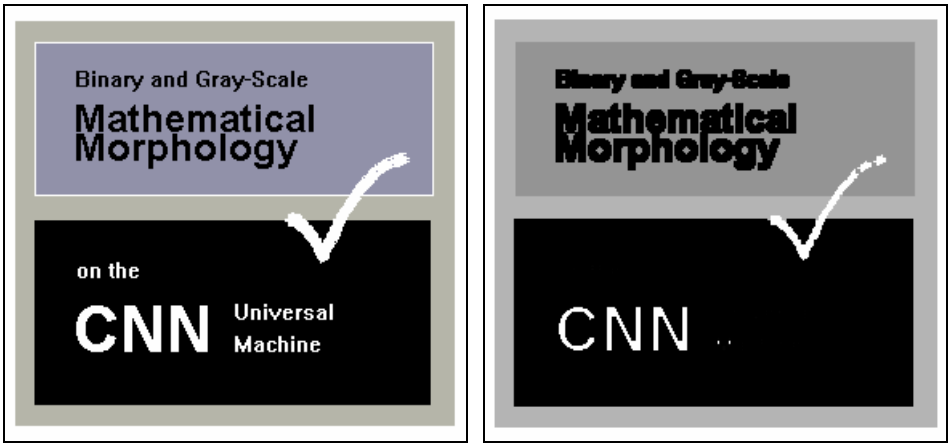
The dilation template is the following:

$$\mathbf{A}=\begin{bmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix},\quad \mathbf{D}=\begin{bmatrix}d_{-1,1} & d_{0,1} & d_{1,1}\\ d_{-1,0} & d_{0,0} & d_{1,0}\\ d_{-1,-1} & d_{0,-1} & d_{1,-1}\end{bmatrix},\quad z=-1$$



where: $\hat{S}(\eta)$ is the real value of the inverted and reflected structuring element set ($\hat{S}=-S(-x)$) at point $\eta=(k,l)$. If it is not defined $d_{ij;kl}\equiv 0$.

Example for grayscale dilation with a 3x3 square shaped zero value structuring element set. The black areas have dilated. Image name: grmorph.bmp; image size: 288x272.



1.3. SPATIAL LOGIC

BlackFiller: ***Drives the whole network into black***

Old names: FILBLACK, BLACK

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 4 |
| --- |

I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X(0) = P**
Output:	**Y(t)⇒Y(∞)** = black image (all pixels are black)

II. Example: image name: madonna.bmp, image size: 59x59; template name: filblack.tem .

initial state

output

WhiteFiller: ***Drives the whole network into white***

Old names: FILWHITE, WHITE

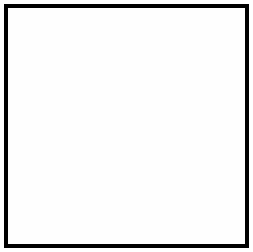
$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{-4}$$

I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X**(0) **= P**
Output:	**Y(t)⇒Y(∞)** = white image (all pixels are white)

II. Example: image name: madonna.bmp, image size: 59x59; template name: filwhite.tem .

initial state



output

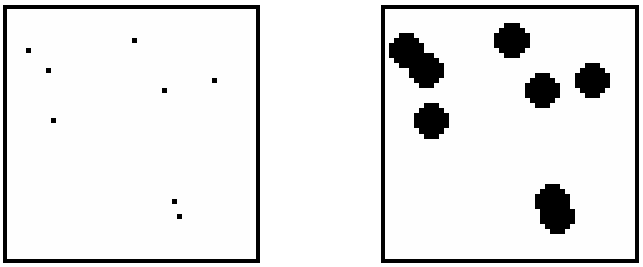
BlackPropagation: ***Starts omni-directional black propagation from black pixels [54]***

$$\mathbf{A} = \begin{bmatrix} 0.25 & 0.25 & 0.25 \\ 0.25 & 3 & 0.25 \\ 0.25 & 0.25 & 0.25 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{3.75}$$

I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X(0) = P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)** = Binary image showing black objects in **P** with increasing black neighborhood (white objects decreasing in size).

II. Example: image name: points.bmp, image size: 50x50; template name: bprop.tem .



input output

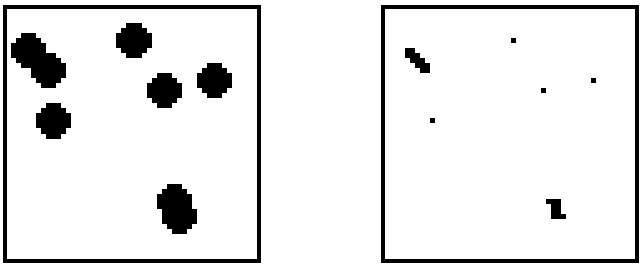
WhitePropagation: ***Starts omni-directional white propagation from white pixels [54]***

$$\mathbf{A} = \begin{bmatrix} 0.25 & 0.25 & 0.25 \\ 0.25 & 3 & 0.25 \\ 0.25 & 0.25 & 0.25 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{-3.75}$$

I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	**X(0) = P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)** = Binary image showing white objects in **P** with increasing white neighborhood (black objects decreasing in size).

II. Example: image name: patches.bmp, image size: 50x50; template name: wprop.tem .



input output

ConcaveLocationFiller: ***Fills the concave locations of objects*** **[22]**

Old names: HOLLOW

$$\mathbf{A} = \begin{bmatrix} 0.5 & 0.5 & 0.5 \\ 0.5 & 2 & 0.5 \\ 0.5 & 0.5 & 0.5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{3.25}$$

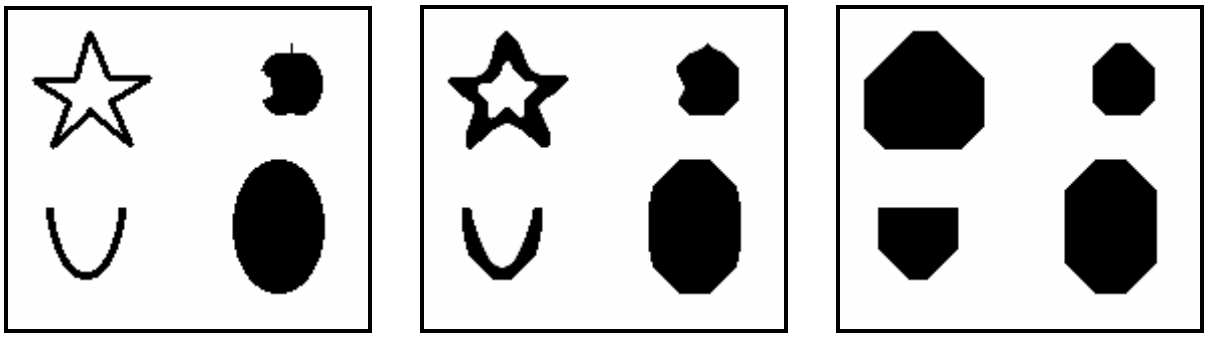
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image in which the concave locations of objects are black.

Remark:

In general, the objects of **P** that are not filled should have at least a 2-pixel-wide contour. Otherwise the template may not work properly.
The template transforms all the objects to solid black concave polygons with vertical, horizontal and diagonal edges only.

II. Example: image name: hollow.bmp, image size: 180x160; template name: hollow.tem .



input output ($t=20\tau_{CNN}$) output ($t=\infty$)

ConcaveArcFiller: ***Fills the concave arcs of objects to prescribed direction***

FILL35:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{2}$$

FILL65:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{3}$$

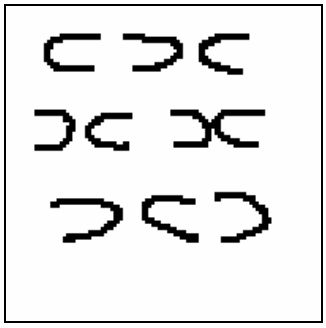
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, y_{ij} = -1 for all virtual cells, denoted by [Y]=-1
Output:	**Y(t)⇒Y(∞)** = Binary image in which those arcs of objects are filled which have a prescribed orientation.

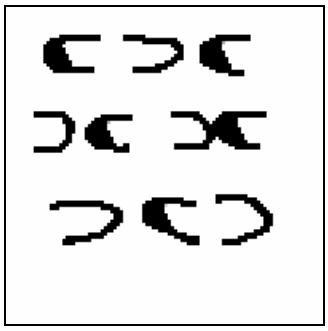
Remark:

In general, the objects of **P** that are not filled should have at least 2 pixel wide contour. Otherwise the template may not work correctly.

II. Example: image name: arcs.bmp, image size: 100x100; template name: arc_fill.tem .



input



output ($t=20\tau_{CNN}$)

SurfaceInterpolation: Interpolates a smooth surface through given points

Old names: INTERP, INTERPOL

A =

0	0	-2	0	0
0	-4	16	-4	0
-2	16	-39	16	-2
0	-4	16	-4	0
0	0	-2	0	0

B =

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

z = 0

I. Global Task

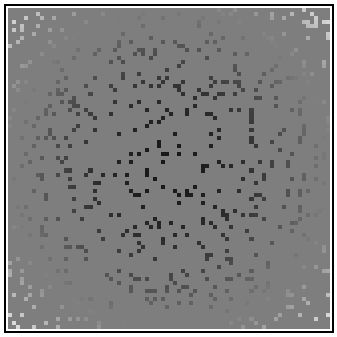
Given:	a static grayscale image $\mathbf{P_1}$ and a static binary image $\mathbf{P_2}$
Input:	**U(t) =** Arbitrary or as a default U(t)=0
Initial State:	$\mathbf{X}(0) = \mathbf{P_1}$
Fixed State Mask:	$\mathbf{X_{fix}} = \mathbf{P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Grayscale image representing an interpolated surface that fits the given points and is as smooth as possible.

Remark:

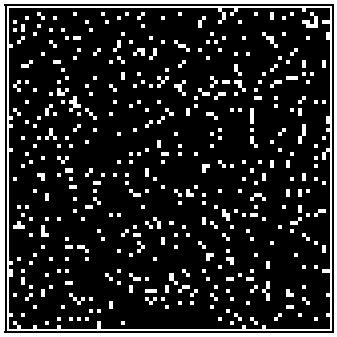
Images $\mathbf{P_1}$ and $\mathbf{P_2}$ are to be constructed as follows: if the altitude of the surface to be interpolated is known at point (i,j), it is preset in $\mathbf{P_1}$ (state) at the position (i,j), and the state is kept fixed ($\mathbf{P_2}$(i,j) = -1) during the transient. If the altitude is not known, then zero is filled into the state ($\mathbf{P_1}$(i,j) = 0) and changing of the state is allowed ($\mathbf{P_2}$(i,j) = 1). For exact solution the feedback template must be space variant at the borders. An approximate result can be obtained by using space invariant network. Further information about the space variant network is available in [27] and [29].

II. Examples

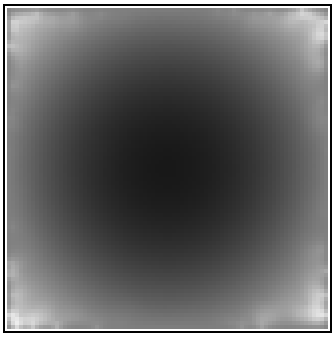
Example 1: Ball surface reconstruction (10% of the points is known). Image names: interp1.bmp, interp2.bmp; image size: 80x80; template name: interp.tem .



initial state

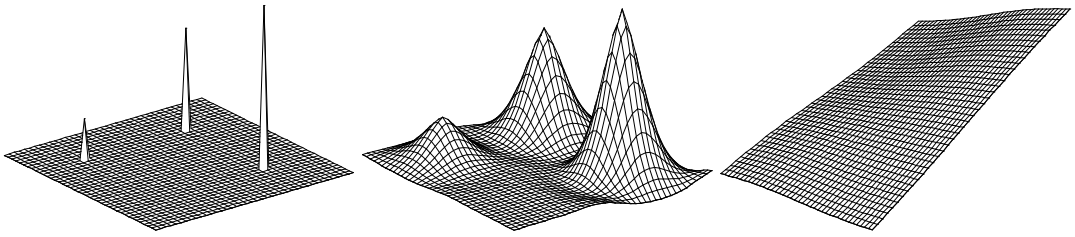


fixed state mask



output

Example 2: Fitting a surface on three given points. Image size: 80x80.



initial state intermediate step output

JunctionExtractor: ***Extracts the junctions of a skeleton*** **[22]**

Old names: JUNCTION

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | 1 | 1 | 1 |
| --- | --- | --- |
| 1 | 6 | 1 |
| 1 | 1 | 1 |

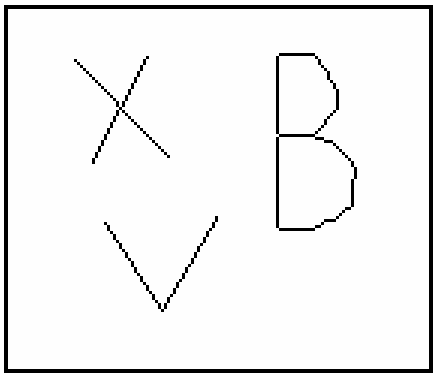
z = -3

I. Global Task

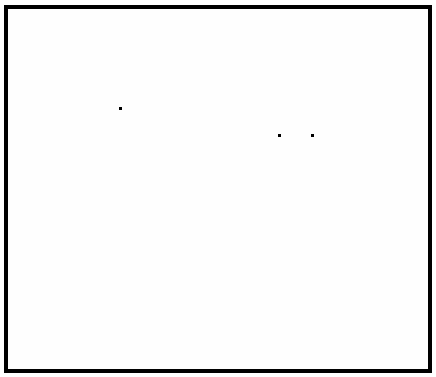
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, u_{ij} = -1 for all virtual cells, denoted by [U] = -1
Output:	**Y(t)⇒Y(∞)** = Binary image showing the junctions of a skeleton.
Template robustness:	ρ = 0.15 .

Remark:
A black pixel is considered to be a junction if it has at least 3 black neighbors.

II. Example: image name: junction.bmp, image size: 140x120; template name: junction.tem .



input



output

JunctionExtractor1: Finding the intersection points of thin (one-pixel thick) lines from two binary images

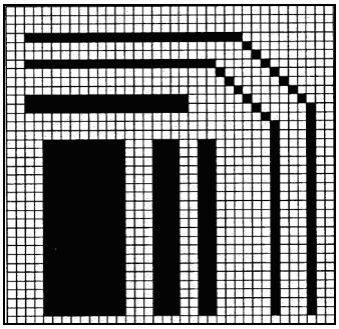
$$\mathbf{A} = \begin{bmatrix} -0.5 & -0.5 & -0.5 \\ -0.5 & 3 & -0.5 \\ -0.5 & -0.5 & -0.5 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} -0.5 & -0.5 & -0.5 \\ -0.5 & 3 & -0.5 \\ -0.5 & -0.5 & -0.5 \end{bmatrix} \quad z = \begin{bmatrix} -8.5 \end{bmatrix}$$

I. Global Task

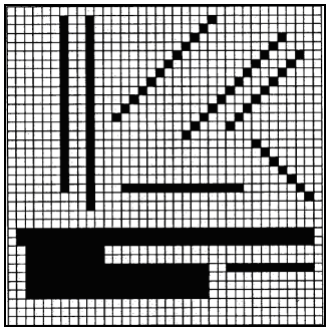
Given:	*two* static binary images **P1** and **P2** containing thin (one-pixel thick) lines or curves, among other (compact) objects
Input:	**U(t) = P1**
Initial State:	**X**(0) **= P2**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image containing all the intersection points between the thin lines contained in the binary images **P1** and **P2**

Remarks: *The two binary images can be interchanged, i.e. we can apply* **P2** *at the input and load* **P1** *as initial state. The feedback and control templates are identical. Even if other (compact) objects are present in the two images, their overlapping is not detected, except intersection points of thin lines.*

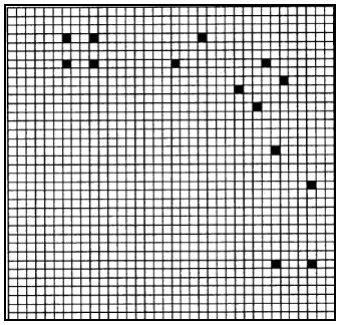
II. Example: image size: 36x36.



Initial state



Intermediate state



Output

LocalConcavePlaceDetector: ***Local concave place detector* [11]**

Old names: LCP

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 2 | 2 | 2 |
| 1 | -2 | 1 |

z = -7

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X**(0) **=** Arbitrary

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0

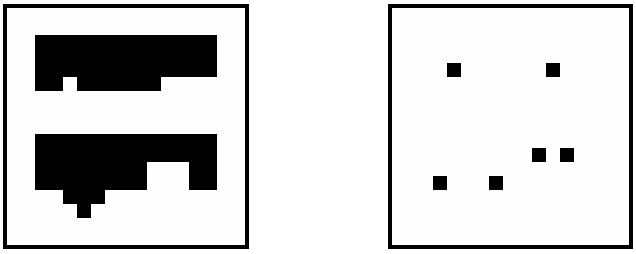
Output: **Y(t)⇒Y(∞)** = Binary image showing the local concave places of **P**.

Template robustness: ρ = 0.24 .

Remark:

Local concave places of the image are pixels having no southern neighbor, but both eastern, western and either a south-western or (permissive) a south-eastern one.

II. Example: image name: lcp_lse.bmp, image size: 17x17; template name: lcp.tem .



input output

LE7pixelVerticalLineRemover: ***Deletes vertical lines not longer than 7 pixels*** **[10]**

Old names: LINCUT7V, CUT7V

A =

0	0	1	0	0
0	0	0.5	0	0
0	0	2	0	0
0	0	0.5	0	0
0	0	1	0	0

B =

0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0

z = -5.5

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= P**

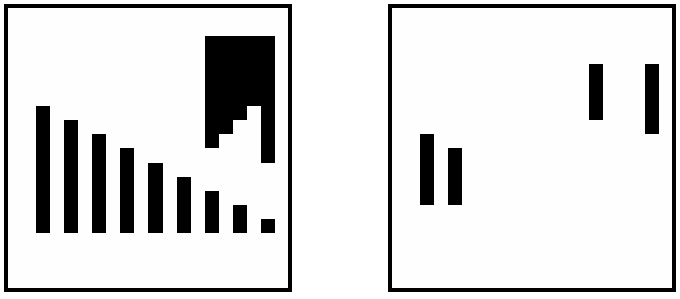
Boundary Conditions: Fixed type, u_{ij} = -1, y_{ij} = -1 for all virtual cells, denoted by [U]=[Y]= -1

Output: **Y(t)⇒Y(∞)** = Binary image where black pixels identify the vertical lines with a length of 8 or more pixels in **P**.

Remark:

The template deletes even those parts of the objects that could be interpreted as vertical lines not longer than 7 pixels.

II. Example: image name: lincut7v.bmp, image size: 20x20; template name: lincut7v.tem



input output

GrayscaleLineDetector: ***Grayscale line detector template***

Old names: LINE3060

A =

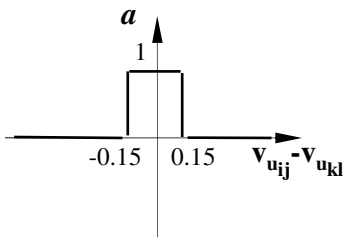
0	0	0
0	1.5	0
0	0	0

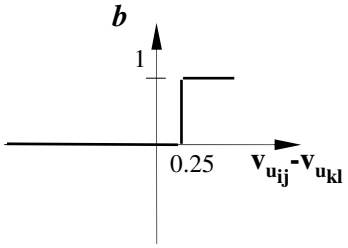
B =

b	***a***	***a***
b	***0***	***a***
a	***b***	***b***

z = -4.5

where ***a*** and ***b*** are defined by the following nonlinear functions:





I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X**(0) **= 0**

Boundary Conditions: Zero-flux boundary condition (duplicate)

Output: **Y(t)⇒Y(∞)** = Binary image where black pixels correspond to the grayscale lines within a slope range of approximately 30° (30°-60°) in **P**.

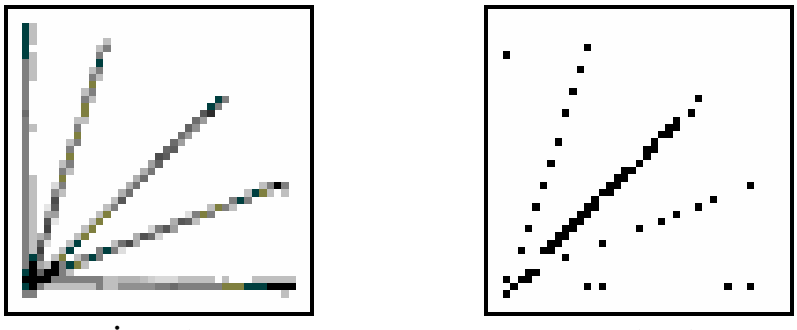
Remark:

It is supposed that the difference between values of a grayscale line and those of the background is not less than 0.25 (see function ***b***). Analogously, the difference between values representing a grayscale line is supposed to be in the interval [-0.15, 0.15] (see function ***a***). The template can easily be tuned for other input assumptions by changing functions ***a*** and ***b***.

The functionality of this template is similar to that of the rotated version of the *GrayscaleDiagonalLineDetector* template.

II. Examples

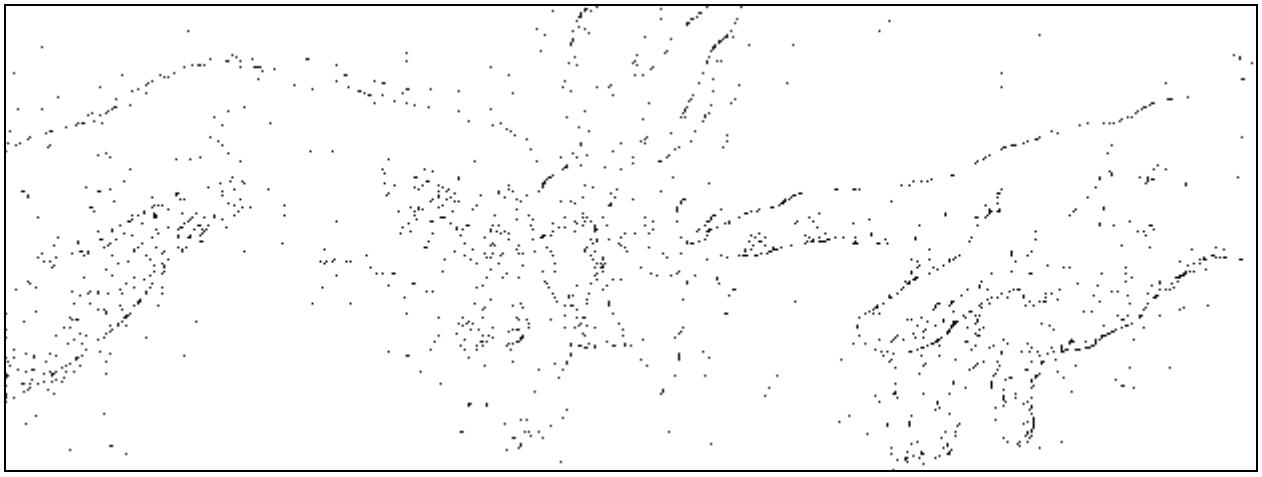
Example 1 (simple): image name: line3060.bmp, image size: 41x42; template name: line3060.tem .



input output

Example 2 (complex): image name: michelan.bmp, image size: 625x400; template name: line3060.tem .

input



output

LE3pixelLineDetector: ***Lines-not-longer-than-3-pixels-detector template*** **[13]**

Old names: LINEXTR3, LGTHTUNE

A =

0	0	0	0	0
0	0.3	0.3	0.3	0
0	0.3	3	0.3	0
0	0.3	0.3	0.3	0
0	0	0	0	0

B =

-1	0	-1	0	-1
0	-1	-1	-1	0
-1	-1	4	-1	-1
0	-1	-1	-1	0
-1	0	-1	0	-1

z = -2

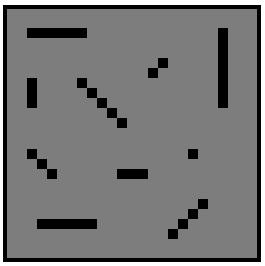
I. Global Task

Given:	static binary image **P** where the background is set to 0
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing only lines not longer than 3 pixels in **P**.

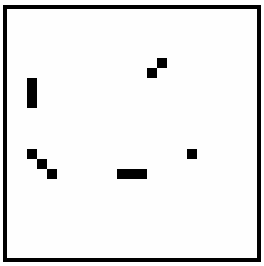
Remark:

The image **P** should contain only 1 pixel wide non-crossing lines. Otherwise the template may not work properly.

II. Example: image name: linextr3.bmp, image size: 25x25; template name: linextr3.tem .



input



output

PixelSearch: ***Pixel search in a given range*** **[72]**

A =

0	0	0
0	2	0
0	0	0

B=

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0

z = -1

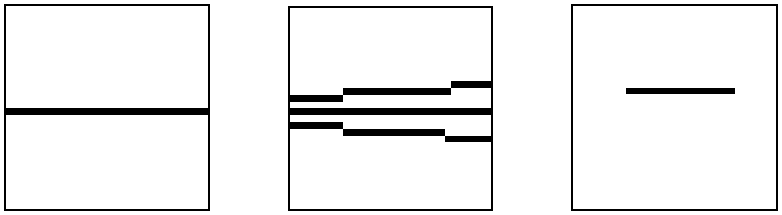
I. Global Task

Given:	static binary image **P, Q**
Input:	**U(t) = P** the reference image
Initial State:	**X**(0) **= Q** pixel whose distance is measured to the reference
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing the pixels being at the specified distance from the reference.

Remark:

This operation keeps those pixels of the initial state where there is a black pixel on the input at the place of the nonzero element of **B**. Based on the above example arbitrary pixel search operations can be implemented by extending the applied template to *NxN*. Although the operation requires *NxN* templates (7x7 in the example), members of this template class can be decomposed into a sequence of 3x3 linear templates (see [73]).

II. Example: image names: input_reference.bmp, initial_available.bmp; image size: 30x30; template name: pixelsearch.tem .



input initial state output

LogicANDOperation: ***Logic AND and Set Intersection ∩ (Conjunction ∧) template***

Old names: LogicAND, LOGAND, AND

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = | -1 |
| --- |

I. Global Task

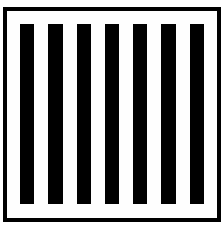
Given: two static binary images $\mathbf{P_1}$ and $\mathbf{P_2}$

Input: $\mathbf{U(t) = P_1}$

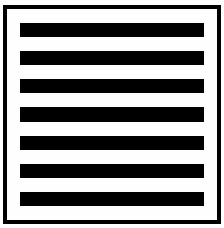
Initial State: $\mathbf{X}(0) = \mathbf{P_2}$

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = binary output of the logic operation "**AND**" between $\mathbf{P_1}$ and $\mathbf{P_2}$. In logic notation, $\mathbf{Y(\infty) = P_1 \wedge P_2}$, where $\wedge$ denotes the "conjunction" operator. In set-theoretic notation, $\mathbf{Y(\infty) = P_1 \cap P_2}$, where $\cap$ denotes the "intersection" operator.

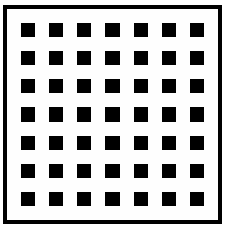
II. Example: image names: logic01.bmp, logic02.bmp; image size: 44x44; template name: logand.tem .



input



initial state



output

LogicDifference1: ***Logic Difference and Relative Set Complement ($P_2 \setminus P_1 = P_2 - P_1$) template*** **[7]**

Old names: LOGDIF, PA-PB

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{-1}$$

I. Global Task

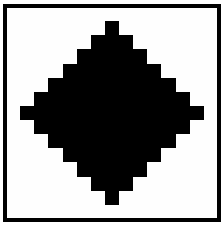
Given: two static binary images $\mathbf{P_1}$ and $\mathbf{P_2}$

Input: $\mathbf{U(t) = P_1}$

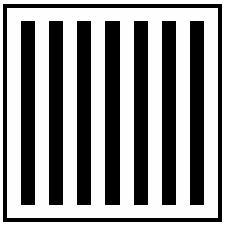
Initial State: $\mathbf{X}(0) = \mathbf{P_2}$

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Binary image representing the set-theoretic, or logic complement of $\mathbf{P_2}$ relative to $\mathbf{P_1}$. In set-theoretic or logic notation, $\mathbf{Y(\infty) = P_2 \setminus P_1}$, or $\mathbf{Y(\infty) = P_2 - P_1}$, i.e., $\mathbf{P_2}$ minus $\mathbf{P_1}$.

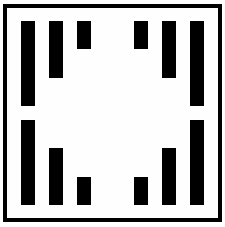
II. Example: image names: logic05.bmp, logic01.bmp; image size: 44x44; template name: logdif.tem .



input



initial state



output

LogicNOTOperation: ***Logic NOT and Set Complementation ($P \rightarrow \overline{P} = P^c$) template***

Old names: LogicNOT, LOGNOT, INV

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | -2 | 0 |
| 0 | 0 | 0 |

z = | 0 |
| --- |

I. Global Task

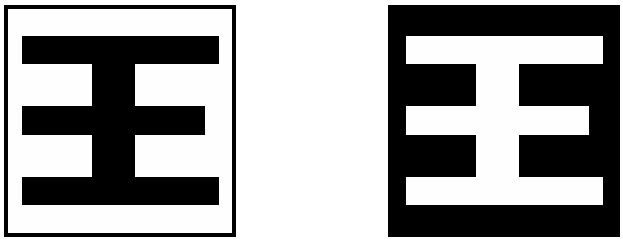
Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X(0) =** Arbitrary or as a default X(0)=0

Output: **Y(t)⇒Y(∞)** = Binary image where each black pixel in **P** becomes white, and vice versa. In set-theoretic or logic notation: $\mathbf{Y(\infty)=P^c=\overline{P}}$, where the bar denotes the "Complement" or "Negation" operator.

II. Example: image name: chineese.bmp; image size: 16x16; template name: lognot.tem .



input output

LogicOROperation: Logic OR and Set Union ∪ (Disjunction ∨) template

Old names: LogicOR, LOGOR, OR

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 2 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$z =$ | 1 |
|---|

I. Global Task

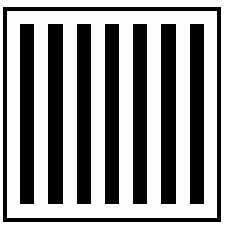
Given: two static binary images $\mathbf{P_1}$ and $\mathbf{P_2}$

Input: $\mathbf{U(t) = P_1}$

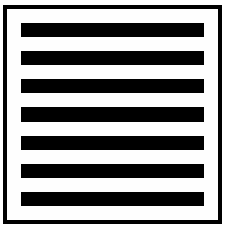
Initial State: $\mathbf{X}(0) = \mathbf{P_2}$

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = binary output of the logic operation **OR** between $\mathbf{P_1}$ and $\mathbf{P_2}$. In logic notation, $\mathbf{Y(\infty) = P_1 \vee P_2}$, where $\vee$ denotes the "disjunction" operator. In set-theoretic notation, $\mathbf{Y(\infty) = P_1 \cup P_2}$ where $\cup$ denotes the "set union" operator.

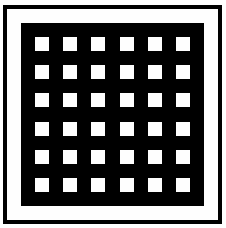
II. Example: image names: logic01.bmp, logic02.bmp; image size: 44x44; template name: logor.tem .



input



initial state



output

LogicORwithNOT: ***Logic "OR" function of the initial state and logic "NOT" function of the input*** **[24]**

Old names: LOGORN, INV-OR

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{1}$$

I. Global Task

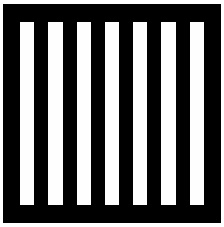
Given: two static binary images $\mathbf{P_1}$ and $\mathbf{P_2}$

Input: $\mathbf{U(t) = P_1}$

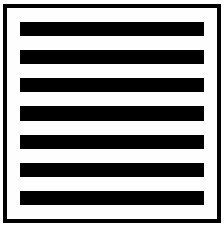
Initial State: $\mathbf{X}(0) = \mathbf{P_2}$

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = binary output of the logic operation **OR** between $\overline{P}_1$ and $\mathbf{P_2}$. In logic notation, $\mathbf{Y}(\infty) = \overline{P}_1 \vee \mathbf{P_2}$, where $\vee$ denotes the "disjunction" operator.

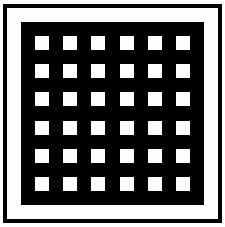
II. Example: image names: logic06.bmp, logic02.bmp; image size: 44x44; template name: logorn.tem .



input



initial state



output

PatchMaker: ***Patch maker template*** **[22]**

Old names: PATCHMAK

A = | 0 | 1 | 0 |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 1 | 0 |

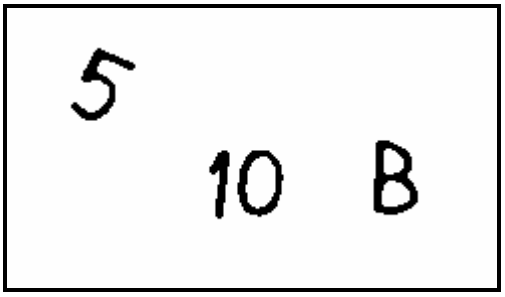
B = | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 4.5

I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	**Y(t)⇒Y(T)** = Binary image with enlarged objects of the input obtained after a certain time t = **T**. The size of the objects depends on time **T**. When **T** → ∞ all pixels will be driven to black.

II. Example: image name: patchmak.bmp; image size: 245x140; template name: patchmak.tem .



input


output

SmallObjectRemover: ***Deletes small objects*** **[22]**

Old names: SMKILLER

$$\mathbf{A} = \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 2 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \qquad \mathbf{B} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \qquad z = \boxed{0}$$

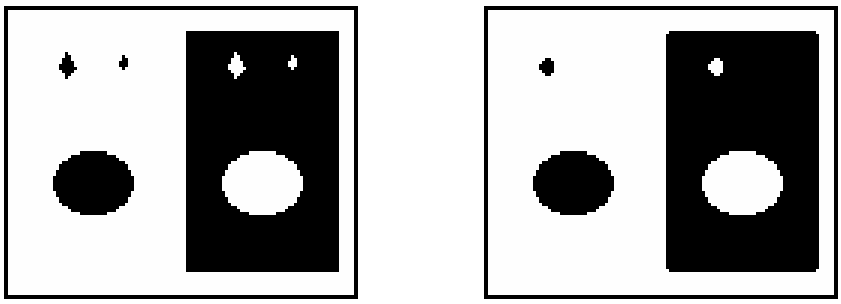
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image representing **P** without small objects.

Remark:

This template drives dynamically white all those black pixels that have more than four white neighbors, and drives black all white pixels having more than four black neighbors.

II. Example: image name: smkiller.bmp; image size: 115x95; template name: smkiller.tem .



input output

BipolarWave: ***Generates black and white waves*** **[52]**

$$\mathbf{A} = \begin{bmatrix} 0.3 & 0.3 & 0.3 \\ 0.3 & 0.8 & 0.3 \\ 0.3 & 0.3 & 0.3 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

I. Global Task

Given:	image **P** containing three gray levels: +1, 0, -1 (black, gray, white)
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	**Y(t)⇒Y(∞) =** Black and white areas, the boundary of which is located at positions where the waves collided.

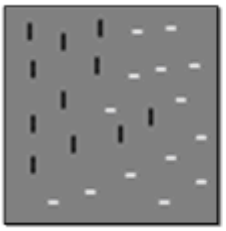
Remark:

The wave starts from black and white pixels and propagates on cells which have zero state (gray color). The final image will contain black and white areas.

II. Example: image size: 64x64; template name: bipolar.tem .

input

output



input

output

1.4. TEXTURE SEGMENTATION AND DETECTION

5x5TextureSegmentation1: Segmentation of four textures by a 5*5 template **[17]**

Old names: TX_HCLC

A =

-3.44	0.86	-1.64	-0.16	-1.02
-1.09	0.16	-2.19	-3.2	3.51
2.50	1.56	3.91	2.66	2.42
0.55	2.89	-0.62	0.47	3.67
-1.80	-0.55	2.50	-0.23	2.34

B =

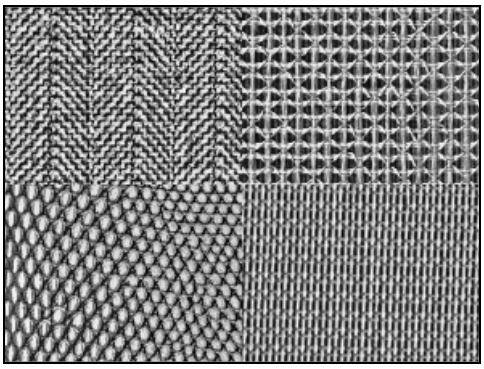
-2.19	-0.23	0.16	-0.63	-0.78
1.64	2.27	-3.2	1.09	2.03
0.08	0.55	0.86	3.52	0.08
0.39	-3.83	-3.12	-2.34	-2.11
0.78	-2.66	-1.17	-1.41	1.02

z = 3.28

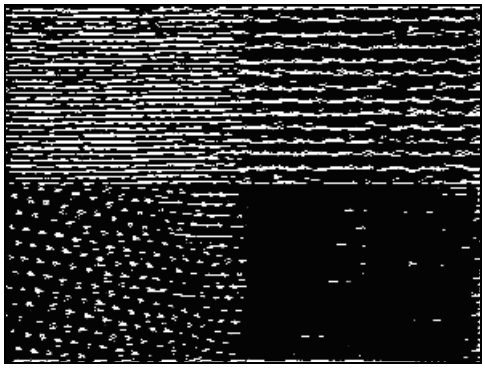
I. Global Task

Given:	static grayscale image **P** representing four textures (herringbone clothes, cloth, lizard skin, cloth) having the same flat grayscale histograms with the same gray average value
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0
Output:	**Y(t)⇒Y(T)** = Nearly binary image representing four patterns that differ in average gray-levels.

II. Example: image name: tx_hclc.bmp, image size: 296x222; template name: tx_hclc.tem .



input



output

3x3TextureSegmentation: ***Segmentation of four textures by a 3*3 template*** **[17]**

Old names: TX_RACC3

A = | 0.86 | 0.94 | 3.75 |
| 2.11 | -2.81 | 3.75 |
| -1.33 | -2.58 | -1.02 |

B = | 0.16 | -1.56 | 1.25 |
| -2.89 | 1.09 | -3.2 |
| 4.06 | 4.69 | 3.75 |

z = 1.8

I. Global Task

Given: static grayscale image **P** representing four textures (raffia, aluminum mesh, 2 clothes) having the same flat grayscale histograms

Input: **U(t) = P**

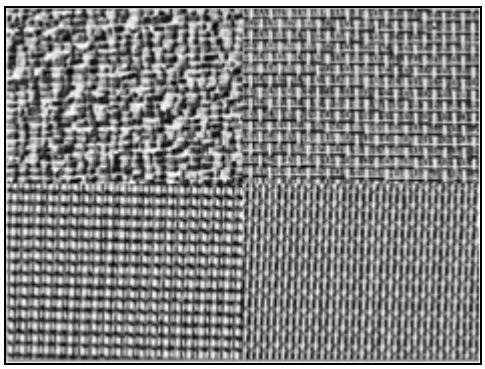
Initial State: **X**(0) **= P**

Boundary Conditions: Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0

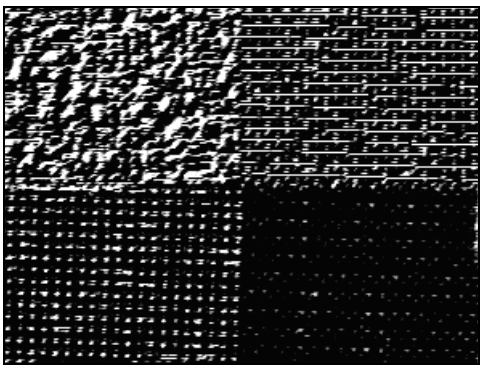
Output: **Y(t)⇒Y(T)** = Nearly binary image representing four patterns that differ in average gray-levels.

Remark:
This template is called "Texture Discrimination" in [44].

II. Example: image name: tx_racc.bmp, image size: 296x222; template name: tx_racc3.tem .



input



output

5x5TextureSegmentation2: Segmentation of four textures by a 5*5 template **[17]**

Old names: TX_RACC5

A =

4.21	-1.56	1.56	3.36	0.62
-2.89	4.53	-0.23	3.12	-2.89
2.65	2.18	-4.68	-3.43	-2.81
3.98	1.56	-1.17	-3.12	-3.20
-3.75	-2.18	3.28	2.19	-0.62

B =

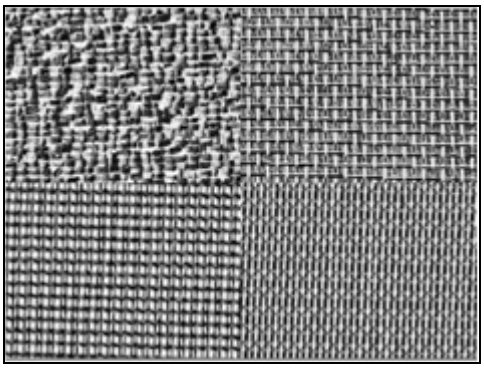
4.06	-5	0.39	2.11	-1.87
3.90	0.31	-1.95	4.84	-0.31
0	-4.06	0.93	-0.31	0.46
-0.62	-5	2.34	0.62	-1.87
3.59	-0.93	0.15	2.81	-1.87

z = -5

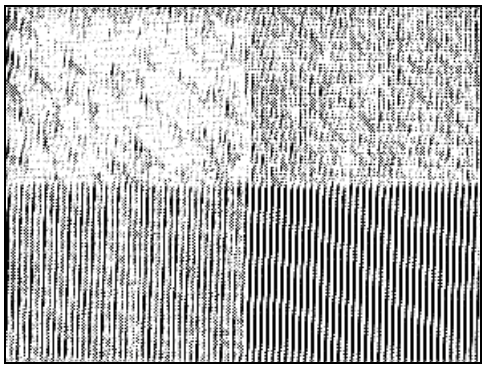
I. Global Task

Given:	static grayscale image **P** representing four textures (raffia, aluminum mesh, 2 clothes) having the same flat grayscale histograms
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0
Output:	**Y(t)⇒Y(T)** = Nearly binary image representing four patterns that differ in average gray-levels.

II. Example: image name: tx_racc.bmp, image size: 296x222; template name: tx_racc5.tem .



input



output

TEXTURE DETECTION

TextureDetector1 (T1_RACC3)

A = | 2.27 | 1.80 | 3.36 |
| -0.70 | -4.45 | 1.41 |
| 3.20 | 3.98 | -0.31 |

B = | -3.91 | 1.25 | 3.05 |
| 0.86 | -3.05 | 3.36 |
| 1.72 | -0.63 | -4.61 |

z = -1.64

TextureDetector2 (T2_RACC3)

A = | 1.56 | 4.38 | 2.42 |
| 4.69 | -3.13 | 1.41 |
| 2.19 | -5 | 0.86 |

B = | -2.81 | 2.42 | -3.75 |
| -5 | -0.39 | -5 |
| 3.67 | 4.22 | 3.13 |

z = -3.20

TextureDetector3 (T3_RACC3)

A = | 1.64 | -1.02 | 1.33 |
| 1.88 | -4.61 | 2.89 |
| 3.28 | 2.03 | 3.75 |

B = | -3.91 | -2.66 | -3.13 |
| 0.94 | 1.48 | -3.13 |
| 1.33 | 0.55 | 2.34 |

z = -2.42

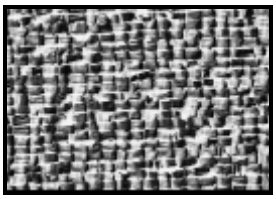
TextureDetector4 (T4_RACC3)

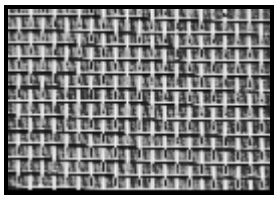
A = | 3.13 | 4.30 | 2.19 |
| -2.81 | 3.13 | 0.16 |
| 1.88 | 4.92 | 4.53 |

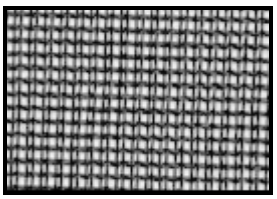
B = | -3.52 | 4.38 | -5 |
| -0.94 | -3.05 | -3.67 |
| 1.41 | -0.63 | -4.38 |

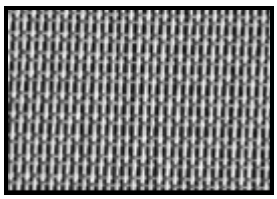
z = -2.42

Textures being detected by these templates


TextureDetector1


TextureDetector2


TextureDetector3


TextureDetector4

I. Global Task

Given: static grayscale image **P** representing textures having the same flat grayscale histograms. One of them is identical to a texture shown above.

Input: **U(t) = P**

Initial State: **X**(0) **= P**

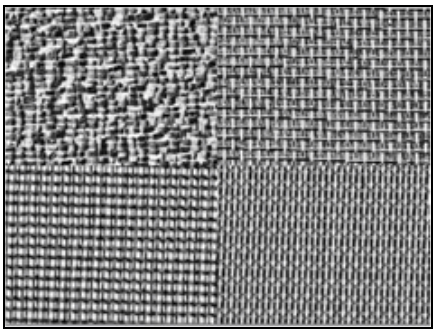
Boundary Conditions: Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0

Output: **Y(t)⇒Y(T)** = Nearly binary image where the detected texture becomes darker than the others.

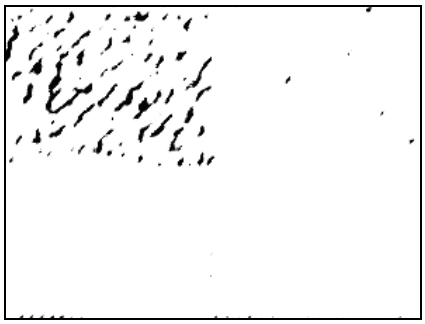
These templates can be used in a classification problem when the number of different examined textures is, for instance, more than 10 and the input textures have the same flat grayscale histograms.

II. Examples image name: tx_racc.bmp; image size: 296x222.

Example 1: Texture detection with the *TextureDetector1* (t1_racc3.tem) template.

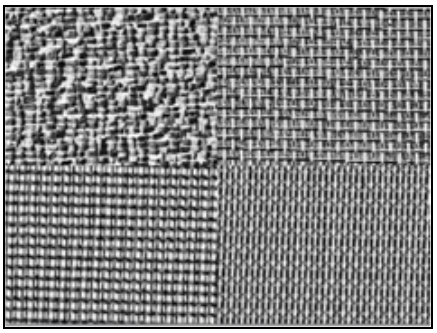


input

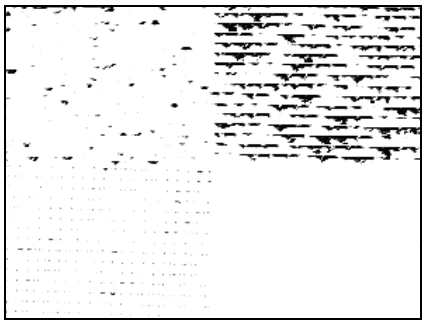


output

Example 2: Texture detection with the *TextureDetector2* (t2_racc3.tem) template.

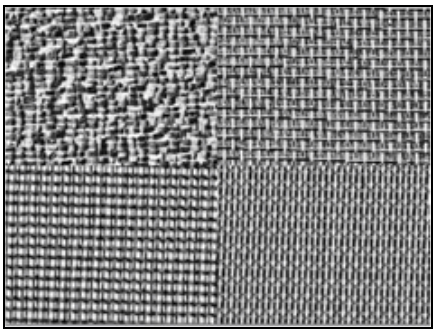


input

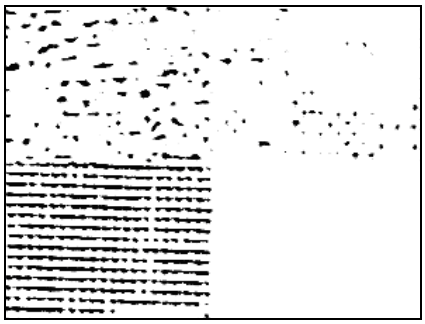


output

Example 3: Texture detection with the *TextureDetector3* (t3_racc3.tem) template.

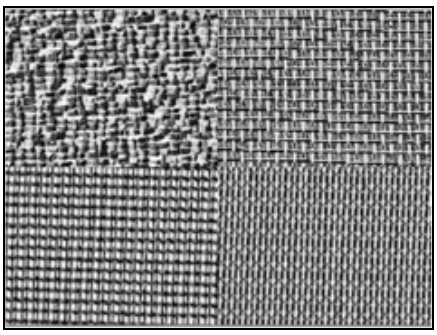


input

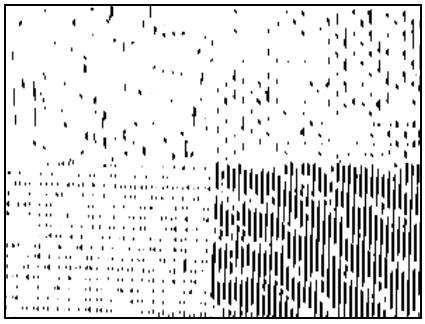


output

Example 4: Texture detection with the *TextureDetector4* (t4_racc3.tem) template.



input



output

1.5. MOTION

ImageDifferenceComputation: ***Logic difference between the initial state and the input pictures with noise filtering*** **[7]**

Old names: LogicDifference2, LOGDIFNF, PA-PB_F1

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0.25 & 0.25 & 0.25 \\ 0.25 & 2 & 0.25 \\ 0.25 & 0.25 & 0.25 \end{bmatrix} \qquad z = \boxed{-4.75}$$

$$\mathbf{A}^{\tau} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B}^{\tau} = \begin{bmatrix} -0.25 & -0.25 & -0.25 \\ -0.25 & -2 & -0.25 \\ -0.25 & -0.25 & -0.25 \end{bmatrix} \qquad \tau = 10\ \tau_{CNN}$$

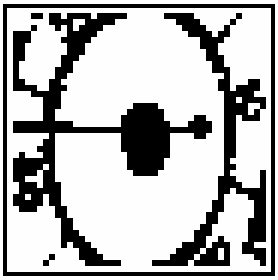
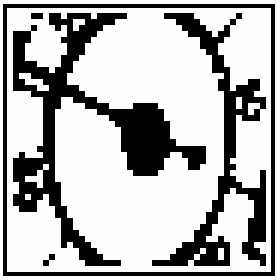
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(T)** = Binary image where black pixels identify the moving parts of **P**.

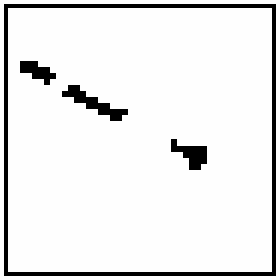
Remark:

Takes the logic difference of the delayed and actual input in continuous mode together with noise filtering. Moving parts of an image can be extracted.

II. Example: image names: logdnfi0.bmp, logdnfi1.bmp; image size: 44x44; template name: logdifnf.tem .

two consecutive steps of the input stream



output

MotionDetection: ***Direction and speed dependent motion detection*** **[12]**

Old names: MotionDetection1, MOTDEPEN, MOVEHOR

$\mathbf{A} =$ | -0.1 | -0.1 | -0.1 |
|---|---|---|
| -0.1 | 0 | -0.1 |
| -0.1 | -0.1 | -0.1 |

$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1.5 | 0 |
| 0 | 0 | 0 |

$z =$ -2

$\mathbf{A}^{\tau} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

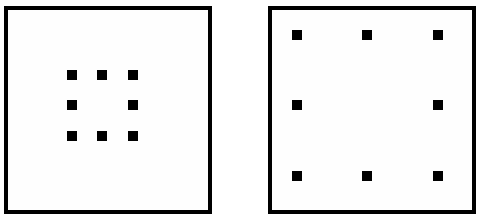
$\mathbf{B}^{\tau} =$ | 0 | 0 | 0 |
|---|---|---|
| 1.5 | 0 | 0 |
| 0 | 0 | 0 |

$\tau = 10\ \tau_{CNN}$

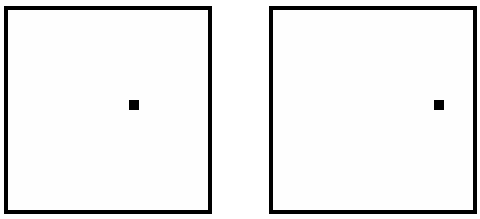
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$, $y_{ij} = 0$ for all virtual cells, denoted by [U]=[Y]=0
Output:	**Y(t)⇒Y(T)** = Binary image representing only objects of **P** moving horizontally to the right with a speed of 1 pixel/delay time.

II. Example: image names: motdep1.bmp, motdep2.bmp; image size: 20x20; template name: motdepen.tem .



two consecutive steps of the input stream



corresponding outputs

***SpeedDetection: Direction independent motion detection* [7]**

Old names: MotionDetection2, MOTINDEP, MD_CONT

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 6 | 0 |
| 0 | 0 | 0 |

$z =$ -2

$\mathbf{A}^{\tau} =$ | 0.68 | 0.68 | 0.68 |
|---|---|---|
| 0.68 | 0.68 | 0.68 |
| 0.68 | 0.68 | 0.68 |

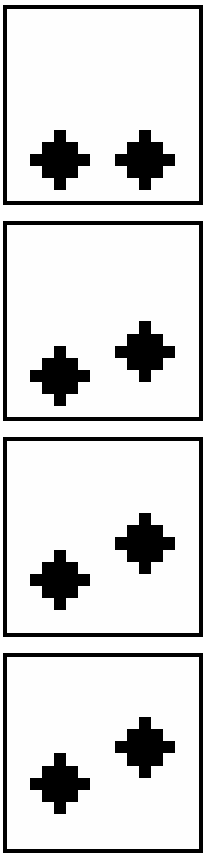
$\mathbf{B}^{\tau} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

$\tau = 10\ \tau_{CNN}$

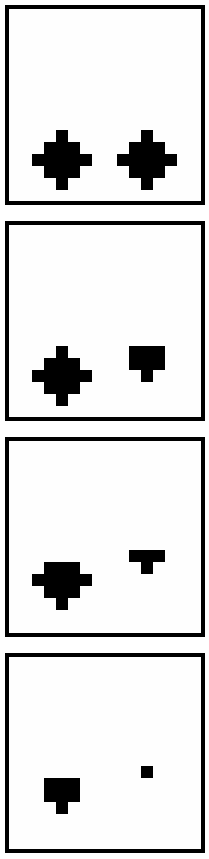
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(T)** = Binary image representing only objects of **P** moving slower than 1 pixel/delay time in arbitrary directions.

II. Example: image names: motind1.bmp, motind2.bmp, motind3.bmp, motind4.bmp; image size: 16x16; template name: motindep.tem .



4 consecutive steps of the input stream

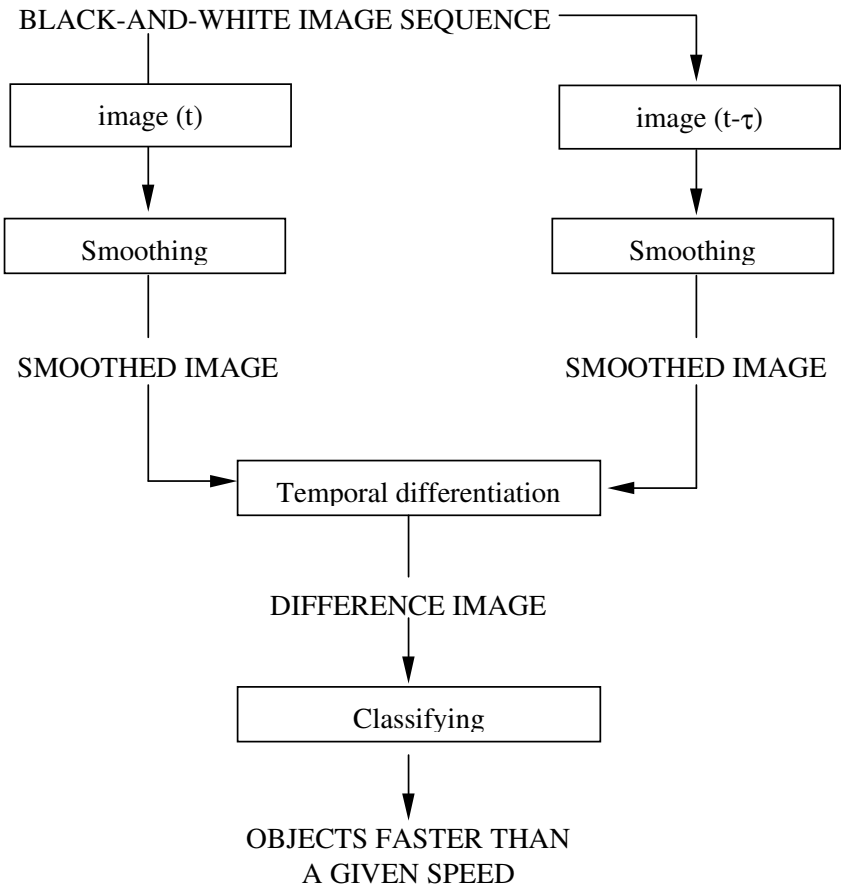


corresponding outputs

SPEED CLASSIFICATION

The algorithm is capable of classifying the speed of black-and-white objects moving parallel to the image plane. It extracts objects moving faster than a given speed determined by the current of the threshold template. Grayscale image sequences can be converted to black-and-white using the *Smoothing* template.

The flow-chart of the algorithm:



Smoothing:

$\mathbf{B}_{smoothing}$ =

0.06	0.13	0.06
0.13	0.24	0.13
0.06	0.13	0.06

Temporal differentiation:

$\mathbf{A}_{diff}$ =

0	0	0
0	1	0
0	0	0

$\mathbf{B}_{diff}$ =

0	0	0
0	0.4	0
0	0	0

[ ] =

0	0	0
0	-0.4	0
0	0	0

Classifying speed:

$\mathbf{A}_{class}$ =

0	0	0
0	2	0
0	0	0

$\mathbf{B}_{class}$ =

0	0	0
0	0	0
0	0	0

z_{class} = *-threshold*

Recalling objects:

$\mathbf{A}_{recall}$ =

0.3	0.3	0.3
0.3	4	0.3
0.3	0.3	0.3

$\mathbf{B}_{recall}$ =

0	0	0
0	5.1	0
0	0	0

Example: Input and output pictures. Image names: speed1.bmp, speed2.bmp; image size: 163x105.

PathTracing: ***Traces the path of moving objects on black-and-white images***

Old names: TRACE

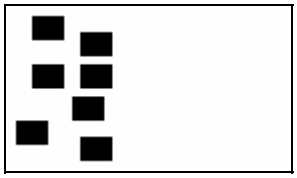
$\mathbf{A}_{11} = \begin{bmatrix} 0.3 & 0.3 & 0.3 \\ 0.3 & 4 & 0.3 \\ 0.3 & 0.3 & 0.3 \end{bmatrix}$ $\mathbf{B}_{11} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 5.1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_1 = \boxed{0}$

$\mathbf{A}_{22} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{A}_{21} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_2 = \boxed{2}$

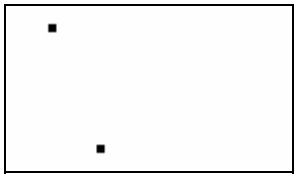
I. Global Task

Given:	a binary image sequence $\mathbf{P_1}$ and a binary image $\mathbf{P_2}$. $\mathbf{P_1}$ represents the objects to be traced, $\mathbf{P_2}$ consists of black pixels marking the objects to be traced.
Input:	$\mathbf{U_1(t) = P_1}$
Initial State:	$\mathbf{X_1}(0) = \mathbf{P_2}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y_1(t) \Rightarrow Y(T)}$ = **B**inary image representing the actual position of the marked objects $\mathbf{Y_2(t) \Rightarrow Y(\infty)}$ = **B**inary image showing the whole path of the marked objects

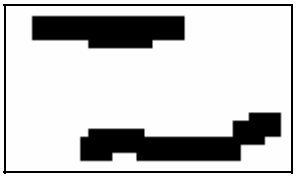
II. Example: image names: trace1.bmp, trace2.bmp; image size: 140x80; template name: trace.tem



starting snapshot of the input sequence



initial state

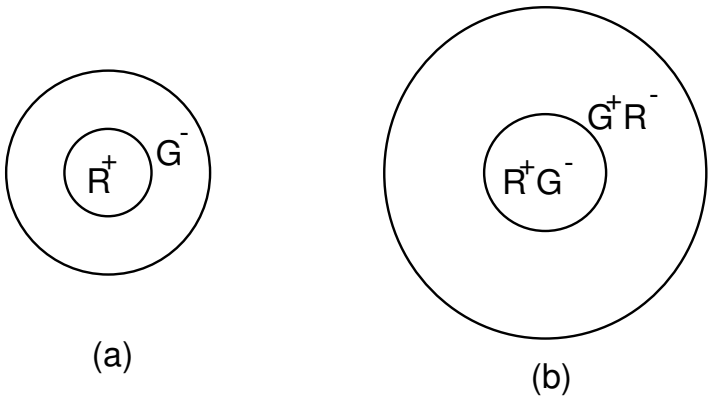


output of the 2. layer

1.6. COLOR

CNN MODELS OF SOME COLOR VISION PHENOMENA: SINGLE AND DOUBLE OPPONENCIES

In the retina, and the visual cortex, there are single and double color opponent cells [23]. Their receptive fields are as follows:



where (a) belongs to the single and (b) belongs to the double opponent cell. The template simulating the single opponent cell has two layers. The input of the first layer is the monochromatic red map, while the second layer gets the green map. The result appears on the second layer. The template is the following:

$\mathbf{B}_{12} =$

0	0	0
0	2	0
0	0	0

$\mathbf{B}_{22} =$

-0.25	-0.25	-0.25
-0.25	0	-0.25
-0.25	-0.25	-0.25

By swapping the layers we get the template generating the G^+R^- single opponents. The output of the R^+G^- and G^+R^- layers provide the input for the first and second layer of the double opponent structure, respectively. The output appears on the second layer. The template is as follows:

$\mathbf{B}_{12} =$

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	2	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

$\mathbf{B}_{22} =$

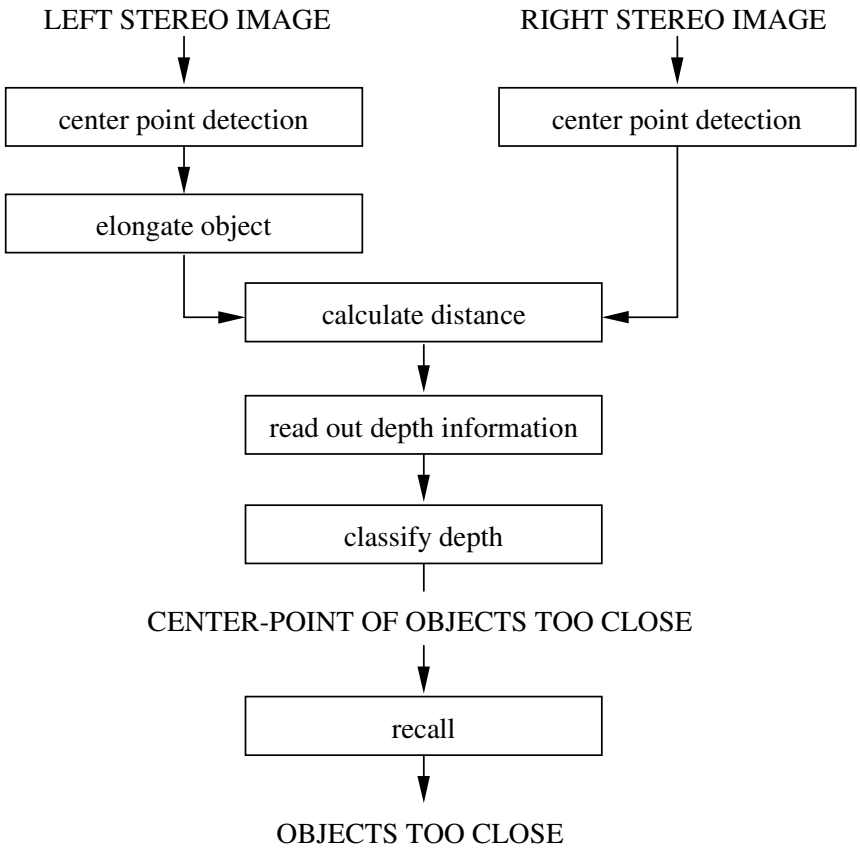
0.02	0.02	0.02	0.02	0.02	0.02	0.02
0.02	0	0	0	0	0	0.02
0.02	0	0	0	0	0	0.02
0.02	0	0	0	0	0	0.02
0.02	0	0	0	0	0	0.02
0.02	0	0	0	0	0	0.02
0.02	0.02	0.02	0.02	0.02	0.02	0.02

1.7. DEPTH

DEPTH CLASSIFICATION

The algorithm determines the depth of black-and-white objects based on a pair of stereo images. It determines whether an object is closer than a given distance or not. The first step of the algorithm is to reduce the objects in both input images to a single pixel; then the distance between corresponding points is calculated. The distance can be thresholded to determine whether the object is too close or not. As a first preprocessing step, grey-scale images can be converted into black-and-white using the *Smoothing* template.

The flow-chart of the algorithm:



Templates:

Elongate objects: add pixels to the top and bottom of each object (use the left image as input)

$\mathbf{A}_{\text{elongate}} =$

0	0	0
0	1	0
0	0	0

$\mathbf{B}_{\text{elongate}} =$

0	3	0
0	3	0
0	3	0

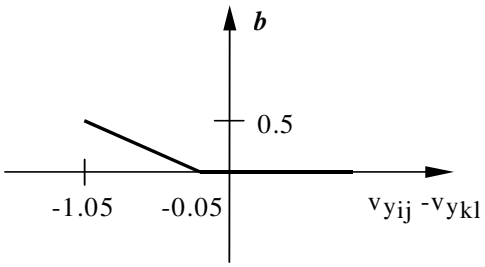
$z_{\text{elongate}} =$ 4.5

Calculate depth: (use the elongated left image as initial state)

$\mathbf{A}_{\text{distance}} =$

0	0	0
b	1	***b***
0	0	0

where ***b*** is defined by the following nonlinear function:



Read out depth: (use the right center points as a fixed state map)

$\mathbf{A}_{\text{read}} =$

0	0	0
0	1	0
0	0	0

$z_{\text{read}} =$ -2

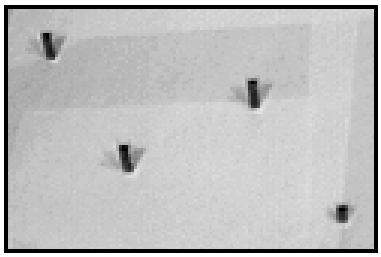
Classify depth:

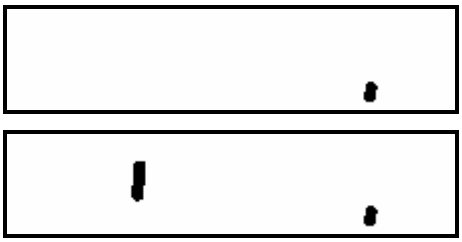
$\mathbf{A}_{\text{class}} =$

0	0	0
0	2	0
0	0	0

$z_{\text{class}} =$ *-threshold*

Example: image name: depth0.bmp; image size: 120x80.





1.8. OPTIMIZATION

GlobalMaximumFinder: ***Finds the global maximum*** **[33]**

Old names: GLOBMAX

A =

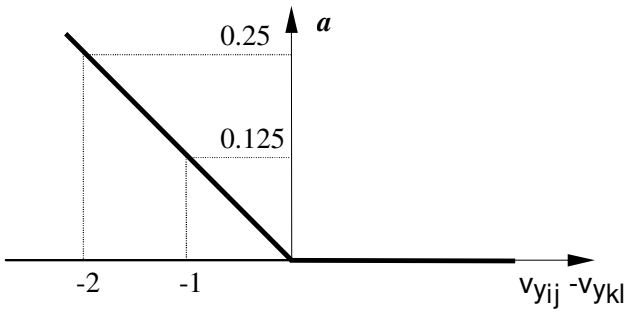
a	***a***	***a***
a	**1**	***a***
a	***a***	***a***

B =

0	0	0
0	0	0
0	0	0

$z =$ 0

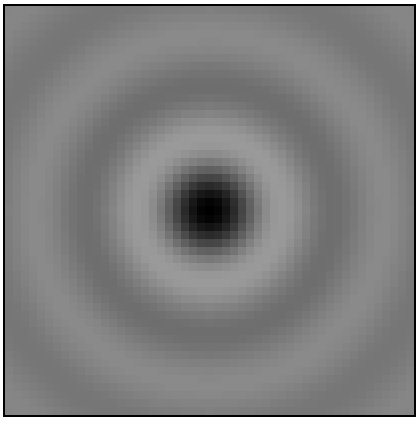
where ***a*** is defined by the following nonlinear function:



I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) =** Arbitrary
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	**Y(t)⇒Y(∞)** = An image where intensities of the pixels are identical and equal to the maximum intensity of pixels in **P**. In other words, the output is filled up with the global maximum of **P**.

II. Example: image name: globmax.bmp, image size: 51x51; template name: globmax.tem .



input

output

1.9. GAME OF LIFE AND COMBINATORICS

HistogramGeneration: ***Generates the one-dimensional histogram of a black-and-white image* [20]**

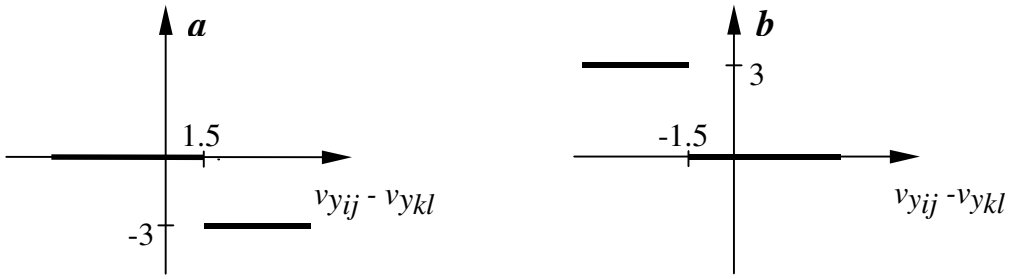
Old names: HistogramComputation, HISTOGR

A = | 0 | 0 | 0 |
| --- | --- | --- |
| ***a*** | 1 | ***b*** |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |
| --- |

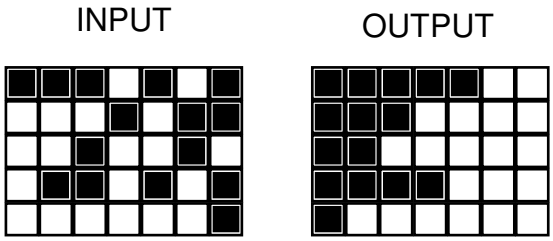
where ***a*** and ***b*** are defined by the following nonlinear functions:



I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image where all black pixels are shifted to the left.

II. Example: image name: histogr.bmp, image size: 7x5; template name: histogr.tem .



GameofLife1Step: ***Simulates one step of the game of life*** **[11]**

Old names: LIFE_1

$\mathbf{A}_{11} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_{11} =$ | -1 | -1 | -1 |
|---|---|---|
| -1 | 0 | -1 |
| -1 | -1 | -1 |

$z =$ -1

$\mathbf{A}_{22} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

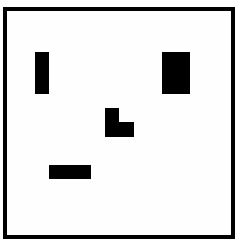
$\mathbf{B}_{21} =$ | -1 | -1 | -1 |
|---|---|---|
| -1 | -1 | -1 |
| -1 | -1 | -1 |

$z =$ -4

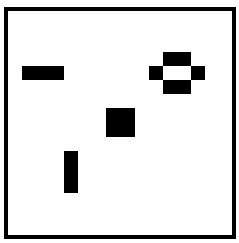
I. Global Task

Given:	static binary image **P**
Inputs:	$\mathbf{U_1(t) = P}$, $\mathbf{U_2(t)}$ = Arbitrary
Initial States:	$\mathbf{X_1}(0) = \mathbf{X_2}(0)$ = Arbitrary
Boundary Conditions:	Fixed type, u_{ij} = -1 for all virtual cells, denoted by [U]= -1
Outputs:	$\mathbf{Y_1(t)}$, $\mathbf{Y_2(t)} \Rightarrow \mathbf{Y_1(\infty)}, \mathbf{Y_2(\infty)}$ = Binary images representing partial results. The desired output is $\mathbf{Y_1(\infty)}$ **XOR** $\mathbf{Y_2(\infty)}$. For the simulation of the following steps of game of life this image should be fed to the input of the first layer.

II. Example: image name: life_1.bmp, image size: 16x16; template name: life_1.tem .



input



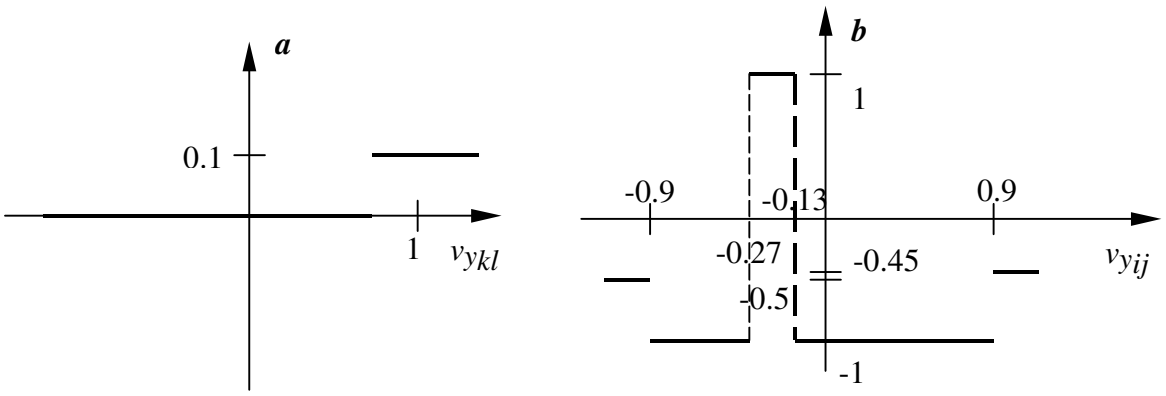
output

GameofLifeDTCNN1: ***Simulates the game of life on a single-layer DTCNN with piecewise-linear thresholding*** **[11]**

Old names: LIFE_1L

$$\mathbf{A} = \begin{bmatrix} a & a & a \\ a & b & a \\ a & a & a \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{0}$$

where ***a*** and ***b*** are defined by the following nonlinear functions:



I. Global Task

Given:	static binary image **P**
Input:	**U(t) =** Arbitrary
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Outputs:	$\mathbf{Y(t)} \Rightarrow \mathbf{Y(2*t_1)}$, $\mathbf{t_1}$= 1,2,… Binary images representing the game of life. The new state appears after every second iteration.

Remark:

The DTCNN with piecewise-linear thresholding is a CNN approximated by the forward Euler integration form using time step 1.

II. Example

See the example of the *GameofLife1Step* template (template name: life_1l.tem).

GameofLifeDTCNN2: ***Simulates the game of life on a 3-layer DTCNN*** **[11]**

Old names: LIFE_DT

$\mathbf{A}_{31} =$

0	0	0
0	1	0
0	0	0

$\mathbf{A}_{13} =$

0.3	0.3	0.3
0.3	0.3	0.3
0.3	0.3	0.3

$z_1 =$ | 1 |

$\mathbf{A}_{32} =$

0	0	0
0	1	0
0	0	0

$\mathbf{A}_{23} =$

-0.6	-0.6	-0.6
-0.6	0	-0.6
-0.6	-0.6	-0.6

$z_2 =$ | -0.8 |

$z_3 =$ | -1.5 |

I. Global Task

Given:	static binary image **P**
Inputs:	$\mathbf{U_1(t) = U_2(t) = U_3(t) =}$ Arbitrary
Initial States:	$\mathbf{X_1}(0) = \mathbf{X_2}(0) = \mathbf{X_3}(0) = \mathbf{P}$
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Outputs:	$\mathbf{Y(t) \Rightarrow Y_3(3,5,7,9,\ldots)}$ = Binary images representing the game of life.

II. Example

See the example of the *GameofLife1Step* template (template name: life_dt.tem).

***MajorityVoteTaker: Majority vote-taker* [20]**

Old names: MAJVOT

The goal of the one dimensional majority vote-taker template is to decide whether a row of an input image contains more black or white pixels, or their number is equal. The effect is realized in two phases. The first template (setting the initial state to 0) gives rise to an image, where the sign of the rightmost pixel corresponds to the dominant color. Namely, it is positive, if there are more black pixels than white ones; it is negative in the opposite case, and is 0 in the case of equality. By using the second template this information can be extracted, which drives the whole network into black or white, depending on the dominant color, or leaves the rightmost pixel unchanged otherwise. The method can easily be extended to two or even three dimensions.

First template

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 0.05 | 0 |
| 0 | 0 | 0 |

$z =$ | 0 |
|---|

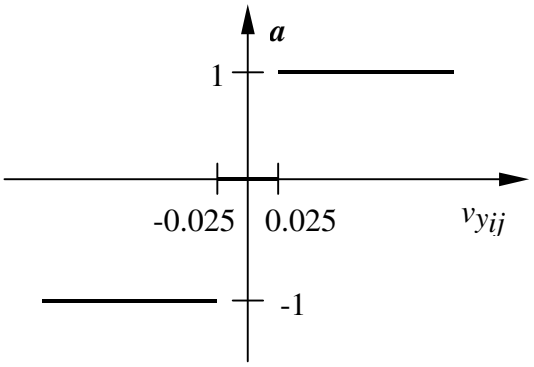
Second template

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | ***a*** | 2 |
| 0 | 0 | 0 |

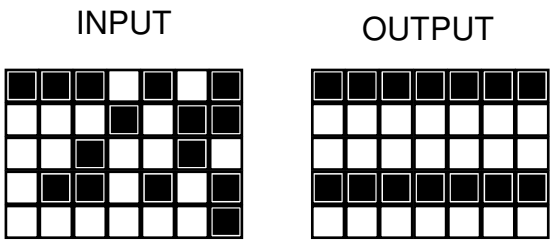
$\mathbf{B} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

$z =$ | 0 |
|---|

where ***a*** is defined by the following nonlinear function:



Example: image name: histogr.bmp, image size: 7x5; template names: majvot1.tem, majvot2.tem.



ParityCounting1: ***Determines the parity of a row of the input image*** **[20]**

Old names: PARITY

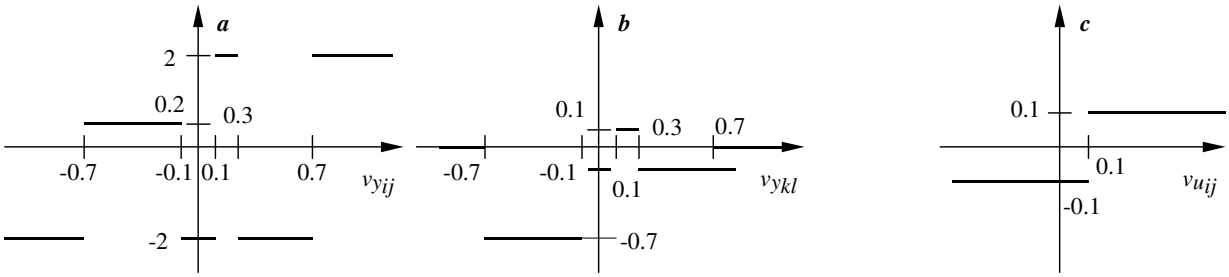
The template determines whether the number of black pixels in a row is even or odd. As a result, the leftmost pixel in the output image corresponds to the parity of the row, namely, black represents odd, while white means even parity. It is also true that each pixel codes the parity of the pixels right to it, together with the pixel itself. Naturally, the parity of a column or a diagonal can be counted in the same manner. The parity of an array can also be determined if columnwise parity is counted on the result of the rowwise parity operation. The initial state should be set to -0.5.

A = | 0 | 0 | 0 |
| 0 | ***a*** | ***b*** |
| 0 | 0 | 0 |

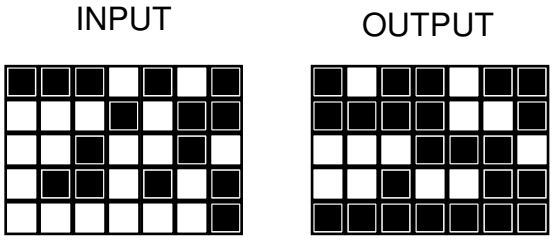
B = | 0 | 0 | 0 |
| 0 | ***c*** | 0 |
| 0 | 0 | 0 |

z = | 0 |

where ***a***, ***b***, and ***c*** are defined by the following nonlinear functions:



Example: image name: histogr.bmp, image size: 7x5; template name: parity1.tem .



ParityCounting2: ***Computes the parity of rows in a black-and-white image***

A = | 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

D = | 0 | 0 | 0 |
| ***d*** | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |

where $\boldsymbol{d} = -v_{u_{ij}} v_{y_{kl}}$.

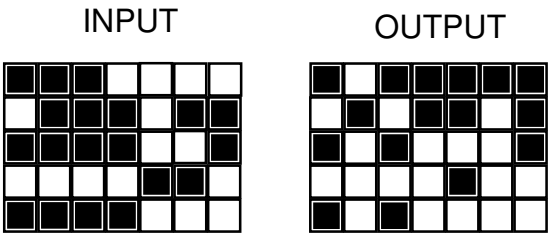
I. Global Task

Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = -1$ for all virtual cells, denoted by [Y]=-1
Output:	**Y(t)⇒Y(∞)** = Binary image representing the parity of rows in P

Remark:

A particular pixel in the output is black if an odd number of black pixels can be found at the left of the particular pixel in the input (including the position of the pixel itself).

II. Example: image name: parity2.bmp, image size: 7x5; template name: parity2.tem .



1-DArraySorting: ***Sorts a one dimensional array*** **[20]**

Old names: SORTING

A one-dimensional array of n values in the [-1,+1] interval can be sorted in descending order in n steps with the following time- and space-varying template. In each odd step, the templates should be applied in the $\mathbf{A}_R\mathbf{A}_L\mathbf{A}_R\mathbf{A}_L\mathbf{A}_R\mathbf{A}_L$ ... pattern, while in each even step in the $\mathbf{A}_L\mathbf{A}_R\mathbf{A}_L\mathbf{A}_R\mathbf{A}_L\mathbf{A}_R$ ... pattern. To suppress side effects, the left and right boundaries should be set to +1 and -1, respectively.

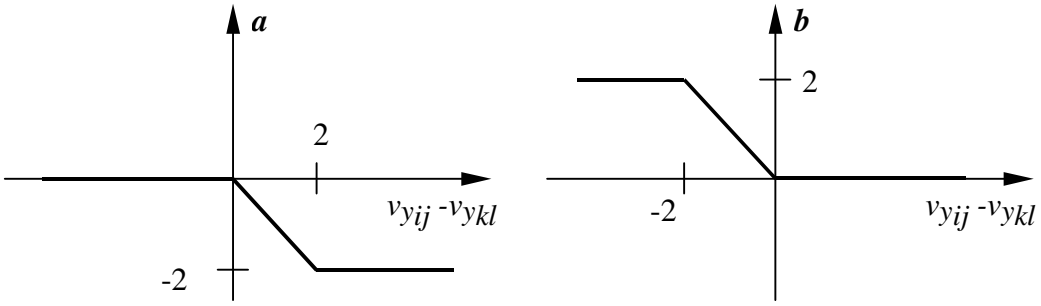
$\mathbf{A}_L =$

0	0	0
a	1	0
0	0	0

$\mathbf{A}_R =$

0	0	0
0	1	***b***
0	0	0

where ***a*** and ***b*** are defined by the following nonlinear functions:



Example: image name: sorting.bmp; image size: 12x1.

Input

Output

1.10. PATTERN FORMATION

SPATIO-TEMPORAL PATTERN FORMATION IN TWO-LAYER OSCILLATORY CNN

Spatio-temporal pattern formation in two-layer oscillatory CNN is studied in [56] and showed e.g. that some exotic types of spiral waves exist on this type of network. As an example, spiral waves might consist of not only two types of motif (black and white patches) but also, for instance, checkerboard patterns. These three types of motifs propagate like spiral waves and transform continuously into each other.

I. Global Task

Task: Generate oscillatory Turing patterns

Given:

Initial States:	$\mathbf{X_1}(0)$**,** $\mathbf{X_2}(0)$ **=** small signal random pattern, otherwise arbitrary
Boundary Conditions:	Zero-flux boundary condition
Outputs:	$\mathbf{Y_1(t)}$**,** $\mathbf{Y_2(t)}$ spatio-temporal oscillatory Turing patterns.

II. CNN architecture

Consider the dynamics of a two-layer autonomous CNN:

$$\left.\begin{aligned}\frac{dx_{1,ij}}{dt} &= -x_{1,ij} + \mathbf{A}_{1,1} * \mathbf{y}_1 + \beta_1 y_{2,ij} \\ \frac{dx_{2,ij}}{dt} &= -x_{2,ij} + \mathbf{A}_{2,2} * \mathbf{y}_2 + \beta_2 y_{1,ij}\end{aligned}\right\},$$ where β_1 and β_2 are the coupling parameters between the two layers.

III. Templates

Turing pattern generating templates extended with coupling parameter (β_1 and β_2). As an example, template A_{11} generates cow patches while template A_{22} generates checker board pattern.

$\mathbf{A}_{11} =$

1	0.1	1
0.1	-2	0.1
1	0.1	1

$\mathbf{B}_{11} =$

0	0	0
0	0	0
0	0	0

$z =$ 0 $\quad \beta_1 =$ 2

$\mathbf{A}_{22} =$

1	-0.1	1
-0.1	-2	-0.1
1	-0.1	1

$\mathbf{B}_{21} =$

0	0	0
0	0	0
0	0	0

$z =$ 0 $\quad \beta_2 =$ 2

IV. Example

Output of the 1st layer (2nd layer behaves in a very similar way):

$t=t_0+1$ [τ_{CNN}]

$t= t_0+2$

$t= t_0+3$

$t= t_0+4$

$t= t_0+5$

$t= t_0+6$

$t= t_0+7$

$t= t_0+8$

SPATIO-TEMPORAL PATTERNS OF AN ASYMMETRIC TEMPLATE CLASS

The template class analyzed in [57] produces novel spatio-temporal patterns that exhibit complex dynamics. The character of these propagating patterns depends on the self-feedback and on the sign of the coupling below the self-feedback template element.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ s & p & q \\ 0 & r & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{z} = \boxed{z}$$

I. Global Task

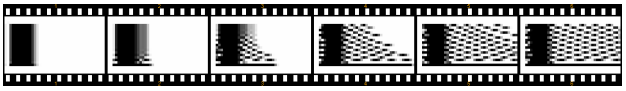
Given:	static binary image **P**
Input:	**U(t) = P**
Initial State:	**X(0) = P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Output:	Generated spatio-temporal pattern the structure of which depends on the sign of the extra template element *r*.

Remark:
The full range CNN model is used.

II. Examples

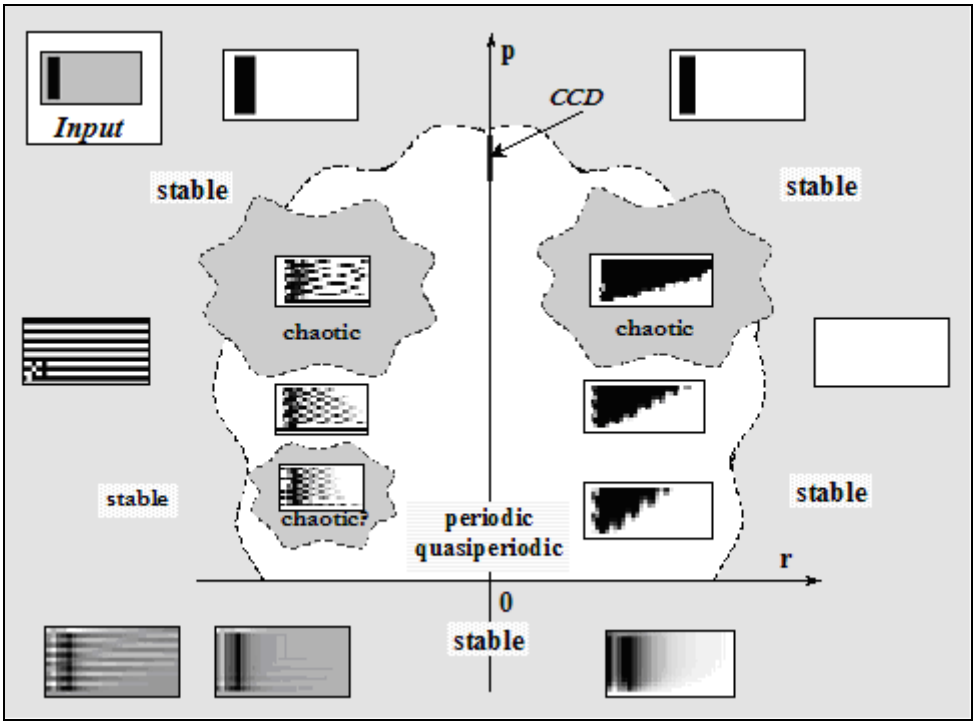
Examples are generated for the *sign-antisymmetric* case having $sq<0$ ($s = -q$).
Patterns generated depend on the sign of the extra template element *r*:

- if $r>0$ a pattern is formed which is solid inside, however its right border is oscillating.
- if $r<0$ a texture-like oscillating pattern is formed.

Coupling sign	*Propagating pattern*	*Snapshots*	*Time evolution*
positive: r > 0	solid inner part, oscillating border cells		
negative: r < 0	texture like oscillating pattern		

The effect of the central template element

If self-feedback is increased the generated pattern becomes more and more irregular and it can become chaotic. The *r-p* plane can be divided into *stable-periodic-chaotic* sub-regions as shown below.



The input & initial state is shown in the upper left corner. It is a three-pixel wide bar. The pictures in the different regions show few typical snapshots of outputs belonging to that region. The arrangement and size of the different regions gives only qualitative information.

1.11. NEUROMORPHIC ILLUSIONS AND SPIKE GENERATORS

HerringGridIllusion: Herring-grid illusion [13]

Old names: HERRING

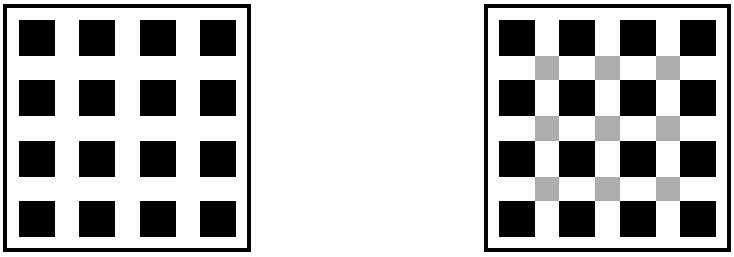
$$\mathbf{B} = \begin{bmatrix} -0.16 & -0.16 & -0.16 & -0.16 & -0.16 \\ -0.16 & -0.40 & -0.40 & -0.40 & -0.16 \\ -0.16 & -0.40 & 4 & -0.40 & -0.16 \\ -0.16 & -0.40 & -0.40 & -0.40 & -0.16 \\ -0.16 & -0.16 & -0.16 & -0.16 & -0.16 \end{bmatrix} \qquad z = \boxed{0}$$

$$\mathbf{A}^{\tau} = \begin{bmatrix} -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.3 & -0.3 & -0.3 & -0.1 \\ -0.1 & -0.3 & 0 & -0.3 & -0.1 \\ -0.1 & -0.3 & -0.3 & -0.3 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \end{bmatrix} \qquad \tau = 3\,\tau_{CNN}$$

I. Global Task

Given:	static binary image **P** with a grid of black squares
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Grayscale image representing **P** with gray patches at the intersections of the grid of black squares.

II. Example: image name: herring.bmp, image size: 256x256; template name: herring.tem .



input output

***MüllerLyerIllusion: Simulates the Müller-Lyer illusion* [13]**

Old names: MULLER

A =

0	0	0	0	0
0	0	0	0	0
0	0	1.3	0	0
0	0	0	0	0
0	0	0	0	0

B =

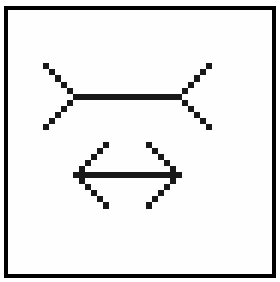
-0.1	-0.1	-0.1	-0.1	-0.1
-0.1	-0.1	-0.1	-0.1	-0.1
-0.1	-0.1	1.3	-0.1	-0.1
-0.1	-0.1	-0.1	-0.1	-0.1
-0.1	-0.1	-0.1	-0.1	-0.1

z = -2.8

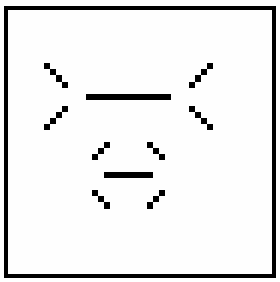
I. Global Task

Given:	static binary image **P** representing two horizontal lines between arrows. The arrows are dark-gray, the background is white
Input:	**U(t) = P**
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=0
Output:	**Y(t)⇒Y(∞)** = Binary image showing that the horizontal line on the top in **P** seems to be longer than the other one.

II. Example: image name: muller.bmp, image size: 44x44; template name: muller.tem .



input



output

SpikeGeneration1: ***Rhythmic burst-like spike generation using 4 ion channels, 2 of them are delayed***

Old names: SPIKE_BU

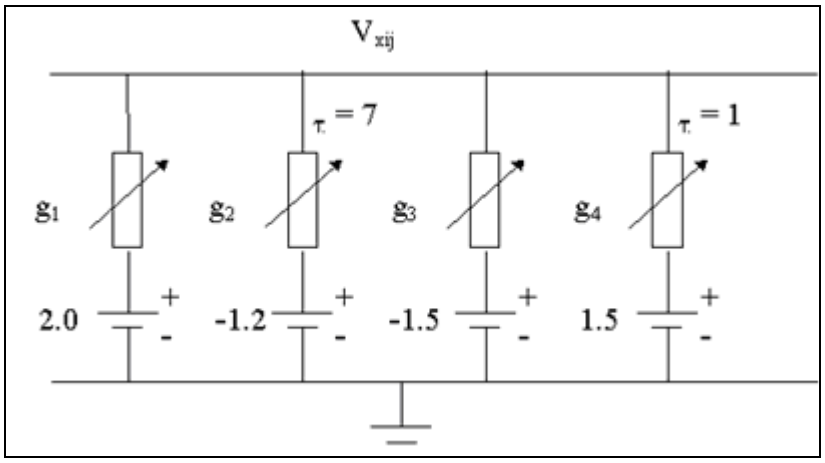
A =

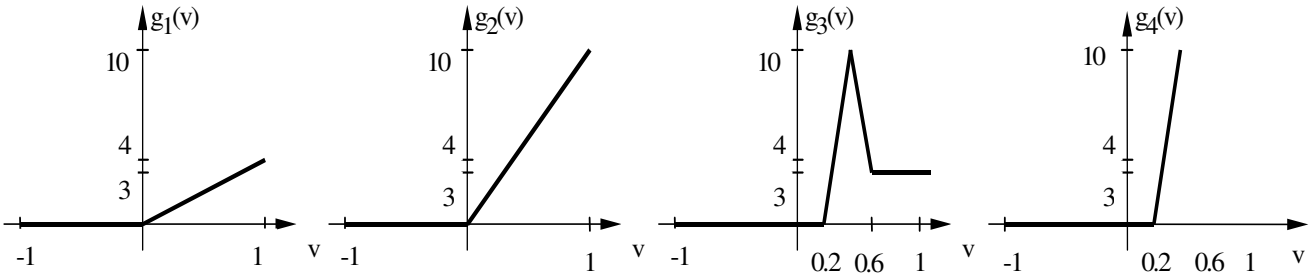
0	0	0
0	0	0
0	0	0

B =

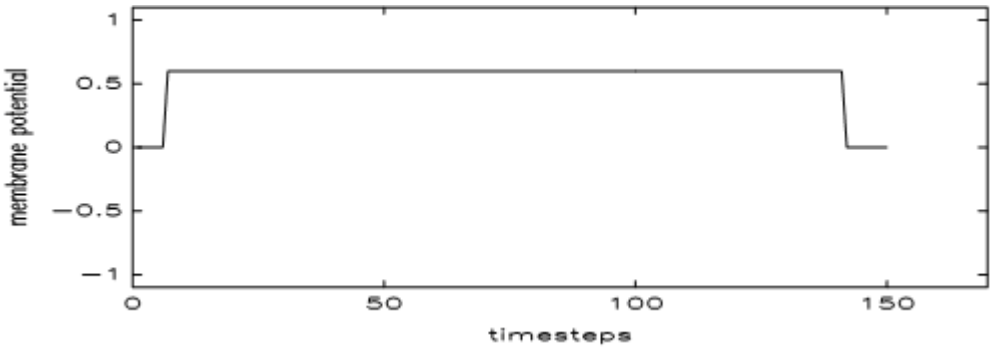
0	0	0
0	1	0
0	0	0

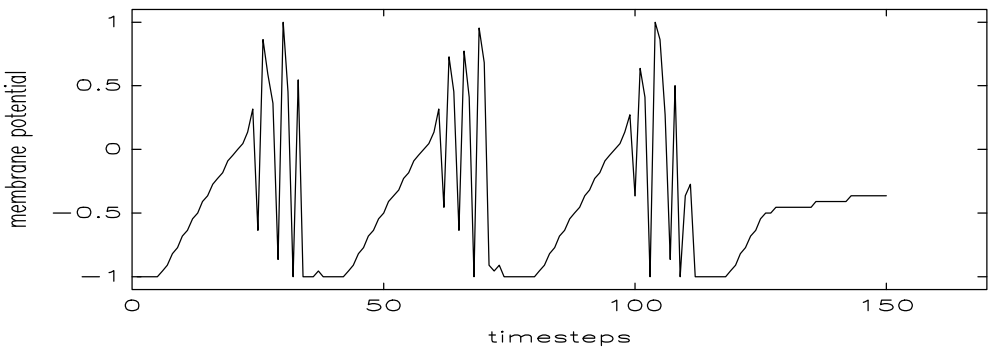
z = 0





Example: Input and output waveforms





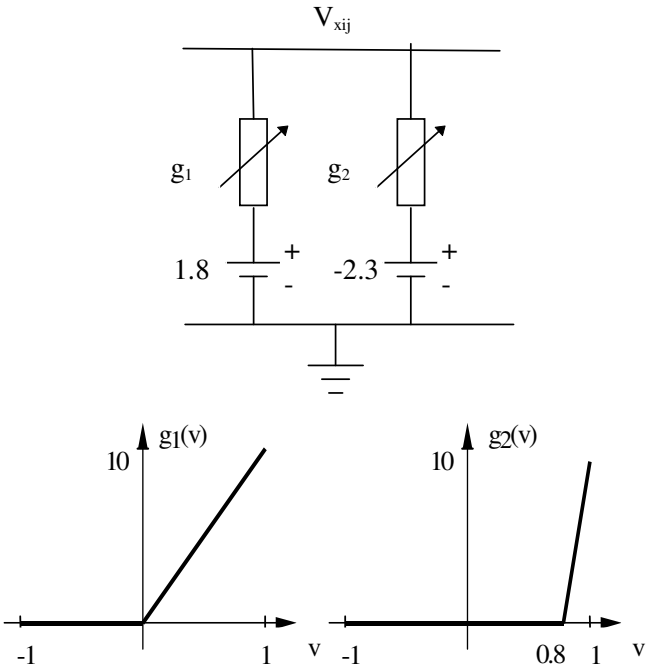
SpikeGeneration2: ***Action potential generation in a neuromorphic way*** **without delay** ***using 2 ion channels***

Old names: SPIKE_N

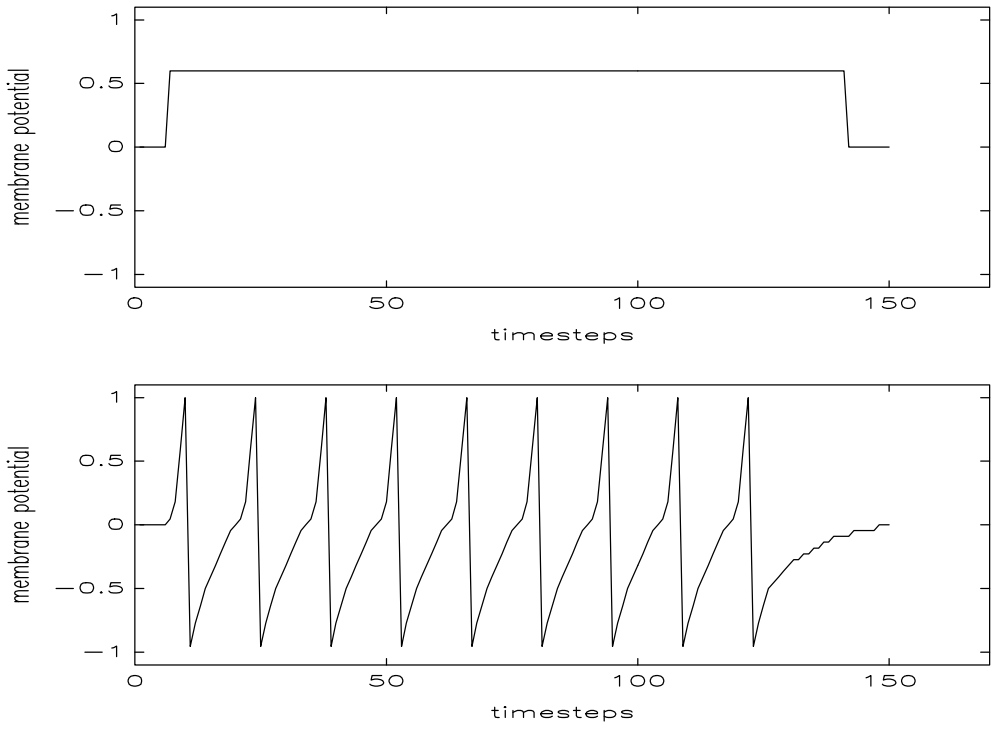
A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 0



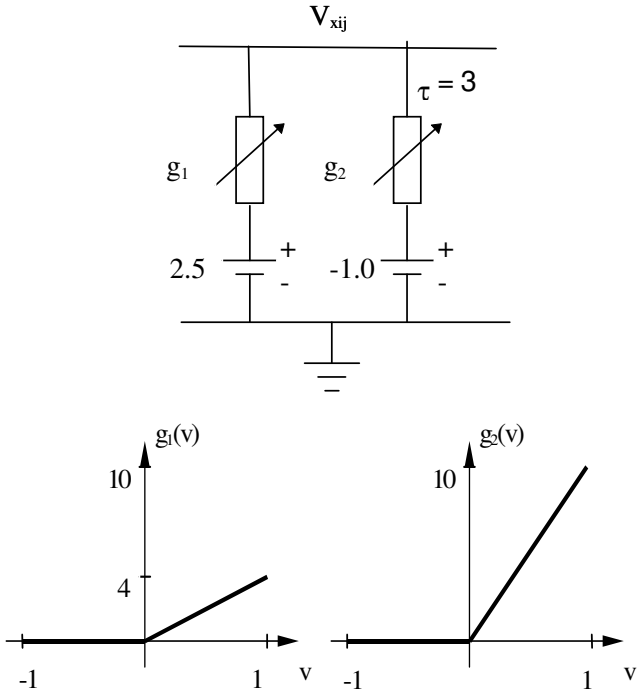
Example: Input and output waveforms



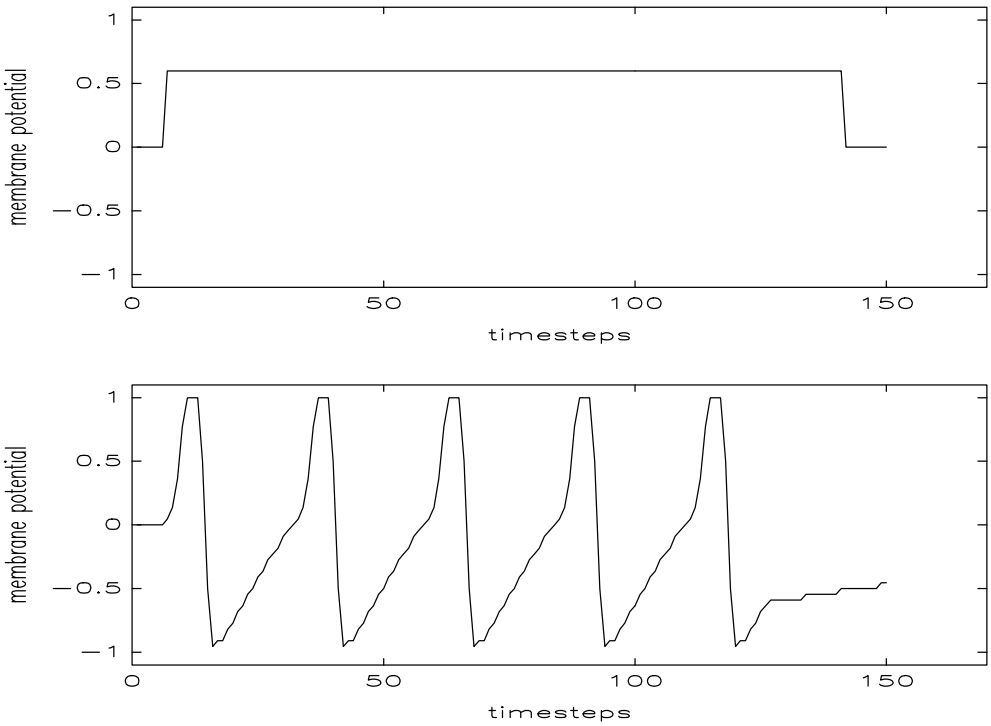
SpikeGeneration3: ***Action potential generation in a* neuromorphic *way, using 2 ion channels where one is* delayed. *Ion channels are modeled with voltage-controlled conductance (VCC) templates***

Old names: SPIKE_ND

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{0}$$



Example: Input and output waveform at each cell

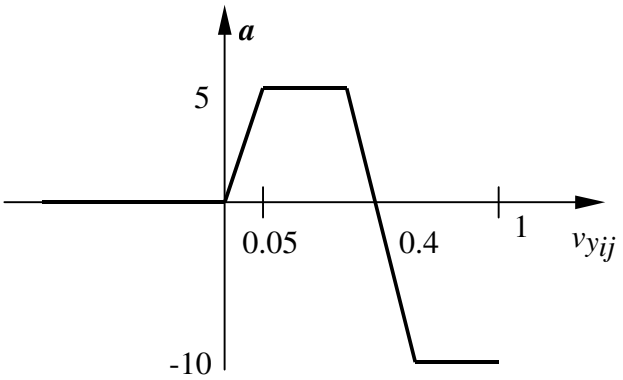


SpikeGeneration4: ***Action potential (spike) generation in a phenomenological way (with a nonlinear multivibrator-like template)***

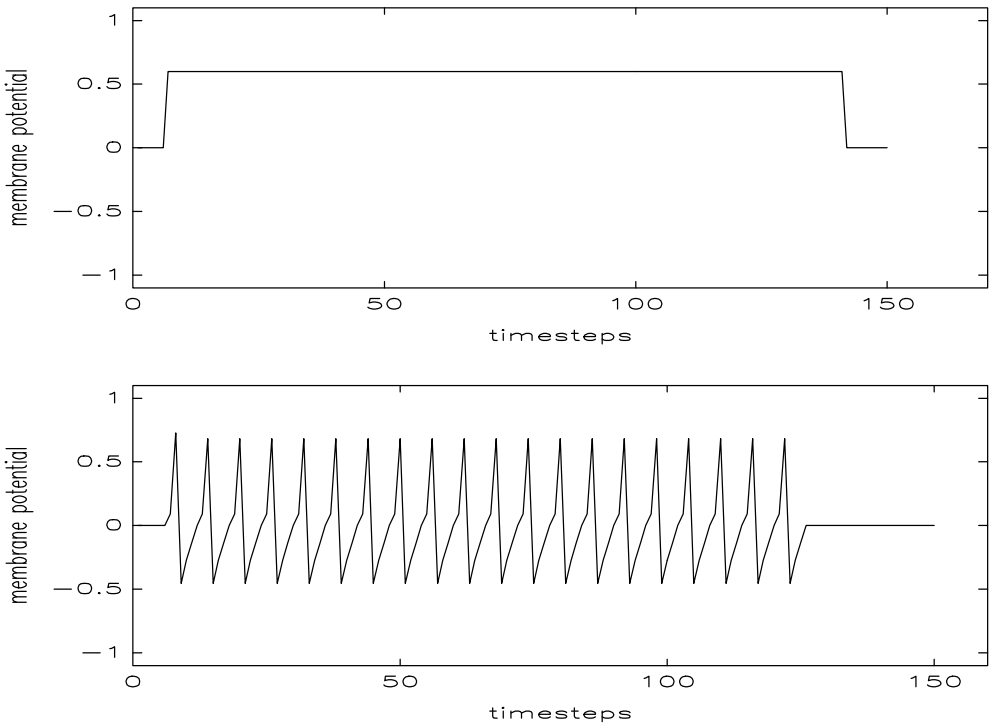
Old names: SPIKE_PH

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \boldsymbol{a} & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{0}$$

where ***a*** is defined by the following nonlinear function:



Example: Input and output waveform at each cell



1.12. CELLULAR AUTOMATA

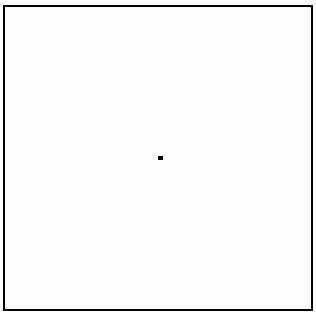
CELLULAR AUTOMATA [44]

Short Description

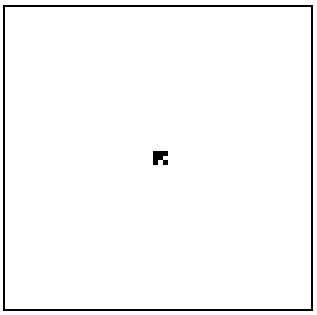
The following simple algorithm simulates the functioning of a cellular automaton. Input and output pictures are binary. The input of the n^{th} iteration is replaced by the output of the $(n-1)^{th}$ iteration.

Typical Example

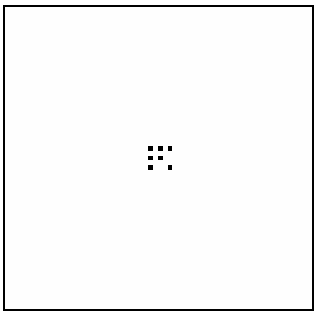
The following example shows a few consecutive states of the simulated cellular automata.



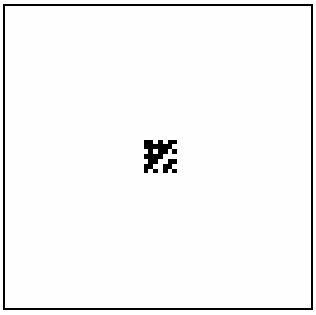
INPUT



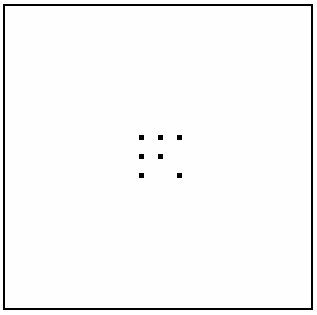
Output of the 1. iteration



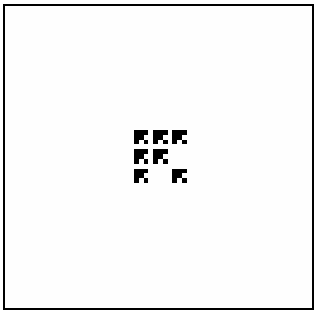
Output of the 2. iteration



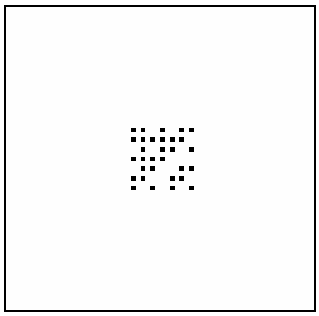
Output of the 3. iteration



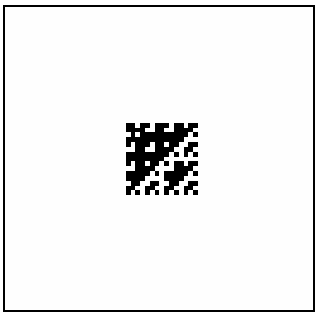
Output of the 4. iteration



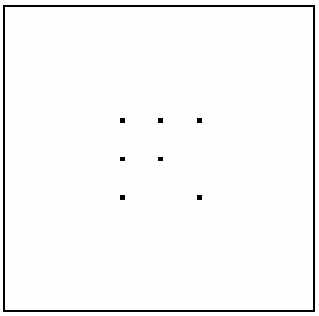
Output of the 5. iteration



Output of the 6. iteration

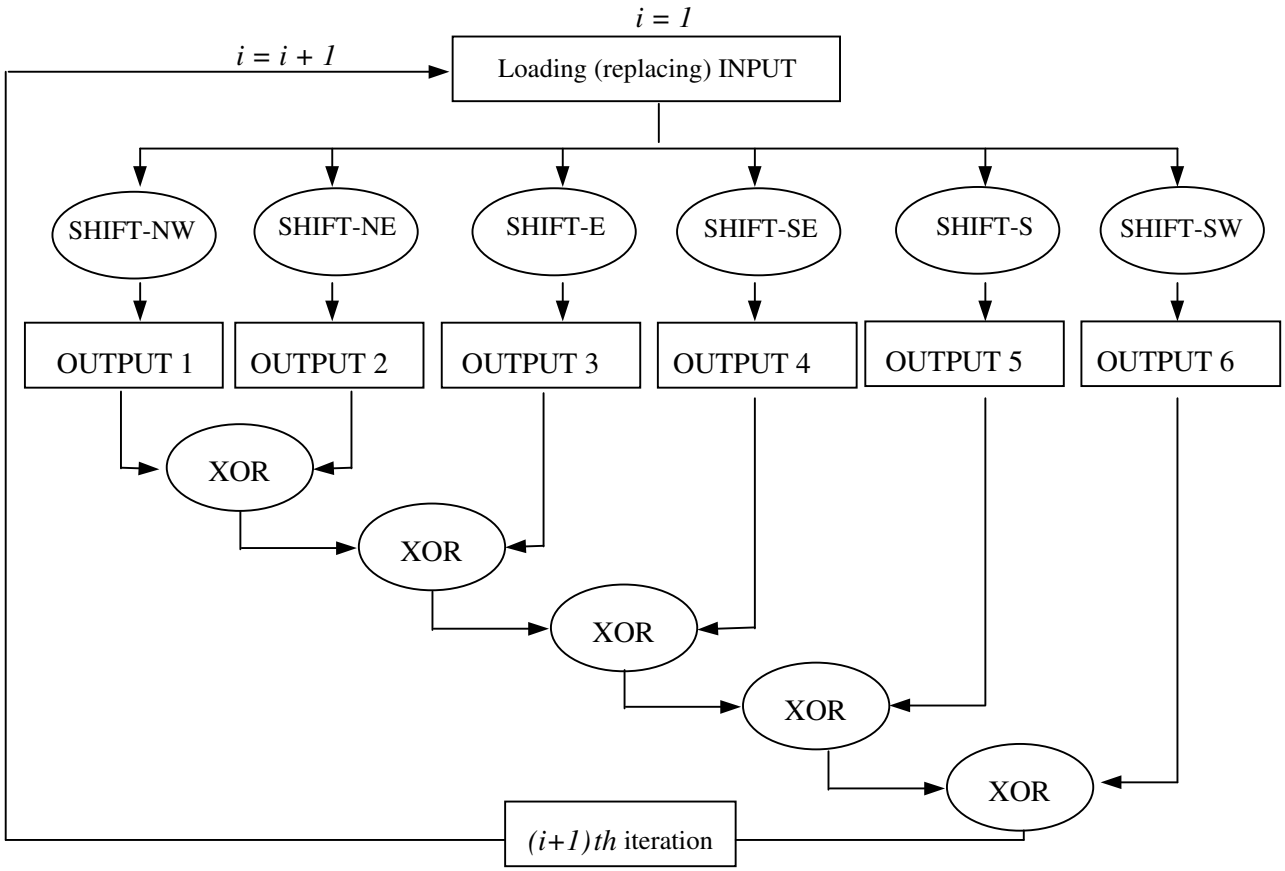


Output of the 7. iteration



Output of the 8. iteration

Block Diagram of the Algorithm



Templates used in the algorithm

Templates used in the algorithm are direction-dependent SHIFT templates, which are as follows:

SHIFT_NW:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

SHIFT_NE:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

SHIFT_E:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

SHIFT_SE:

A = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 1 |

z = | 0 |

SHIFT_S:

A = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |

z = | 0 |

SHIFT_SW:

A = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 1 | 0 | 0 |

z = | 0 |

ALPHA source

```
/* CELLAUT.ALF */
/* Performs a function of the cellular automata; */

PROGRAM cellaut(in);
CONSTANT
ONE = 1;
TIME = 5;
TIMESTEP = 1.0;
ENDCONST;

CHIP_SET simulator.eng;
/* Chip definition section */
A_CHIP
SCALARS

IMAGES
 im1: BINARY;
 im2: BINARY;
 im3: BINARY;
ENDCHIP;

/* Chip set definition section */
E_BOARD
SCALARS
cycle: INTEGER;
IMAGES
 input:  BINARY;
ENDBOARD;
```

```
/* Definition of analog operation symbol table */
OPERATIONS FROM cellaut.tms;

PROCESS cellaut;
 USE (shift_nw, shift_ne, shift_e, shift_se, shift_s, shift_sw);

 SwSetTimeStep (TIMESTEP);
 HostLoadPic(in, input);
 im1:= input;
 cycle:=1;

 REPEAT UNTIL (cycle > 30) ;
 input:=im1;
 HostDisplay(input, ONE);
    shift_nw (im1, im1, im2, TIME, PERIODIC);
    im2 := im1 XOR im2;

   shift_ne(im1, im1, im3, TIME, PERIODIC);
    im2 := im3 XOR im2;

   shift_e(im1, im1, im3, TIME, PERIODIC);
    im2 := im3 XOR im2;

   shift_se(im1, im1, im3, TIME, PERIODIC);
    im2 := im3 XOR im2;

   shift_s(im1, im1, im3, TIME, PERIODIC);
    im2 := im3 XOR im2;

   shift_sw(im1, im1, im3, TIME, PERIODIC);
    im2 := im3 XOR im2;

  im1:=im2;
 cycle := cycle + 1;
 ENDREPEAT;
ENDPROCESS;
ENDPROG;
```

GENERALIZED CELLULAR AUTOMATA [44]

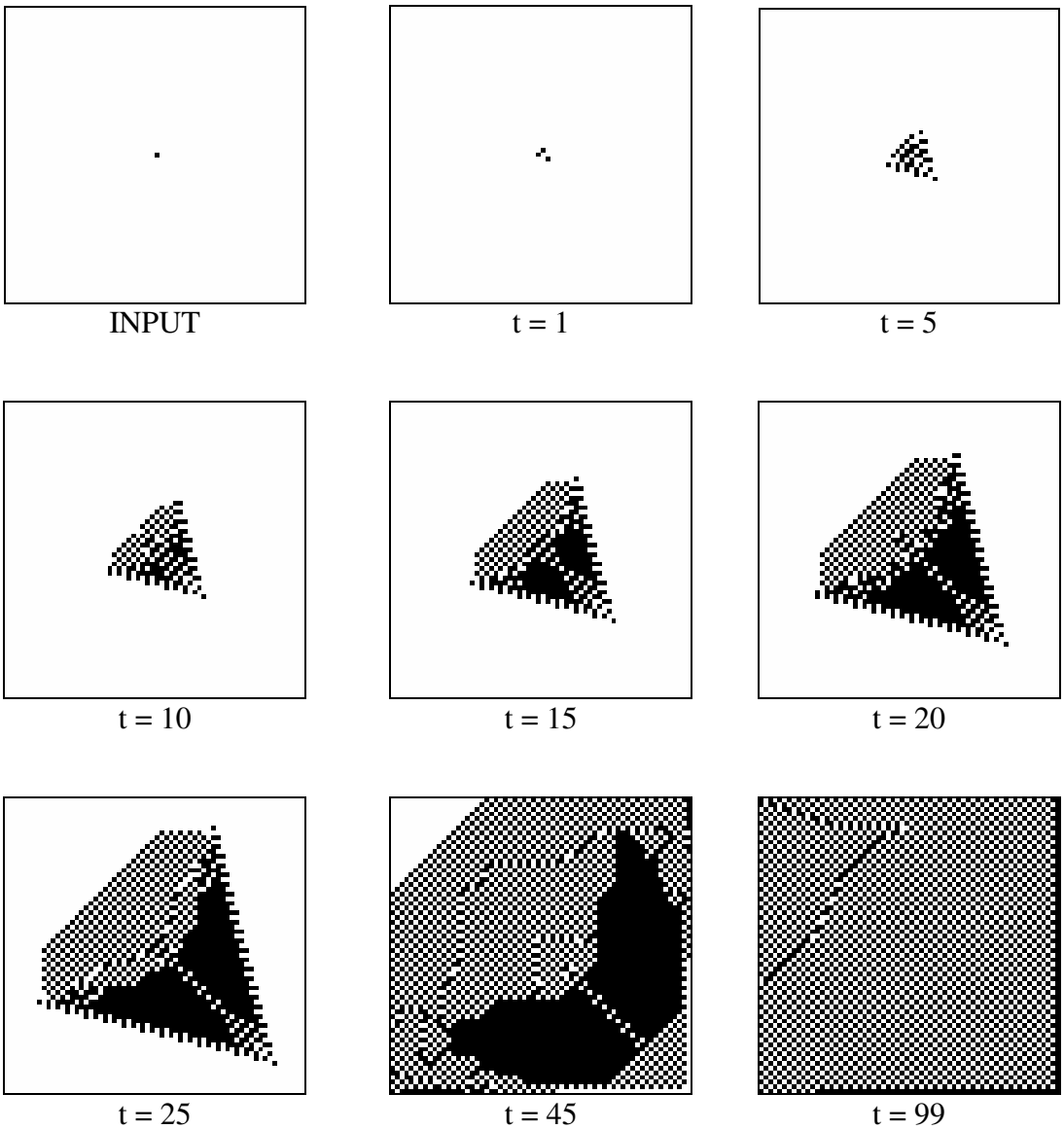
Definition: A Generalized Cellular Automaton (GCA) is a CNN with a single binary input/output template. The output is fed back to the input (the initial state is the same and is prescribed). If B = 0 then the output is fed back to the initial state.

Short Description

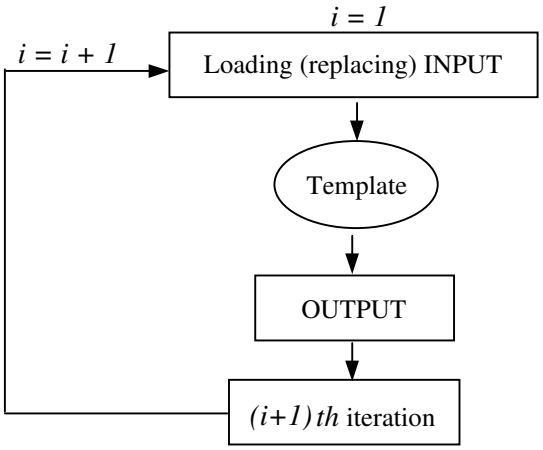
The following simple algorithm simulates the functioning of a general cellular automaton. Input and output pictures are binary. In each consecutive step the initial state is equal to zero. The input of the n^{th} iteration is replaced by the output of the $(n-1)^{th}$ iteration.

Typical Example

The following example shows a few consecutive steps of the simulated general cellular automaton.



Block Diagram of the Algorithm



Templates used in the algorithm

A =

0	0.5	0
0.5	2	-1
0	-1	0

B =

0	0.5	0
0.5	-0.5	0.5
0	0.5	0

z = 0.5

1.13. OTHERS

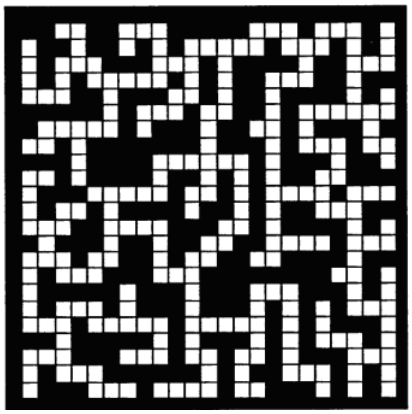
PathFinder: ***Finding all paths between two selected points through a labyrinth*** **[61]**

$$\mathbf{A} = \begin{bmatrix} 0.5 & 4 & 0.5 \\ 4 & 12 & 4 \\ 0.5 & 4 & 0.5 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{8}$$

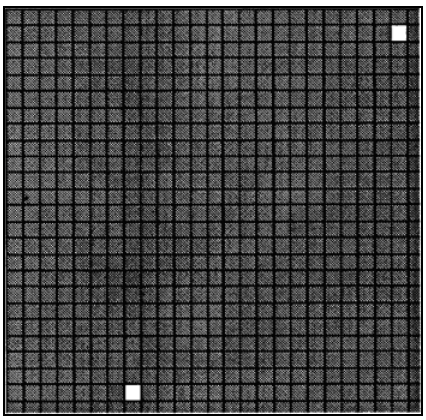
I. Global Task

Given:	static binary image **P** representing a labyrinth made of one-pixel thick white curves on a black background
Input:	**U(t) = 0**, except for two white pixels which mark the desired points in the labyrinth
Initial State:	**X**(0) **= P**
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = Binary image containing all the paths connecting the marked points (made of white curves against a black background)

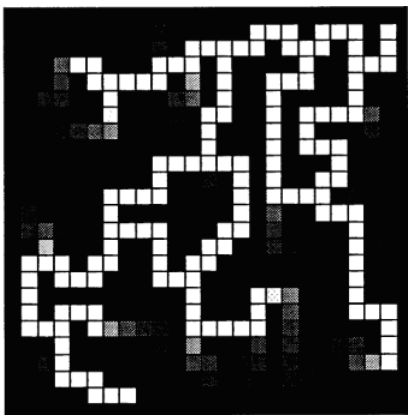
II. Example: image size: 25x25.



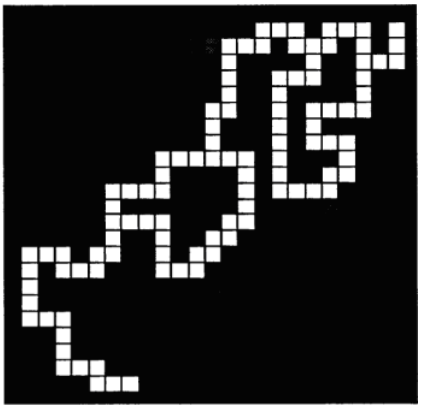
Initial state



Input



Intermediate result



Output

ImageInpainting: ***Interpolation-based image restoration [58]***

Old names: NEL_AINTPOL3

$$\mathbf{A} = \begin{bmatrix} -0.05 & 0.3 & -0.05 \\ 0.3 & 0 & 0.3 \\ -0.05 & 0.3 & -0.05 \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} 0 & d & 0 \\ d & 0 & d \\ 0 & d & 0 \end{bmatrix}$$

where $d = \lambda sign(y_{ij} - x_{kl})$, $\lambda \in [-1, 0]$, with (**B**=0, z=0).

I. Global Task

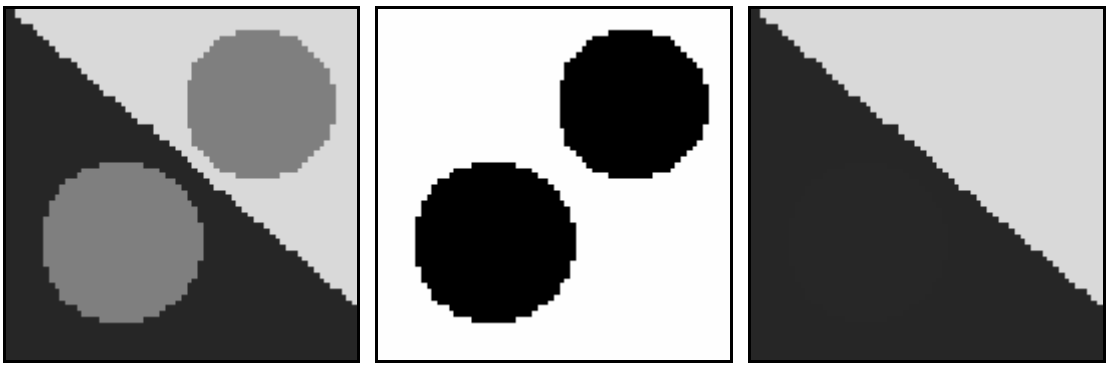
Given:	static grayscale image **P** (image to be restored = missing or damaged image) and static binary image **M**
Input:	**U(t) =** Arbitrary (in the examples we choose **U(t)= P**)
Initial State:	**X**(0) **= P**
Boundary Conditions:	Zero-flux boundary condition (duplicate)
Fixed State Mask:	$\mathbf{X_{fix}=M}$**.** It is necessary to use a mask image that does not change the elements of the image that are known at the beginning, but allow the computing of unknown elements. The existence of a mask image presumes that the user knows the positions of the elements that need to be computed.
Output:	**Y(t)⇒Y(∞)** = Grayscale image representing the restored missing or damaged image.

Remark:

Image inpainting is an interpolation problem where an image with missing or damaged parts is restored.

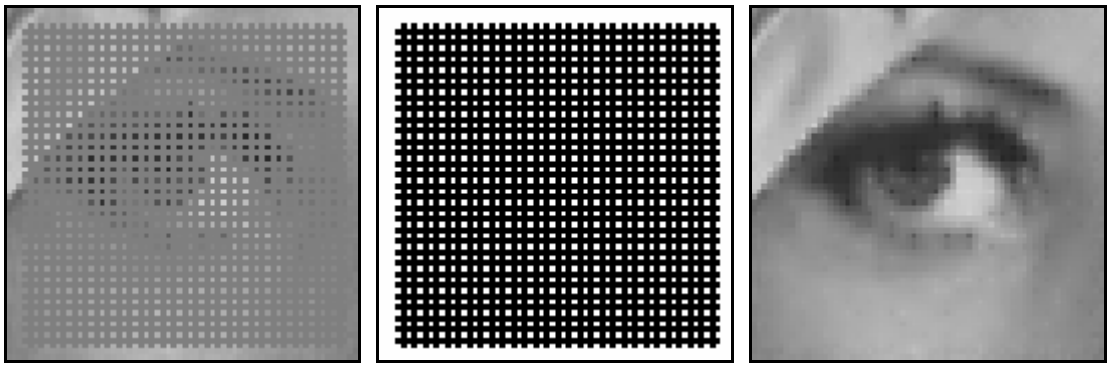
II. Examples

Example 1: image name: inphole.bmp, image size: 64x64; template name: nel_aintpol3.tem .



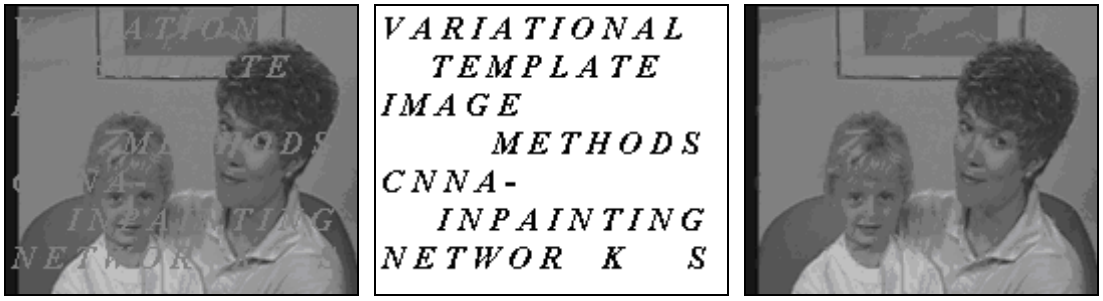
input mask output

Example 2: image name: inpeye.bmp, image size: 64x64; template name: nel_aintpol3.tem .



input mask output

Example 3: image name: inpaintig.bmp, template name: nel_aintpol3.tem .



input mask output

ImageDenoising: Image denoising based on the total variational (TV) model of Rudin-Osher-Fatemi [59, 60]

A = | 0 | *a* | 0 |
| *a* | 1 | *a* |
| 0 | *a* | 0 |

D = | 0 | 0 | 0 |
| 0 | *d* | 0 |
| 0 | 0 | 0 |

where $a = \lambda sign(x_{ij} - x_{kl})$ $\lambda \in [-1, 0]$ and $d = 2\alpha(x_{ij} - u_{ij})$ $\alpha \in [0, 1]$, with (**B**=0, z=0).

I. Global Task

Given: static grayscale image **P** (image to be denoising)

Input: **U(t) =** Arbitrary (in the examples we choose **U(t)= P**)

Initial State: **X**(0) **= P**

Boundary Conditions: Zero-flux boundary condition (duplicate)

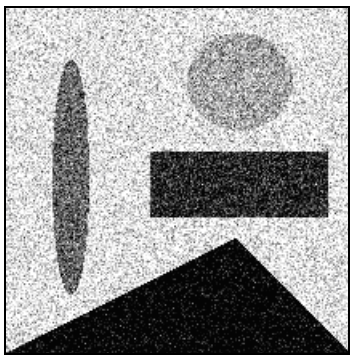
Output: **Y(t)⇒Y(∞)** = Grayscale image representing the denoising image.

Remark:

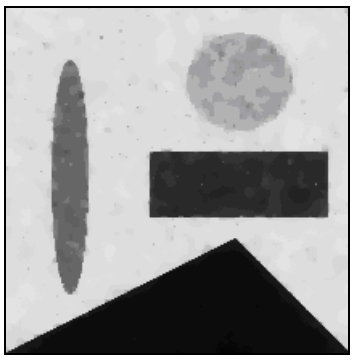
The noise is white Gaussian and additive.

II. Examples

Example 1: image name: osrufa5.bmp, image size: 214x216; templates name: osrufa2.tem.



input



output

Example 2: image name: cameraman10.bmp, image size: 256x256; templates name: osrufa.tem.

input

output

Orientation-SelectiveLinearFilter: ***IIR linear filter with orientation-selective low-pass (LP) frequency response, oriented at an angle*** φ ***with respect to an axis of the frequency plane*** **[62]**

Given the orientation angle φ, first the 3×3 *orientation matrix* O_φ is obtained:

$$O_\varphi = -\frac{1}{2}\left(\sin 2\varphi \cdot \begin{bmatrix} 0 & 1 & -1 \\ 1 & -2 & 1 \\ -1 & 1 & 0 \end{bmatrix} + \cos 2\varphi \cdot \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \right)$$

then another two templates depending only on φ are obtained:

$$B_\varphi = O_0 - 0.4532 \cdot O_\varphi^{'} + 0.0194 \cdot O_\varphi * O_\varphi$$

$$A_\varphi = O_0 + 0.0468 \cdot O_\varphi^{'} + 0.0012 \cdot O_\varphi * O_\varphi$$

where "$*$" = matrix convolution; O_0 is a 5×5 zero matrix with the central element of value 1; the 5×5 matrix $O_\varphi^{'}$ is the 3×3 matrix O_φ bordered with zeros in order to be summed with O_0 and $O_\varphi * O_\varphi$. In the two templates A_φ, B_φ (5×5), the marginal elements are generally negligible and can be discarded, so A_φ, B_φ can be reduced to size 3×3 with a minimum error.

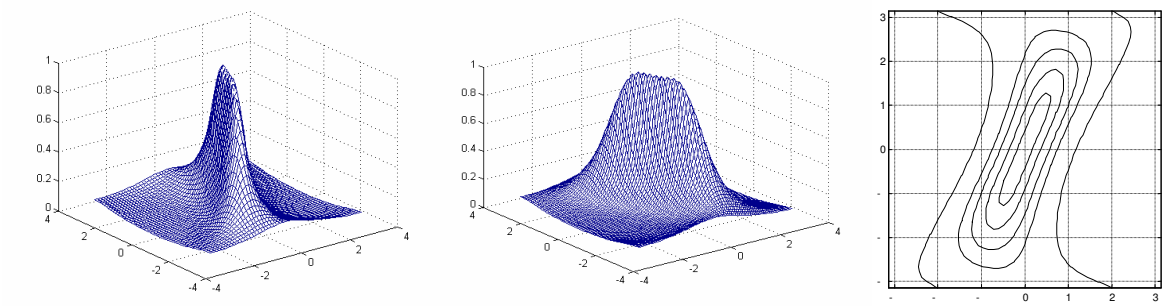
An LP oriented filter with the general spatial transfer function $H_\varphi(\omega_1,\omega_2) = \dfrac{B(\omega_1,\omega_2)}{A(\omega_1,\omega_2)}$ can be designed using the templates, with parameter $p<0$ specifying selectivity:

$$B(\omega_1,\omega_2) = A_\varphi(\omega_1,\omega_2)$$

$$A(\omega_1,\omega_2) = -(p+1)\cdot A_\varphi(\omega_1,\omega_2) + p \cdot B_\varphi(\omega_1,\omega_2)$$

Example: An LP oriented filter with $p_a = -37.6$ and $\varphi = \pi/8$ is realized with:

$$A = \begin{bmatrix} -0.1413 & 8.4406 & 5.8977 \\ -3.2964 & -23.6541 & -3.2964 \\ 5.8977 & 8.4406 & -0.1413 \end{bmatrix}; \quad B = \begin{bmatrix} 0.0005 & 0.0464 & 0.0331 \\ -0.0199 & -1.9623 & -0.0199 \\ 0.0331 & 0.0464 & 0.0005 \end{bmatrix}$$



Frequency response of a selective oriented LP filter ($p_a = -37.6$ and $\varphi = \pi/8$) viewed from two angles and constant level contours

Complex-Gabor: ***Filtering with a complex-valued Gabor-type filter*** **[53]**

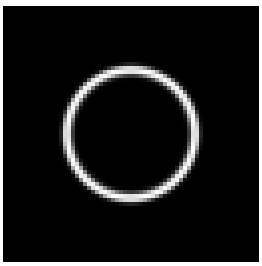
$$\mathbf{A} = \begin{bmatrix} 0 & e^{-j\Omega_y} & 0 \\ e^{j\Omega_x} & -\left(3+\lambda^2\right) & e^{-j\Omega_x} \\ 0 & e^{j\Omega_y} & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \lambda^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad z = \boxed{0}$$

where Ω_x and Ω_y control the spatial frequency tuning of the filter and λ controls the bandwidth. Note that the off-center elements are complex valued $\left(j=\sqrt{-1}\right)$. The state is also assumed to be complex valued. Note that the center element of the template presented here differs from that presented in [53] by 1 because we assume the standard CNN equation here, whereas [53] used an equation without the resistive loss term.

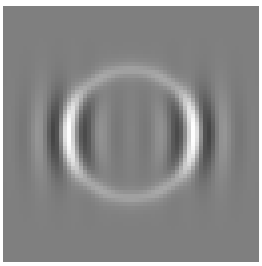
I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary or as a default *X*(t)=0. Note that the state is assumed to be complex valued.
Boundary Conditions:	Fixed type, y_{ij} = 0 for all virtual cells, denoted by [Y]=0
Output:	**Y(t)⇒Y(∞)** = grayscale image representing the output of the Gabor-type filter. The real part of the output is the output of an even-symmetric Gabor-type filter. The imaginary part of the output is the output of an odd symmetric Gabor type filter.

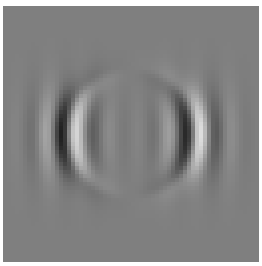
II. Example: image name: annulus.bmp, image size: 64x64; template name: cgabor.tem .



P (input)



Real(Y)



Imag(Y)

where $\lambda = 0.2$, $\Omega_x = 2\pi/8$, $\Omega_y = 0$.

Two-Layer Gabor: ***Two-layer template implementing even and odd Gabor-type filters***

$$\mathbf{A_{11}} = \begin{bmatrix} 0 & \cos(\Omega_y) & 0 \\ \cos(\Omega_x) & -(3+\lambda^2) & \cos(\Omega_x) \\ 0 & \cos(\Omega_y) & 0 \end{bmatrix} \quad \mathbf{B_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \lambda^2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z_1 = \boxed{0}$$

$$\mathbf{A_{22}} = \begin{bmatrix} 0 & \cos(\Omega_y) & 0 \\ \cos(\Omega_x) & -(3+\lambda^2) & \cos(\Omega_x) \\ 0 & \cos(\Omega_y) & 0 \end{bmatrix} \quad \mathbf{B_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z_2 = \boxed{0}$$

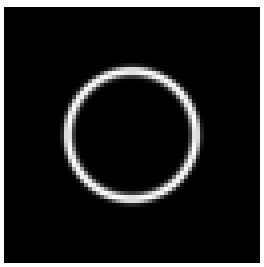
$$\mathbf{A_{12}} = \begin{bmatrix} 0 & \sin(\Omega_y) & 0 \\ -\sin(\Omega_x) & 0 & \sin(\Omega_x) \\ 0 & -\sin(\Omega_y) & 0 \end{bmatrix} \quad \mathbf{A_{21}} = \begin{bmatrix} 0 & -\sin(\Omega_y) & 0 \\ \sin(\Omega_x) & 0 & -\sin(\Omega_x) \\ 0 & \sin(\Omega_y) & 0 \end{bmatrix}$$

where Ω_x and Ω_y control the spatial frequency tuning of the filter and λ controls the bandwidth. This template is equivalent to the Complex-Gabor template, where we have separated the real and imaginary parts to two layers.

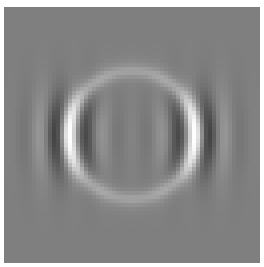
I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	$\mathbf{X_1}(0) = \mathbf{X_2}(0) =$ Arbitrary or as a default $X_1(t)= X_2(t)=0$.
Boundary Conditions:	Fixed type, $y_{ij} = 0$ for all virtual cells, denoted by [Y]=0
Output:	$\mathbf{Y_1(t)} \Rightarrow \mathbf{Y_1(\infty)}$ = grayscale image representing the output of the even-symmetric Gabor-type filter.
	$\mathbf{Y_2(t)} \Rightarrow \mathbf{Y_2(\infty)}$ = grayscale image representing the output of the odd-symmetric Gabor-type filter.

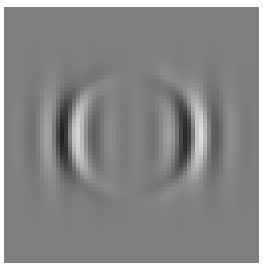
II. Example: image name: annulus.bmp, image size: 64x64; template name: cgabor.tem .



P (input)



Y_1



Y_2

where $\lambda = 0.2$, $\Omega_x = 2\pi/8$, $\Omega_y = 0$.

LinearTemplateInverse: ***Inverse of a linear template operation using dense support of input pixels [55]***

A Linear Template to be inverted

$\mathbf{A} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

$\mathbf{B} =$ | -0.03 | -0.10 | -0.02 |
|---|---|---|
| 0 | 0.50 | -0.20 |
| -0.03 | -0.10 | -0.02 |

$z =$ 0

Example: image names: LenaS.bmp; image size: 128x128; template name: CS2.tem.

TEST INPUT

TEST OUTPUT

Old names: Linear Template Inverse (A_i=1-B; B_i=1-A; A and B see above)

$\mathbf{A_i} =$ | 0.03 | 0.10 | 0.02 |
|---|---|---|
| 0 | 0.50 | 0.20 |
| 0.03 | 0.10 | 0.02 |

$\mathbf{B_i} =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$z =$ 0

I. Global Task

Given: a linear template as well as two static gray scale images $\mathbf{P_1}$ (result of the linear template operation (see the test template above and its output) and $\mathbf{P_2}$. (masked version of the original image). $\mathbf{P_3}$ is a binary version of $\mathbf{P_2}$ providing the fixed state mask for CNN operation. $\mathbf{P_3}$ indicates the positions of supporting pixels where the interpolation is fixed.. The result of the inverse of a linear template operation is computed rapidly using masked diffusion even if the template cannot be inverted (linear template – convolution kernel - have zero Eigen values).

Input: $\mathbf{U(t) = P_1}$

Initial State: $\mathbf{X(0) = P_2}$

Fixed State Mask: $\mathbf{X_{fix} = P_3}$

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]= [Y]=0

Output: $\mathbf{Y(t) \Rightarrow Y(\infty)}$ = Gray scale image containing the inverse of the B template operation (B=1-A).

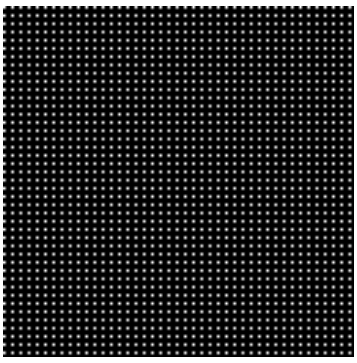
Remark:

The inverse operation of a linear template can be easily performed by using a dense support even if the theoretical inverse converges too slowly or if theoretically the template operation cannot be inverted.

II. Example: image names: LenaSCs.bmp, LenaSMask.bmp, MaskS.bmp; image size: 128x128; template name: DiffM2.tem .

INPUT (P_1)

INITIAL STATE (P_2)



MASK (P_3)

OUTPUT

Translation(dx,dy): *Translation by a fraction of pixel (dx,dy) with -1 ≤dx≤1 and -1 ≤dy ≤1*

$$\mathbf{A} = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \qquad \mathbf{B} = \begin{array}{|c|c|c|}\hline a\,b\,\mathrm{H}\,\mathrm{V} & (1-a)\,b\,\mathrm{H} & a\,b\,\mathrm{H}\,(1-\mathrm{V}) \\ \hline a\,(1-b)\,\mathrm{V} & (1-a)\,(1-b) & a\,(1-b)\,(1-\mathrm{V}) \\ \hline a\,b\,(1-\mathrm{H})\,\mathrm{V} & (1-a)\,b\,(1-\mathrm{H}) & a\,b\,(1-\mathrm{H})\,(1-\mathrm{V}) \\ \hline \end{array} \qquad z = \boxed{0}$$

where

- $H = 1$, if $dy > 0$; $H = 0$, otherwise
- $V = 1$, if $dx > 0$; $V = 0$, otherwise
- $a = |dx|$ $b = |dy|$

I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 1$ for all virtual cells, denoted by [U]=1
Output:	**Y(t)⇒Y(∞)** = Grayscale image **P** shifted horizontally by *dx* and vertically by *dy*.

Remark:

Translations by a value greater than one pixel can be achieved by applying the same template many times. For example, if $dx > dy > 1$:

ny = trunc(dy); $fy = dy - ny$;
nx = trunc(dx); $fx = dx - nx$;
- Apply template *Translation(fx,fy)*
- repeat *nx-ny* times *Translation (1,1)*
- repeat *nx* times *Translation (1,0)*

II. Example image name: lenna.bmp, image size: 256x256; dx = -10.5; dy = 4.7

INPUT

OUTPUT

Rotation: ***Image rotation by angle ϕ around (O_x, O_y)***

The *Rotation* algorithm is based on a space-variant version of the *Translation(dx,dy)* template:

A =

0	0	0
0	0	0
0	0	0

B =

$a\ b$ H V	(1 - a) b H	$a\ b$ H (1 - V)
a (1 - b) V	(1 - a) (1 - b)	a (1 - b) (1 - V)
$a\ b$ (1 - H) V	(1 - a) b (1 - H)	$a\ b$ (1 - H) (1 - V)

z =

0

where

- $dx(i,j) = O_x + (j - O_x)\cos\varphi - (i - O_y)\sin\varphi - j$
- $dy(i,j) = O_y + (j - O_x)\sin\varphi + (i - O_y)\cos\varphi - i$
- $H = 1$, if $dy(i,j) > 0$; $H = 0$, otherwise
- $V = 1$, if $dx(i,j) > 0$; $V = 0$, otherwise
- $a = |dx(i,j)|$ $b = |dy(i,j)|$

If $|dx|>1$ or $|dy|>1$ for some i,j, an exact rotation is not possible with a neighborhood order 1.

We can perform an approximation with an error depending on ϕ, by applying the *Translation* template many times with the following algorithm:

- Compute the integer and fractional parts:
  $nx(i,j)$ = trunc($dx(i,j)$); $fx(i,j) = dx(i,j) - nx(i,j)$
  $ny(i,j)$ = trunc($dy(i,j)$); $fy(i,j) = dy(i,j) - ny(i,j)$
  m=max(($nx(i,j)$,$ny(i,j)$))

- Apply space-variant template *Translation(fx,fy)*

- for k = 1 to m:

  if $|nx(i,j)| - k > 0$ then $kx(i,j)$ = sign($nx(i,j)$) else $kx(i,j) = 0$
  if $|nx(i,j)| - k > 0$ then $ky(i,j)$ = sign($ny(i,j)$) else $ky(i,j) = 0$
  Apply the space-variant template *Translation(kx, kx)*

- Apply the low-pass template:

A =

0	0	0
0	0	0
0	0	0

B =

0.11	0.11	0.11
0.11	0.11	0.11
0.11	0.11	0.11

z =

0

- Apply the hi-pass template:

A =

0	0	0
0	0	0
0	0	0

B =

-0.04	-0.12	-0.04
-0.12	1.57	-0.12
-0.04	-0.12	-0.04

z =

0

I. Global Task

Given:	static grayscale image **P**
Input:	**U(t) = P**
Initial State:	**X**(0) **=** Arbitrary (in the examples we choose $x_{ij}(0)=0$)
Boundary Conditions:	Fixed type, $u_{ij} = 1$ for all virtual cells, denoted by [U]=1
Output:	**Y(t)⇒Y(∞)** = Grayscale image **P** rotated by ϕ around (O_x, O_y).

II. Example: image name: lenna.bmp, image size: 256x256; $\phi = -10°$; $O_x = 0$; $O_y = 0$

INPUT

OUTPUT

Chapter 2. Subroutines and Simpler Programs

BLACK AND WHITE SKELETONIZATION

Old names: SKELBW

Task description and algorithm

The algorithm finds the skeleton of a black-and-white object. The 8 templates should be applied circularly, always feeding the output back to the input before using the next template [19].

The templates of the algorithm:

SKELBW1:

$\mathbf{A}_1 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_1 =$ | 1 | 1 | 0 |
|---|---|---|
| 1 | 5 | -1 |
| 0 | -1 | 0 |

$z_1 =$ -1

SKELBW2:

$\mathbf{A}_2 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_2 =$ | 2 | 2 | 2 |
|---|---|---|
| 0 | 9 | 0 |
| -1 | -2 | -1 |

$z_2 =$ -2

SKELBW3:

$\mathbf{A}_3 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_3 =$ | 0 | 1 | 1 |
|---|---|---|
| -1 | 5 | 1 |
| 0 | -1 | 0 |

$z_3 =$ -1

SKELBW4:

$\mathbf{A}_4 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_4 =$ | -1 | 0 | 2 |
|---|---|---|
| -2 | 9 | 2 |
| -1 | 0 | 2 |

$z_4 =$ -2

SKELBW5:

$\mathbf{A}_5 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_5 =$ | 0 | -1 | 0 |
|---|---|---|
| -1 | 5 | 1 |
| 0 | 1 | 1 |

$z_5 =$ -1

SKELBW6:

$\mathbf{A}_6 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_6 =$ | -1 | -2 | -1 |
|---|---|---|
| 0 | 9 | 0 |
| 2 | 2 | 2 |

$z_6 =$ -2

SKELBW7:

$\mathbf{A}_7 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_7 =$ | 0 | -1 | 0 |
|---|---|---|
| 1 | 5 | -1 |
| 1 | 1 | 0 |

$z_7 =$ -1

SKELBW8:

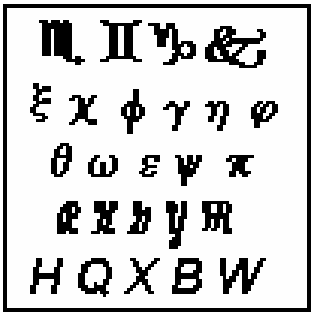
$\mathbf{A}_8 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_8 =$ | 2 | 0 | -1 |
|---|---|---|
| 2 | 9 | -2 |
| 2 | 0 | -1 |

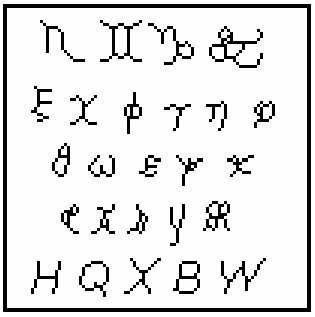
$z_8 =$ -2

The robustness of templates SKELBW1 and SKELBW2 are ρ(SKELBW1) = 0.18 and ρ(SKELBW2) = 0.1, respectively. Other templates are the rotated versions of SKELBW1 and SKELBW2, thus their robustness values are equal to the mentioned ones.

Example: image name: skelbwi.bmp, image size: 100x100; template names: skelbw1.tem, skelbw2.tem, …, skelbw8.tem.

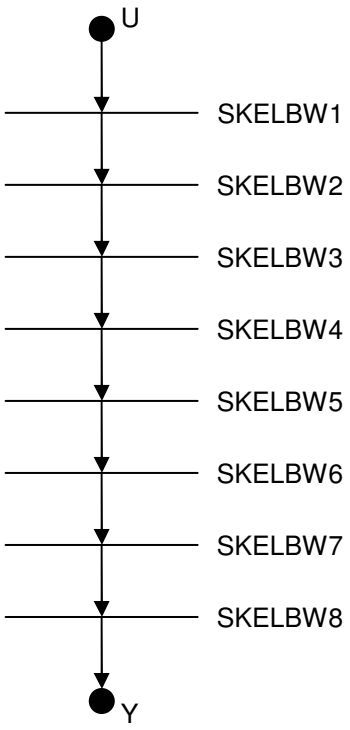


input



output

UMF diagram



GRAYSCALE SKELETONIZATION

Old names: SKELGS

Task description and algorithm

The algorithm finds the skeleton of grayscale objects. The algorithm uses 8 sets of templates, skeletonizing the objects circularly. Each step contains two templates. First, appropriate pixels are selected by the *selection* templates; afterwards these are used as fixed state masks for the *replacement*. The result of the replacement is fed back to the input. Only 4 of the 8 required templates are shown, the others can easily be generated by rotating functions *a* and *b*, as indicated in the first 3 steps [19].

The templates of the algorithm:

Selection templates: *Replacement:*

$\mathbf{A}_1 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_1 =$ | ***a*** | ***a*** | 0 |
|---|---|---|
| ***a*** | 0 | ***b*** |
| 0 | ***b*** | 0 |

$z_1 =$ -4.5

$\mathbf{A}_1 =$ | ***c*** | ***c*** | 0 |
|---|---|---|
| ***c*** | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{A}_2 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_2 =$ | ***a*** | ***a*** | ***a*** |
|---|---|---|
| 0 | 0 | 0 |
| ***b*** | ***b*** | 0 |

$z_2 =$ -4.5

$\mathbf{A}_2 =$ | ***c*** | ***c*** | ***c*** |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{A}_3 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_3 =$ | 0 | ***a*** | ***a*** |
|---|---|---|
| ***b*** | 0 | ***a*** |
| 0 | ***b*** | 0 |

$z_3 =$ -4.5

$\mathbf{A}_3 =$ | 0 | ***c*** | ***c*** |
|---|---|---|
| 0 | 1 | ***c*** |
| 0 | 0 | 0 |

. . . . . .

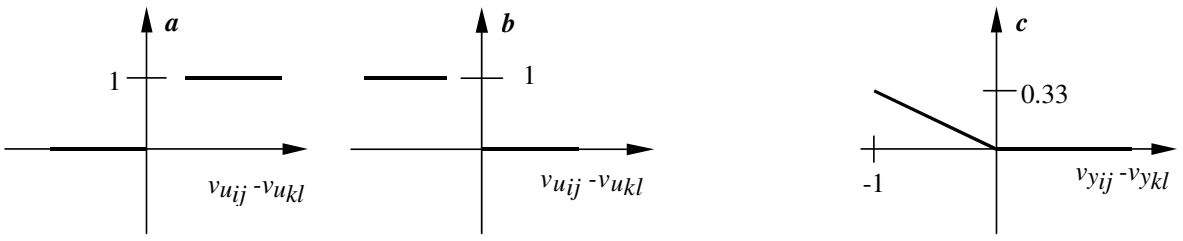
$\mathbf{A}_8 =$ | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

$\mathbf{B}_8 =$ | ***a*** | 0 | 0 |
|---|---|---|
| ***a*** | 0 | ***b*** |
| ***a*** | 0 | ***b*** |

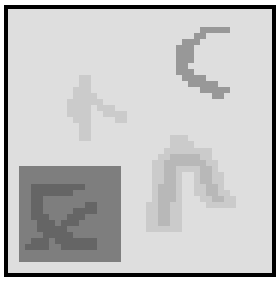
$z_8 =$ -4.5

$\mathbf{A}_8 =$ | ***c*** | 0 | 0 |
|---|---|---|
| ***c*** | 1 | 0 |
| ***c*** | 0 | 0 |

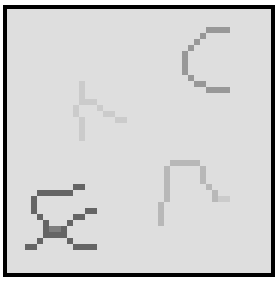
where ***a***, ***b*** and ***c*** are defined by the following nonlinear functions:



Example 1: image name: skelg2i.bmp; image size: 44x44.

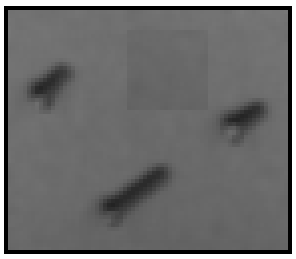


input

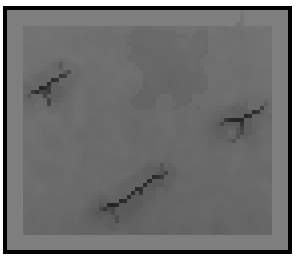


output

Example 2: image name: skelg1.bmp; image size: 70x60.



input



output

GRADIENT CONTROLLED DIFFUSION

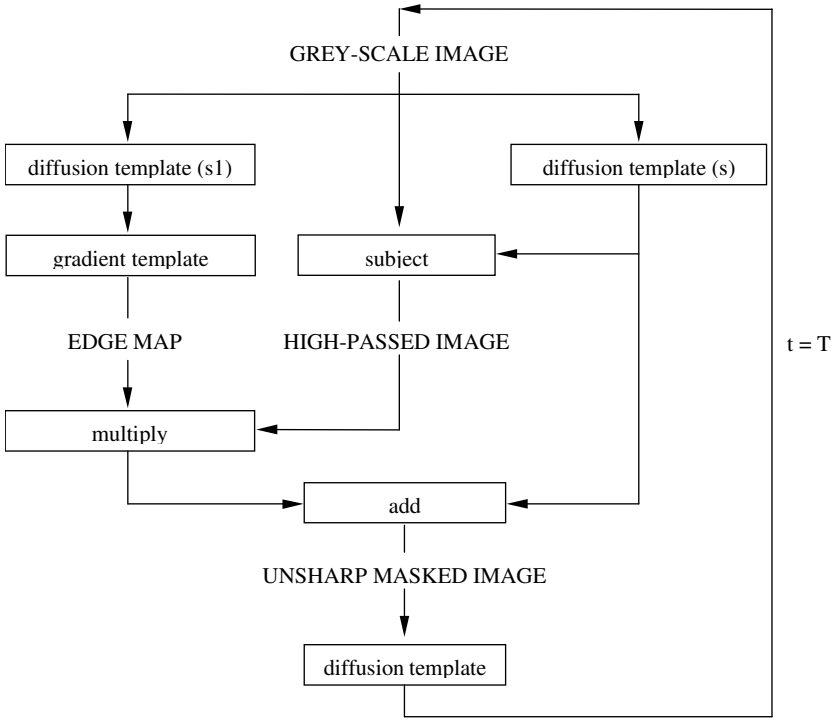
Task description and algorithm

Performs edge-enhancement during noise-elimination [17,25,30]. The equation used for filtering is as follows:

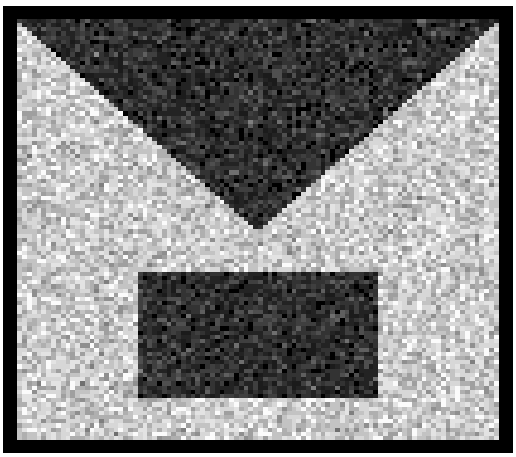
$$\frac{\partial I}{\partial t} = \Delta\left[I(x, y, t) \cdot \left(1 - k \cdot \left|grad\left(G(s) * I(x, y, t)\right)\right|\right)\right]$$

Here $I(x, y, t)$ is the image changing in time, $G(s)$ is the Gaussian filter with aperture s, k is a constant value between 1 and 3. Both the Gaussian filtering and the Laplace operator (Δ) is done by the *HeatDiffusion* (diffusion) template with different diffusion coefficients. The *ThresholdedGradient* (gradient) template can also be found in this library. This equation can be used for noise filtering without decreasing the sharpness of edges.

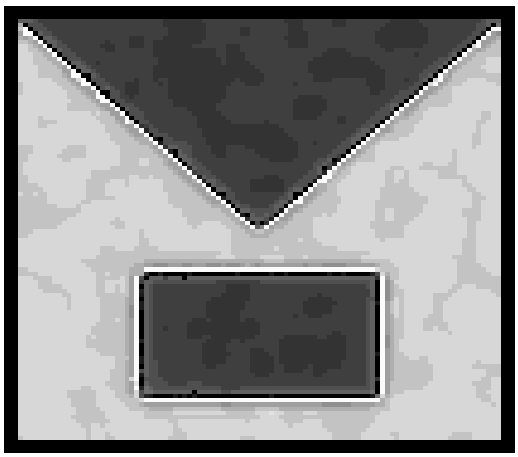
The flow-chart of the algorithm:



Example: image name: laplace.bmp; image size: 100x100.



input



output

SHORTEST PATH

Task description and algorithm

Two points given, the subroutine finds the shortest path connecting them. A labyrinth can also be defined, where the walls (black pixels) denote forbidden cells. The algorithm runs in a time proportional to the length of the shortest path. The algorithm can further be generalized to connect several points. It was also used to design the layout of printed circuit boards.
The subroutine contains two phases. In the first phase, the black-and-white labyrinth should be applied to the input, and one of the two points to the initial state (a white point against a black background). In the second step, when the shortest route has been selected, the output of the first step serves as input, and the other point to be connected is the initial state (black point on a white background). In this step four templates are to be applied cyclically.

The templates of the algorithm:

Explore:

$\mathbf{A}_{explore}$=

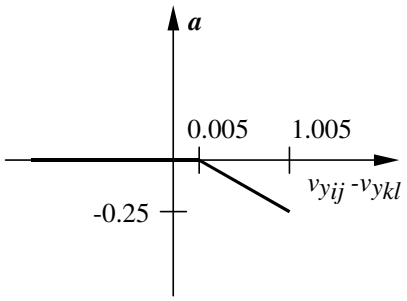
0	***a***	0
a	1	***a***
0	***a***	0

$\mathbf{B}_{explore}$ =

0	0	0
0	3	0
0	0	0

$z_{explore}$ = 3

where ***a*** is defined by the following nonlinear function:



Select:

$\mathbf{A}_{right}$=

0	0	0
1	3	0
0	0	0

$\mathbf{A}_{down}$=

0	1	0
0	3	0
0	0	0

$\mathbf{A}_{left}$ =

0	0	0
0	3	1
0	0	0

$\mathbf{A}_{up}$ =

0	0	0
0	3	0
0	1	0

$\mathbf{B}_{right}$=

0	0	0
b	0	0
0	0	0

$\mathbf{B}_{down}$=

0	***b***	0
0	0	0
0	0	0

$\mathbf{B}_{left}$ =

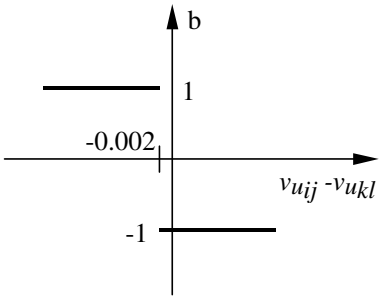
0	0	0
0	0	***b***
0	0	0

$\mathbf{B}_{up}$ =

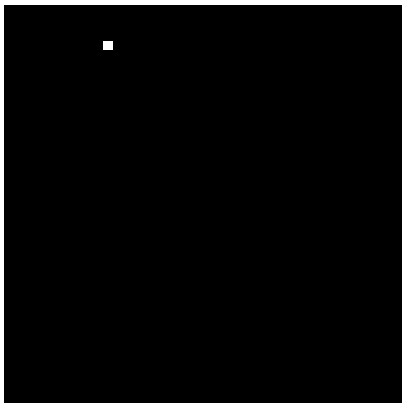
0	0	0
0	0	0
0	***b***	0

z_{right}= 1 z_{down}= 1 z_{left} = 1 z_{up} = 1

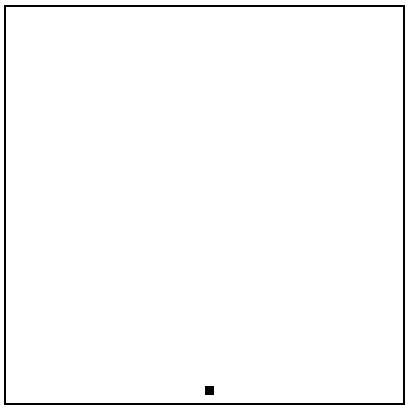
where ***b*** is defined by the following nonlinear function:



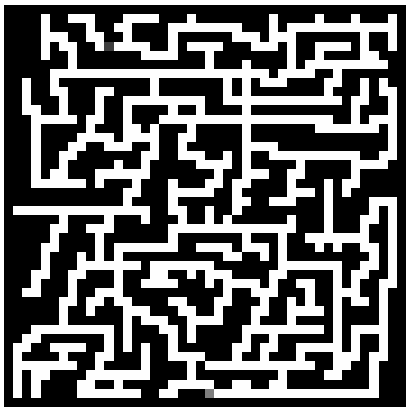
Example : image names: shpath1.bmp, shpath2.bmp, shpath3.bmp; image size: 44x44.



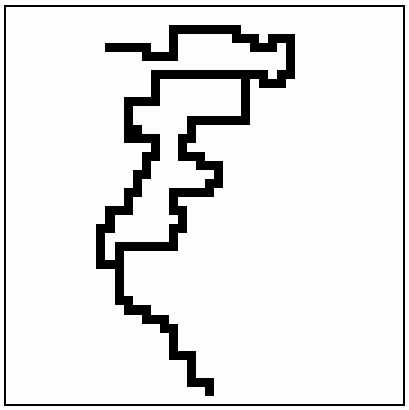
initial state of the 1. step



initial state of the 2. step



labyrinth

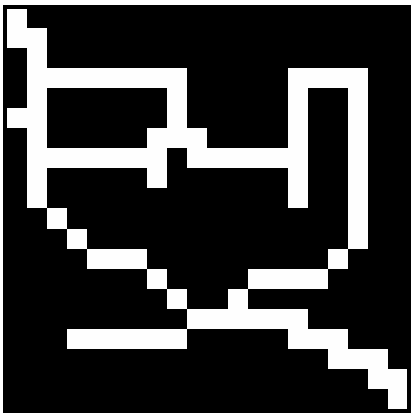


shortest paths

J-FUNCTION OF SHORTEST PATH

Task description and algorithm

A labyrinth is defined as a binary picture, where the white points (-1) represent the free cells, and black pixels (+1) represent the obstacles.



The task is to find the shortest path from a given startpoint to a given endpoint. The task can be solved through a so-called J-function, which gives for every point the length of the shortest path from the startpoint to the given point in some appropriate measure (the length of the path is scaled down into the interval [-1,+1] in the CNN solution).
The minimum in a given neighborhood can be computed through a difference-controlled template. Thus the computation can be carried out through a two layer nonlinear CNN. The first layer represents the J-function, and on the second layer the actual value of the J-function increased by the unit length is computed. The startpoint is given by a binary image, where a white pixel represents the startpoint and the other points are black. The J-function is obtained as an output picture, which values are related to the value of J. Obstacles have J=1, i.e. they are black.

The templates of the algorithm:

Layer1 : Minimum selection:

$\mathbf{A_{11}}=$

0	0	0
0	1	0
0	0	0

$\mathbf{D_{12}}=$

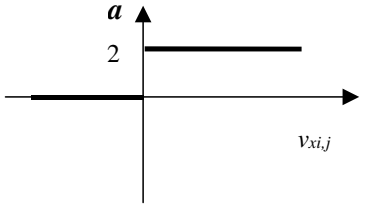
d	***d***	***d***
d	0	***d***
d	***d***	***d***

$\mathbf{B_{11}}=$

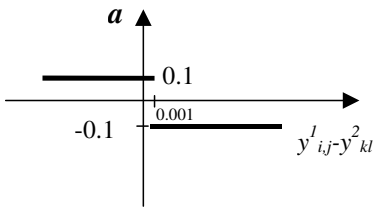
0	0	0
0	***a***	0
0	0	0

$z=$ -0.8

where ***a*** is defined by the following nonlinear function:



and ***d*** is defined by the following function:



Layer2: Increased J-function:

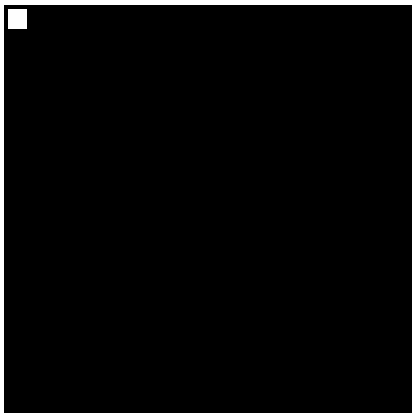
$\mathbf{A_{21}}=$

0	0	0
0	1	0
0	0	0

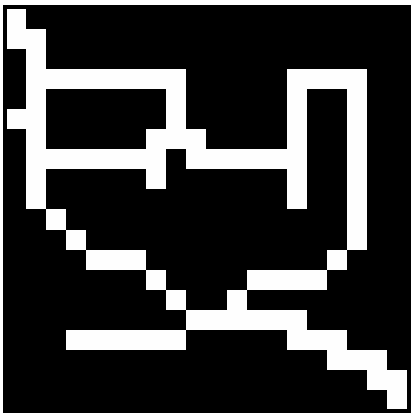
$z =$ | ***c*** |

where *c*=2/maximal path length

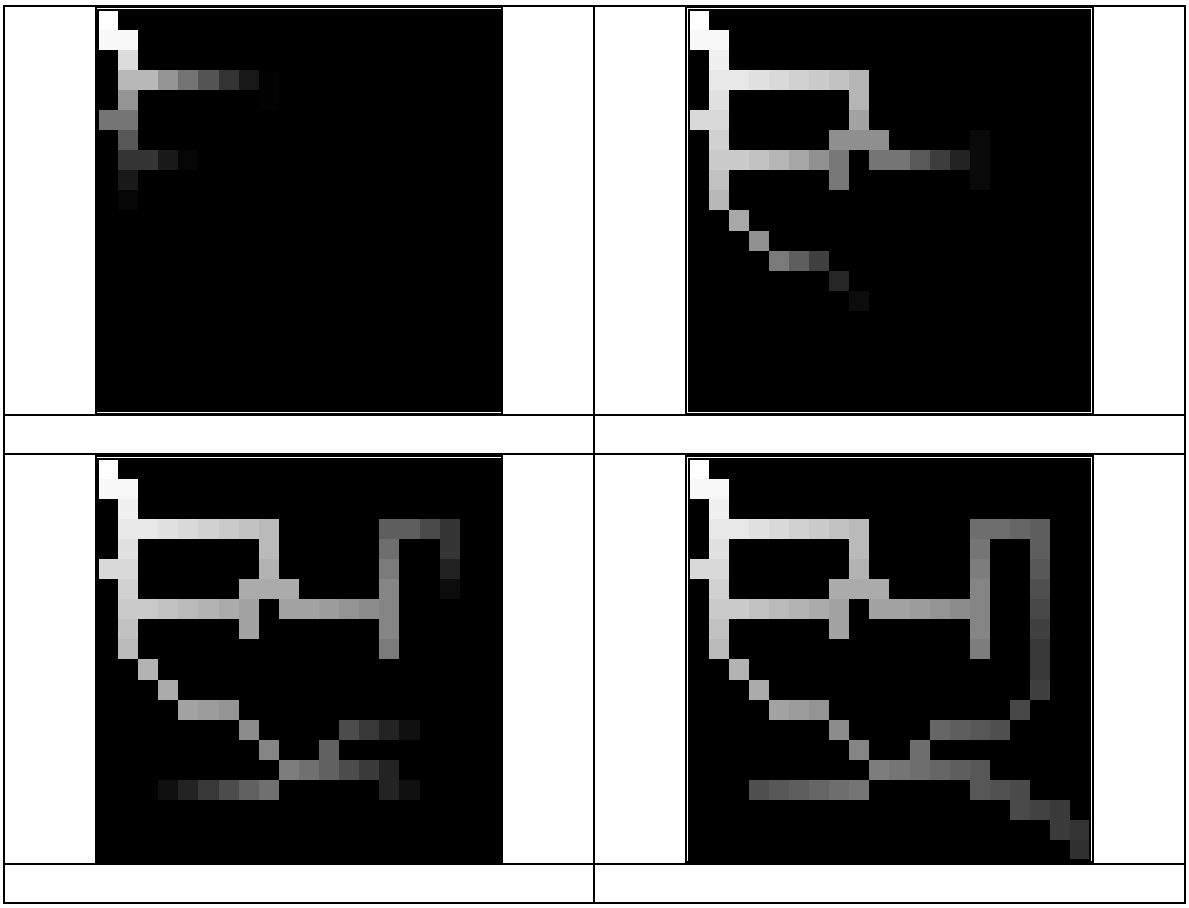
Example: image names: init.bmp, mask.bmp; image sizes: 20x20.



initial state



labirinth

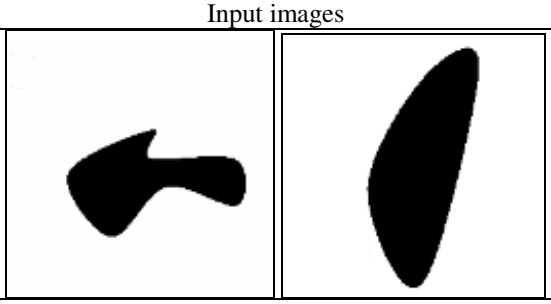
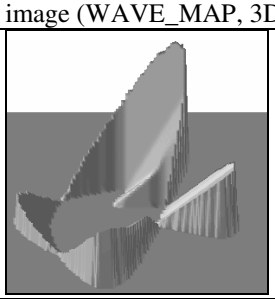


Transient sampled at different time values

NONLINEAR WAVE METRIC COMPUTATION

Task description and algorithm

This algorithm implements a wave metric for object comparison based on nonlinear spatio-temporal wave process. The method uses nonlinear wave propagation to explore the properties of objects. From the intersections of object-pairs wave fronts propagate through the unions of objects and the time evolution of the process is recorded on a special wave map. This wave map contains aggregated information about the dynamics of the wave process; and several different measures can be extracted from this map focusing on the desired aspect of the comparison. The key steps of metric computations are wave based transformation of objects, wave map generation where the associated gray-scale values are related to the time required for the wave to reach a given position, and distance calculation from the wave map. The detailed description of the method can be found in [46]. Two types of implementation are presented, namely, a dynamic solution where wave map is generated on a two-layer CNN architecture in a single transient, using nonlinear template interactions (two-layer UMF machine), and an iterative type of method which was implemented on the 64x64 I/O CNN-UM chip [47]. The main advantage of the method is that several object pairs can be compared to each other at the same time.

Input images	Output image (WAVE_MAP, 3D view)
	

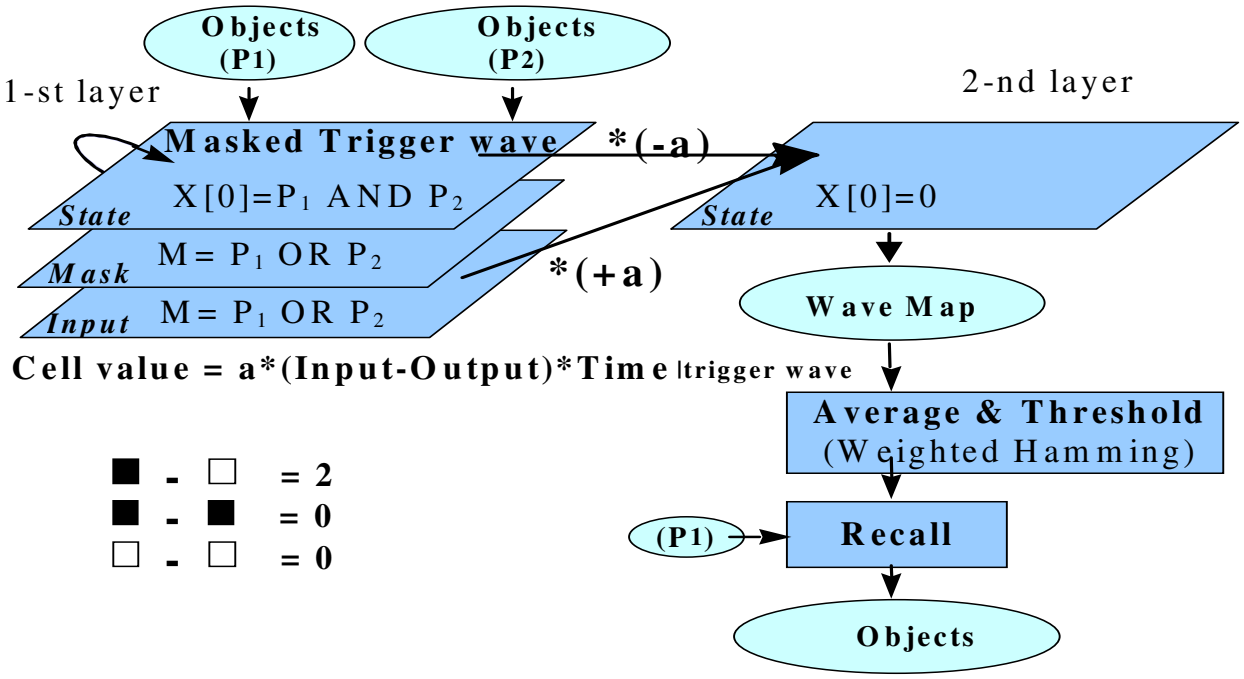
Input Parameters

U_1, U_2	Input binary images (object set A and set B)
B_WAVE	Binary wave propagation
WAVE_MAPPING	Depending on the machine it can be a nonlinear template or a simple linear
WAVE_MAP	Spatial map encoding dynamics of wave propagation
DIFFUS and THRESHOLD	Approximation of Integrated Hausdorff metric [1]

Output Parameters

MARKERS	Result of comparison: marker points of selected objects. These are to be used for further processing. Application example can be found in [2].

Dynamic implementation



Template for trigger wave generation

$\mathbf{A}_{1,1} =$

0.41	0.59	0.41
0.59	2	0.59
0.41	0.59	0.41

$\mathbf{B} =$

0	0	0
0	0	0
0	0	0

$z =$ 4.5

Template for current filling

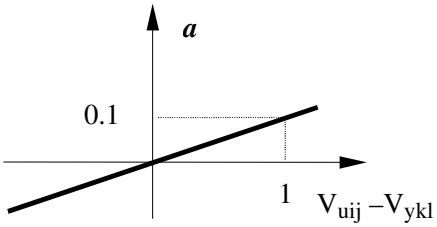
$\mathbf{A}_{1,2} =$

0	0	0
0	*a*	0
0	0	0

$\mathbf{B} =$

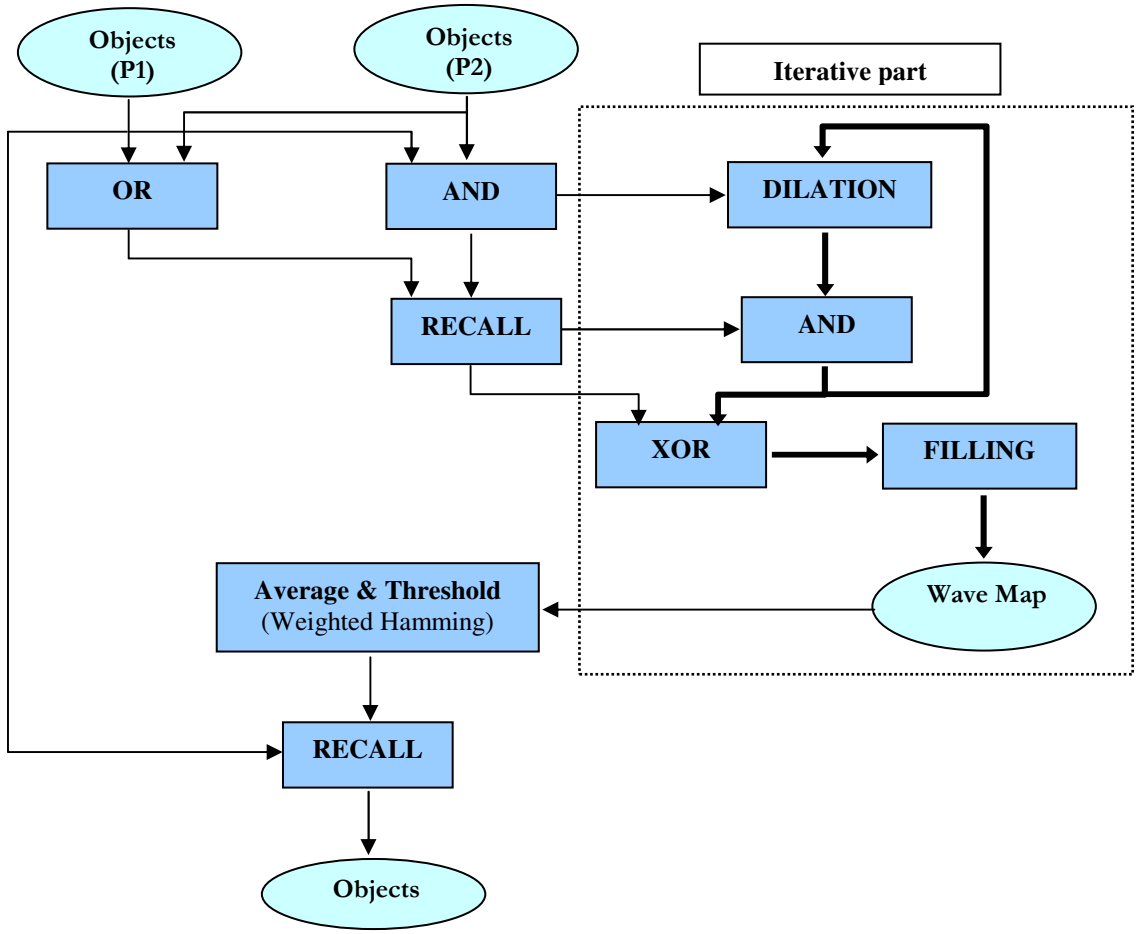
0	0	0
0	0	0
0	0	0

$z =$ 0



Iterative type implementation

The dilation template



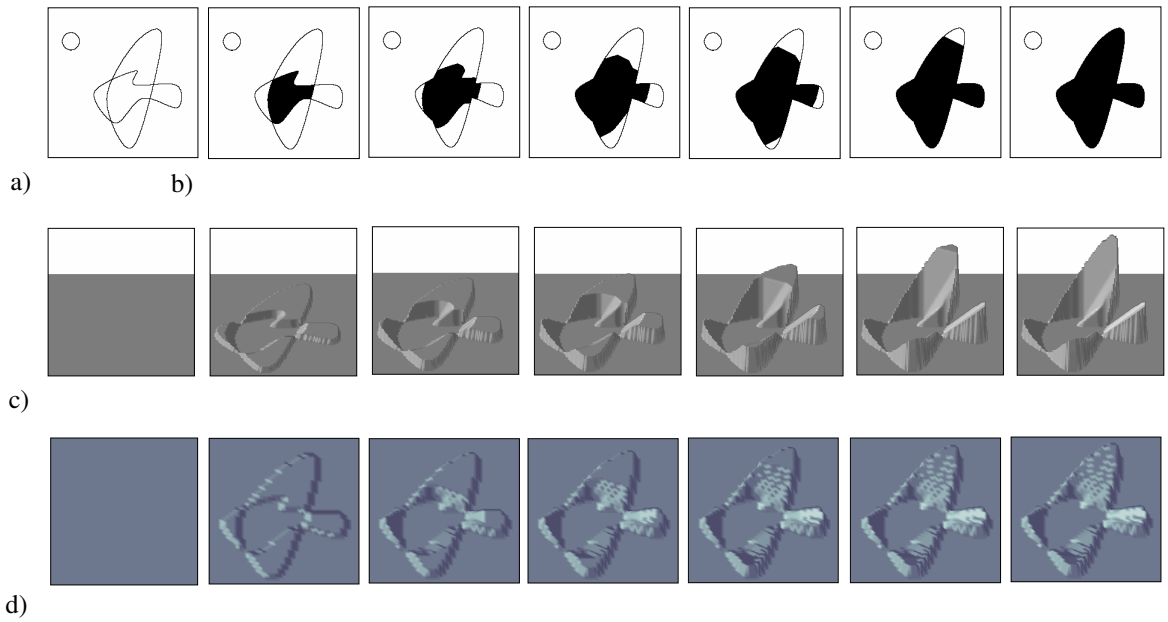
A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

z = 8.5

The other cited templates can be found in this template library.

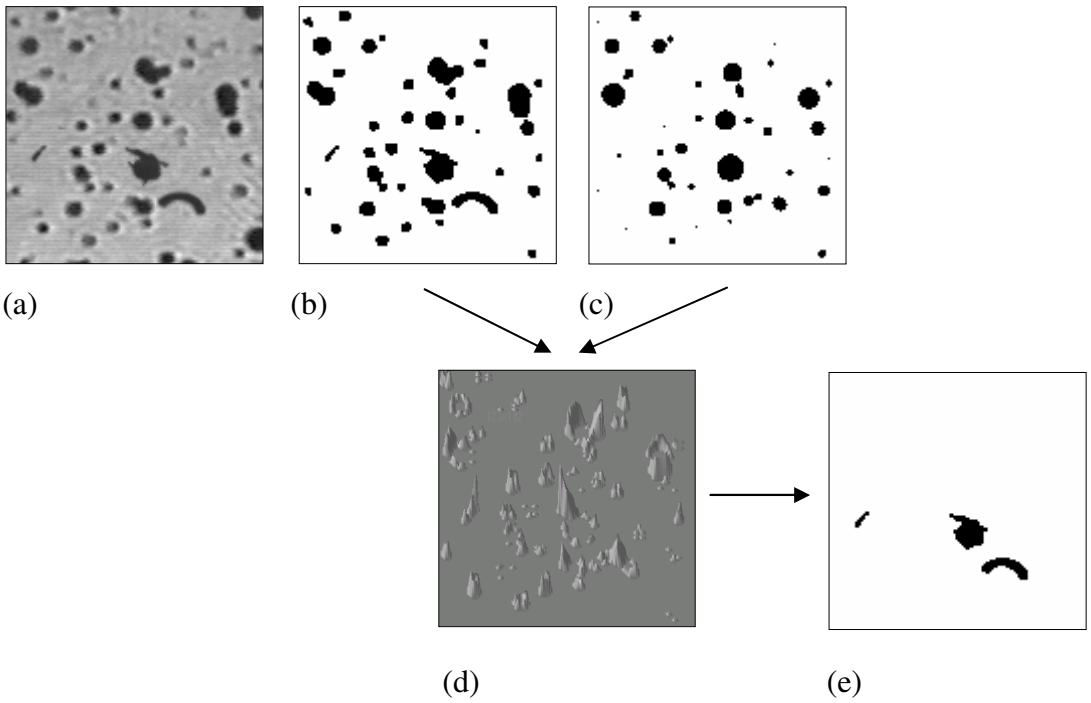
Example – 1



Wave Map generation. a) Outlines of two partially overlapping point sets, b) Trigger wave spreads from the intersection through the union of contiguous parts of point sets until all the points become triggered, c) Wave map generated by increasing intensities of pixels until trigger wave reaches them, simulation result d) Consecutive steps of generating Wave Map on the 64x64 I/O CNN-UM chip.

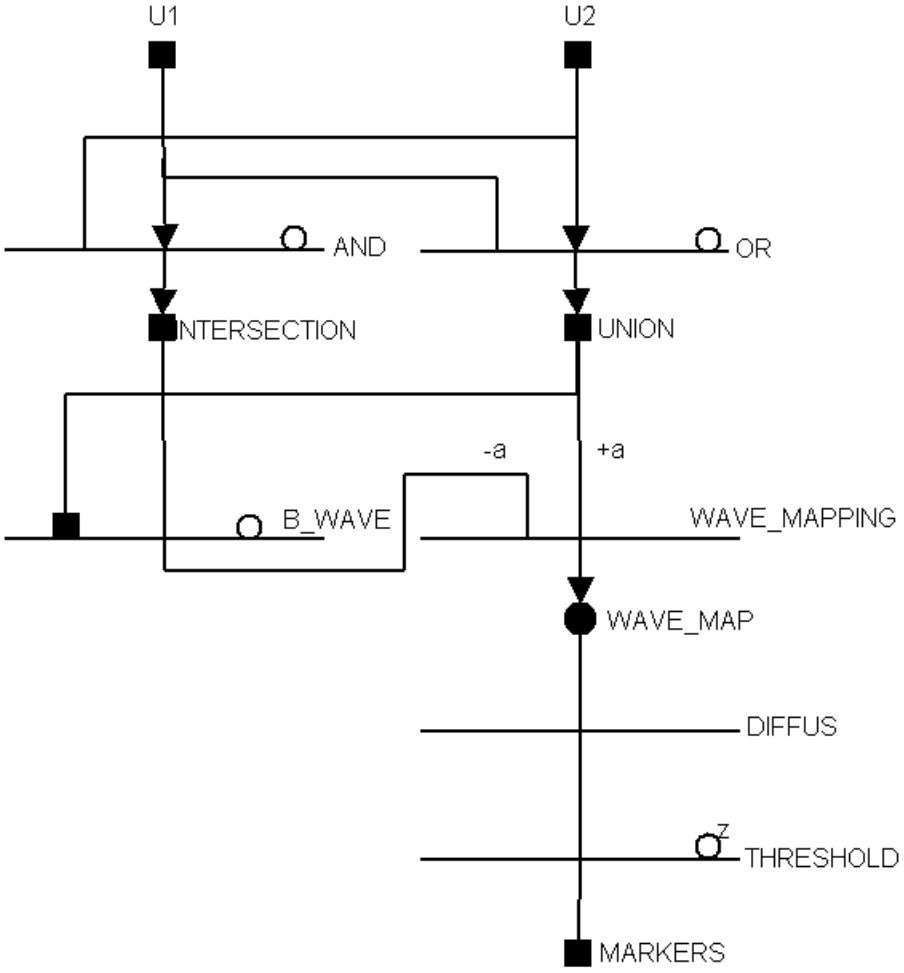
Example – 2

A possible application of the nonlinear wave metric was presented in [48]. The study addressed the problem of conditional basement maintenance of engines, concerning the on-line monitoring system that should forecast engine malfunction. Optical sensing of the oil flow of the engine was applied and debris particles were detected by using model-based object recognition. The comparison of object-model pairs was performed via nonlinear wave metric.



Bubble-debris classification algorithm (a) original gray-scale image, (b) adaptive threshold, (c) bubble models, (d) wave map, (e) detected debris particles

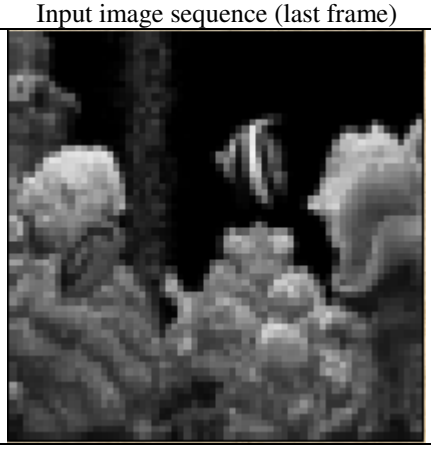
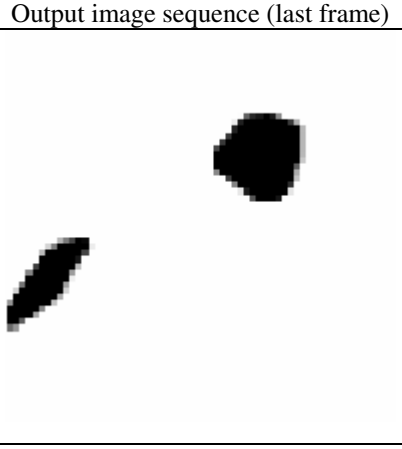
UMF diagram


U1
U2
AND
OR
INTERSECTION
UNION
-a
+a
B_WAVE
WAVE_MAPPING
WAVE_MAP
DIFFUS
Z
THRESHOLD
MARKERS

MULTIPLE TARGET TRACKING

Task description and algorithm

A simple thresholded-difference method is required to maintain both hardware and processing time complexity low. The history of detected targets is used to assure the quality of tracking until target is disappeared from the scene [63].

Input image sequence (last frame)	Output image sequence (last frame)
	

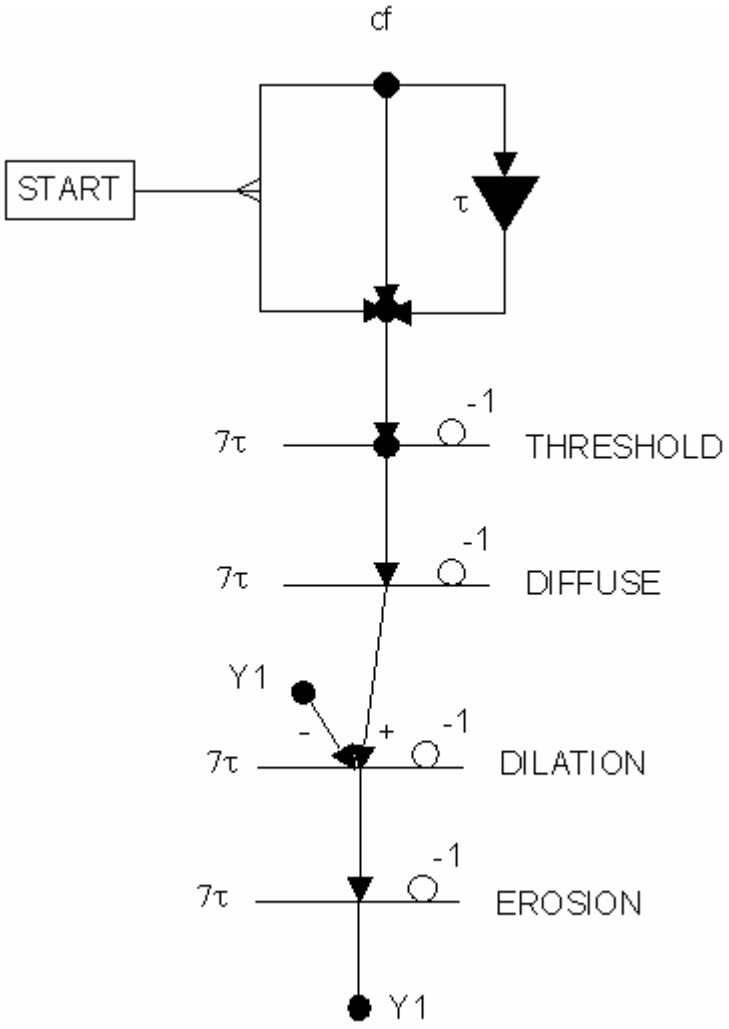
Input Parameters

cf	Current frame (input image)

Output Parameters

Y1	Detected targets in the present frame (output image)

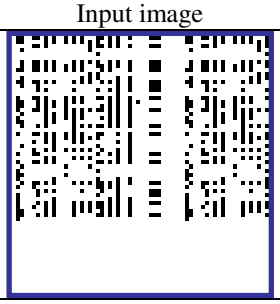
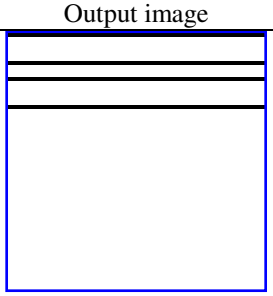
UMF diagram


cf
START
τ
-1
7τ
THRESHOLD
-1
7τ
DIFFUSE
Y1
-
+
-1
7τ
DILATION
-1
7τ
EROSION
Y1

MAXIMUM ROW(S) SELECTION

Task description and algorithm

This algorithm detects rows where the number of black pixels is the greatest [64]. Two different implementations are presented: a longer one with simple linear templates and binary storage (A), and a shorter one with non-linear templates (B).

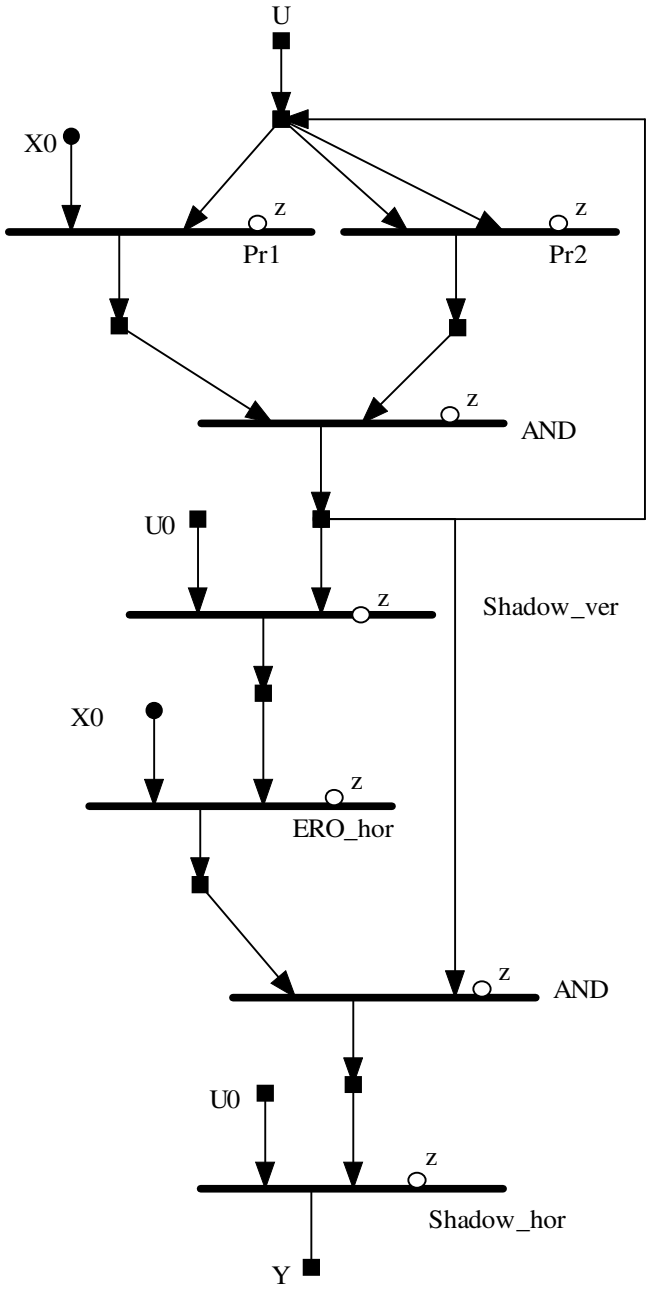
Input image	Output image
	

Input Parameters: –

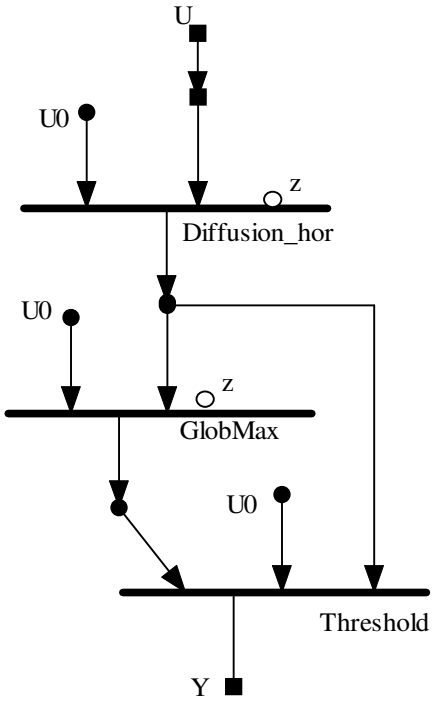
Remarks:

- The hardware complexity (chip area) of the (A) solution is much lower than the (B) solution
- The theoretical scalability in space (increasing the array size) is better in the (B) solution
- The practical scalability in space (tiling the input) is better in the (A) solution
- The computational time (speed and power consumption) depends on the available boundary conditions

UMF diagram



A

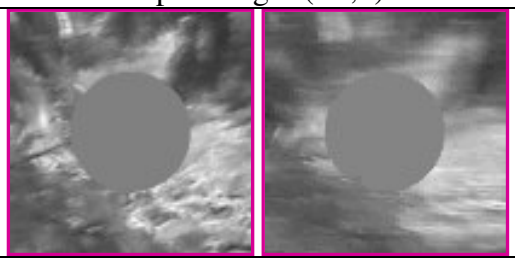


B

SUDDEN ABRUPT CHANGE DETECTION

Task description and algorithm

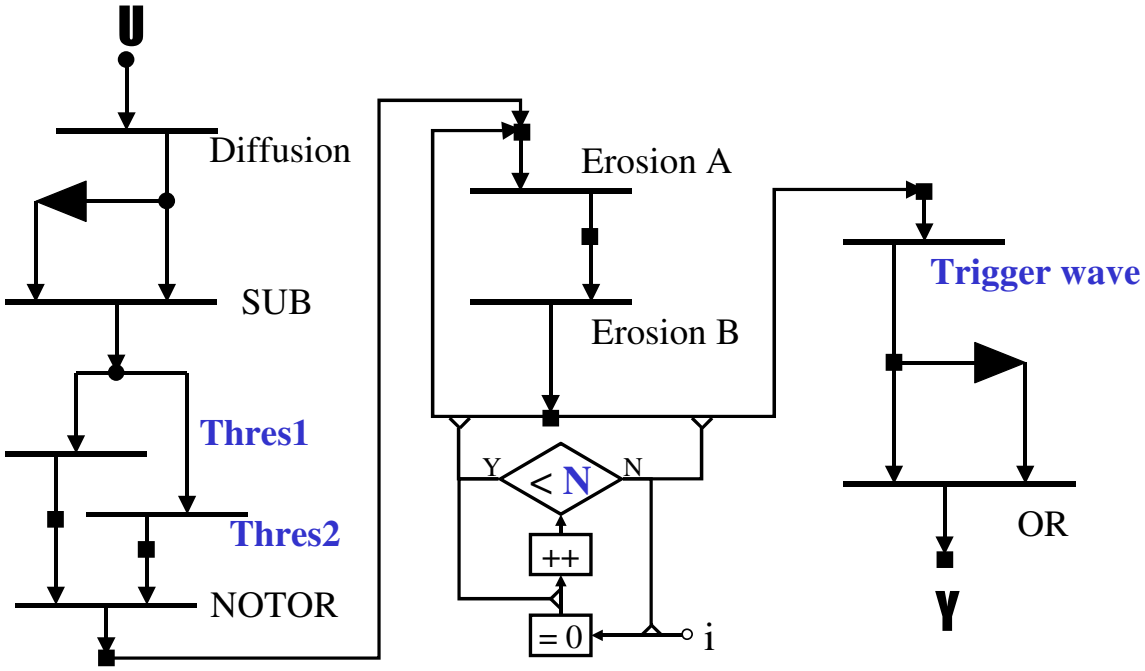
This algorithm detects the places and time instances in a video flow where a sudden abrupt change is happened based on neurobiological measurements [65]. The generated map can be used to re-initialize the history-based processing in a general image flow processing application [66].

Input images (t-1, t)	Output image (t)
	

Input Parameters

U	Input grayscale image flow
Thres	Threshold level
N	Number of the erosion steps
Trigger wave	Expansion size ~ trigger wave running time

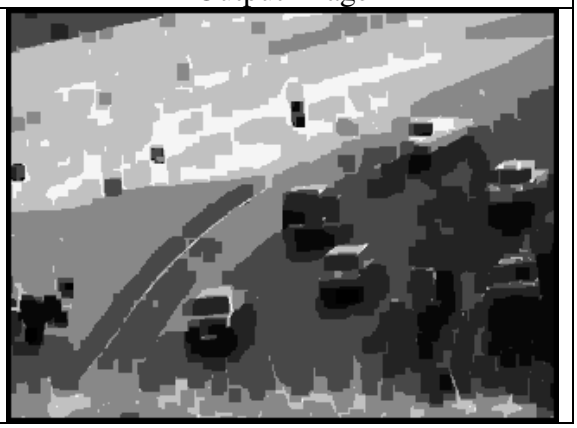
UMF diagram


Diffusion
SUB
Thres1
Thres2
NOTOR
Erosion A
Erosion B
Y
< N
N
++
= 0
i
Trigger wave
OR

HISTOGRAM MODIFICATION WITH EMBEDDED MORPHOLOGICAL PROCESSING OF THE LEVEL-SETS

Task description and algorithm

This algorithm is designed for simultaneous contrast enhancement, noise suppression and shape enhancement [67]. An adaptive multi-thresholded output with shape enhancement can be obtained if a morphological processing is embedded at each gray-scale level considered. This could be implemented either through a multi-step erosion and dilation operations or using trigger-waves that approximate a continuous-scale binary morphology with flat structuring elements

Input image	Output image
	

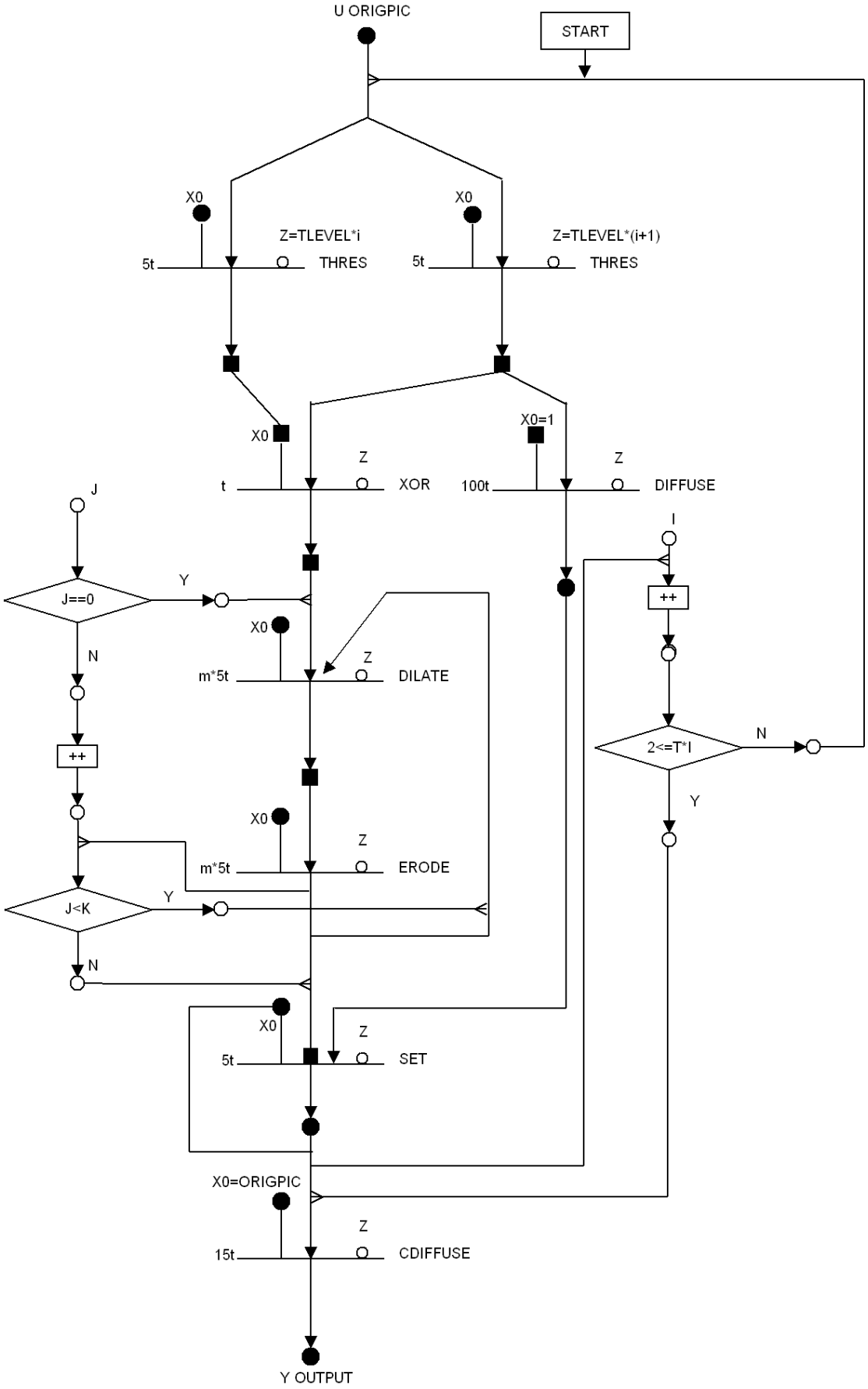
Input Parameters

U	Input image
I,J	Indexes
K	Number of morphological steps
M	(Def = 1) Heuristic value
T(LEVEL)	Number of levels

Output Parameters

Y	Output image

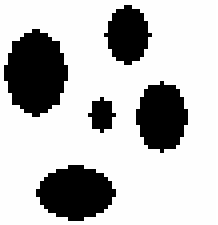
UMF diagram


U ORIGPIC
START
X0
Z=TLEVEL*i
5t
THRES
X0
Z=TLEVEL*(i+1)
5t
THRES
X0
Z
t
XOR
X0=1
Z
100t
DIFFUSE
J
J==0
Y
N
X0
Z
m*5t
DILATE
++
X0
Z
m*5t
ERODE
J<K
Y
N
I
++
2<=T*I
N
Y
X0
Z
5t
SET
X0=ORIGPIC
Z
15t
CDIFFUSE
Y OUTPUT

OBJECT COUNTER

Task description and algorithm

This algorithm counts the distinct objects in a binary image. The routine is a member of the iterative structure family. The algorithm iteratively selects and removes one single object at a time from a set of objects in a picture until no black pixel is present. One counts the number of the necessary iterations can tell the number of the objects in the original input picture.

Input image	Output (scalar)
	5 objects

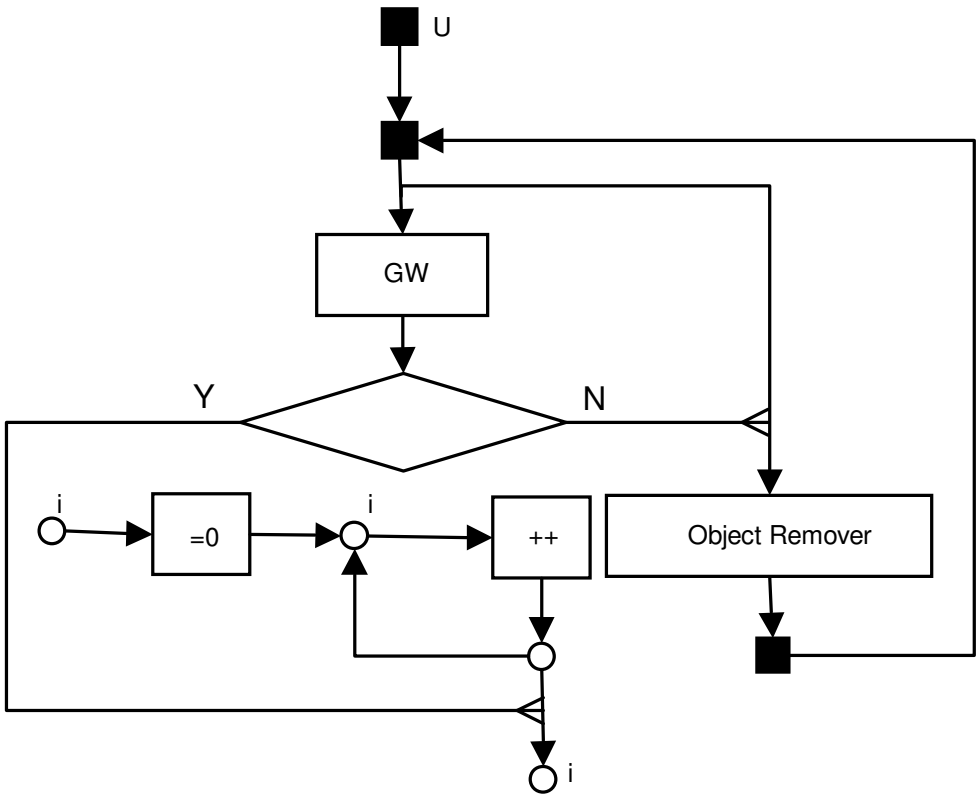
Input Parameters

U	Input image

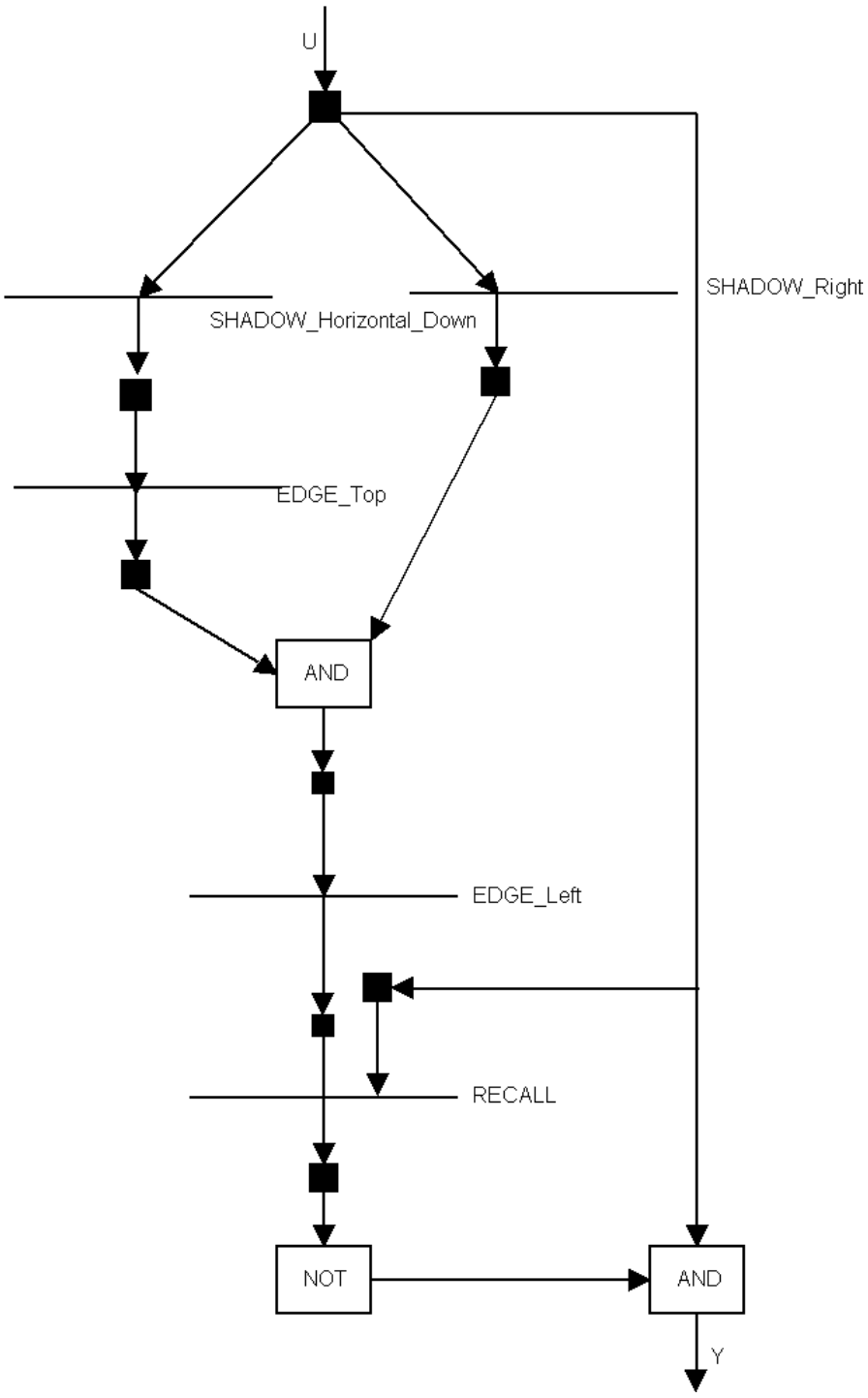
Output Parameters

I	Number of detected objects

UMF diagram



Subroutine **Object Remover**



HOLE DETECTION IN HANDWRITTEN WORD IMAGES

Task description and algorithm

This algorithm [68] detects the holes in a handwritten word image and classifies them into four classes that are used as features in handwriting recognition:

- holes (holes like the one in letter ‘o’)
- small holes (holes like the one in letter ‘e’)
- upper holes (holes in the ascender region)
- lower holes (holes in the descender region)

Input image	Output image
	

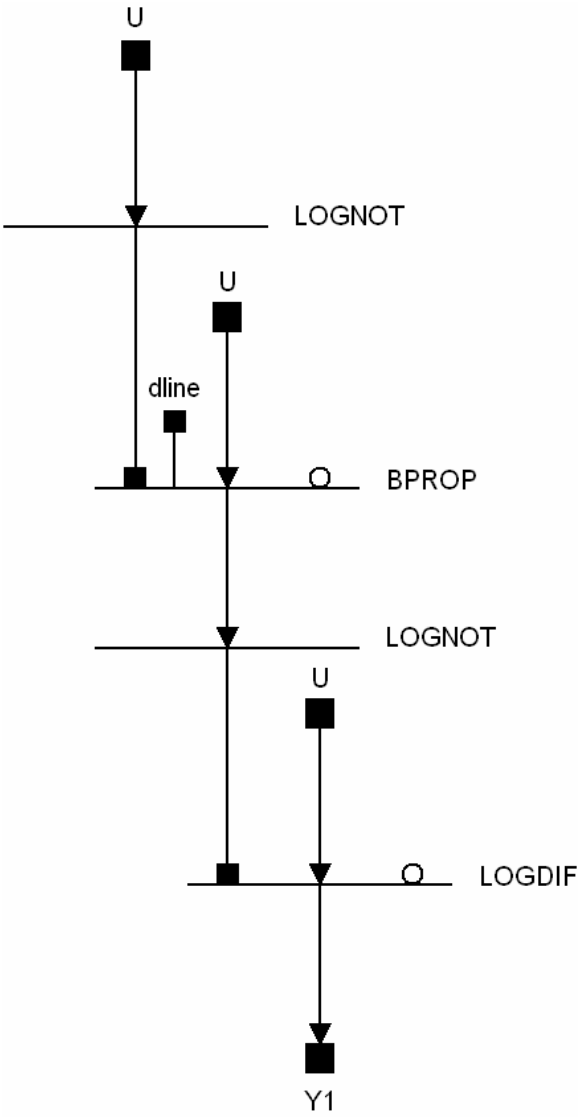
Input Parameters

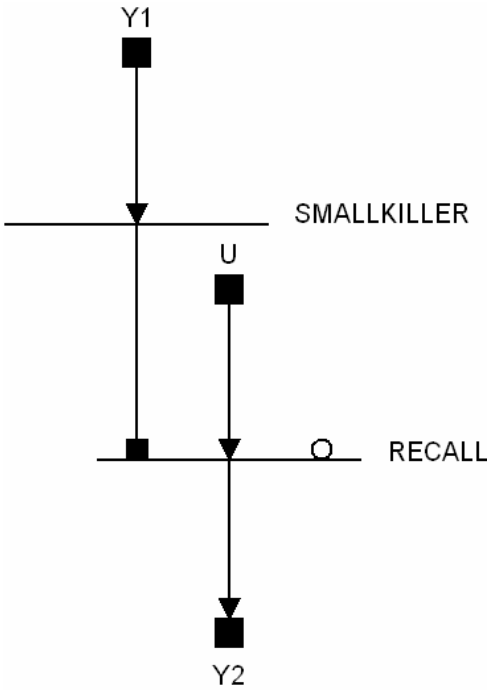
U	Input image
uline / dline	Upper / lower baselines

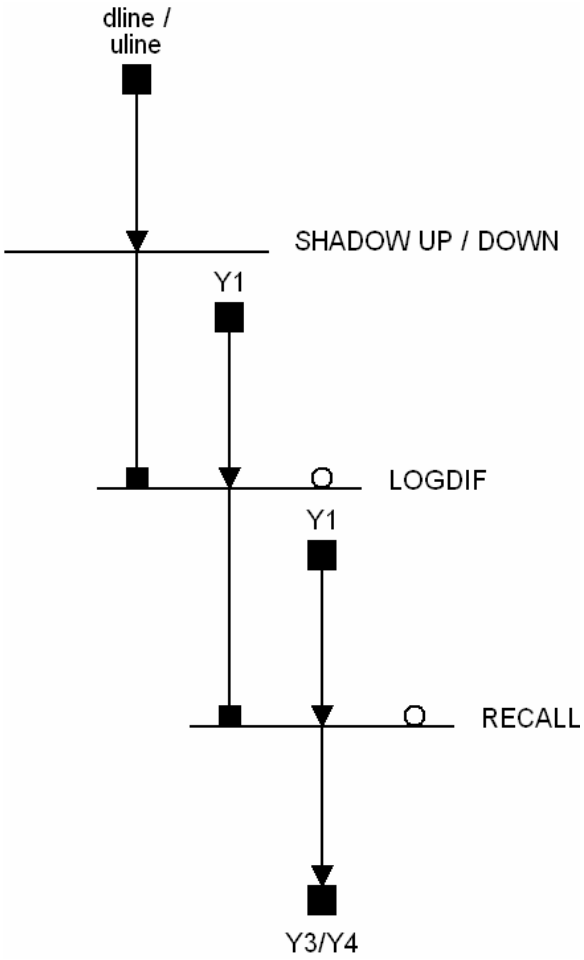
Output Parameters

Y1	Holes detected
Y2	Small holes
Y3 / Y4	Upper / lower holes

UMF diagram


U
LOGNOT
U
dline
BPROP
LOGNOT
U
LOGDIF
Y1


Y1
SMALLKILLER
U
RECALL
Y2


dline /
uline
SHADOW UP / DOWN
Y1
LOGDIF
Y1
RECALL
Y3/Y4

AXIS OF SYMMETRY DETECTION ON FACE IMAGES

Task description and algorithm

This algorithm detects axis of symmetry on face images [69]. For estimating the axis of symmetry we take an assumption that the region of nose on a face is the most vertically detailed region of face

Input image	Output image
	

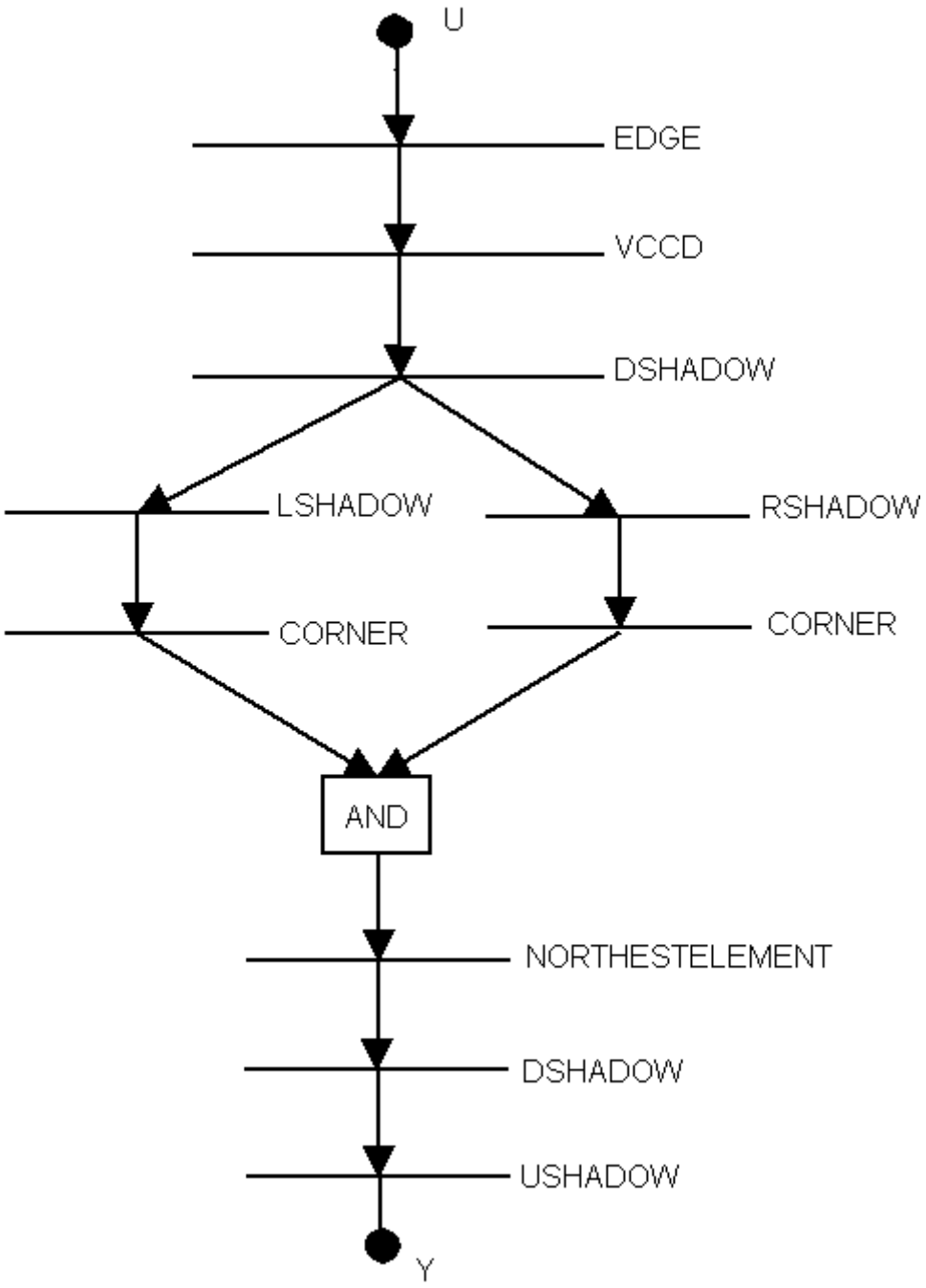
Input Parameters

U	Face image

Output Parameters

Y	Axis of symmetry detected

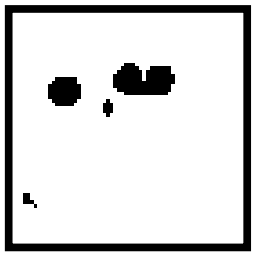
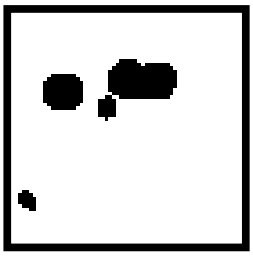
UMF diagram


U
EDGE
VCCD
DSHADOW
LSHADOW
RSHADOW
CORNER
CORNER
AND
NORTHESTELEMENT
DSHADOW
USHADOW
Y

ISOTROPIC SPATIO-TEMPORAL PREDICTION CALCULATION BASED ON PREVIOUS DETECTION RESULTS

Task description and algorithm

This algorithm calculates the likely position of moving targets based on their current detected position [70].

Input image	Output image
	

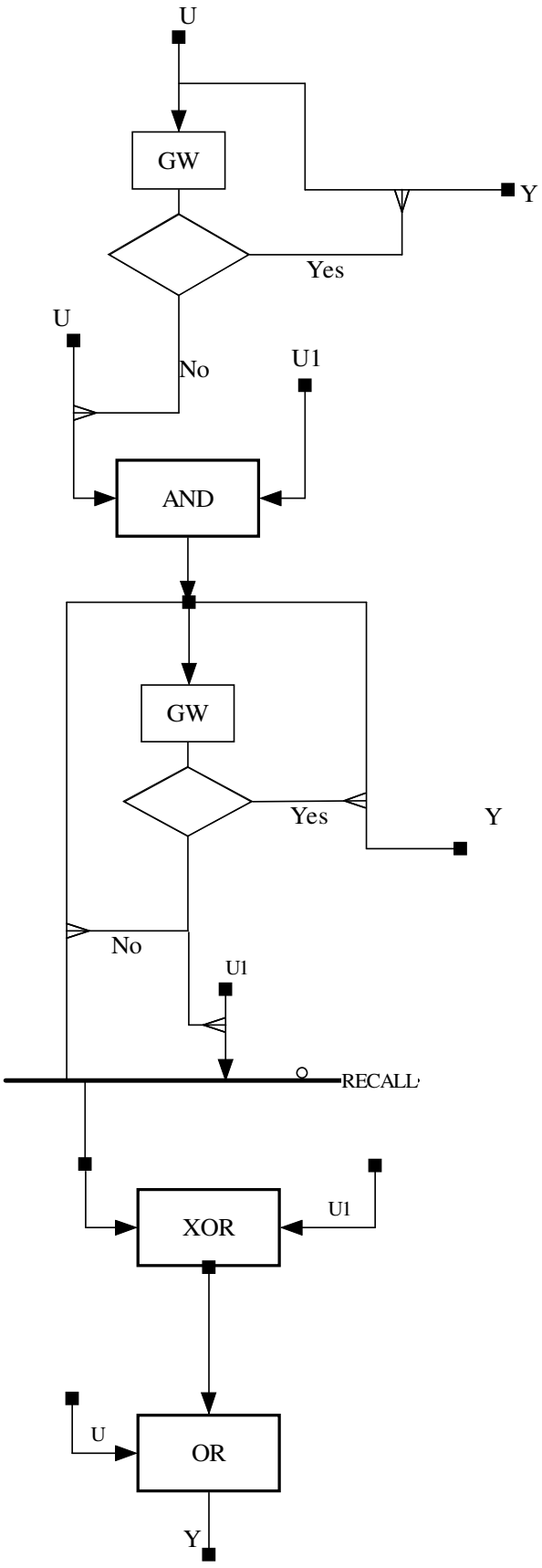
Input Parameters

U	Input image
U1	Previous prediction image

Output Parameters

Y	Current prediction

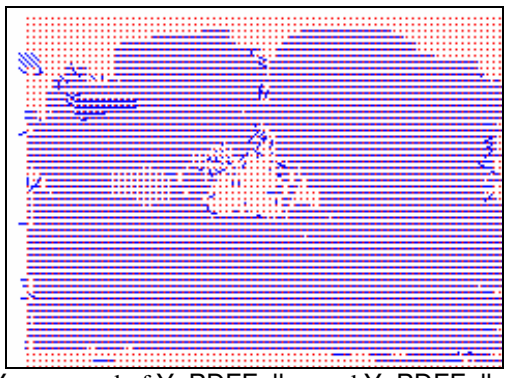
UMF diagram



MULTI SCALE OPTICAL FLOW

Task description and algorithm

This algorithm selects the most likely motion vectors in each pixel based upon two subsequent frames of a video sequence [71].
The algorithm consists of three recursively embedded steps operating on the current and previous frames of the sequence. The outer step is responsible for cycling on scales, which factorizes the "scale" from "lscale_low" up to the limit in each cycle. This scalar value is provided for the Probability Distribution Function (PDF) generator (PDFFull(scale)) which in turn creates 2Nx2N images where N is the maximal displacement of frames to be detected, measured in pixels. The schema depicts the case in which *N=1*. Finally, the most embedded subroutine (PDF(scale)) is responsible for generating a single PDF in a single direction. In the operation, it realizes L_1 norm between its two inputs, diffuses the result and finally puts in a *exp(-x)* function pixel-wise.

Input images	Output image
 Ucurr  Uprev	 Y composed of Y_PDFFull_x and Y_PDFFull_x

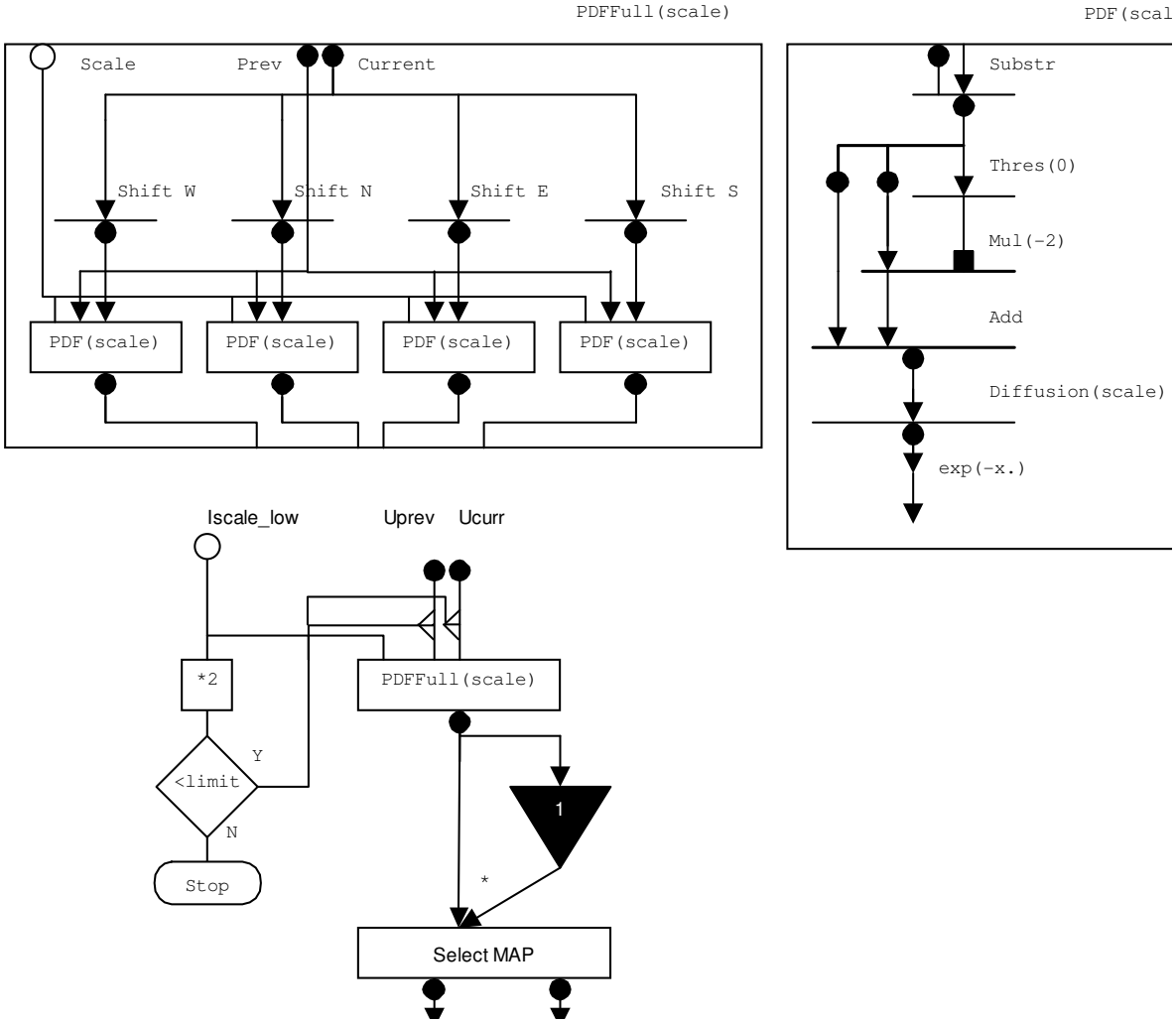
Input Parameters

U	Input image
Ucurr	The second frame of the pair
Uprev	The first frame of the pair

Output Parameters

Y	Output image
Y_PDFFull_x	X coordinate of the most likely motion vectors
Y_PDFFull_y	Y coordinate of the most likely motion vectors

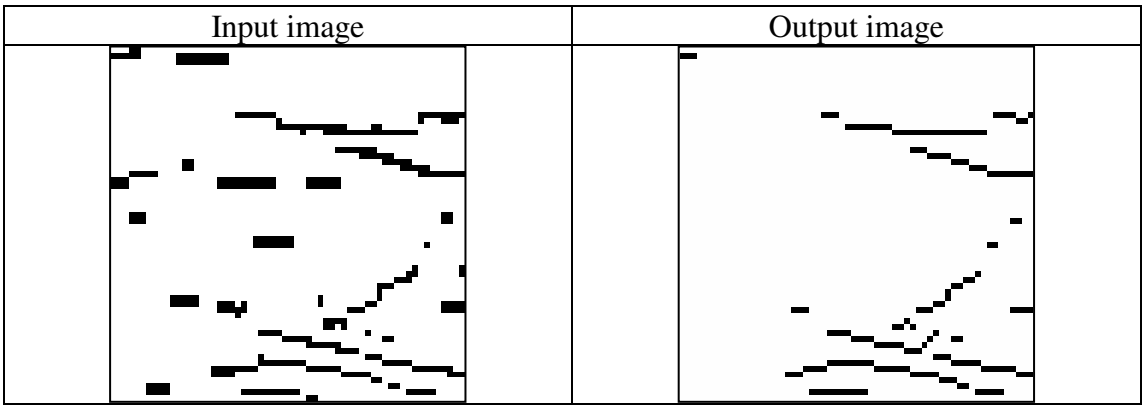
UMF diagram


PDFFull(scale)
Scale
Prev
Current
Shift W
Shift N
Shift E
Shift S
PDF(scale)
PDF(scale)
PDF(scale)
PDF(scale)
PDF(scal
Substr
Thres(0)
Mul(-2)
Add
Diffusion(scale)
exp(-x.)
Iscale_low
Uprev
Ucurr
*2
PDFFull(scale)
Y
<limit
N
Stop
1
*
Select MAP

BROKEN LINE CONNECTOR

Task description and algorithm

This algorithm fill gaps between neighboring object which onsets or offsets is closer than a given threshold [72]. The size of filled gaps can be controlled through the iteration of dilation template operation.



Input Parameters

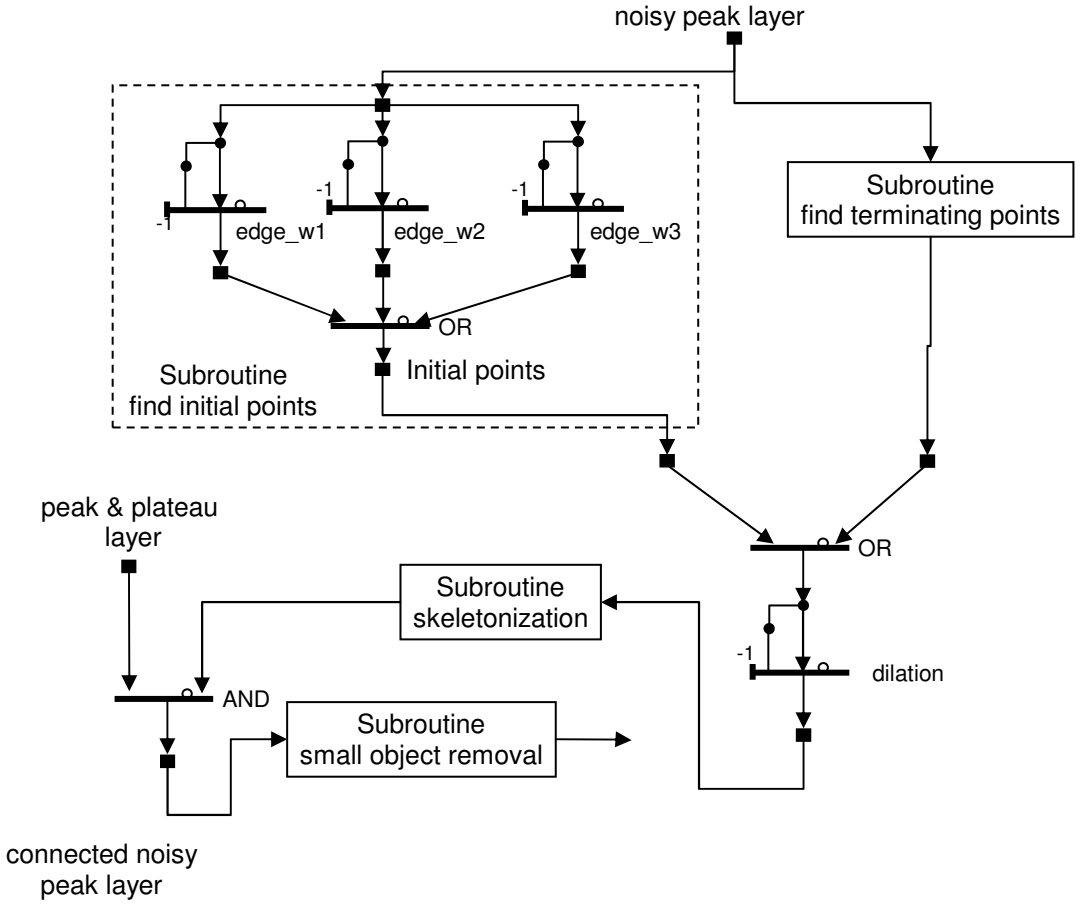
noisy peak layer	Input image

Output Parameters

Connected peak layer	Connected objects

Remarks

UMF diagram

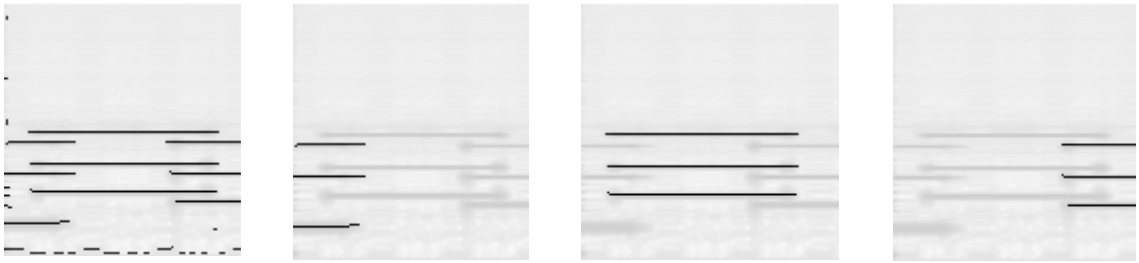

noisy peak layer
Subroutine
find terminating points
-1
-1
-1
edge_w1
edge_w2
edge_w3
OR
Initial points
Subroutine
find initial points
peak & plateau
layer
OR
Subroutine
skeletonization
-1
dilation
AND
Subroutine
small object removal
connected noisy
peak layer

COMMON AM

Task description and algorithm

The algorithm extracts frequency bands to a distinct layer which onsets and offsets are synchronized [72]. This algorithm produces similar groups like the common amplitude-modulation group that was observed by psychoacoustic experiments of the human hearing.

Input image Y1 Y20 Y 87



Input Parameters

peak layer	Layer containing the representative frequency bands (peak layer)

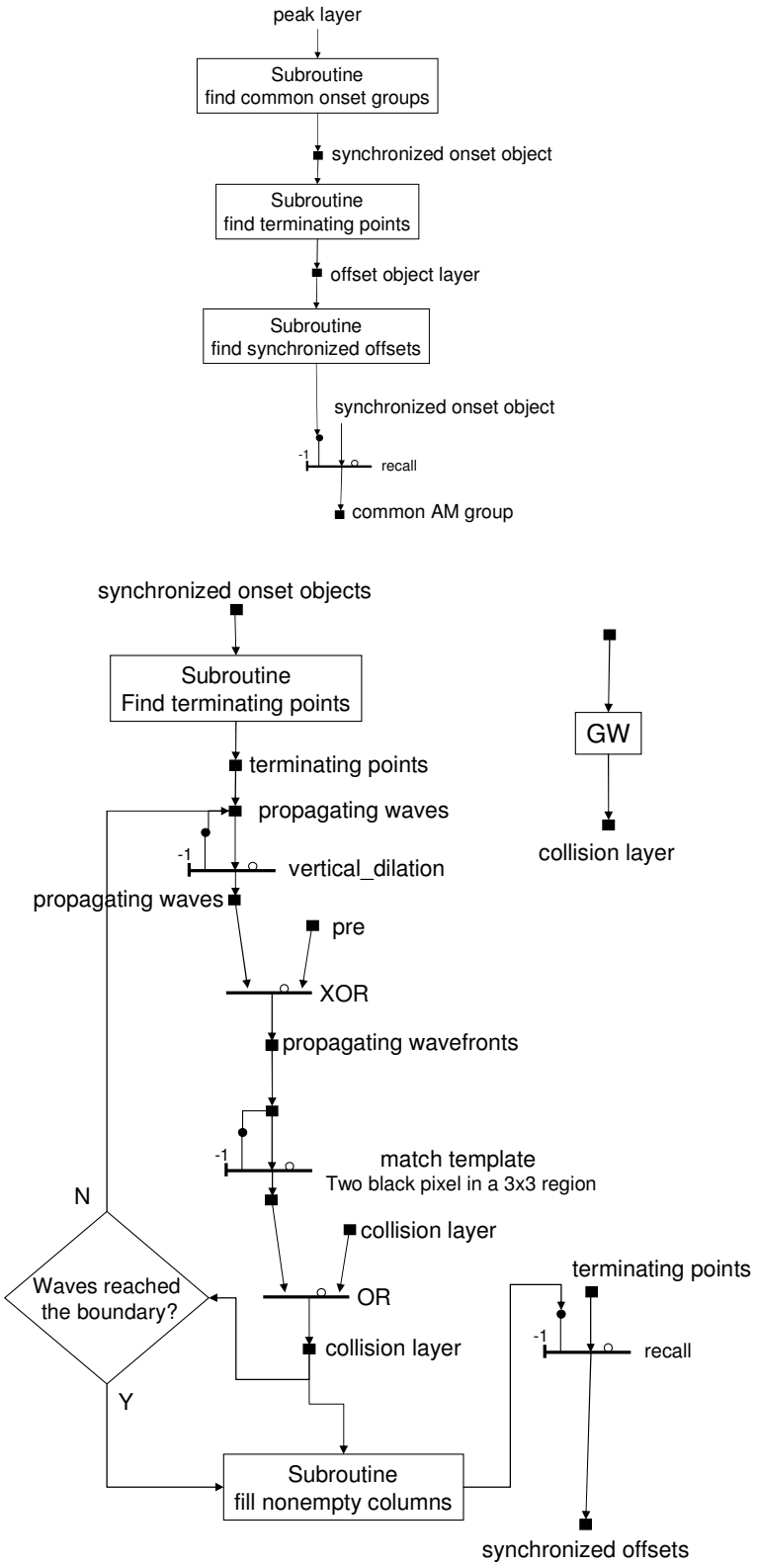
Output Parameters

common AM layer	Bands with synchronized onset and offset from

Remarks

The time tolerance of offsets can be controlled through the iteration number of the vertical wave propagation step producing the *propagating waves* layer.

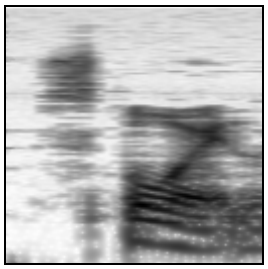
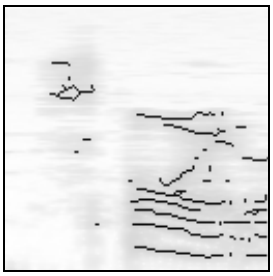
UMF diagram


peak layer
Subroutine
find common onset groups
synchronized onset object
Subroutine
find terminating points
offset object layer
Subroutine
find synchronized offsets
synchronized onset object
-1
recall
common AM group
synchronized onset objects
Subroutine
Find terminating points
terminating points
propagating waves
-1
vertical_dilation
propagating waves
pre
XOR
propagating wavefronts
-1
match template
Two black pixel in a 3x3 region
collision layer
N
Waves reached
the boundary?
OR
collision layer
Y
Subroutine
fill nonempty columns
GW
collision layer
terminating points
-1
recall
synchronized offsets

FIND OF COMMON FM GROUP

Task description and algorithm

Psychoacoustic experiments shows that human hearing system handles object with common frequency modulation as coming from the same source. This algorithm finds objects that is modulated by the same frequency i.e. their vertical distance is constant [72].

Original image	Representative frequency bands (input image)
	
Reference curve	Common FM group
	

Input Parameters

peak layer	Input image with lines in which some of them vertical distance is constant. (parallel curve)

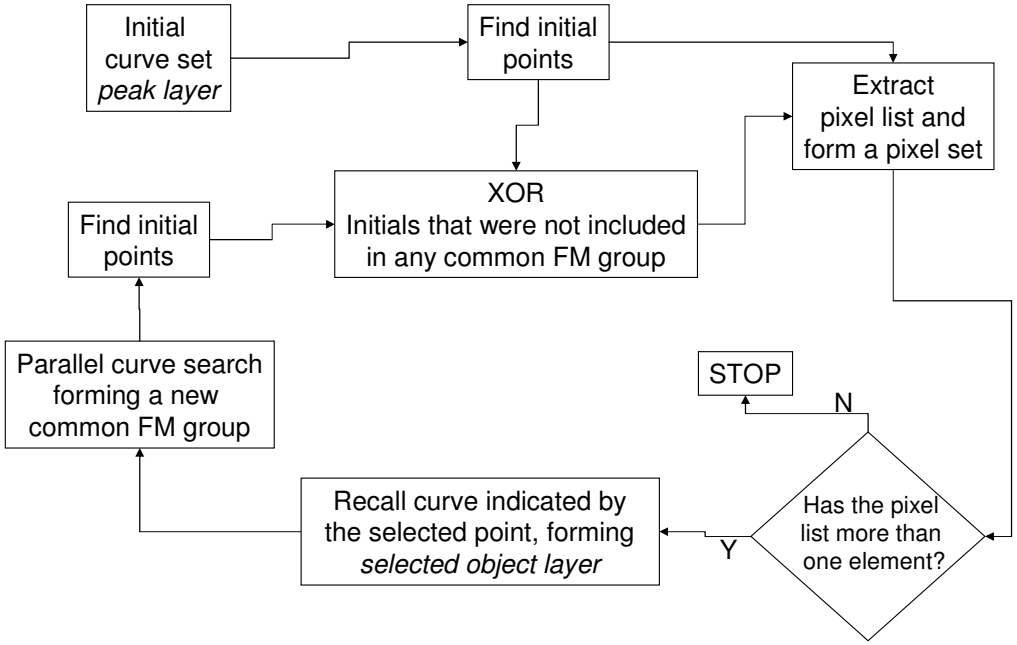
Output Parameters

common FM group	Curves parallel to a given reference

Remarks

The registration of curves is processed by a digital computer. The digital computer selects the reference curves which initials are still in the maintained pixel list.

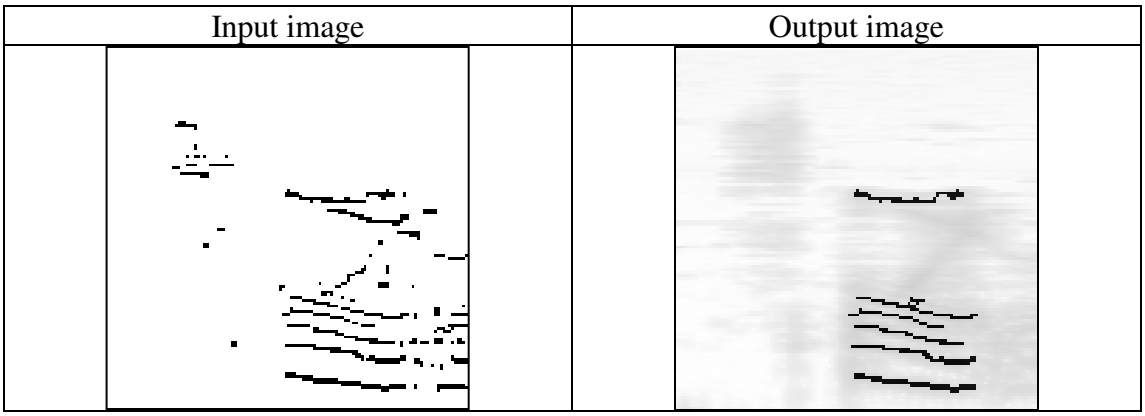
UMF diagram


Initial
curve set
peak layer
Find initial
points
Extract
pixel list and
form a pixel set
XOR
Initials that were not included
in any common FM group
Find initial
points
Parallel curve search
forming a new
common FM group
STOP
N
Has the pixel
list more than
one element?
Y
Recall curve indicated by
the selected point, forming
selected object layer

FIND COMMON ONSET/OFFSET GROUPS

Task description and algorithm

This algorithm produces series of layers containing objects with synchronized onset [72].



Input Parameters

peak layer	Input image

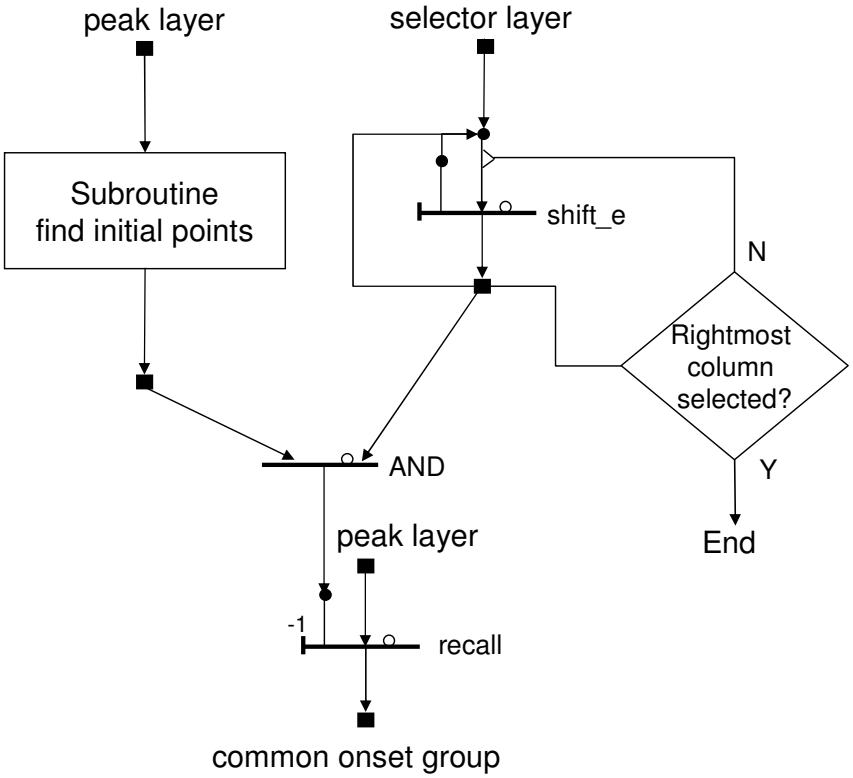
Output Parameters

common onset group	Object with synchronized onset

Remarks

Tolerance of time asynchrony is set by the width of vertical line shifted on the *selector layer*.

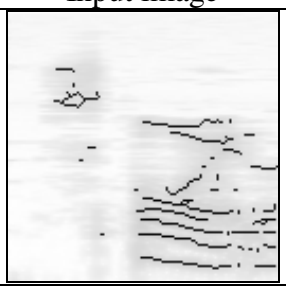
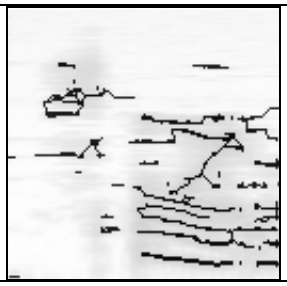
UMF diagram


peak layer
selector layer
Subroutine
find initial points
shift_e
N
Rightmost
column
selected?
Y
End
AND
peak layer
-1
recall
common onset group

CONTINUITY

Task description and algorithm

This algorithm connects object whose ending and initial is closer than a given threshold [72].

Input image	Connected curves
	

Input Parameters

peak layer	Input image

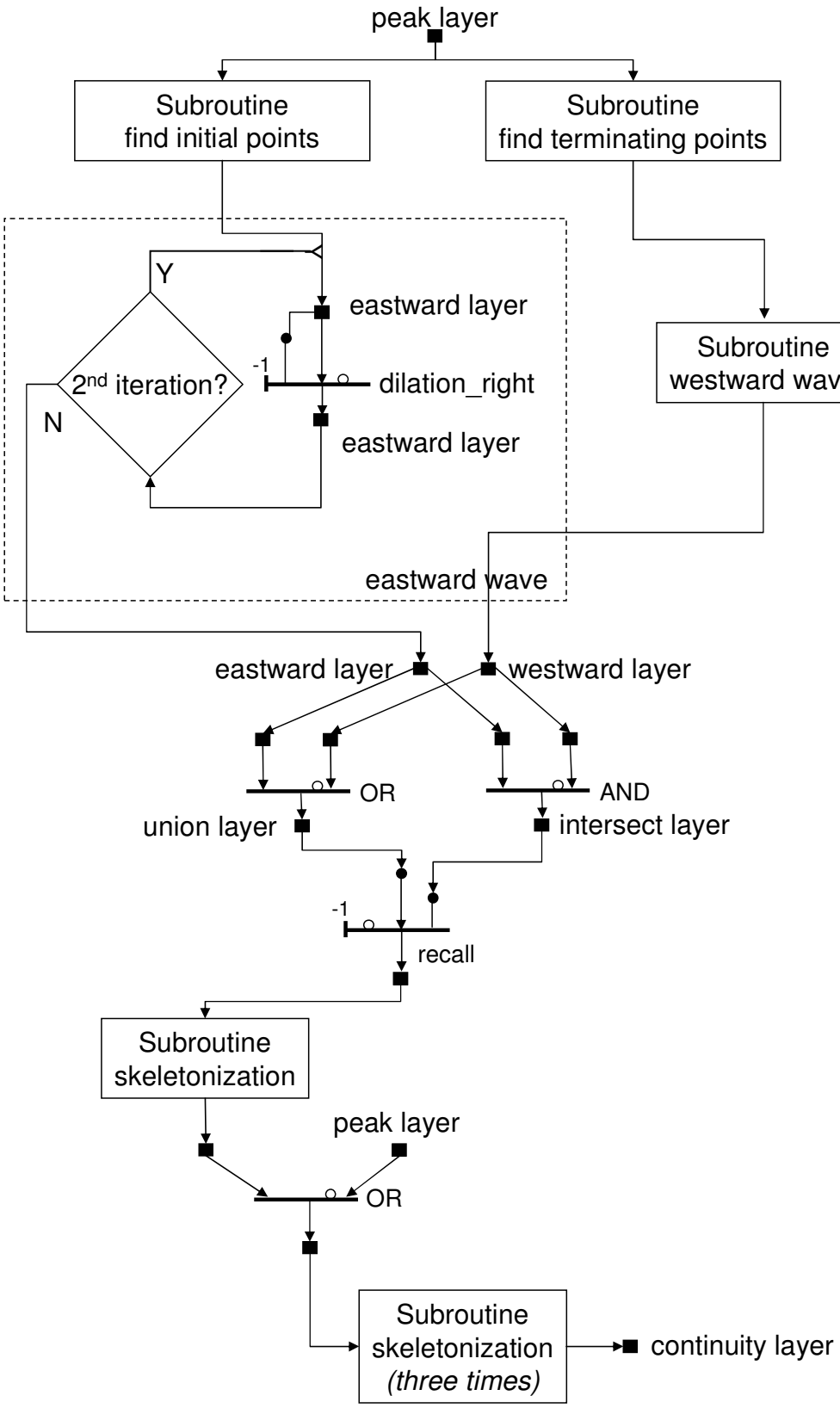
Output Parameters

continuity layer	Connected curves

Remarks

The spanned distance depends on the iteration of *dilation_right* template.

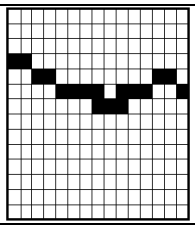
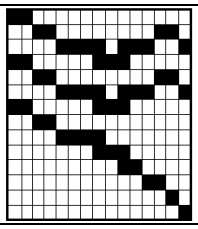
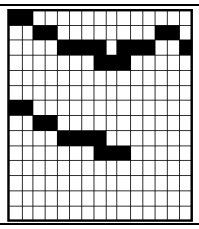
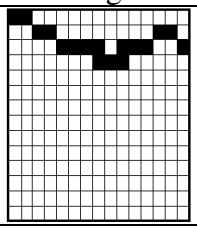
UMF diagram


peak layer
Subroutine
find initial points
Subroutine
find terminating points
Y
eastward layer
Subroutine
westward wave
-1
2nd iteration?
dilation_right
N
eastward layer
eastward wave
eastward layer
westward layer
OR
AND
union layer
intersect layer
-1
recall
Subroutine
skeletonization
peak layer
OR
Subroutine
skeletonization
(three times)
continuity layer

PARALLEL CURVE SEARCH

Task description and algorithm

This algorithm finds parallel curves to a selected one from an initial curve set [72].

Reference curve	Initial curve set	Output of the pixel-search step to a given distance	Output of the *keep almost complete curves* algorithm
			

Input Parameters

peak layer	Peak layer
selected object layer	Reference curve

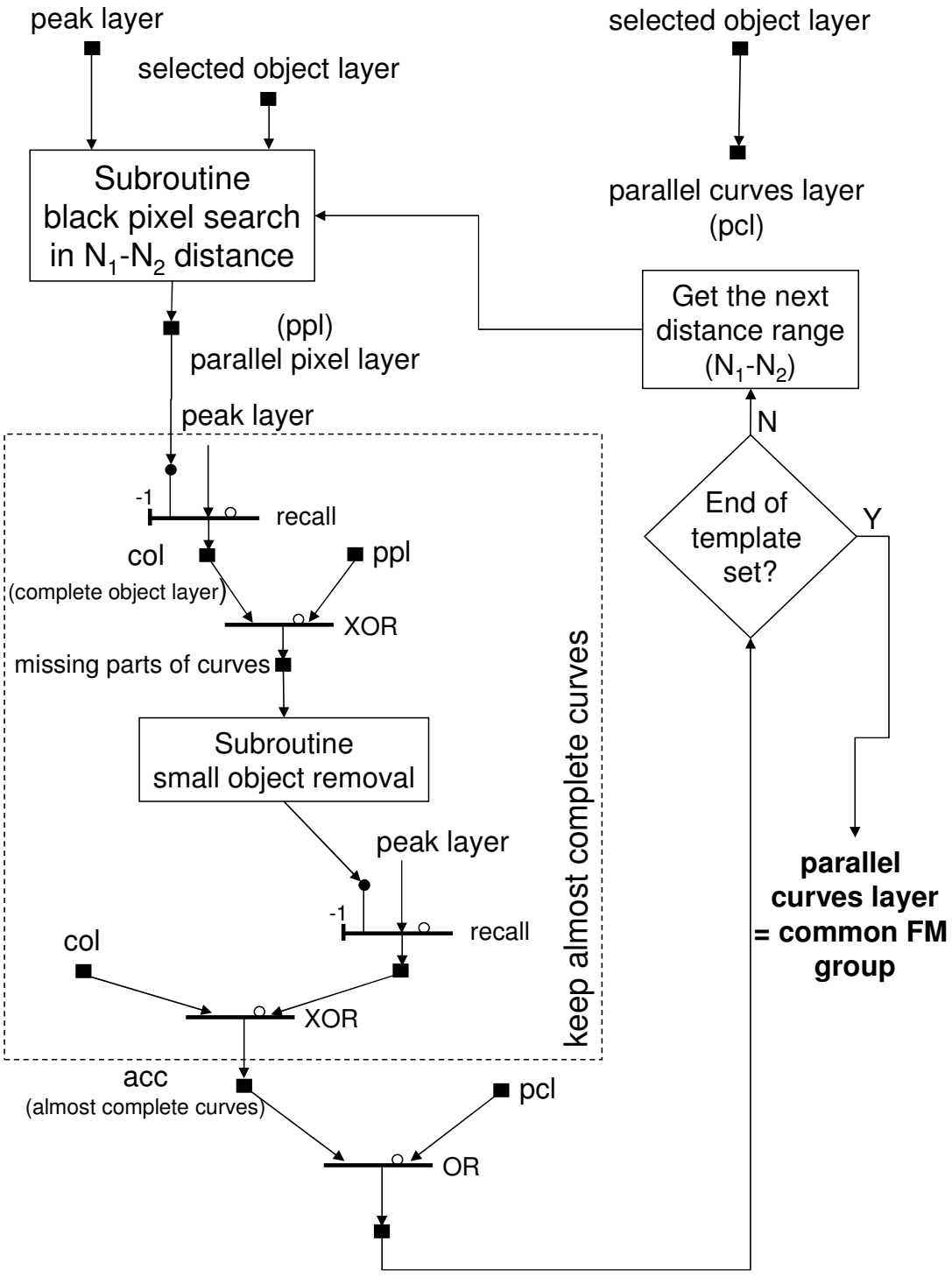
Output Parameters

parallel curves layer	Curves parallel to the reference

Remarks

The search region is controlled through the black pixel search template class.

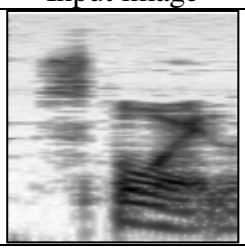
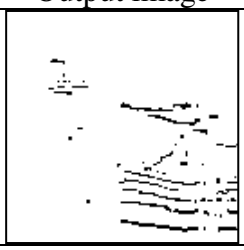
UMF diagram


peak layer
selected object layer
Subroutine
black pixel search
in N1-N2 distance
(ppl)
parallel pixel layer
peak layer
-1
recall
col
(complete object layer)
ppl
XOR
missing parts of curves
Subroutine
small object removal
peak layer
-1
recall
col
XOR
acc
(almost complete curves)
pcl
OR
keep almost complete curves
selected object layer
parallel curves layer
(pcl)
Get the next
distance range
(N1-N2)
N
End of
template
set?
Y
parallel
curves layer
= common FM
group

PEAK-AND-PLATEAU DETECTOR

Task description and algorithm

This algorithm detects horizontal the peaks and plateaus on a grayscale image [72].

Input image	Output image
	

Input Parameters

magnitude image	Input image

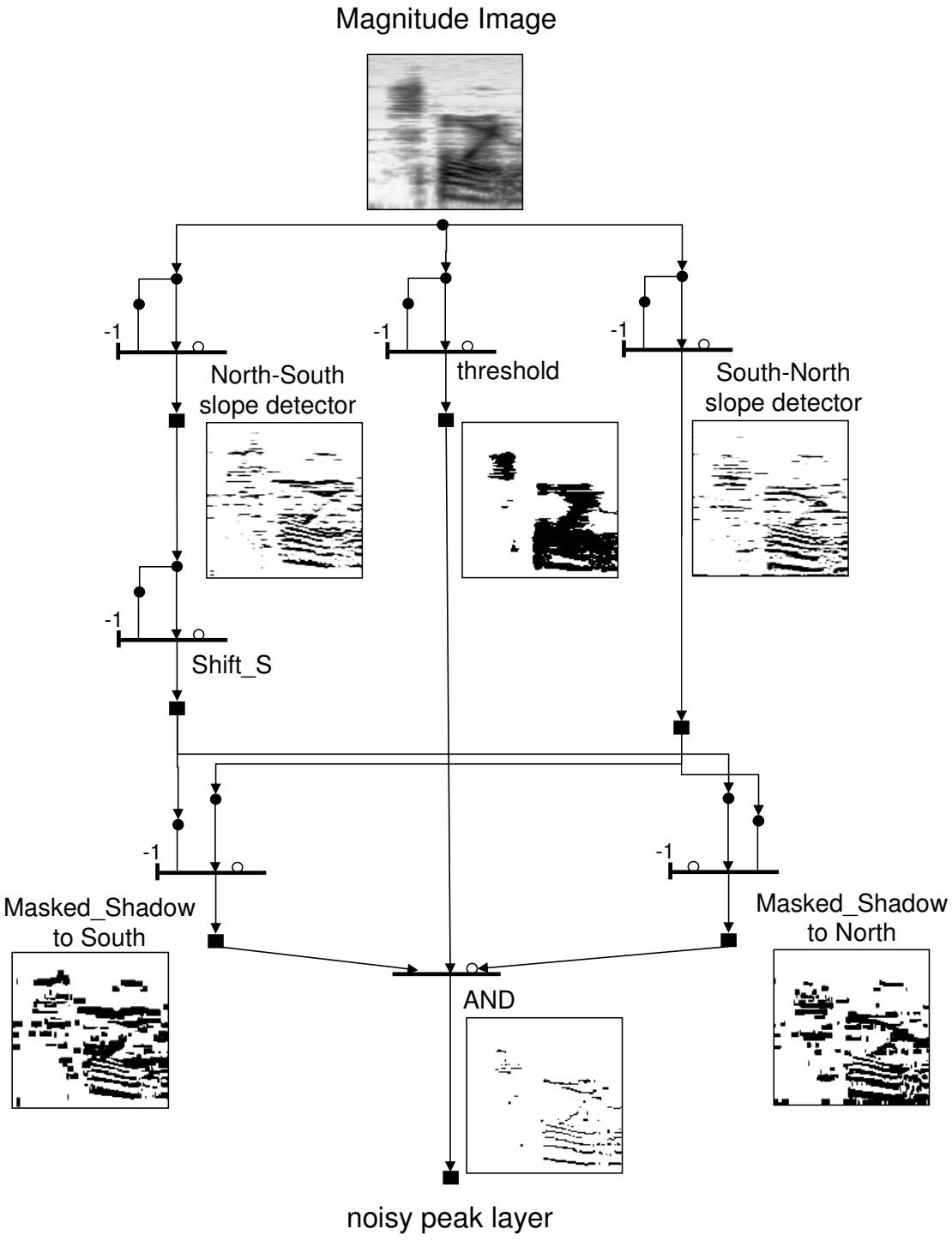
Output Parameters

noise peak layer	Peaks and plateaus

Remarks

There could be a few pixel wide gaps in the detected peaks and plateaus while vertical interconnecting pixels do not cause black pixels to emerge. This gaps can be filled by the *broken line connector* algorithm.

UMF diagram


Magnitude Image
-1
North-South
slope detector
-1
threshold
-1
South-North
slope detector
-1
Shift_S
-1
-1
Masked_Shadow
to South
Masked_Shadow
to North
AND
noisy peak layer

GLOBAL DISPLACEMENT DETECTOR

Task description and algorithm

This algorithm calculates the most probable global displacement vector of the input scene. The diagram shows the calculation steps for the vertical coordinate of the displacement vector. The horizontal coordinate can be calculated in an analogous way by exchanging vertical and horizontal directions.

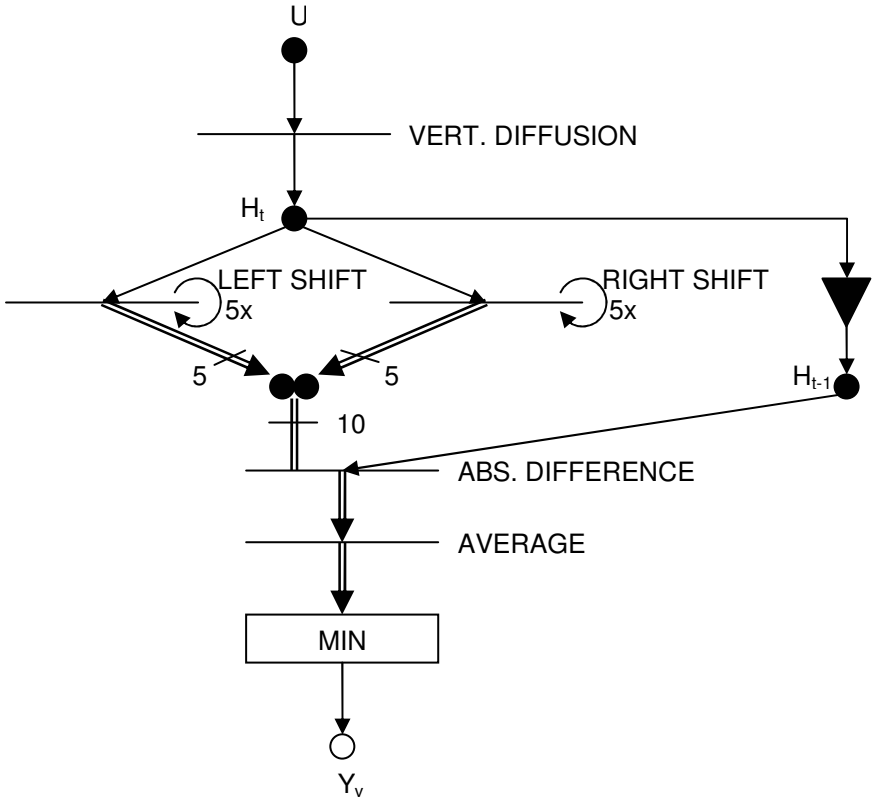
Input Parameters

U	Input image

Output Parameters

Y_h	Horizontal displacement
Y_v	Vertical displacement

UMF diagram

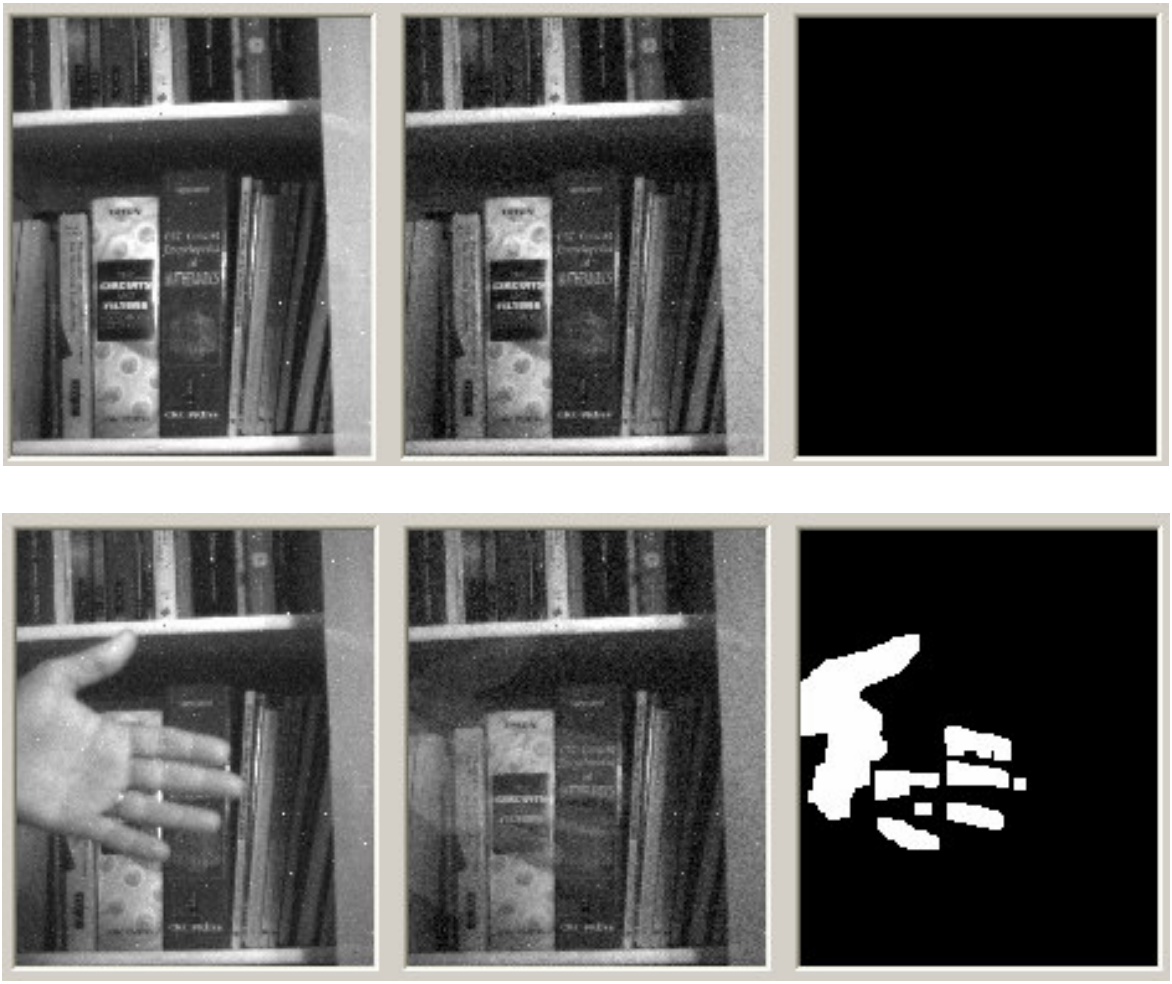

U
VERT. DIFFUSION
H_t
LEFT SHIFT
5x
RIGHT SHIFT
5x
5
5
H_{t-1}
10
ABS. DIFFERENCE
AVERAGE
MIN
Y_v

ADAPTIVE BACKGROUND AND FOREGROUND ESTIMATION

Task description and algorithm

This algorithm continuously estimates the background and foreground of a video flow.

Input image Background Foreground



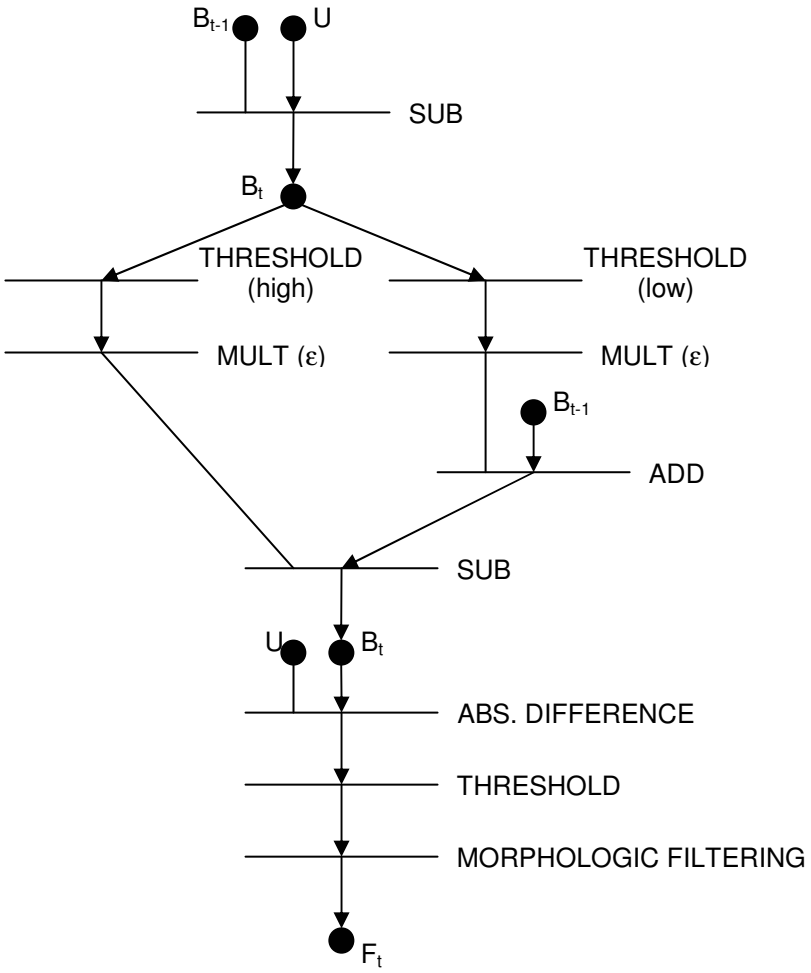
Input Parameters

U_t	Input frame at time t
B_{t-1}	Background estimation after previous frame

Output Parameters

B_t	Updated background estimation
F_t	Current foreground estimation

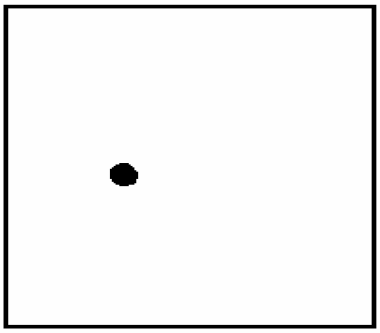
UMF diagram


B_{t-1}
U
SUB
B_t
THRESHOLD (high)
THRESHOLD (low)
MULT (ε)
MULT (ε)
B_{t-1}
ADD
SUB
U
B_t
ABS. DIFFERENCE
THRESHOLD
MORPHOLOGIC FILTERING
F_t

BANK-NOTE RECOGNITION

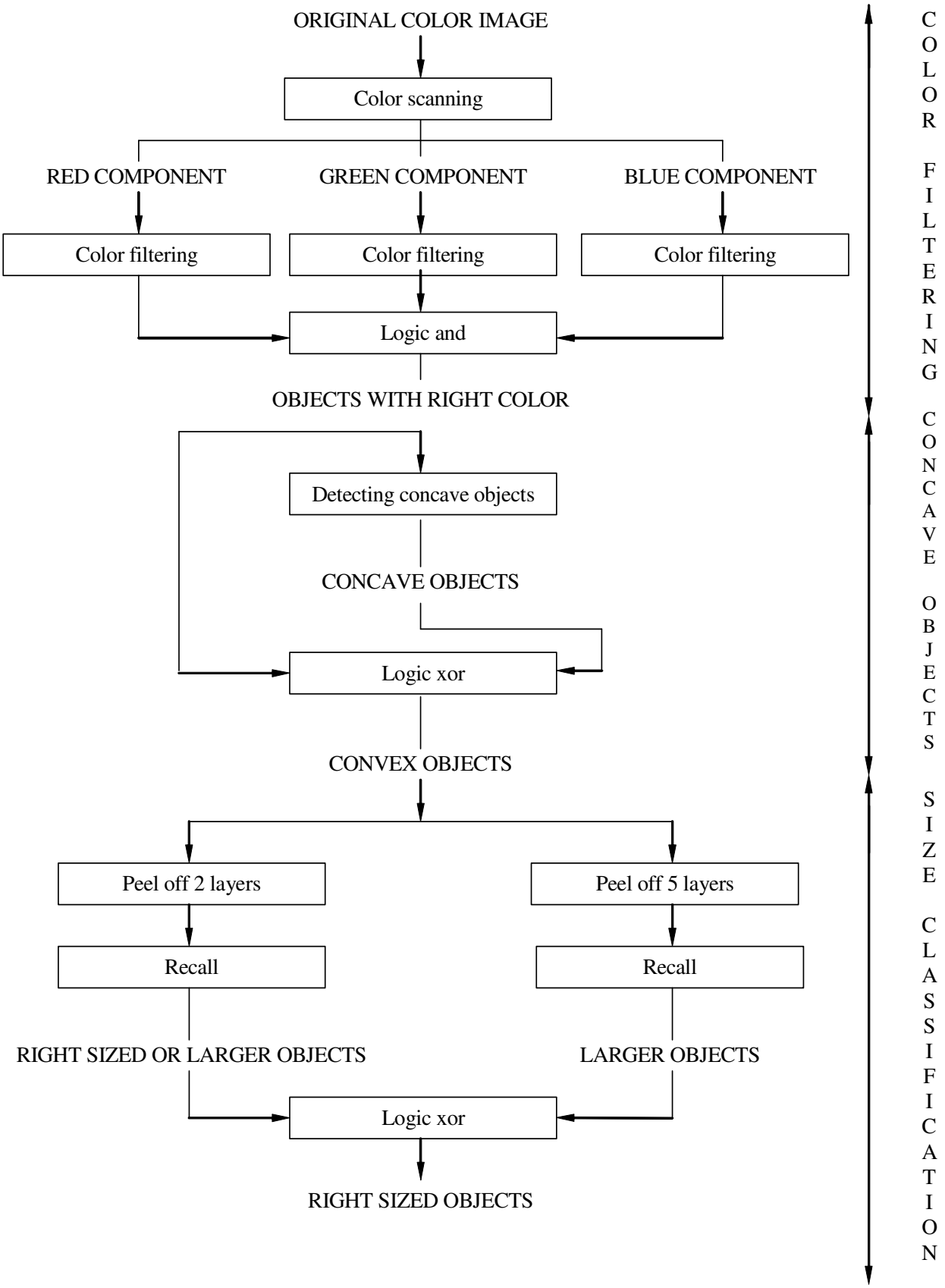
This algorithm identifies American bank-notes on color images. The bank-notes can be in the image with arbitrary offset and rotation. The positioned algorithm finds the green and black circles common to all US bank-notes. It analyses color, shape and size. The algorithm can be separated into three parts. These parts are indicated in the flow-chart. The detailed description can be found in [22]. The templates can be found in this template library.

*Example***:** A grayscale version of a color input image, and the extracted black circle.



The flow-chart of the algorithm:

This chart contains only half of the algorithm, and finds only one circle. The other circle can be found with a similar method, but with different parameters in the color filtering.


ORIGINAL COLOR IMAGE
Color scanning
RED COMPONENT
GREEN COMPONENT
BLUE COMPONENT
Color filtering
Color filtering
Color filtering
Logic and
OBJECTS WITH RIGHT COLOR
Detecting concave objects
CONCAVE OBJECTS
Logic xor
CONVEX OBJECTS
Peel off 2 layers
Peel off 5 layers
Recall
Recall
RIGHT SIZED OR LARGER OBJECTS
LARGER OBJECTS
Logic xor
RIGHT SIZED OBJECTS
COLOR FILTERING
CONCAVE OBJECTS
SIZE CLASSIFICATION

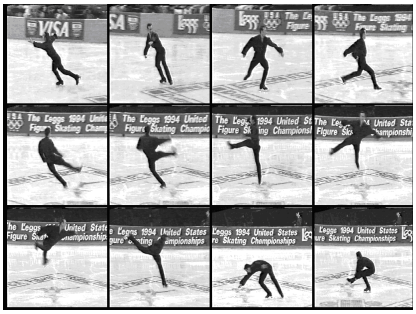
CALCULATION OF A CRYPTOGRAPHIC HASH FUNCTION [37]

Short Description

The CNN analogic program described here performs the basic operations needed to calculate a hash value, when a set of binary images and a key vector are given. The current version of the Alpha compiler cannot interpret sequences of key bits or image sequences, therefore the Alpha source code listed below contains only two images and two key bits.
The first image is loaded to the chip, and its columns (as binary vectors) are multiplied by the key vector. This multiplication is performed as a sequence of shift-add operations. Then, the next binary image is added to modulo 2, and the multiplication is performed once more.

Typical Example

Gray-scale images (or video sequences) must be quantized and cut into pieces according to the chip size. The following pictures show an input sequence and a typical hash result; the latter, of course, heavily depends on the key bits.

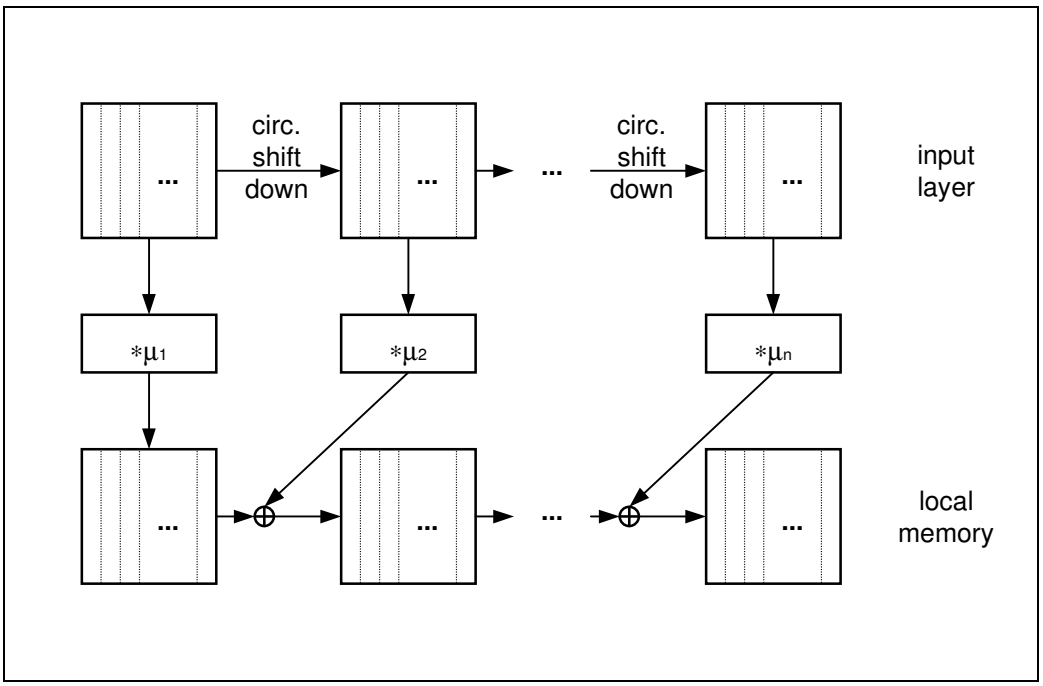


INPUT SEQUENCE (12x256x256x8 bits)

HASH RESULT (21x20 bits)

Flow-diagram of the algorithm

The following diagram shows the CNN implementation of the vector multiplication needed in the hashing process:



Templates used in the algorithm

The algorithm uses only a vertical shift operation and local logic. The shift operation is performed via the following template:

SHIFTSOU template:

A = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 2 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = | 0 |

DETECTION OF MAIN CHARACTERS

Short Description

The algorithm has three major steps. In the first step, only a part of the noise is discarded, but the main features are coming out fine. In the second step a more aggressive filter were applied. After this step, only some parts of the largest objects remained on the image. In the last step, the main characters are reconstructed from the previous two results.

Typical Example

The speciality of this example is that the input image was stored in an extremely high-density optical memory [40]. It is corrupted with noise heavily.

In this example, the image size is 318x93. This image was automatically cut to about 300 20x22 image tiles and processed one after the other on the chip. The algorithm was executed on the 20x22 CNN chip [41]. Experimental results of the main feature extractor algorithm are as follows.

(a)

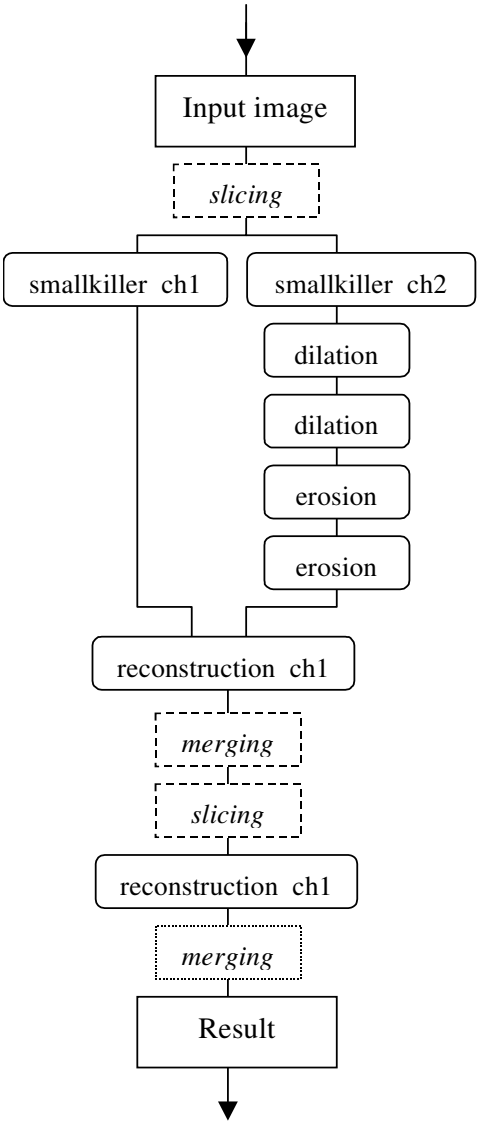
(b)

(c)

(d)

Block Diagram of the Algorithm

The flowchart of the main feature extractor algorithm is as follows.



Templates used in the algorithm

SMALLKILLER_CH1:

A = | 1 | 1 | 1 |
| 1 | 2 | 1 |
| 1 | 1 | 1 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = -1.7

SMALLKILLER_CH2:

A = | 1 | 1 | 1 |
| 1 | 2 | 1 |
| 1 | 1 | 1 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = -2

DILATION:

A = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 1 | 0 |
| 1 | 1 | 1 |
| 0 | 1 | 0 |

z = 4.5

EROSION:

A = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

B = | 0 | 1 | 0 |
| --- | --- | --- |
| 1 | 1 | 1 |
| 0 | 1 | 0 |

z = -5.5

RECONSTRUCTION_CH:

A = | 0 | 1 | 0 |
| --- | --- | --- |
| 1 | 3 | 1 |
| 0 | 1 | 0 |

B = | 0 | 0 | 0 |
| --- | --- | --- |
| 0 | 3 | 0 |
| 0 | 0 | 0 |

z = -1.25

FAULT TOLERANT TEMPLATE DECOMPOSITION [49, 50]

Short Description

An idea of fault tolerant template decomposition in the case of local boolean operators (binary input/output templates) will be outlined here. Due to parameter deviations of current analog VLSI implementations, templates generated theoretically do not work properly. A solution to this problem is applying a sequence of so called fault tolerant templates, that "make no faults". Here two examples of such a decomposition will be presented: the Local Concave Place (LCP) detector template and the JUNCTION template from this template library will be decomposed into a sequence of two fault tolerant templates.

Typical Examples

Example 1: LocalConcavePlaceDetector (LCP) template decomposition

LCP:

$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 2 & 2 \\ 1 & -2 & 1 \end{bmatrix}$ $z = \boxed{-5}$

Minimized form of the function: $F(u) = u_6 u_5 u_4 \bar{u}_2 (u_1 + u_3)$

Sub-functions chosen: $F_1(u) = u_6 u_5 u_4 \bar{u}_2$ $F_2(u) = u_1 + u_3$

Fault tolerant template sequence:

LCP1:

$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & -1 & 0 \end{bmatrix}$ $z = \boxed{-3}$

LCP2:

$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ $z = \boxed{1}$

Result: *LCP* $\Leftrightarrow$ *LCP1* ***AND*** *LCP2,*
where $\rho(\text{LCP}) = 0.24$, while $\rho(\text{LCP1}) = 0.5$ and $\rho(\text{LCP2}) = 0.71$.

Example 2: Decomposition of the JunctionExtractor template

JunctionExtractor:

$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 6 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ $z = \boxed{-3}$

Minimized form of the function: $F(u) = u_5(u_1 u_2 u_3 + u_1 u_2 u_4 + u_1 u_2 u_5 + ... + u_6 u_8 u_9 + u_7 u_8 u_9)$

Sub-functions chosen: $F_1(u) = u_5$

$$F_2(u) = u_1 u_2 u_3 + u_1 u_2 u_4 + u_1 u_2 u_5 + ... + u_6 u_8 u_9 + u_7 u_8 u_9$$

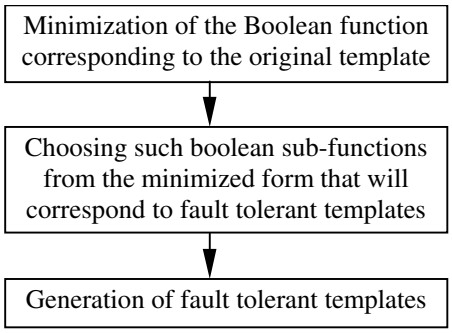
Fault tolerant template sequence:

JUNC1:

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad z = \boxed{3}$$

Result: *JunctionExtractor* $\Leftrightarrow$ INPUT ***AND*** *JUNC1*
where ρ(JunctionExtractor) = 0.15, and ρ(JUNC1) = 0.35 .

Block Diagram of the Algorithm



Templates used in the algorithm

Templates used in examples are shown in the “Typical Examples” session.

ALPHA source

Alpha source of Example1

```
PROGRAM LCP (input; output);
/* DECOMPOSITION OF THE LOCAL CONCAVE PLACE DETECTOR TEMPLATE */
/* BY USING TWO FAULT TOLERANT TEMPLATES */
CONSTANT
  White = -1.0;
  ONE = 1;
  TWO = 2;
  TIME = 5;
ENDCONST;
/* Chip set definition section */
CHIP_SET simulator.eng;
A_CHIP
SCALARS
IMAGES
 LLM1: BINARY;
 LLM2: BINARY;
 LLM3: BINARY;
ENDCHIP;
E_BOARD
SCALARS

IMAGES
 P: BINARY;
```

```
ENDBOARD;

OPERATIONS FROM LCP.tms;
PROCESS LCP_DECOMP;
  USE (lcp1, lcp2);
  HostLoadPic(input, P);
  HostDisplay(P, ONE);
  LLM1 := P;
  lcp1 (LLM1, LLM1, LLM2, TIME, White);
  lcp2 (LLM1, LLM1, LLM3, TIME, White);
  LLM1 := LLM3;
  LLM1 := LLM1 AND LLM2;
  P := LLM1;
  HostDisplay(P, TWO);
  HostSavePic(output, P);
ENDPROCESS;
ENDPROG;
```

Alpha source of Example2

```
PROGRAM JUNCTION (input; output);
/* DECOMPOSITION OF THE JUNCTION TEMPLATE */
/* BY USING ONE FAULT TOLERANT TEMPLATE AND A LOGIC OPERATION */
CONSTANT
  White = -1.0;
  ONE = 1;
  TWO = 2;
  TIME = 5;
ENDCONST;
/* Chip set definition section */
CHIP_SET simulator.eng;
A_CHIP
SCALARS
IMAGES
 LLM1: BINARY;
 LLM2: BINARY;
 LLM3: BINARY;
ENDCHIP;
E_BOARD
SCALARS
IMAGES
 P: BINARY;
ENDBOARD;
OPERATIONS FROM junction.tms;
PROCESS JUNCTION_DECOMP;
  USE (junc1);

  HostLoadPic(input, P);
  HostDisplay(P, ONE);
  LLM1 := P;
  junc1 (LLM1, LLM1, LLM2, TIME, White);
```

```
  LLM1 := LLM1 AND LLM2;
  P := LLM1;
  HostDisplay(P, TWO);
  HostSavePic(output, P);
ENDPROCESS;
ENDPROG;
```

Comments

Fault tolerant template generation results in a sequence of “reliable” templates.

GAME OF LIFE

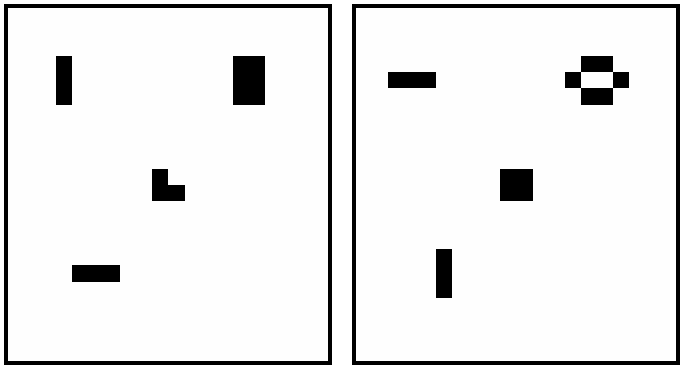
Short Description

The following simple algorithm simulates the Game of Life. Both input and output pictures are binary. Rules of the game:

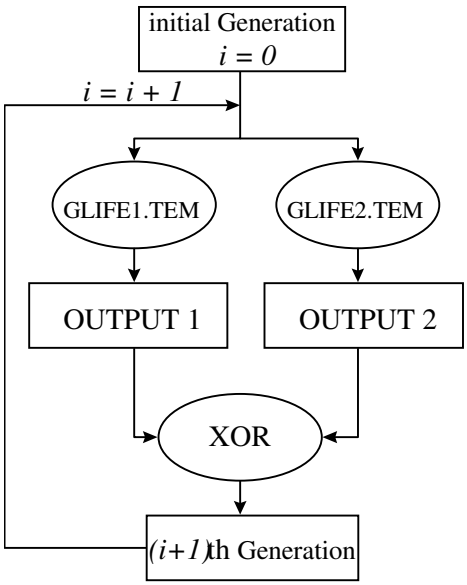
- a black pixel turns white if it has *more than three* or *less than two* black neighbors.
- a white pixel turns black if it has *exactly three* black neighbors.

Typical Example

The following example shows two consecutive generations of the Game of Life simulated by CNN.



Block Diagram of the Algorithm



Templates used in the algorithm

The corresponding templates can be found in this template library as LIFE_1.

ALPHA source

```
PROGRAM GAMEOFLIFE (inputFC);
/* THIS PROGRAM SIMULATES TEN CYCLES OF THE GAME OF LIFE  */

CONSTANT                              /* constant declaration */
```

```
   NULL = 0;
   NUM_GEN = 9;                          /* Number of generations */
   ONE = 1;
   WHITE = -1.0;
   TIME = 5;
   TIMESTEP = 0.5;
ENDCONST;
CHIP_SET simulator.eng;
A_CHIP
SCALARS
IMAGES
  L1: BINARY;          /* LLM1 */
  L2: BINARY;                /* LLM2 */
  L3: BINARY;                /* LLM3 */
  L4: BINARY;                /* LLM4 */
ENDCHIP;
E_BOARD
SCALARS
 var: INTEGER;
IMAGES
 LargeInp: BINARY;
 LargeOut: BINARY;
ENDBOARD;
OPERATIONS FROM gameoflife.tms;
FUNCTION game_of_life;
 USE (glife1, glife2);
   L4 := NULL;                                 /* Zero state */
   glife1 (L3, L4, L1, TIME, WHITE);           /* Template1 execution */
   glife2 (L3, L4, L2, TIME, WHITE);           /* Template2 execution */
   L4 := L1 XOR L2;                      /* XOR operation */
ENDFUNCT;
PROCESS freichen;                        /* here starts the main routine */
 USE ();
  SwSetTimeStep (TIMESTEP);
  HostLoadPic(inputFC, LargeInp);
  L4 := LargeInp;
  REPEAT var := 1 TO NUM_GEN BY 1;
   L3 := L4;                             /* Reload i-th genaration to input */
   game_of_life;                         /* Simulating one generation of the Game of Life */
   LargeOut := L4;                       /* copying the image from chip to board */
   HostDisplay(LargeOut, ONE);
  ENDREPEAT;
 ENDPROCESS;
ENDPROG;
```

Comments

The templates can be found by using the TemMaster [38] template design software package.

HAMMING DISTANCE COMPUTATION

In the theory of information processes it is a common problem that, given a code received in a noisy channel and the set of legal code words, we have to determine the code word nearest in some metric to the received one. In the case of binary codes the Hamming distance is the most common choice to measure the distance. Here a 4-step method is given presented, which selects the legal code closest to the input.

The first step compares the input to all legal code words. In order for this to happen, the m legal code words should be put in a single image, each code being a separate row, while the input should be written in another image m times. This step can be performed by the logic **XOR** template. In the second step the number of differences is calculated, after feeding the output of the previous step back to the input and setting the initial state to 0. In the third step the minimum distance is determined, and finally, the best matching code word is selected. For this to be realized, the output of the previous step should be used as initial state, and that of step 2 as input [20].

Templates:

Differences: *Min. distance:* *Best matching*

$\mathbf{A}_2 =$

0	0	0
0	0	1
0	0	0

$\mathbf{A}_3 =$

0	***b***	0
0	1	0
0	***b***	0

$\mathbf{A}_4 =$

0	0	0
0	2	0
0	0	0

$\mathbf{B}_2 =$

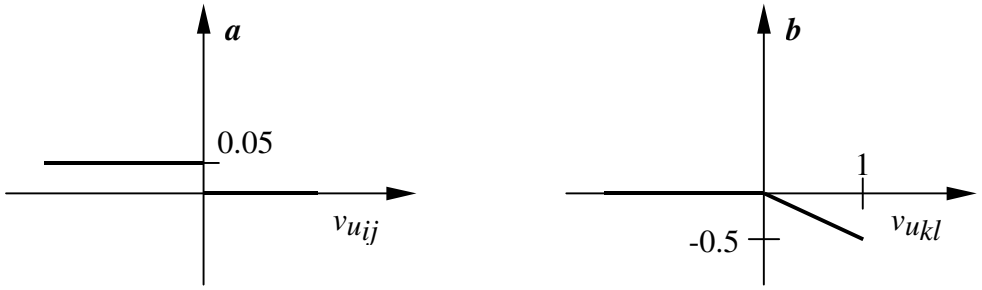
0	0	0
0	***a***	0
0	0	0

$\mathbf{B}_4 =$

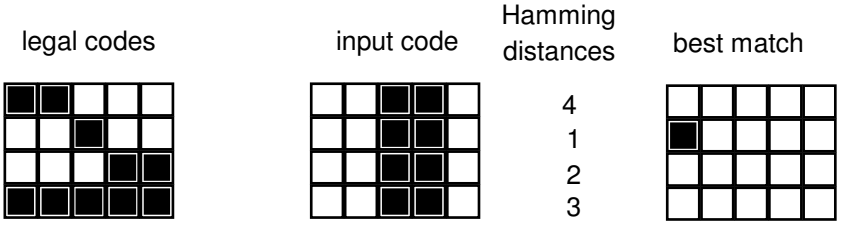
0	0	0
0	-1	0
0	0	0

$z_4 =$ | 0.02 |

where ***a*** and ***b*** are defined by the following nonlinear functions:



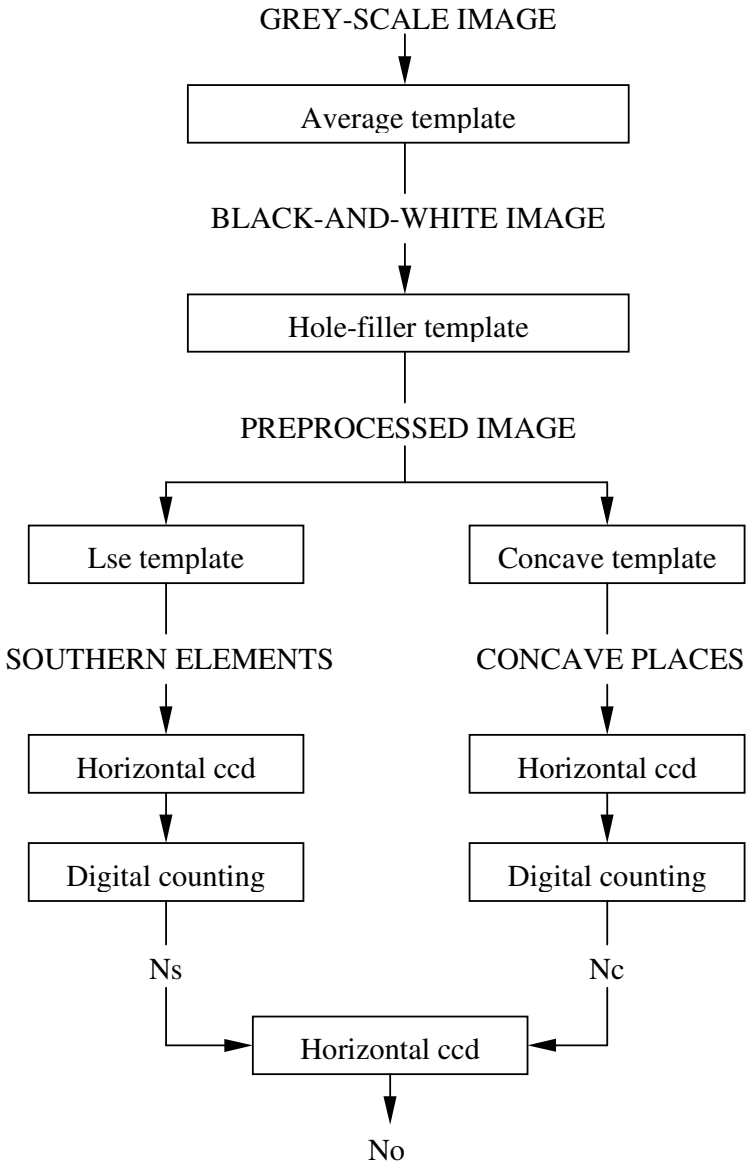
Example:



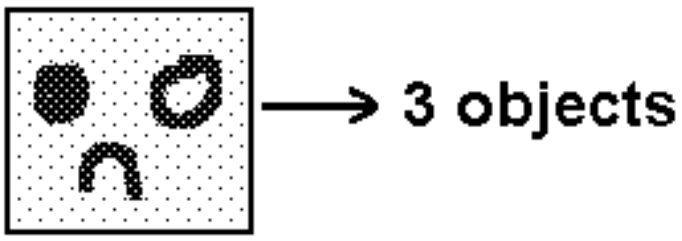
OBJECT COUNTING

This algorithm counts the connected objects on a grayscale image. The algorithm is detailed in [11]. The cited templates can be found in this template library.

The flow-chart of the algorithm:



Example: The algorithm is demonstrated on a grayscale image containing 3 objects. Image name: objcount.bmp; image size: 91x83.



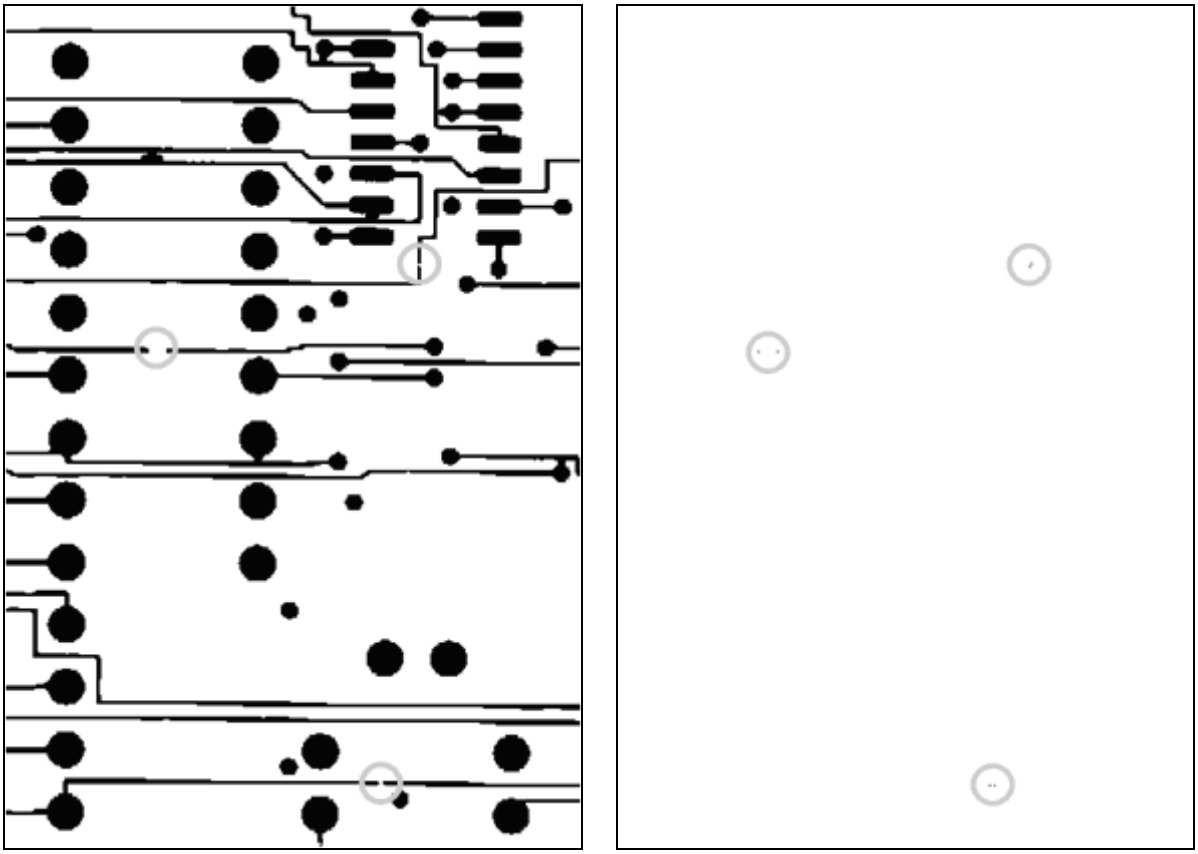
OPTICAL DETECTION OF BREAKS ON THE LAYOUTS OF PRINTED CIRCUIT BOARDS [39]

Short Description

The input images of a printed circuit board or an artwork film are scanned in by using a scanner. For detecting breaks, the basic idea of the algorithm is that a wire has to be terminated in a pad, in a via hole or in another wire.

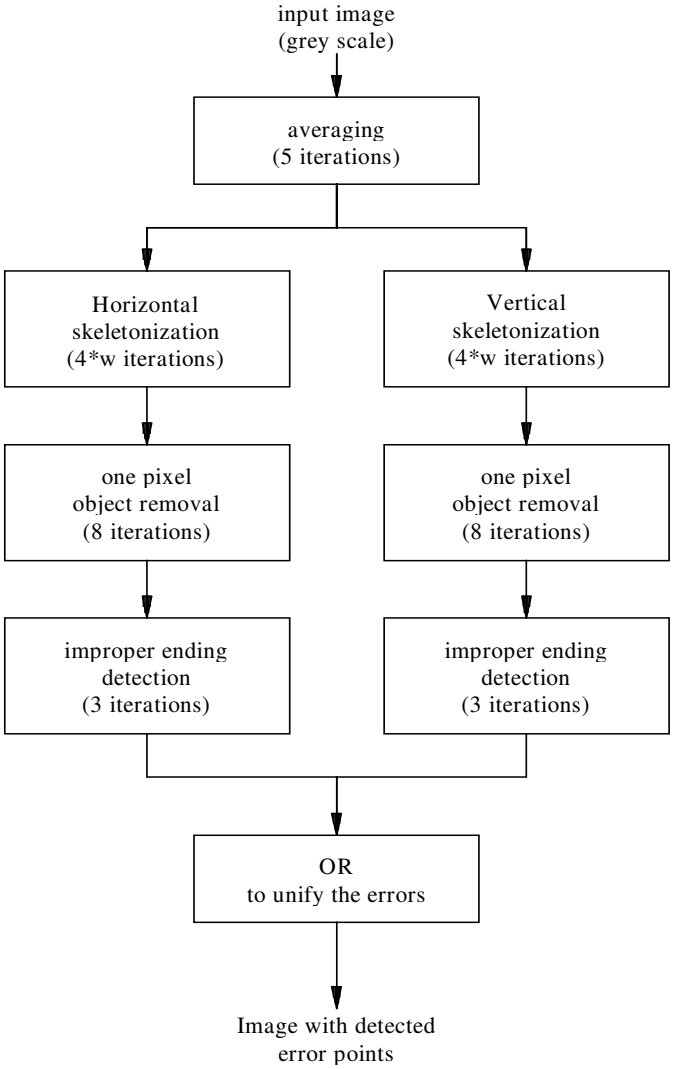
Typical Example

This example was run on the CNN Hardware Accelerator Board (CNN-HAB).



The input and result of the wire break detection analogic CNN algorithm

Flow-diagram of the algorithm



Templates used in the algorithm

The one-pixel-object removal and the OR templates can be found in this template library as *SmallObjectRemover* and *LogicOR*, respectively.

HorSkelL: ***horizontal skeleton from left***

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0.5 & 0 & 0.125 \\ 0.5 & 0.5 & -0.5 \\ 0.5 & 0 & 0.125 \end{bmatrix} \quad z = \boxed{-1}$$

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X(0) =** Arbitrary (in the examples we choose $x_{ij}(0)=0$)

Boundary Conditions: Fixed type, $u_{ij} = 0$, for all virtual cells, denoted by [U]=[0]

Output: **Y(t)⇒Y(∞)** = Binary image, peeling the black pixels from left of a wire

Remark:

The template *HorSkelR* (horizontal skeleton from right) can be obtained by rotating *HorSkelL* by 180°). The *VerSkelT* and *VerSkelB* templates (rotating the *HorSkelR* and *HorSkelL* templates by 90°) are used for horizontal line skeletonization.

DeadEndV: ***finds the endings of vertical wires***

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} -0.25 & -0.25 & -0.25 \\ -0.25 & 0.5 & -0.25 \\ -0.25 & -0.25 & -0.25 \end{bmatrix} \qquad z = \boxed{-5.8}$$

I. Global Task

Given: static binary image **P**

Input: **U(t) = P**

Initial State: **X(0) =** Arbitrary (in the examples we choose $x_{ij}(0)=0$)

Boundary Conditions: Fixed type, $u_{ij} = 0$ for all virtual cells, denoted by [U]=[0]

Output: **Y(t)⇒Y(∞)** = Binary image of the endings of the vertical wires

Remark:

The *DeadEndH* templates (rotating the *DeadEndV* template by 90°) are used to detect the endings of horizontal wires.

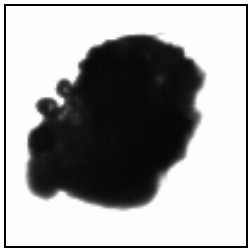
ROUGHNESS MEASUREMENT VIA FINDING CONCAVITIES [18]

Short Description

This simple CNN analogic program detects concavities of objects. This can be used for surface roughness measurement.
The basic idea here is to find the concave parts of objects. First, the gray-scale image is converted into a binary image via a thresholding operation. Next, pixels being located at concave places are driven into black by using the "hollow" template. This template turns black all those white pixels which have at least four black direct neighbors. Next, we extract concavities of objects by using the logical XOR operation between the thresholded image and filled image.

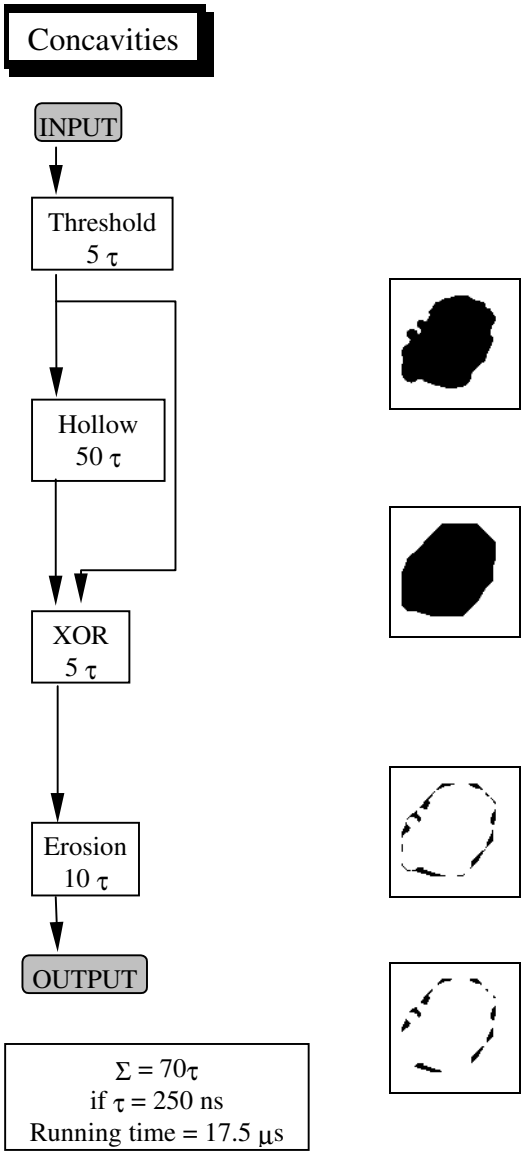
Typical Example

The following example shows the detected concave parts of an object.


INPUT


OUTPUT

Flow-diagram of the algorithm



Templates used in the algorithm

Templates can be found in this template library as *Threshold*, *ConcaveLocationFiller* (HOLLOW), and *Erosion*, respectively. The exact template values are presented below.

Threshold: converting gray-scale image to binary image

A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = 0

ConcaveLocationFiller: fills the concave locations of objects

This template turns black all those white pixels which have at least four black direct neighbors. We call concave those white pixels which are surrounded by black pixels from at least four of the eight possible directions. The network transient must be stopped after a given amount of time, depending on the size of the largest holes to be filled in.

A = | 0.5 | 0.5 | 0.5 |
| 0.5 | 2 | 0.5 |
| 0.5 | 0.5 | 0.5 |

B = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

z = 3.5

Erosion: ***eroding picture with a given structuring element***

Erosion represents the probing of an image to see where some primitive shapes fit inside the image. The primitive is called structuring element placed in the **B** term.

A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

z = -8.5

ALPHA source

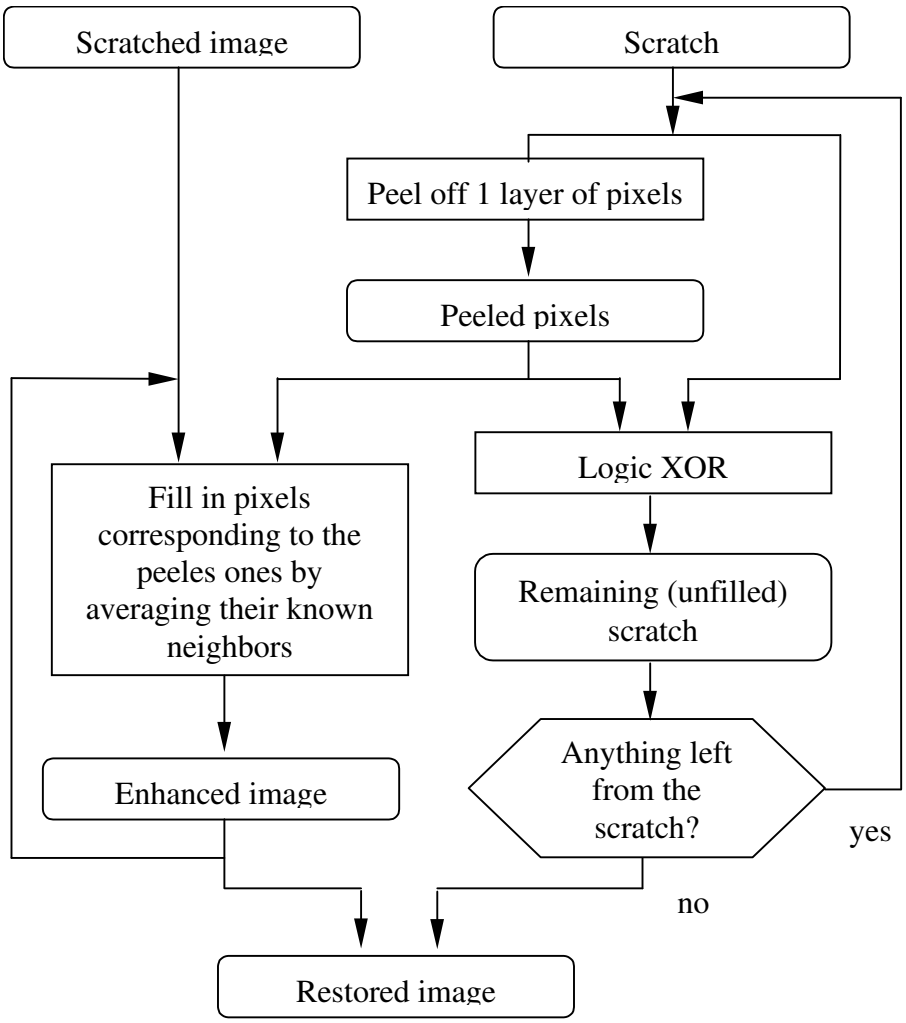
```
/* THE PROGRAM DETECTS CONCAVITIES OF OBJECTS */
PROGRAM concave (in; out);
CONSTANT
ONE = 1;
TWO = 2;
WHITE = -1.0;
TIME1 = 50;
TIME2 = 10;
ENDCONST;
/* Chip set definition section */
CHIP_SET simulator.eng;
A_CHIP
SCALARS
IMAGES
 c1: BINARY;
 c2: BINARY;
 c3: BINARY;
 c4: BINARY;
ENDCHIP;
E_BOARD
SCALARS
IMAGES
 bi1: BINARY;
 bi2: BINARY;
ENDBOARD;
OPERATIONS FROM concave.tms;

PROCESS concave;
USE (thres, hollow, erosion);
  HostLoadPic(in, bi1);
  HostDisplay(bi1, ONE);
  c1 := bi1;
  thres (c1, c1, c1, TIME1, WHITE);
  hollow (c1, c1, c2, TIME1, WHITE);
```

```
  c3 := c1 XOR c2;
  erosion(c3, c3, c1, TIME2, WHITE);
  bi2 := c3;
  HostDisplay(bi2, TWO);
ENDPROCESS;
ENDPROG;
```

SCRATCH REMOVAL

On photocopier machines, the glass panel often gets scratched, which scratch is then copied together with the material, resulting in a visually annoying copy. The following algorithm is capable of removing such scratches assuming that the location of the scratch is known in advance. This is a valid assumption, since the scratches can automatically be detected e.g. by copying a blank sheet of paper. The algorithm removes the scratches gradually, peeling off pixels circularly [19].

The flow-chart of the algorithm:



Smoothing:

Selection templates: *Fill templates:*

$\mathbf{A}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B}_1 = \begin{bmatrix} -0.5 & 0 & 0 \\ -0.5 & 0.5 & 0 \\ -0.5 & 0 & 0 \end{bmatrix}$ $z_1 = \boxed{-1.5}$ $\mathbf{B}_1 = \begin{bmatrix} 0.33 & 0 & 0 \\ 0.34 & 0 & 0 \\ 0.33 & 0 & 0 \end{bmatrix}$

$\mathbf{A}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B}_2 = \begin{bmatrix} -0.5 & -0.5 & 0 \\ -0.5 & 0.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_2 = \boxed{-1.5}$ $\mathbf{B}_2 = \begin{bmatrix} 0.34 & 0.33 & 0 \\ 0.33 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

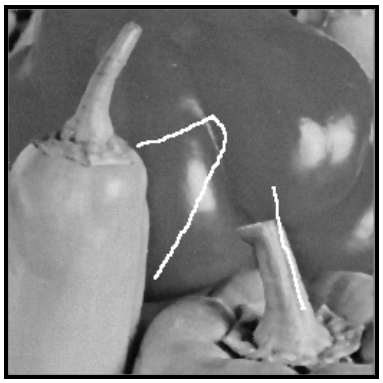
$\mathbf{A}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B}_3 = \begin{bmatrix} -0.5 & -0.5 & -0.5 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $z_3 = \boxed{-1.5}$ $\mathbf{B}_3 = \begin{bmatrix} 0.33 & 0.34 & 0.33 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

. . . . . .

$\mathbf{A}_8 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ $\mathbf{B}_8 = \begin{bmatrix} 0 & 0 & 0 \\ -0.5 & 0.5 & 0 \\ -0.5 & -0.5 & 0 \end{bmatrix}$ $z_8 = \boxed{-1.5}$ $\mathbf{B}_8 = \begin{bmatrix} 0 & 0 & 0 \\ 0.33 & 0 & 0 \\ 0.34 & 0.33 & 0 \end{bmatrix}$

Example: image names: peppers.bmp, scratch.bmp; image size: 256x256.

Original image



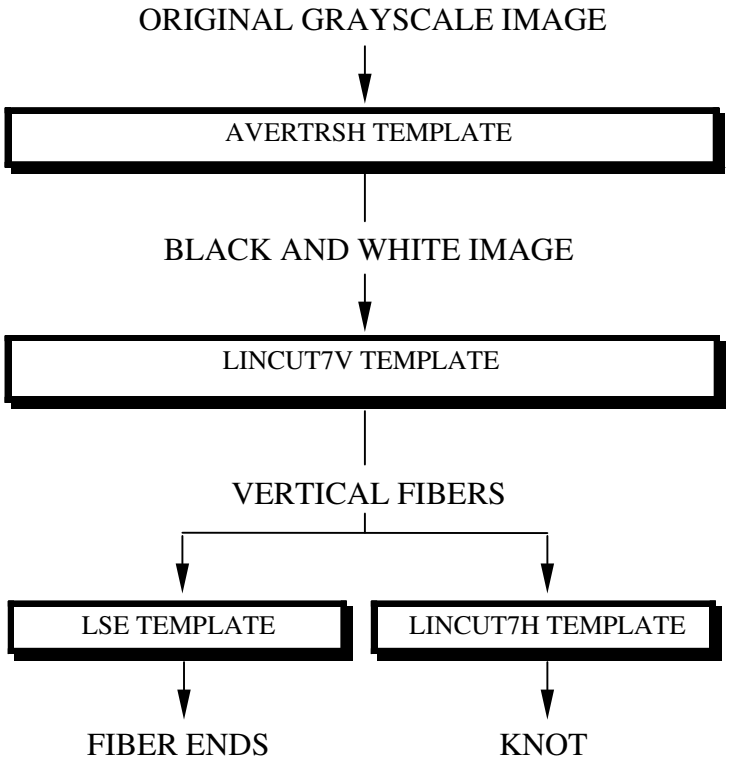
Scratched image

Restored image

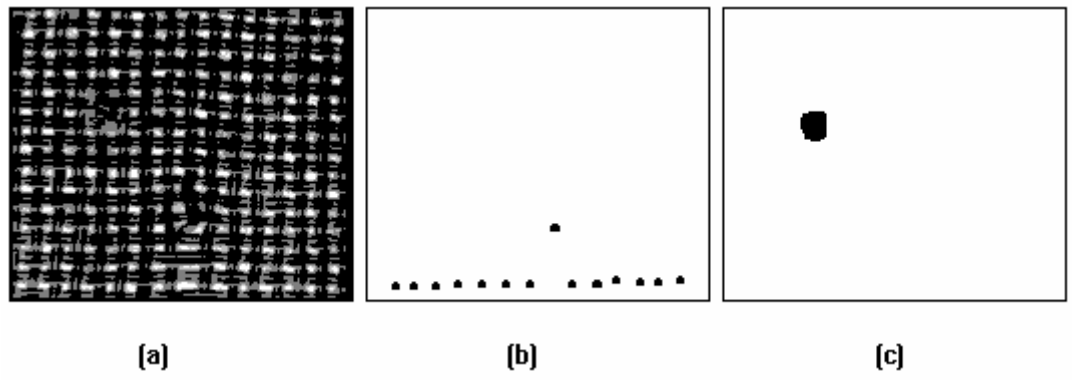
TEXTILE PATTERN ERROR DETECTION

This algorithm finds the knots and fiber breakings in a loose-waved textile. The algorithm is detailed in [10]. The cited templates can be found in this template library. (The lincut7h is the rotated version of lincut7v (*LE7pixelVerticalLineRemover*))

The flow-chart of the algorithm:



Example: The algorithm is demonstrated on a piece of table cloth. (a): the original image, (b) the southern ends of the fibers (the inside one indicates the fault), (c): the knot. Image name: textpatt.bmp; image size: 170x145.

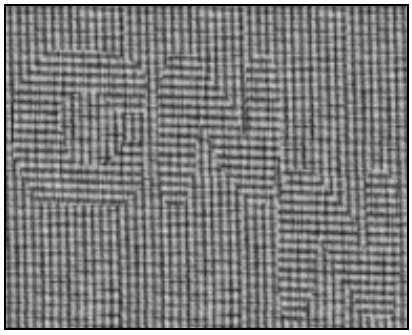


TEXTURE SEGMENTATION I [17]

Short Description

This simple CNN analogic program is able to separate two predefined types of binary textures mixed up in images.
The first step is a template execution, which separates the textures by local ratio of the black-and-white pixels (ROB). In the next step, a small killer template increases the separation. Then an erosion and small killer templates are used for improving the segmentation.

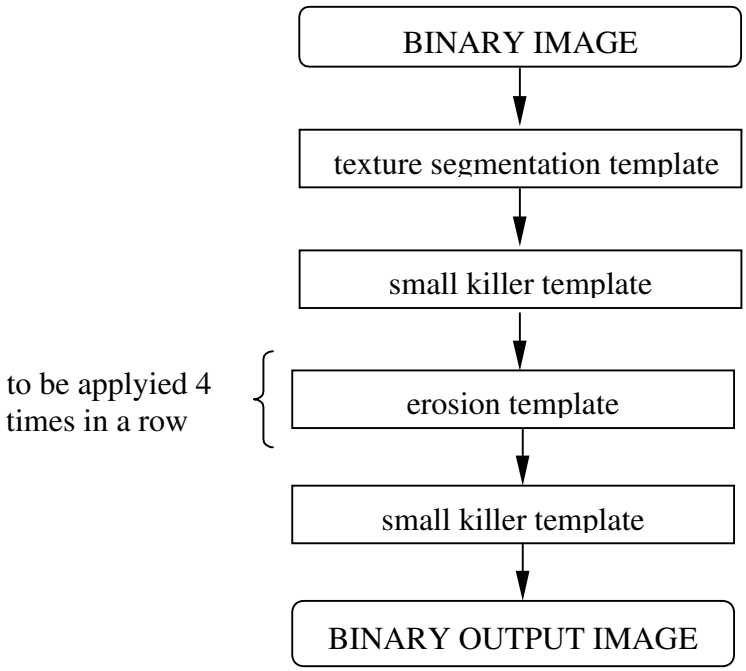
Typical Example



INPUT

OUTPUT

Flow-diagram of the algorithm



Templates used in the algorithm

The small killer template can be found in this template library as *SmallObjectRemover*.
The rest of templates used in this algorithm are described as follows:

Texture segmentation template (tx_hclc1.tem):

A =

-3.44	0.86	-1.64	-0.16	-1.02
-1.09	0.16	-2.19	-3.2	3.51
2.50	1.56	3.91	2.66	2.42
0.55	2.89	-0.62	0.47	3.67
-1.80	-0.55	2.50	-0.23	2.34

B =

-2.19	-0.23	0.16	-0.63	-0.78
1.64	2.27	-3.2	1.09	2.03
0.08	0.55	0.86	3.52	0.08
0.39	-3.83	-3.12	-2.34	-2.11
0.78	-2.66	-1.17	-1.41	1.02

z = 4.8

Erosion template (erosion1.tem):

A =

0	0	0
0	1	0
0	0	0

B =

0.5	1	0.5
1	1	1
0.5	1	0.5

z = -6

ALPHA source

```
/* TEXTURE_1.ALF */
/* Separates two predefined  types of binary textures */
PROGRAM texture_1(in; out);
CONSTANT
ONE = 1;
TWO = 2;
THREE = 3;
FOUR = 4;
FIVE = 5;
SIX = 6;
BLACK = 1;
TimeStep005 = 0.05;
TimeStep01 = 0.1;
TimeStep03 = 0.3;
TimeStep1 = 1;
TimeStep2 = 2;
TIME1 = 1;
TIME2 = 2;
TIME4 = 4;
TIME03 = 0.3;
ENDCONST;
/* Chip set definition section */
CHIP_SET simulator.eng;
A_CHIP
SCALARS
IMAGES
 im1: BINARY;
 im2: BINARY;
 im3: BINARY;
 im4: BINARY;
ENDCHIP;

E_BOARD
```

```
SCALARS
Loop: INTEGER;
IMAGES
 input:  BINARY;
 output: BINARY;
ENDBOARD;
/* Definition of analog operation symbol table */
OPERATIONS FROM texture_1.tms;
PROCESS texture_1;
 USE (tx_hclc1, smkiller, erosion1);
 HostLoadPic(in, input);
 HostDisplay(input, ONE);
 im1:= input;
    SwSetTimeStep (TimeStep005);
     tx_hclc1(im1, im1, im2, TIME1, ZEROFLUX);
     output:=im2;
     HostDisplay(output, TWO);
    SwSetTimeStep (TimeStep01);
    smkiller (im2, im2, im3, TIME03, BLACK);
    output:=im3;
    HostDisplay(output, THREE);
    SwSetTimeStep (TimeStep01);
    REPEAT Loop:= 1 to 4 BY 1;
      erosion1(im3, im3, im3, TIME2,  BLACK);
    ENDREPEAT;
    output:=im3;
    HostDisplay(output, FOUR);
   SwSetTimeStep (TimeStep01);
    smkiller(im3, im3, im4, TIME4, ZEROFLUX);
    output:=im4;
    HostDisplay(output, FIVE);
ENDPROCESS;
ENDPROG;
```

Comments

Using the genetic template learning algorithm, a wide range of texture types can be segmented with high accuracy. In order to achieve flat segments propagating-type filter template(s) could be used.

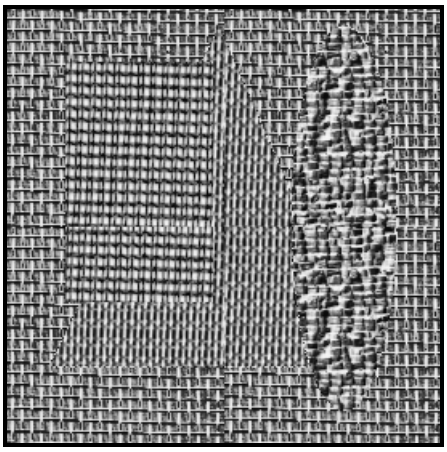
TEXTURE SEGMENTATION II [17]

Short Description

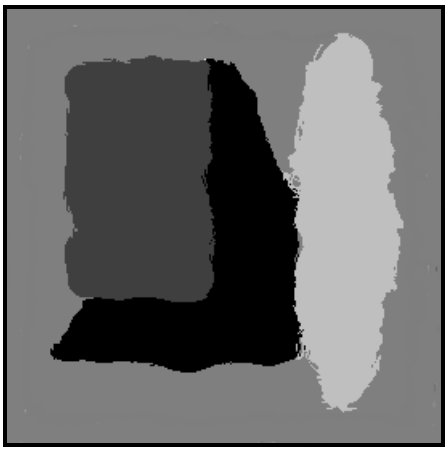
This simple CNN analogic program is able to separate 4 (maybe more) types of textures mixed up in images.
We run 4 or 5 templates consecutively. The first step in an iteration is a template execution which separates the textures by local ratio of the black-and-white pixels (ROB). The chip may physically scan the textured surface or the texture-segments of the whole image are executed separately by reading in from the A-RAM.

Typical Example

The following example can be evaluated by the CNN Engine Board using the next-generation CNN chip.

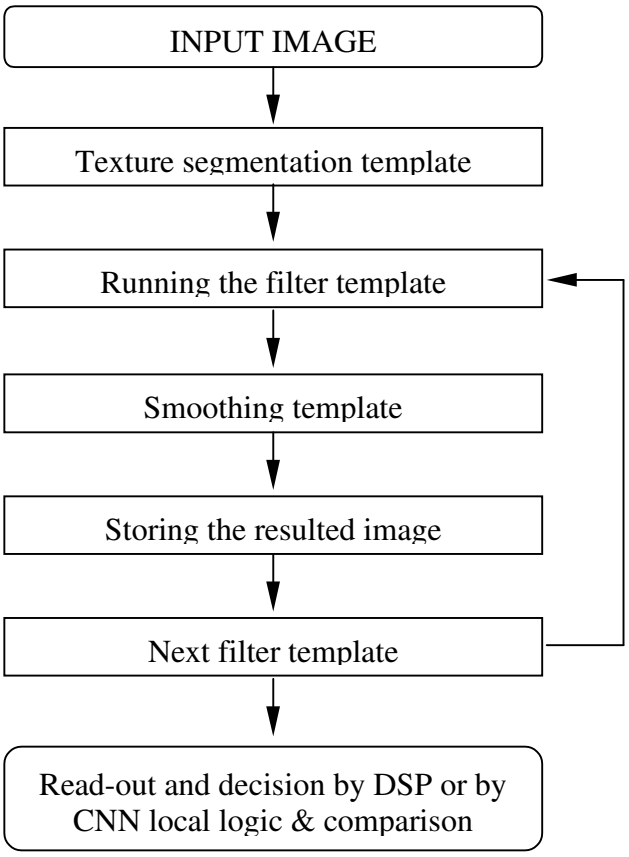


INPUT



OUTPUT

Flow-diagram of the algorithm



Templates used in the algorithm

Texture segmentation templates can be found in this library as *3x3TextureSegmentation*, *5x5TextureSegmentation#*, *Smoothing*.

The Alpha language description of the core of the texture segmentation algorithm is as follows:

ALPHA source

```
/* TEXTURE_2.ALF */
/* Classifies 4 different textured images */
PROGRAM texture_2(in);
CONSTANT
  ONE = 1;
  TWO = 2;
  THREE = 3;
  FOUR = 4;
  WHITE = -1.0;
  RUNTIME_3 = 3;
  TIMESTEP = 0.2;
ENDCONST;
/* Chip description file */
CHIP_SET simulator.eng;
/* Chip variables */
A_CHIP
SCALARS
IMAGES
  ci1: ANALOG;
```

```
  ci2: BINARY;
ENDCHIP;
/* Board variables */
E_BOARD
SCALARS
  GLOB_COUNT: REAL;
IMAGES
  input: BYTE;
  display: BINARY;
ENDBOARD;
/* Template list */
OPERATIONS FROM texture_2.tms;
PROCESS texture_2;
USE (tx_hclc1);
  HostLoadPic(in, input);
  SwSetTimeStep (TIMESTEP);
  ci1:=input;
  tx_hclc1 (ci1, ci1, ci2, RUNTIME_3, ZEROFLUX);
  display:=ci2;
  HostDisplay(display, ONE);
ENDPROCESS;
ENDPROG;
```

VERTICAL WING ENDINGS DETECTION OF AIRPLANE-LIKE OBJECTS [51]

Short Description

One of the characteristic features of objects, which the human recognition is based on, is the local curvature. This algorithm detects property, namely, the locations of a binary image where the local edges are convex from north. By this method, for example, the wing endings of an airplane can be detected.

In the first part of the algorithm local shadows are created with appropriate templates in the image into the 35°, 65°, 125°, 155°, -155°, -115°, -65° and -25° directions. The generation of shadows depends on the local curvature of edges. As a result we get eight images. Then four by four, groups of images get selected from the eight images and the logic **AND** operation of four images (within each group) is performed. With this step we can enhance the direction selectivity of the fill operation. In the next step we take the logic difference of each of these images and the original image. Then the undesired arc locations (the orientations of these locations are orthogonal to the preferred one) are subtracted from the resulting images. This way we get two images containing patches which denote the possible wing endings. In the last phase shadows are created, starting from these patches into appropriate directions according to the direction represented by the patch. The logic **AND** of the two shadow images and the arc location images one by one yields two images whose union gives the final result.

Some incorrectly detected points can be seen on the resulting image. More sophisticated subtracting and shadowing methods are able to remove these points. The algorithm is invariant to small rate rotation and distortion.

Templates used in the algorithm

FILL35:

A = | 1 | 0 | 1 |
|---|---|---|
| 0 | 2 | 0 |
| 1 | 1 | 0 |

B = | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 2

FILL65:

A = | 1 | 0 | 0 |
|---|---|---|
| 1 | 2 | 0 |
| 0 | 0 | 2 |

B = | 0 | 2 | 0 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |

z = 3

FILL125:

A = | 1 | 0 | 0 |
|---|---|---|
| 0 | 2 | 1 |
| 1 | 0 | 1 |

B = | 0 | 0 | 0 |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 2

FILL155:

A = | 0 | 0 | 2 |
|---|---|---|
| 0 | 2 | 0 |
| 1 | 1 | 0 |

B = | 0 | 0 | 0 |
|---|---|---|
| 2 | 0 | 0 |
| 0 | 0 | 0 |

z = 3

FILL-155:

A = | 0 | 1 | 1 |
| 0 | 2 | 0 |
| 1 | 0 | 1 |

B = | 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 2

FILL-115:

A = | 2 | 0 | 0 |
| 0 | 2 | 1 |
| 0 | 0 | 1 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 2 | 0 |

z = 3

FILL-65:

A = | 1 | 0 | 1 |
| 1 | 2 | 0 |
| 0 | 0 | 1 |

B = | 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

z = 2

FILL-25:

A = | 0 | 1 | 1 |
| 0 | 2 | 0 |
| 2 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 2 | 0 |

z = 3

SHADOW90:

A = | 0 | -1 | 0 |
| 0.3 | 2 | 0.3 |
| 0.4 | 1 | 0.4 |

B = | 0 | 0 | 0 |
| 0 | 1.4 | 0 |
| 0 | 0 | 0 |

z = 2.5

SHADOW270:

A = | 0.4 | 1 | 0.4 |
| 0.3 | 2 | 0.3 |
| 0 | -1 | 0 |

B = | 0 | 0 | 0 |
| 0 | 1.4 | 0 |
| 0 | 0 | 0 |

z = 2.5

LOGANDN:

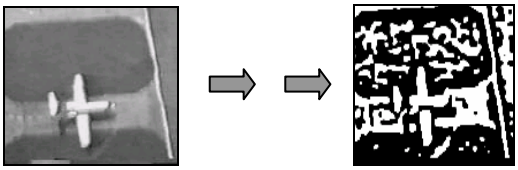
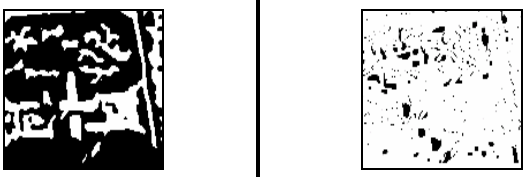
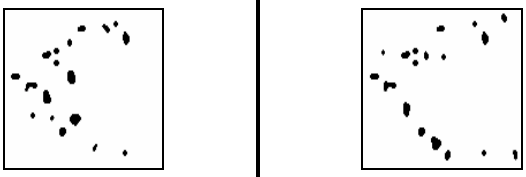
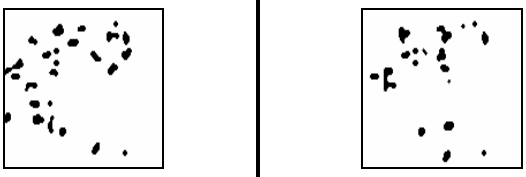
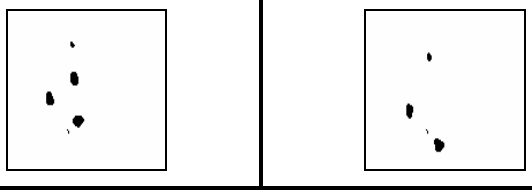
A = | 0 | 0 | 0 |
| 0 | 2 | 0 |
| 0 | 0 | 0 |

B = | 0 | 0 | 0 |
| 0 | -1 | 0 |
| 0 | 0 | 0 |

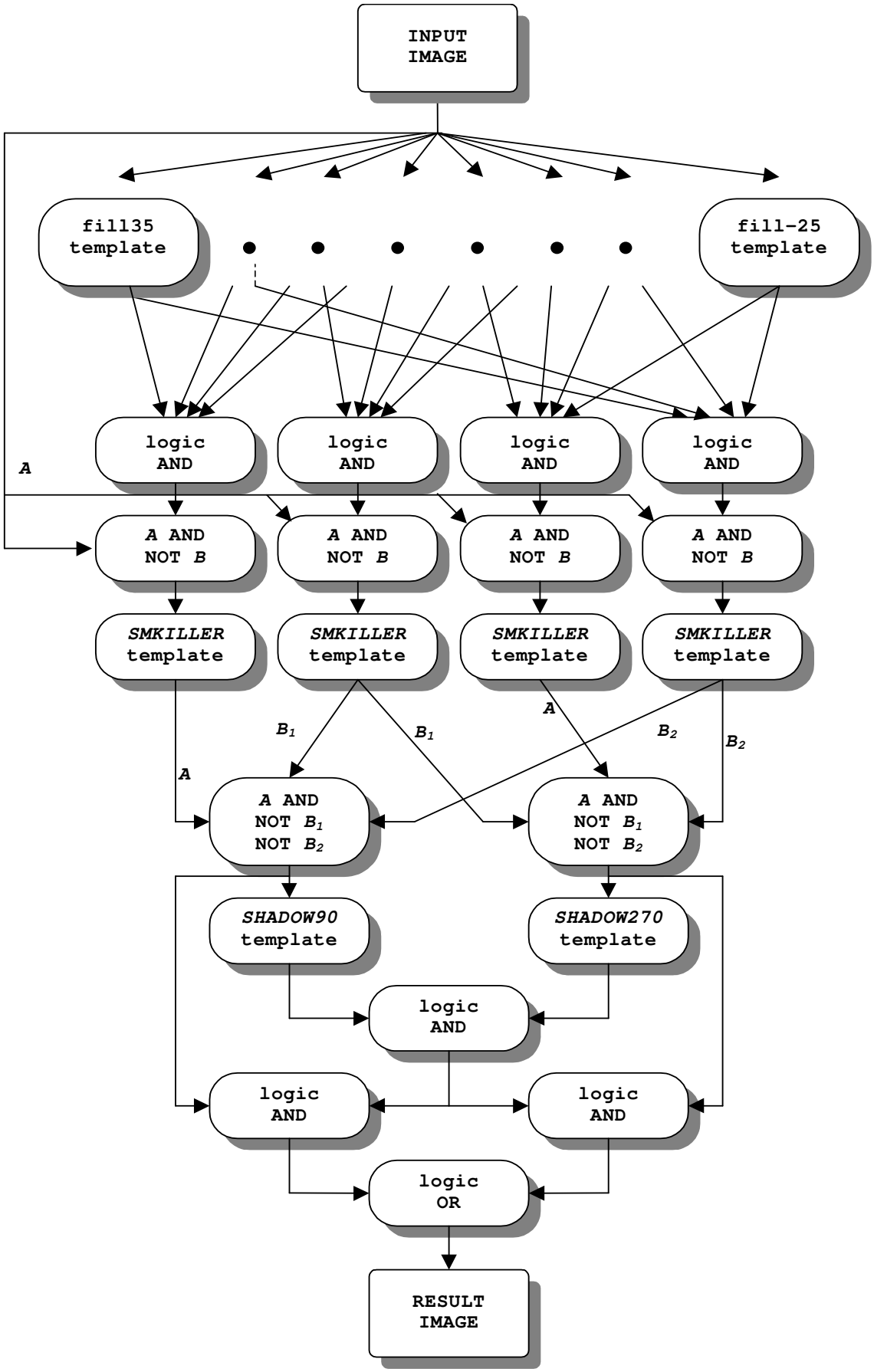
z = -1

The other templates used in the algorithm are available in this library.

Typical Example

Airplane wing endings detection	
Input image	
Concavities into all eight directions (only the first 4 are displayed)	
*Make logic **AND** of 4 subsequent images (only the first 4 are displayed). Logic difference to the original image*	
*The result of **SMKILLER** template containing the desired concave locations*	
*The result of **SMKILLER** template containing the locations which are orthogonal to the desired concave locations*	
*After subtracting (logic **AND**) the orthogonal directions*	
The result (masked with the original)	

Flow-diagram of the algorithm



ALPHA source

```
/* airplane.ALF                                    */
/* Performs airplane wing ending detection */

PROGRAM airplane(in; out);
CONSTANT
ONE = 1;
TWO = 2;
THREE = 3;
FOUR = 4;
FIVE = 5;
SIX = 6;
SEVEN = 7;
EIGHT = 8;
NINE = 9;
SIXTY = 60;
WHITE = -1.0;
TIME = 25;
TEN =10;
TS =0.9;
ENDCONST;

/* Chip definiton section                          */
CHIP_SET simulator.eng;
A_CHIP
SCALARS

IMAGES
input: BINARY;          /* input */
arc35: BINARY;          /* arc */
arc65: BINARY;          /* arc */
arc125: BINARY;         /* arc */
arc155: BINARY;         /* arc */
arc_35: BINARY;         /* arc */
arc_65: BINARY;         /* arc */
arc_125: BINARY;        /* arc */
arc_155: BINARY;        /* arc */
wing_up: BINARY;
wing_left: BINARY;
wing_down: BINARY;
wing_right: BINARY;
shadow_up: BINARY;
shadow_down: BINARY;
shadow_intrsct: BINARY;
output: BINARY;
ENDCHIP;

/* Chip set definition section                     */
E_BOARD
SCALARS
```

```
IMAGES
ENDBOARD;

/* Definition of analog operation symbol table                    */
OPERATIONS FROM airplane.tms;

PROCESS airplane;
USE (fill35, fill65, fill125, fill155, fill_115, fill_65, fill_25, fill_155, logdif, smkiller, shadow0,
shadow90, shadow180, shadow270);
SwSetTimeStep(TS);
HostLoadPic(in, input);
HostDisplay(input, ONE);

/* Filling into different directions */
fill35 ( input , input , arc35 , TIME , WHITE );
fill65 (input,input,arc65, TIME, WHITE);
fill125 (input,input,arc125, TIME, WHITE);
fill155 (input,input,arc155, TIME, WHITE);
fill_25 (input,input,arc_35, TIME, WHITE);
fill_65 (input,input,arc_65, TIME, WHITE);
fill_115 (input,input,arc_125, TIME, WHITE);
fill_155 (input,input,arc_155, TIME, WHITE);

/* Enhance direction selectivity to degree 90*/
wing_up:=     arc35 AND arc65;
wing_up:= wing_up AND arc125;
wing_up:= wing_up AND arc155;
logdif (input,wing_up,wing_up, TWO, WHITE);
smkiller (wing_up,wing_up,wing_up, TEN, WHITE);
HostDisplay(wing_up, THREE);

/* Enhance direction selectivity to degree 180 */
wing_left:=    arc125 AND arc155;
wing_left:= wing_left AND arc_125;
wing_left:= wing_left AND arc_155;
logdif (input,wing_left,wing_left, TWO, WHITE);
smkiller (wing_left,wing_left,wing_left, TEN, WHITE);
HostDisplay(wing_left, FOUR);

/* Enhance direction selectivity to degree 270*/
wing_down:= arc_35 AND arc_65 ;
wing_down:= wing_down AND arc_125;
wing_down:= wing_down AND arc_155;
logdif (input,wing_down,wing_down, TWO, WHITE);
smkiller (wing_down,wing_down,wing_down,TEN, WHITE);
HostDisplay(wing_down, FIVE);

/* Enhance direction selectivity to degree 0*/
wing_right:=  arc_35 AND arc_65;
wing_right:= wing_right AND arc35;
wing_right:= wing_right AND arc65;
```

```
logdif (input,wing_right,wing_right, TWO, WHITE);
smkiller (wing_right,wing_right,wing_right, TEN, WHITE);
HostDisplay(wing_right, TWO);

/* Removing of horizontally directed arcs*/
wing_up:= wing_up AND1NOT2 wing_right;
wing_up:= wing_up AND1NOT2 wing_left;
smkiller (wing_up,wing_up,wing_up, TEN, WHITE);
/* Shadow generation to degree 90*/
shadow90(wing_up, wing_up, shadow_up, SIXTY, WHITE );
HostDisplay(shadow_up, SIX);

/* Removing of horizontally directed arcs*/
wing_down:= wing_down AND1NOT2 wing_right;
wing_down:= wing_down AND1NOT2 wing_left;
smkiller (wing_down,wing_down,wing_down, TEN, WHITE);

/* Shadow generation to degree 270*/
shadow270(wing_down,wing_down,shadow_down, SIXTY, WHITE);
HostDisplay(shadow_down, SEVEN);

/* Distance classification*/
shadow_intrsct:=shadow_up AND shadow_down;
HostDisplay(shadow_intrsct, EIGHT);

/* Resulting wing endings (up and down)*/
output:=wing_up OR wing_down;
output:=output AND shadow_intrsct;
HostDisplay(output, NINE);

ENDPROCESS;
ENDPROG;
```

PEDESTRIAN CROSSWALK DETECTION [75]

Mihály Radványi

This algorithm detects the possible location of a pedestrian crosswalk in an image, and based estimates the confidence value of a crosswalk being present in the image.

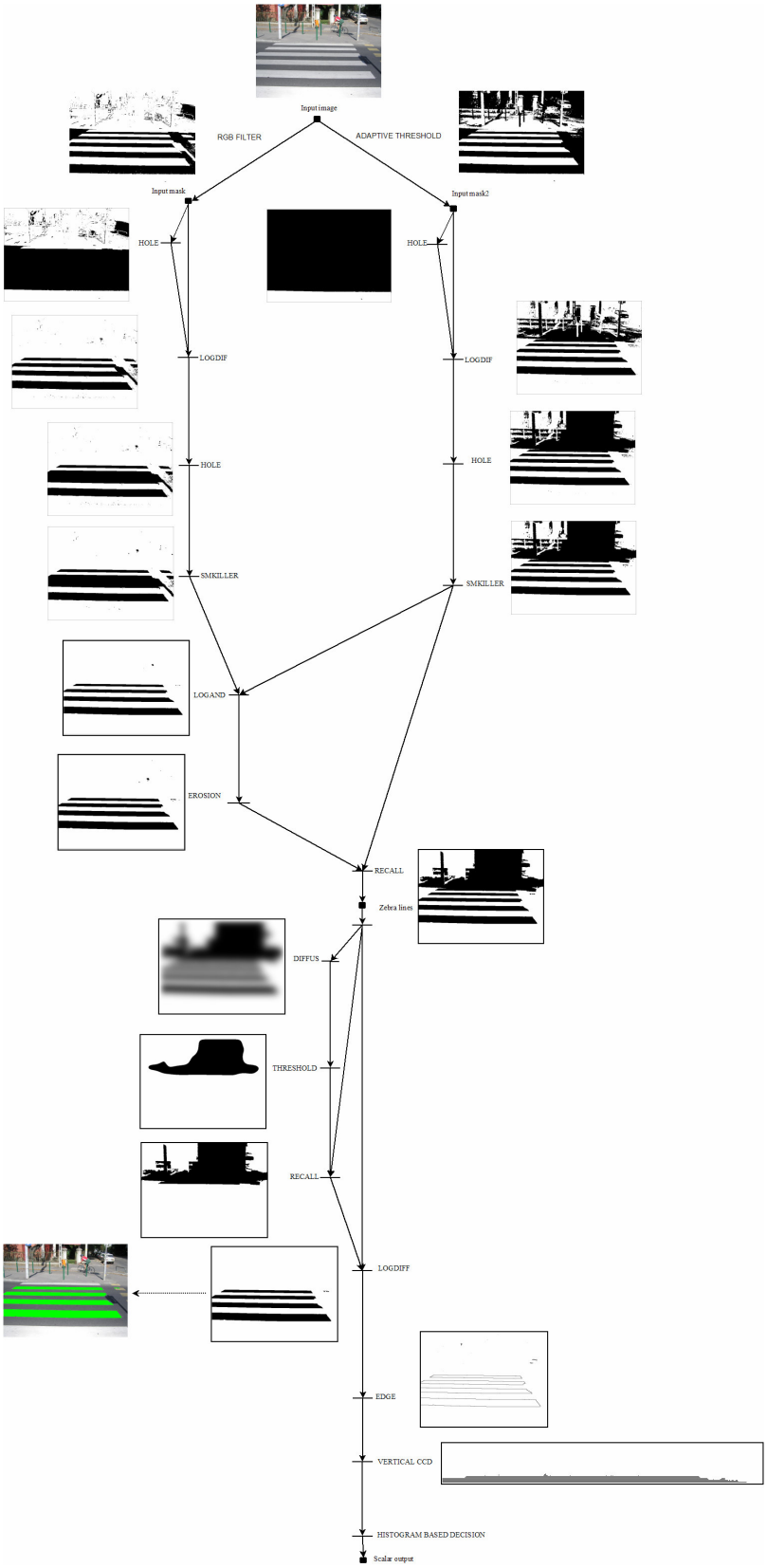
Input Parameters

U	Input image
RGB filter intervals	Possible RGB values of the road surface

Output Parameters

Y	Crosswalk confidence value

UMF diagram


Input image
RGB FILTER
ADAPTIVE THRESHOLD
Input mask
Input mask2
HOLE
HOLE
LOGDIF
LOGDIF
HOLE
HOLE
SMKILLER
SMKILLER
LOGAND
EROSION
RECALL
Zebra lines
DIFFUS
THRESHOLD
RECALL
LOGDIFF
EDGE
VERTICAL CCD
HISTOGRAM BASED DECISION
Scalar output

Chapter 3. IMPLEMENTATION ON PHYSICAL CELLULAR MACHINE

3.1. ARCHITECURE DEFINITIONS

Optimal implementation methods of 2D operators are completely different on different architectures. On the other hand, the implementation efficiency of the different operator on different architectures varies drastically. A large number of different 2D operators and complex algorithms have been described in the previous chapters of this book. In this chapter, their implementation methods and efficiency will be analyzed on different architectures. We are considering the following architectures:

- DSP-memory architecture (in particular DaVinci processors from TI [93]) as a reference;
- Pass-through architecture (CASTLE [99][98], Falcon [91], C-MVA [96]);
- Coarse-grain cellular parallel architecture (Xenon [100]);
- Fine-grain fully parallel cellular architecture with mixed-mode processing (SCAMP [90], Q-Eye [94]);
- Fine-grain fully parallel cellular architecture with continuous time processing (ACE-16k [84], ACLA [87][88]).
- Multi-core inhomogeneous array computing architecture with high-performance kernels (CELL [92]);
- Many-core hierarchical graphic processor unit (GPU [104][105]).

The chapter will be organized as follows. First, seven different basic architectures are briefly described. Then, the operators are grouped according to their execution methods on the different architectures. It is followed by the analysis of the implementation. Finally, an architecture selection guide is shown.

Classic DSP-memory architecture

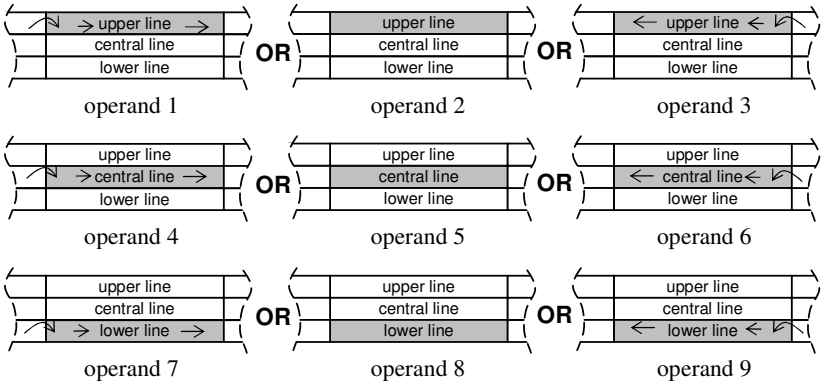
Here we assume 32 bit DSP architecture with cache memory large enough to store the required number of images and the program internally. In this way, we have to practically estimate/measure the required DSP operations. Most of the modern DSPs have numerous MACs and ALUs. To avoid comparing these DSP architectures, which would lead too far from our original topic, we use the DaVinci video processing DSP by Texas Instrument, as a reference.

We use 3×3 convolution as a measure of grayscale performance. The data requirements of the calculation are 19 bytes (9 pixels, 9 kernel values, result), however, many of these data can be stored in registers, hence, only an average of a four-data accesses (3 inputs, because the 6 other ones have already been accessed for the previous pixel position, and one output) is needed for each convolution. From a computational point of view, it needs 9 multiple-add (MAC) operations. It is very typical that the 32 bit MACs in a DSP can be split into four 8 bit MACs, and other auxiliary ALUs help loading the data to the registers in time. Measurement shows that, for example, the Texas DaVinci family with the TMS320C64x core needs only about 1.5 clock cycles to complete a 3×3 convolution.

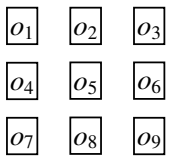
The operands of the binary operations are stored in 1 bit/pixel format, which means that each 32bit word represents a 32×1 segment of an image. Since the DSP's ALU is a 32 bit long unit, it can handle 32 binary pixels in a single clock cycle. As an example, we examine how a 3×3 square shaped erosion operation is executed. In this case erosion is a nine input OR operation where the inputs are the binary pixels values within the 3×3 neighborhood. Since the ALU of the DSP does not contain 9 input OR gate, it is executed sequentially on 32 an entire

32×1 segment of the image. The algorithm is simple: the DSP has to prepare the 9 different operands, and apply bit-wise OR operations on them.

Figure 1 shows the generation method of the first three operands. In the figure a 32×3 segment of a binary image is shown (9 times), as it is represented in the DSP memory. Some fractions of horizontal neighboring segments are also shown. The first operand can be calculated by shifting the upper line with one bit position to the left and filling in the empty MSB with the LSB of the word from its right neighbor. The second operand is the un-shifted upper line. The position and the preparation of the remaining operands are also shown in Figure *1*a.



(a)



(b)

$e_1 = o_1$ OR o_2 OR o_3 OR o_4 OR o_5 OR o_6 OR o_7 OR o_8 OR o_9

(c)

Figure 1. Illustration of the binary erosion operation on a DSP. (a) shows the 9 pieces of 32×1 segments of the image (operands), as the DSP uses them. The operands are the shaded segments. The arrows indicate shifting of the segments. To make it clearer, consider a 3×3 neighborhood as it is shown in (b). $o_{1,}$ $o_{2,\ \ldots}$ o_9 are the operands. The DSP does the same, but on 32 pixels parallel.

This means that we have to apply 10 memory accesses, 6 shifts, 6 replacements, and 8 OR operations to execute a binary morphological operation for 32 pixels. Due to the multiple cores and the internal parallelism, the Texas DaVinci spends 0.5 clock cycles with the calculation of one pixel.

In the low power low cost embedded DSP technology the trend is to further increase the clock frequency, but most probably, not higher than 1 GHz, otherwise, the power budget cannot be kept. Moreover, the drawback of these DSPs is that their cache memory is too small, which cannot be increased significantly without significant cost increase. The only way to significantly increase the speed is to implement a larger number of processors, however, that requires a new way of algorithmic thinking, and software tools.

The DSP-memory architecture is the most versatile from the point of views of both in functionality and programmability. It is easy to program, and there is no limit on the size of the processed images, though it is important to mention that in case of an operation is executed on an image stored in the external memory, its execution time is increasing roughly with an order of magnitude. Though the DSP-memory architecture is considered to be very slow, as it is shown

later, it outperforms even the processor arrays in some operations. In QVGA frame size, it can solve quite complex tasks, such as video analytics in security applications on video rate [95]. Its power consumption is in the 1-3W range. Relatively small systems can be built by using this architecture. The typical chip count is around 16 (DSP, memory, flash, clock, glue logic, sensor, 3 near sensor components, 3 communication components, 4 power components), while this can be reduced to the half in a very basic system configuration.

Pass-through architectures

The basic idea of this pass-through architecture is to process the images line-by-line, and to minimize both the internal memory capacity and the external IO requirements. Most of the early image processing operations are based on 3×3 neighborhood processing, hence 9 image data are needed to calculate each new pixel value. However, these 9 data would require very high data throughput from the device. As we will see, this requirement can be significantly reduced by applying a smart feeder arrangement.

Figure 2 shows the basic building blocks of the pass-through architecture. It contains two parts, the memory (feeder) and the neighborhood processor. Both the feeder and the neighborhood processor can be configured 8 or 1 bit/pixel wide, depending on whether the unit is used for grayscale or binary image processing. The feeder contains, typically, two consecutive whole rows and a row fraction of the image. Moreover, it optionally contains two more rows of the mask image, depending on the input requirements of the implemented neighborhood operator. In each pixel clock period, the feeder provides 9 pixel values for the neighborhood processor and the mask value optionally if the operation requires it. The neighborhood processor can perform convolution, rank order filtering, or other linear or nonlinear spatial filtering on the image segment in each pixel clock period. Some of these operators (e.g., *hole finder*, or a *CNN* emulation with A and B templates) require two input images. The second input image is stored in the mask. The outputs of the unit are the resulting and, optionally, the input and the mask images. Note that the unit receives and releases synchronized pixels flows sequentially. This enables to cascade multiple pieces of the described units. The cascaded units form a chain. In such a chain, only the first and the last units require external data communications, the rest of them receives data from the previous member of the chain and releases the output towards the next one.

An advantageous implementation of the row storage is the application of FIFO memories, where the first three positions are tapped to be able to provide input data for the neighborhood processor. The last positions of rows are connected to the first position of the next row (Figure 2). In this way, pixels in the upper rows are automatically marching down to the lower rows.

The neighborhood processor is of special purpose, which can implement one or a few different kinds of operators with various attributes and parameter. They can implement convolution, rank-order filters, grayscale or binary morphological operations, or any local image processing functions (e.g. Harris corner detection, Laplace operator, gradient calculation, etc,). In architectures CASTLE [99][98] and Falcon [91], e.g., the processors are dedicated to convolution processing where the template values are the attributes. The pixel clock is matched with that of the applied sensor. In case of a 1 megapixel frame at video rate (30 FPS), the pixel clock is about 30 MHz (depending on the readout protocol). This means that all parts of the unit should be able to operate at least at this clock frequency. In some cases the neighborhood processor operates on an integer multiplication of this frequency, because it might need multiple clock cycles to complete a complex calculation, such as a 3×3 convolution. Considering ASIC or FPGA implementations, clock frequency between 100-300 MHz is a feasible target for the neighborhood processors within tolerable power budget.

The multi-core pass-through architecture is built up from a sequence of such processors. The processor arrangement follows the flow-chart of the algorithm. In case of multiple iterations of the same operation, we need to apply as many processor kernels, as many iterations we need.

This easily ends up requiring a few dozens of kernels. Fortunately, these kernels, especially in the black-and-white domain, are relatively inexpensive, either on silicon, or in FPGA.
Depending on the application, the data-flow may contain either sequential segments or parallel branches. It is important to emphasize, however, that the frame scanning direction cannot be changed, unless the whole frame is buffered, which can be done in external memory only. This introduces a relatively long (dozens of millisecond) additional latency.

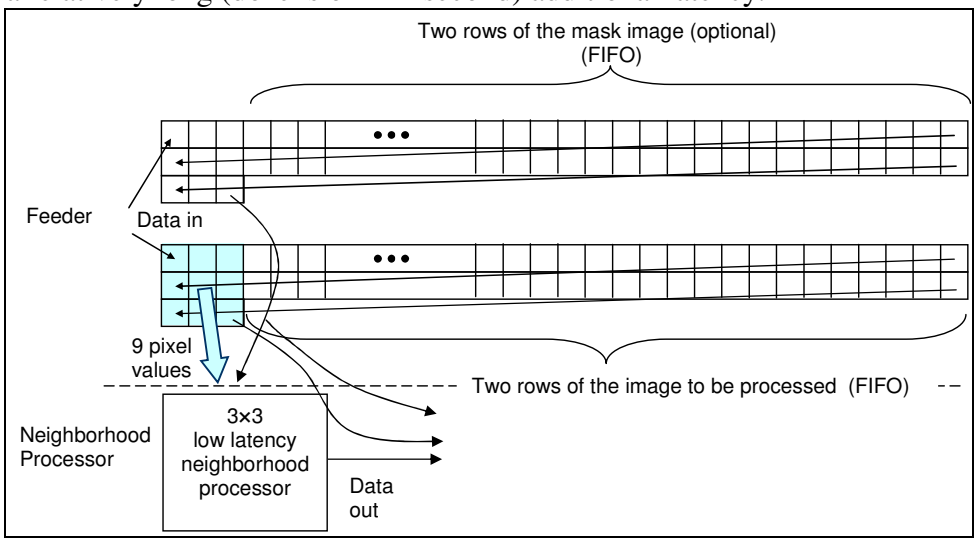


Figure 2. One processor and its memory arrangement in the pass-through architecture.

For capability analysis, here we use the Spartan 3ADSP FPGA (XC3SD3400A) from Xilinx as a reference, because this low-cost, medium performance FPGA was designed especially for embedded image processing. It is possible to implement roughly 120 grayscale processors within this chip, as long as the image row length is below 512, or 60 processors, when the row length is between 512 and 1024.

Coarse-grain cellular parallel architectures

The coarse-grain architecture is a locally interconnected 2D cellular processor arrangement, as opposed to the pass-through one. A specific feature of the coarse-grain parallel architectures is that each processor cell is topographically assigned to a number of pixels (e.g., an 8×8 segment of the image), rather than to a single pixel only. Each cell contains a processor and some memory, which is large enough to store few bytes for each pixel of the allocated image segment. Exploiting the advantage of the topographic arrangement, the cells can be equipped with photo sensors enabling to implement a single chip sensor-processor device. However, to make this sensor sensitive enough, which is the key in high frame-rate applications, and to keep the pixel density of the array high, at the same time, certain vertical integration techniques are needed for photosensor integration.
In the coarse-grain architectures, each processor serves a larger number of pixels, hence we have to use more powerful processors, than in the one-pixel per processor architectures. Moreover, the processors have to switch between serving pixels frequently, hence more flexibility is needed that an analog processor can provide. Therefore, it is more advantageous to implement 8 bit digital processors, while the analog approach is more natural in the one pixel per processor (fine-grain) architectures. (See the next subsection.)
As an example for the coarse-grain architecture, we briefly describe the Xenon chip [100]. As can be seen in Figure 3, Xenon chip [100] is constructed of an 8×8, locally interconnected cell arrangement. Each cell contains a sub-array of 8×8 photosensors; an analog multiplexer; an 8 bit

AD converter; an 8 bit processor with 512 bytes of memory; and a communication unit of local and global connections. The processor can handle images in 1, 8, and 16 bit/pixel representations, however, it is optimized for 1 and 8 bit/pixel operations. Each processor can execute addition, subtraction, multiplication, multiply-add operations, comparison, in a single clock cycle on 8 bit/pixel data. It can also perform 8 logic operations on 1 bit/pixel data in packed-operation mode in a single cycle. Therefore, in binary mode, one line of the 8×8 sub-array is processed jointly, similarly to the way we have seen in the DSP. However, the Xenon chip supports the data shifting and swapping from hardware, which means that the operation sequence, what we have seen in Figure 1 takes 9 clock cycles only. (The swapping and the accessing the memory of the neighbors do not need extra clock cycles.) Besides, the local processor core functions, Xenon can also perform a global OR function. The processors in the array are driven in a single instruction multiple data (SIMD) mode.

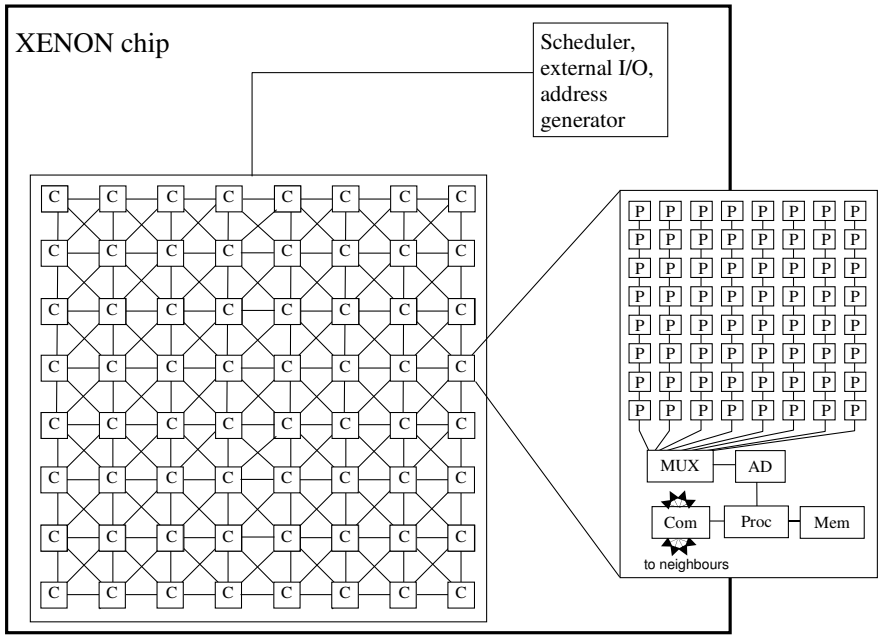


Figure 3. Xenon is a 64 core coarse-grain cellular parallel architecture (C stands for processor cores, while P represents pixels).

Xenon is implemented on a 5x5mm silicon die with 0.18 micron technology. The clock cycle can go up to 100MHz. The layout is synthesized, hence the resulting 75micron equivalent pitch is far from being optimal. It is estimated that through aggressive optimization it could be reduced to 40 micron (assuming a bump bonded sensor layer), which would almost double the resolution achievable on the same silicon area. The power consumption of the existing implementation is under 20mW.

Fine-grain fully parallel cellular architectures with discrete time processing

The fine-grain, fully parallel architectures are based on rectangular processor grid arrangements where the 2D data (images) are topographically assigned to the processors. The key feature here is that for the fine-grain arrangement there is a one-to-one correspondence between the pixels and the processors. This certainly means that at the same time the composing processors must be simpler and less powerful, than in the previous, coarse-grain case. Therefore, fully parallel architectures are typically implemented in analog domain, though bit-sliced digital approach is also feasible.

In the discussed cases, the discrete time processing type fully parallel architectures are equipped with a general purpose, analog processor, and an optical sensor in each cell. These sensor-

processors can handle two types of data (image) representations: grayscale and binary. The instruction set of these processors include addition, subtraction, scaling (with a few discrete factors only), comparison, thresholding, and logic operations. Since it is a discrete time architecture, the processing is clocked. Each operation takes 1-4 clock cycles. The individual cells can be masked. Basic spatial operations, such as convolution, median filtering, or erosion, can be put together as sequences of these elementary processor operations. In this way the clock cycle counts of a convolution, a rank order filtering, or a morphologic filter are between 20 and 40 depending on the number of weighting coefficients.

It is important to note that in case of the discrete time architectures (both coarse- and fine-grain), the operation set is more elementary (lower level) than on the continuous time cores (see the next section). While in the continuous time case (CNN like processors) the elementary operations are templates (convolution, or feedback convolution) [77][78], in the discrete time case, the processing elements can be viewed as RISC (reduced instruction set) processor cores with addition, subtraction, scaling, shift, comparison, and logic operations. When a full convolution is to be executed, the continuous time architectures are more efficient. In the case of operations when both architectures apply a sequence of elementary instructions in an iterative manner (e.g., rank order filters), the RISC is the superior, because its elementary operators are more versatile more accurate, and faster.

The internal analog data representation has both architectural and functional advantages. From architectural point of view, the most important feature is that no AD converter is needed on the cell level, because the sensed optical image can be directly saved in the analog memories, leading to significant silicon space savings. Moreover, the analog memories require smaller silicon area than the equivalent digital counterparts. From the functional point of view, the topographic analog and logic data representations make the implementation of efficient diffusion, averaging, and global OR networks possible.

The drawback of the internal analog data representation and processing is the signal degradation during operation or over time. According to experience, accuracy degradation was more significant in the old ACE16k design [84] than in the recent Q-Eye [94] or SCAMP [90] ones. While in the former case 3-5 grayscale operations led to significant degradations, in the latter ones even 10-20 grayscale operations can conserve the original image features. This makes it possible to implement complex nonlinear image processing functions (e.g., rank order filter) on discrete time architectures, while it is practically impossible on the continuous ones (ACE16k).

The two representatives of discrete time solutions, SCAMP and Q-Eye, are slightly similar in design. The SCAMP chip was fabricated by using 0.35 micron technology. The cell array size is 128×128. The cell size is 50×50 micron, and the maximum power consumption is about 200mW at 1.25MHz clock rate. The array of Q-Eye chip has 144×176 cells. It was fabricated on 0.18 micron technology. The cell size is about 30×30 micron. Its speed and power consumption range is similar to that of the SCAMP chip. Both SCAMP and Q-Eye chips are equipped with single-step mean, diffusion, and global OR calculator circuits. Q-Eye chip also provides hardware support for single-step binary 3×3 morphologic operations.

Fine-grain fully parallel cellular architecture with continuous time processing

Fully parallel cellular continuous time architectures are based on arrays of spatially interconnected dynamic asynchronous processor cells. Naturally, these architectures exhibit fine-grain parallelism, to be able to perform continuous time spatial waves physically in the continuous value electronic domain. Since these are very carefully optimized, special purpose circuits, they are super-efficient for computations they were designed to perform. We have to emphasize, however, that they are not general purpose image processing devices. Here we mainly focus on two designs. Both of them can generate continuous time spatial-temporal propagating

waves in a programmable way. While the output of the first one (ACE-16k [84]) can be in the grayscale domain, the output of the second one (ACLA [87][88]) is always in the binary domain.
The ACE-16k [84] is a classical CNN Universal Machine type architecture equipped with feedback and feed-forward template matrices [78], sigmoid type output characteristics, dynamically changing state, optical input, local (cell level) analog and logic memories, local logic, diffusion and averaging network. It can perform full-signal range type CNN operations [79]. Therefore, it can be used in retina simulations or other spatial-temporal dynamical system emulations, as well. Its typical feed-forward convolution execution time is in the 5-8 microsecond range, while the wave propagation speed from cell-to-cell is up to 1 microsecond. Though its internal memories, easily re-programmable convolution matrices, logic operations, and conditional execution options make it attractive to use as a general purpose high-performance sensor-processor chip for the first sight, its limited accuracy, large silicon area occupation (~80×80 micron/cell on 0.35 micron 1P5M STM technology), and high power consumption (4-5 Watts) prevent the immediate usage in various vision application areas.
The other architecture in this category is the Asynchronous Cellular Logic Array (ACLA) [87], [88]. This architecture is based on spatially interconnected logic gates with some cell level asynchronous controlling mechanisms, which allow ultra high-speed spatial binary wave propagation only. Typical binary functionalities implemented on this network are: *trigger wave, reconstruction, hole finder, shadow*, etc. Assuming more sophisticated control mechanism on the cell level, it can even perform *skeletonization* or *centroid* calculations. Their implementation is based on a few minimal size logic transistors, which makes them hyper-fast, extremely small, and power-efficient. They can reach 500 ps/cell wave propagation speed, with 0.2mW power consumption for a 128×128 sized array. Their very small area requirement (16×8 micron/cell on 0.35 micron 3M1P AMS technology) makes them a good choice to be implemented as a co-processor in any fine-grain array processor architecture.

Multi-core heterogeneous processors array with high-performance kernels (CELL)

The Cell Broadband Engine Architecture (CBEA) [92] is designed to achieve high computing performance with better area/performance and power/performance ratios than the conventional multi-core architectures. The CBEA defines a heterogeneous multi-processor architecture where general purpose processors called Power Processor Elements (PPE) and SIMD processors called Synergistic Processor Elements (SPE) are connected via a high speed on-chip coherent bus called Element Interconnect Bus (EIB) (Figure 4). The processors run maximum on 3.2GHz.
The PPE is a conventional dual-threaded 64bit PowerPC processor which can run existing operating systems without modification and can control the operation of the SPEs. To simplify processor design and achieve higher clock speed instruction reordering is not supported by the PPE. The EIB is not a bus as suggested by its name but a ring network which contains 4 unidirectional rings where two rings run counter to the direction of the other two. The frequency of these rings is half of the frequency of the processors. Each of these rings can transfer 16bits/cycle between two neighboring element on the bus. When farther elements communicate, the data is relayed by each element between the communicating ones. This increase latency, moreover the relaying elements cannot use that particular bus for their own purposes during the transfer.
The dual-channel Rambus XDR memory interface provides very high 25.6GB/s memory bandwidth to external memory. I/O devices can be accessed via two Rambus FlexIO interfaces where one of them (the Broadband Interface (BIF)) is coherent and makes it possible to directly connect two Cell processors.
Similarly to the PPE the SPEs are also in-order processors. Data for the instructions is provided by the very large 128 element register file where each register is 16byte wide. Therefore SIMD instructions of the SPE works on 16byte wide vectors for example: four single precision floating

point numbers or eight 16bit integers. The SPEs support logic operations also. They can handle up to 128 bits in one single step. The SPEs can only address their local 256kB SRAM memory, while they can access the main memory of the system by DMA instructions.

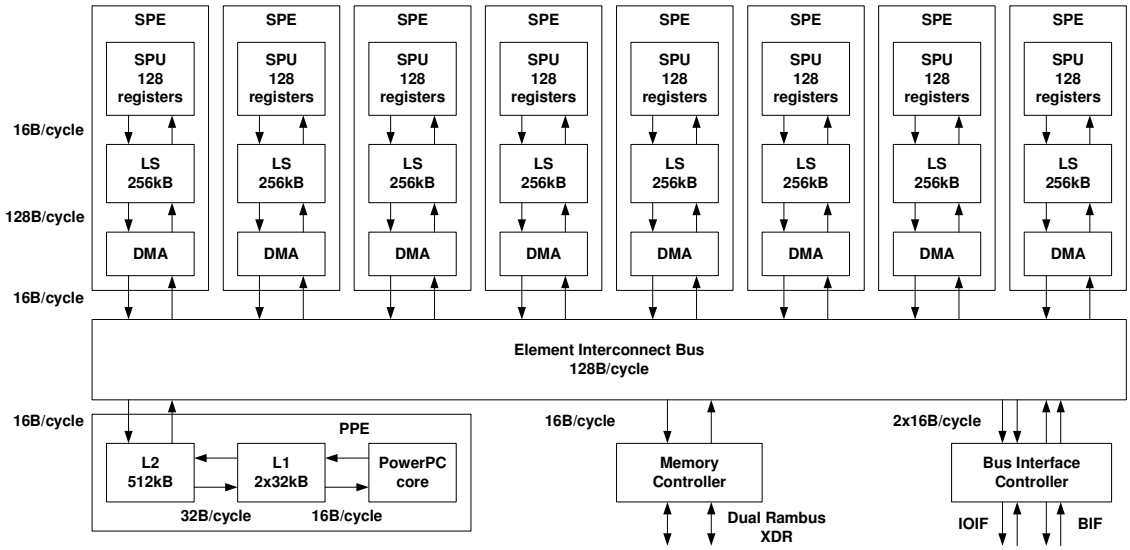


Figure 4. Block diagram of the Cell processor

Many-core hierarchical graphic processor unit (GPU)

Figure 5 shows the hardware model of NVIDIA's GPU architecture. It is constructed of groups of scalar-based processors. Eight of these processors are grouped with an Instruction Unit to form a single-instruction-multiple-data (SIMD) multiprocessor (MP). The number of MPs are varying from 1 to 16, which means 8 to 128 parallel processors. As to its external data storage, it uses 1800 MHz DDR3 RAM modules; however internal cache memories are also available to feed the large number of processors. Three ways for caching were introduced. A 16 KB per MP high-speed universal purpose local shared memory is available beside the constant (1D aware) and texture (2D aware) caching global memories. Both the constant and the texture memories are read only from the processors. I/O instructions accessing the shared memory using cache converges to 1-2 cycle while direct access to the global memories cost 100-200 clock cycles.
The GPU is operated on a way that the programmer defines very large number of simple threads (like executing a convolution on 4 consecutive pixels in a row), and these threads are distributed among the processors. The distribution (mapping) and the order of execution is not in the hand of the programmer, it is done automatically. Hence, we cannot analyze different mapping strategies here. Rather than that, we will show the execution speed what we could reach in different 2D operator classes.

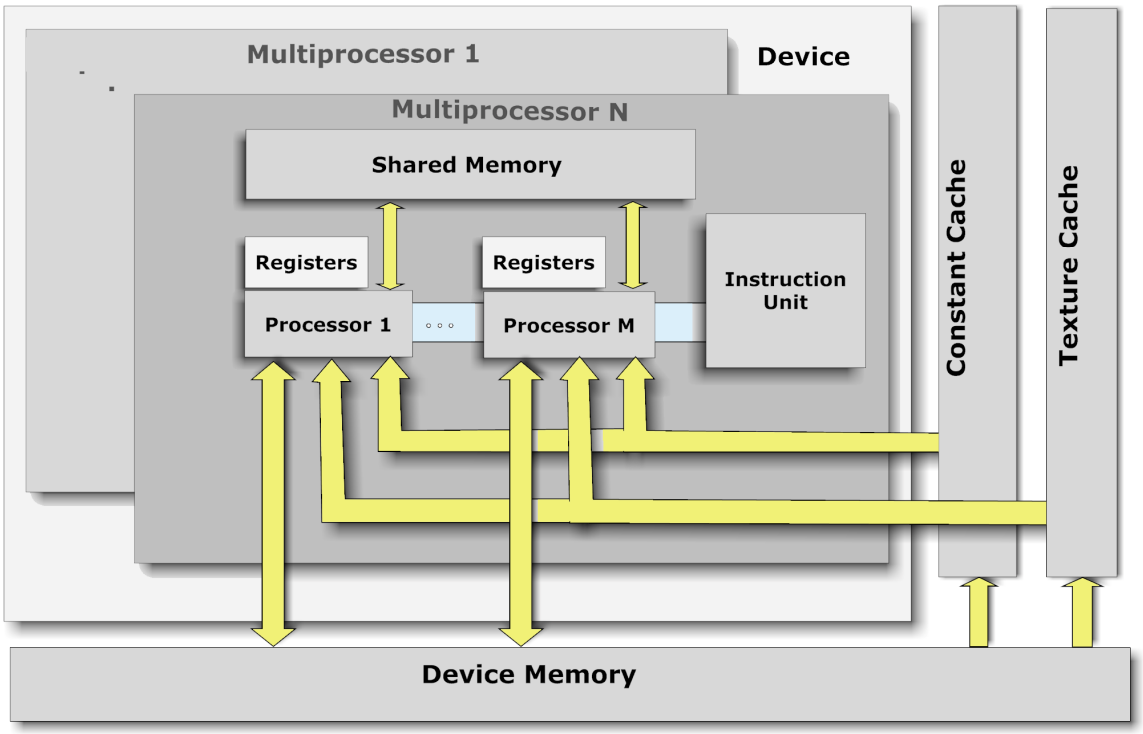


Figure 5. Block diagram of the GPU processor

3.2. IMPLEMENTATION AND EFFICIENCY ANALYSIS OF VARIOUS OPERATORS

Based on the implementation methods, in this section, we introduce a new 2D operator categorization. Then, the implementation methods on different architectures are described and analyzed from the efficiency aspect.
Here we examine only the 2D single-step neighborhood operators, and the 2D, neighborhood based wave type operators. The more complex, but still local operators (such as Canny edge detector) can be built up by using these primitives, while other operators (such as Hough or Fourier transform) require global processing, which is not supported by these architectures.

Categorization of 2D operators

Due to their different spatial-temporal dynamics, different 2D operators require different computational approaches. The categorization (Figure 6) was done according to their implementation methods on different architectures. It is important to emphasize that we categorize operators (functionalities) here, rather than wave types, because the wave types are not necessarily inherited by the operator itself, but rather by its implementation method on a particular architecture. As we will see, the same operator is implemented with different spatial wave dynamic patterns on different architectures. The most important 2D operators, including all the CNN operators [97]are considered here.
The first distinguishing feature is the location of active pixels [97]. If the active pixels are located along one or few one-dimensional stationary or propagating curves at a time, we call the operator front-active. If the active pixels are everywhere in the array, we call it area-active.
The common property of the front-active propagations is that the active pixels are located only at the propagating wave fronts [80]. This means that at the beginning of the wave dynamics (transient) some pixels become active, others remain passive. The initially active pixels may initialize wave fronts which start propagating. A propagating wave front can activate some

further passive pixels. This is the mechanism how the wave proceeds. However, pixels apart from a waveform cannot become active [97]. This theoretically enables us to compute only the pixels which are along the front lines, and not waste efforts on the others. The question is which are the architectures that can take advantage of such a spatially selective computation.

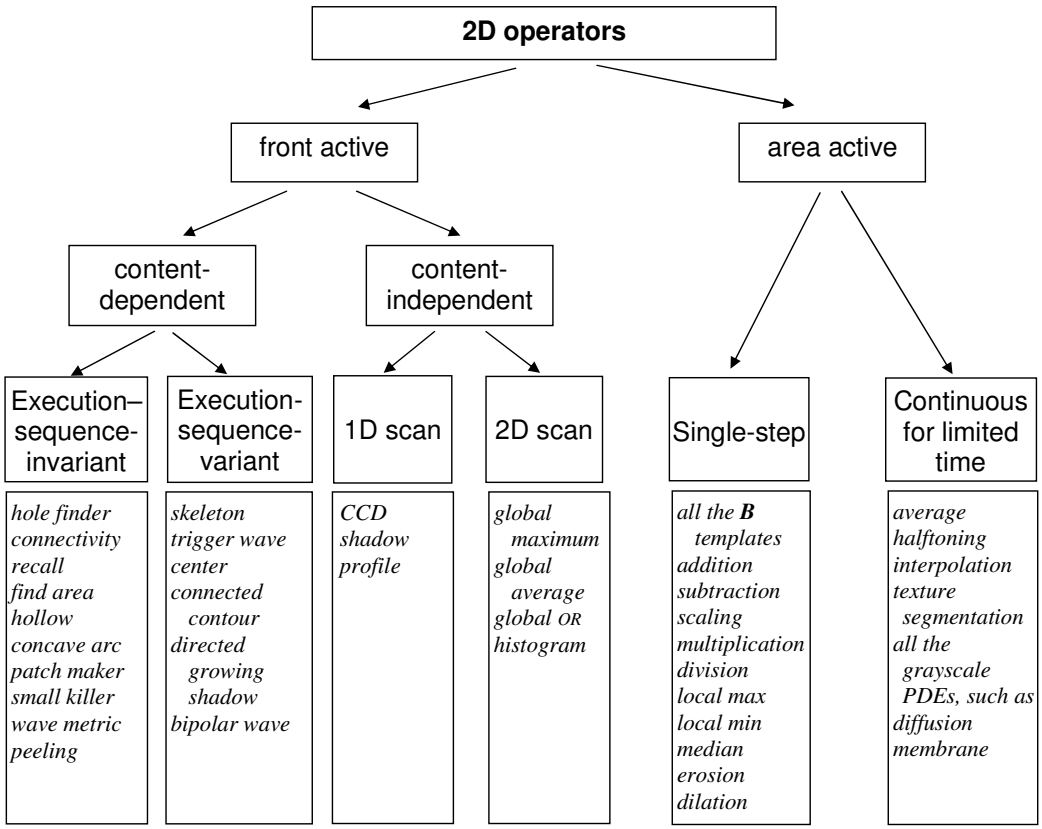


Figure 6. 2D local operator categorization

The front active operators such as *reconstruction*, *hole finder*, or *shadow* are typically binary waves. In CNN terms, they have binary inputs and outputs, positive self-feedback, and space invariant template values. Figure 6 contains three exemptions: *global max*, *global average*, and *global OR*. These functions are not wave type operators by nature; however, we will associate a wave with them which solves them efficiently.

The front active propagations can be content-dependent or content-independent. The content-dependent operator class contains most of the operators where the direction of the propagation depends on the local morphological properties of the objects (e.g., shape, number, distance, size, connectivity) in the image (e.g., *reconstruct*). An operator of this class can be further distinguished as execution-sequence-variant (*skeleton*, etc) or execution-sequence-invariant (*hole finder, recall, connectivity*, etc). In the first case the final result may depend on the spatial-temporal dynamics of the wave, while in the latter it does not. Since the content-dependent operator class contains the most interesting operators with the most exciting dynamics, they are further investigated in the next subsection.

We call the operators content-independent when the direction of the propagation and the execution time do not depend on the shape of the objects (e.g., *shadow*). According to propagation, these operators can be either one- (e.g., *CCD*, *shadow, profile [102]*) or two-dimensional (*global maximum, global OR, global average, histogram*). Content-independent operators are also called single-scan, for their execution requires a single scanning of the entire

image. Their common feature is that they reduce the dimension of the input 2D matrices to vectors (*CCD, shadow, profile, histogram*) or scalars (*global maximum, global average, global OR*). It is worth to mention that on the coarse- and fine-grain topographic array processors the *shadow, profile* and *CCD* are content-dependent operators, and the number of the iterations (or analog transient time) depends on the image content only. The operation is completed, when the output is ceased to change. Generally, however, , it is less efficient to include a test to detect a stabilized output, than to let the operator run in as many cycles as it would run in the worst case.
The area active operator category contains the operators where all the pixels are to be updated continuously (or in each iteration). A typical example is *heat diffusion*. Some of these operators can be solved in a single update of all the pixels (e.g., all the CNN **B** templates *[102]*), while others need a limited number of updates (*halftoning, constrained heat diffusion*, etc.).
The fine-grain architectures update every pixel location in fully parallel in each time instance. Therefore, the area active operators are naturally the best fit for these computing architectures.

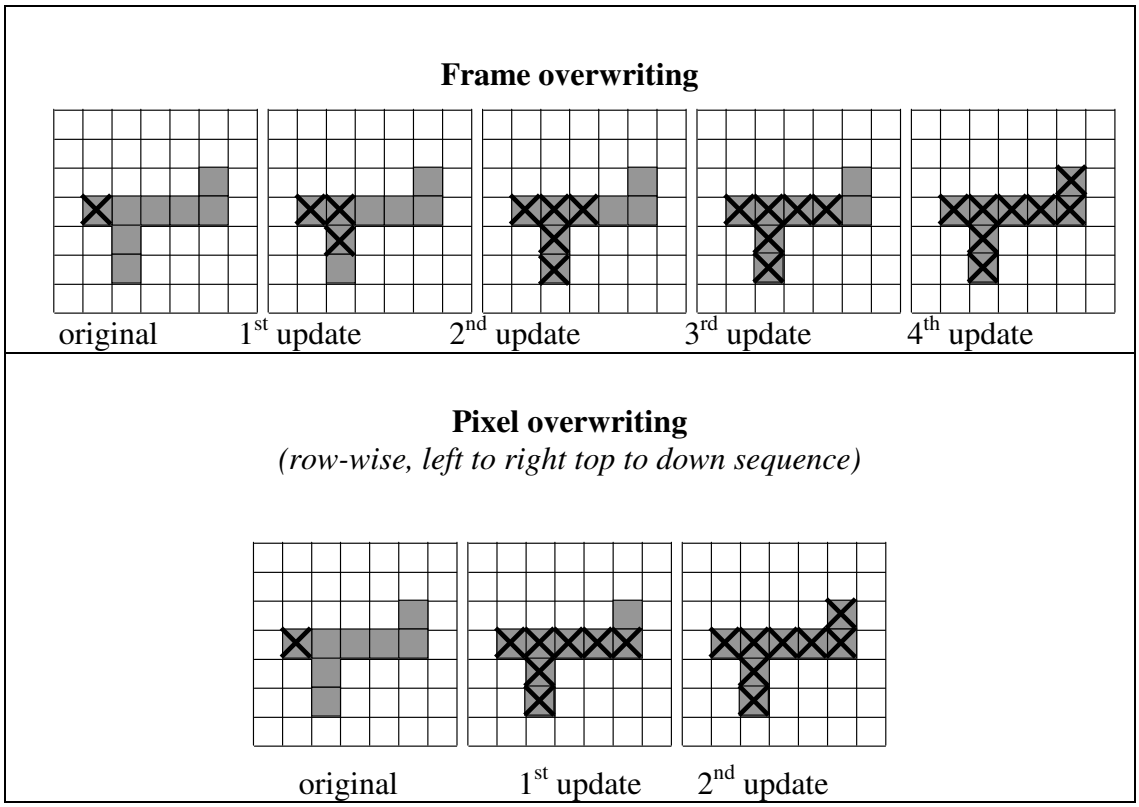


Figure 7. Execution-invariant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that the propagation starts from the marked pixel (denoted by X), and it can go on within the grey domain, proceeding one pixel in each update. In the figure, we can see the results of each update. Update means calculating the new states of all the pixels in the frame.

Execution-sequence-variant versus execution-sequence-invariant operators.

The crucial difference in fine-grain and pass-through architectures is in their state overwriting methods. In the fine-grain architecture the new states of all the pixels are calculated in parallel, and then the previous one is overwritten again in parallel, before the next update cycle is commenced. In the pass-through architecture, however, the new state is calculated pixel-wise, and it is selectable whether to overwrite a pixel state before the next pixel is calculated (pixel

overwriting), or to wait until the new state value is calculated for all the pixels in the frame (frame overwriting). In this context, update means the calculation of the new state for an entire frame. Figure 7 and Figure 8 illustrate the difference between the two overwriting schemes. In case of an execution-sequence-variant operation, the result depends on the frame overwriting schemes.
Here the calculation is done pixel-wise, left to right and row-wise top to down. As we can see, overwriting each pixel before the next pixel's state is calculated (pixel overwriting) speeds up the propagation in the directions which corresponds to the direction the calculation proceeds.
Based on the above, it is easy to draw the conclusion that the two updating schemes lead to two completely different propagation dynamics and final results in execution-variant cases. One is slower, but controlled, the other one is faster, but uncontrolled. The first can be used in cases when speed maximization is the only criterion, while the second is needed when the shape and the dynamics of the propagating wave front count. We called the former case execution-sequence-invariant operators, the latter one execution-sequence-variant operators (Figure 6).

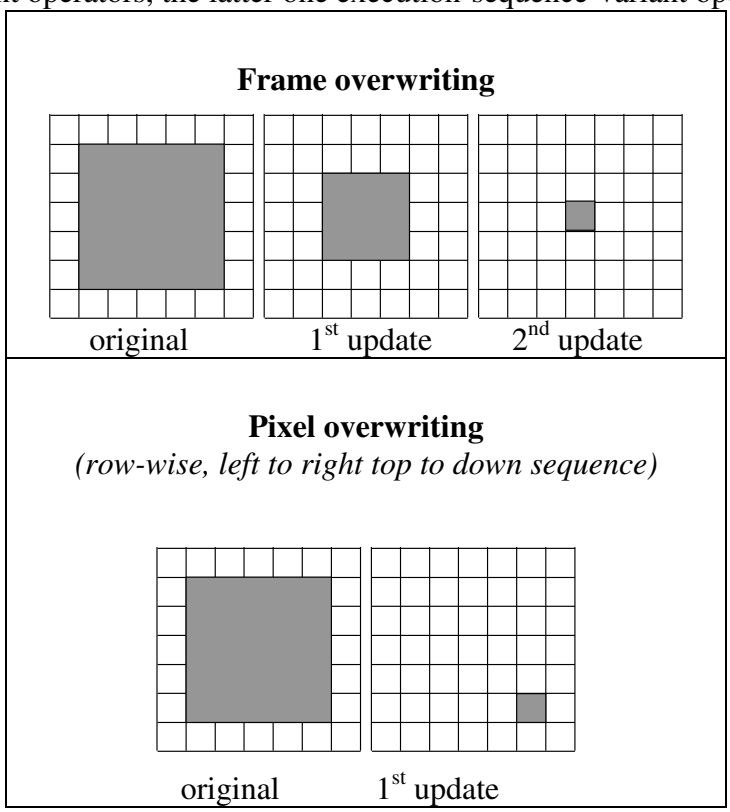


Figure 8. Execution-variant sequence in different overwriting schemes. Given an image with grey objects against white background. The propagation rule is that those pixels of the object, which has both object and background neighbor should became background. In this case, the subsequent peeling leads to find the centroid in the frame overwriting method, while it extracts one pixel of the object in the pixel overwriting mode.

In the fine-grain architecture we can use the frame overwriting scheme only. In the coarse-grain architecture both pixel overwriting and frame overwriting methods can be selected within the individual sub-arrays. In this architecture, we may determine even the calculation sequence, which enables speed-ups in different directions in different updates. Later, we will see an example to illustrate how the *hole finder* operation propagates in this architecture. In the pass-through architecture, we may decide which one to use, however, we cannot change the direction

of the propagation of the calculation, without paying a significant penalty for it in memory size and latency time.

Processor utilization efficiency of the various operation classes

In this subsection, we will analyze the implementation efficiency of various 2D operators from different aspects. We will study both the execution methods and the efficiency from the processor utilization aspect. Efficiency is a key question, because in many cases one or a few wave fronts sweep through the image, and one can find active pixels only in the wave fronts, which is less than one percent of the pixels, hence, there is nothing to calculate in the rest of image. We define a measure of efficiency of processor utilization with the following form:

$$\eta = O_r / O_t \quad (1)$$

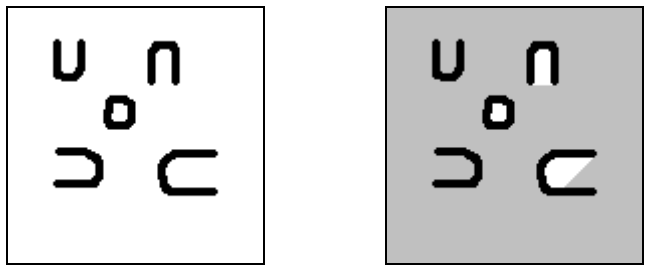
where:

- O_r: the minimum number of required elementary steps to complete an operation, assuming that the inactive pixel locations are not updated
- O_t: is the total number of elementary steps performed during the calculation by all the processors in the particular processor architecture.

The efficiency of processor utilization figure will be calculated in the following where it applies, because this is a good parameter (among others) to compare the different architectures.

Execution-sequence-invariant content-dependent front-active operators.

A special feature of content-dependent operators is that the path and length of the path of the propagating wave front drastically depend on the image contents itself. For example, the range of the necessary frame overwritings with a hole finder operation varies from zero overwriting to n/2 in a fine-grain architecture, assuming n×n pixel array size. Hence, neither the propagation time, nor the efficiency can be calculated without knowing the actual image.
Since the gap between the worst and best case is extremely high, it is not meaningful to provide these limits. Rather, it makes more sense to provide approximations for certain image types. But before that, we examine how to implement these operators on the studied architectures. For this purpose, we will use the *hole finder* operator, as an example. Here we will clearly see how the wave propagation follows different paths, as a consequence of varying propagation speed corresponding to different directions. Since this is an execution-sequence-invariant operation, it is certain that wave fronts with different trajectories lead to the same good result.
The *hole finder* operation, that we will study here, is a "grass fire" operation, in which the fire starts from all the boundaries at the beginning of the calculation, and the boundaries of the objects behave like firewalls. In this way, at the end of the operation, only the holes inside objects remain unfilled.
The *hole finder* operation may propagate to any direction. On a **fine-grain architecture** the wave fronts propagate one pixel steps in each update. Since the wave fronts start from all the edges, they meet in the middle of the image in typically n/2 updates, unless there are large structured objects with long bays which may fold the grass fire into long paths. In case of a text for example, where there are relatively small non-overlapping objects (with diameter k) with large but not spiral like holes, the wave stops after $n/2+k$ operations. In case of an arbitrary camera image with an outdoor scene, in most cases $3*n$ updates are enough to complete the operation, because the image may easily contain large objects blocking the straight paths of the wave front.
On a **pass-through architecture**, thanks to the pixel overwrite scheme, the first update fills up most of the background (Figure 9). Filling in the remaining background requires typically k updates, assuming the largest concavity size with k pixels. This means that on a pass-through architecture, roughly $k+1$ steps are enough, considering small, non-overlapping objects with size k.



(a) (b)

Figure 9. Hole finder operation calculated with a pass-through architecture. (a): original image. (b): result of the first update. (The freshly filled up areas are indicated with grey, just to make it more comprehensible. However, they are black on the black-and-white image, same as the objects.)

In the **coarse-grain architecture** we can also apply the pixel overwriting scheme within the *N*×*N* sub-arrays (Figure 10). Therefore, within the sub-array, the wave front can propagate in the same way, as in the pass-through architecture. However, it cannot propagate beyond the boundary of the sub-array, in a single update. In this way, the wave front can propagate N positions in the direction which correspond to the calculation directions, and one pixel in the other directions, in each update. In this way, in *n/N* updates, the wave-front can propagate n positions in the supported directions. However, the *k* sized concavities in other directions would require *k* more steps. To avoid these extra steps, without compromising the speed of the wave-front, we can switch between the top-down and the bottom-up calculation directions after each update. The resulting wave-front dynamics is shown in Figure 11. This means that for an image, containing only few, non-overlapping small objects with concavities, we need about *n/N+k* steps to complete the operation.

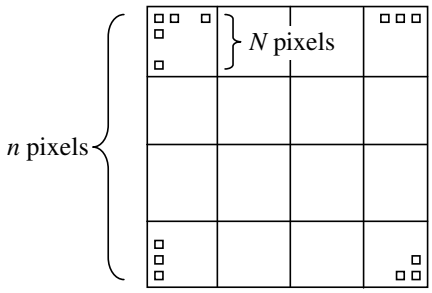


Figure 10. Coarse-grain architecture with n×n pixels. Each cell is to process an N×N pixel sub-array.

The **DSP-memory architecture** offers several choices depending on the internal structure of image. The simplest is to apply pixel overwriting scheme, and switch the direction of the calculation. In case of binary image representation, only the vertical directions (up or down) can be efficiently selected, due to the packed 32 pixel line segment storage and handling. In this way the clean vertical segments (columns of background with maximum one object) are filled up after the second update, and filling up the horizontal concavities would require k steps.

The **CELL architecture** can be considered as the combination of the pass-through architecture and the DSP. Each of the SPEs build up *2j+1* consecutive lines of the image, and execute *j* updates in each SPE, and send the lines over to the next SPE. The number of update depends on the processor load of the operator. If the operator is simple, than multiple updates will be needed, otherwise the data transfer between the processor will cause bottleneck. Since there are bus-rings

into both directions, two data flows can be started parallel. The upper part of the image can be processed and passed from left to right, while the lower part can be processed and passed from right to left. In this way, the wave-front starts propagating from both up and down parallel. The results will be calculated in *(k+1)/8* step similarly to the pass-through. The 8 times speedup is coming from the number of the SPEs processing the image parallel.

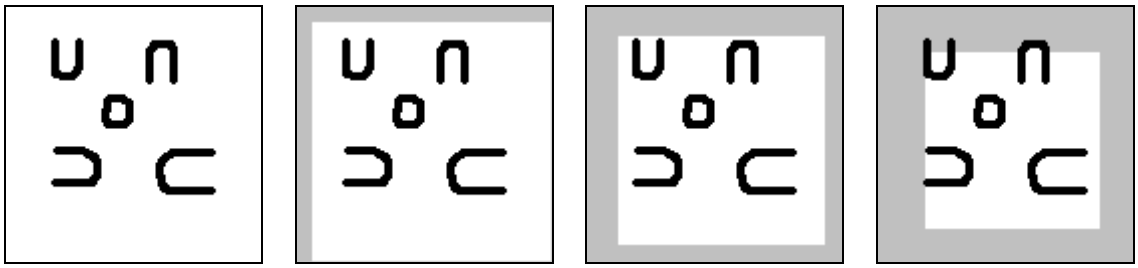


Figure 11. Hole finder operation calculated in a coarse-grain architecture. The first picture shows the original image. The rest shows the sequence of updates, one after the other. The freshly filled-up areas are indicated with grey (instead of black) to make it easier to follow the dynamics of calculation.

Execution-sequence-variant content-dependent front active operators

The calculation method of the execution-sequence-variant content-dependent front active operators is very similar to that of their execution-sequence-invariant counterparts. The only difference is that in each of the architectures the frame overwriting scheme should be used. This does not make any difference in fine-grain architectures, however, it slows down all the other architectures significantly. In the DSP-memory architectures, it might even make sense to switch to one byte/pixel mode, and calculate updates at the wave fronts only.

1D content-independent front active operators (1D scan).

In the 1D content-independent front active category, we use the vertical shadow (north to south) operation as an example. In this category, varying the orientation of propagation may cause drastic efficiency differences on the non-topographic architectures.

On a **fine-grain discrete time** architecture the operator is implemented in a way that in each time instance, each processor should check the value of its upper neighbor. If it is +1 (black), it should change its state to +1 (black), otherwise the state should not change. This can be implemented in one single step in a way, that each cell executes an *OR* operation with its upper neighbor, and overwrites its state with the result. This means that in each time instance the processor array executed n^2 operations, assuming $n \times n$ pixel array size.

In discrete time architectures, each time instance can be considered as a single iteration. In each iteration the shadow wave front moves by one pixel to the south, that is we need n steps for the wave front to propagate from the top row to the bottom (assuming boundary condition above the top row). In this way, the total number of operations, executed during the calculation is n^3. However, the strictly required number of operations is n^2, because it is enough to do these calculations at the wave front, only ones in each row, starting from the top row, and going down row by row, rolling over the results from the front line to the next one. In this way, the efficiency of the processor utilization in *vertical shadow* calculation in the case of fine-grain discrete time architectures is

$$\eta = 1/n \qquad (2)$$

Considering computational efficiency, the situation is the same in **fine-grain continuous architectures**. However, from the point of power efficiency the Asynchronous Cellular Logic Network [88] is very advantageous, because only the active cells in the wave front consume switching power. Moreover, the extraordinary propagation speed (500 ps/cell) compensates for the low processor utilization efficiency.

If we consider a **coarse-grain architecture** (Figure 10), the *vertical shadow* operation is executed in a way that each cell executes the above *OR* operation from its top row, and goes on from the top downwards in each column. This means that $N \times N$ operations are required for a cell to process its sub-array. It does not mean, however, that in the first $N \times N$ steps the whole array is processed correctly, because only the first cell row has all the information for locally finalizing the process. For the rest of the rows their upper boundary condition have not "arrived", hence at these locations correct operations cannot be performed. Thus, in the first $N \times N$ steps, the first N rows were completed only. However, the total number of operation executed by the array during this time is

$$O_{NxN}=N*N * n/N * n/N=n*n, \quad (3)$$

because there are $n/N * n/N$ processors in the array, and each processor is running all the time. To process also the rest of the lines we need to perform

$$O_t=O_{NxN} * n/N=n^3/N. \quad (4)$$

The resulting efficiency is:

$$\eta=N/n \quad (5)$$

It is worth to stop at this result for a while. If we consider a fine-grain architecture (N=1), the result is the same as we obtained in (2). Its optimum is N=n (one processor per column) when the efficiency is 100%. It turns out that in case of *vertical shadow* processing, the efficiency increases by increasing the number of the processor columns, because in that case, one processor has to deal with less columns. However, the efficiency does not increase when the number of the processor rows is increased. (Indeed, one processor/column is the optimal, as it was shown.) Thought the unused processor cells can be switched off with minor extra effort to increase power efficiency, but it would certainly not increase processor utilization.

Pass-through architecture as well as **DSP-memory architecture** can execute *vertical shadow* operation with 100% processor utilization, because there are no multiple processors in a column working parallel.

We have to note, however, that shadows to other three directions are not as simple as the one to downwards. In **DSP architectures**, *horizontal shadows* cause difficulties, because the operation is executed parallel on a 32×1 line segment, hence only one of the positions (where the actual wave front is located) performs effectual calculation. If we consider a left to right shadow, this means that once in each line (at left-most black pixel), the shadow propagation should be calculated precisely for each of the 32 positions. Once the "shadow head" (the 32 bit word, which contains the left-most black pixel) is found, and the shadow is calculated within this word, the task is easier, because all the rest of the words in the line should be filled with black pixels, independently of their original content. Thus the overall resulting cost of a *horizontal shadow* calculation on a **DSP-memory architecture** can be even *20* times higher than that of a *vertical shadow* for a 128×128 sized image. Similar situation might happen in **coarse-grain architectures**, if they handle $n \times 1$ binary segments.

While **pass-through architectures** can execute the *left to right and top to bottom shadows* in a single update at each pixel location, the other directions would require n updates, unless the direction of the pixel flow is changed. The reason for such a high inefficiency is that in each update, the wave front can propagate only one step in the opposite direction.

The CELL architecture operates similar to the DSP in this case. However, in case of vertical shadow, the operation will be bandwidth limited. This means that even one SPE cannot be fully

exploited due to the inadequate amount of data transfer. In case of horizontal shadow, the processor load enables the usage of multiple SPEs.

2D content-independent front active operators (2D scan).

The operators belonging to the 2D content-independent front active category require simple scanning of the frame. In *global max* operation for example, the actual maximum value should be passed from one pixel to another one. After we scanned all the pixels, the last pixel carries the global maximum pixel value.

In **fine-grain architectures** this can be done in two phases. First, in n comparison steps, each pixel takes over the value of its upper neighbor, if it is larger than its own value. After n steps, each pixel in the bottom row contains the largest value of its column. Then, in the second phase after the next n horizontal comparison steps, the global maximum appears at the end of the bottom row. Thus, to obtain the final result requires $2n$ steps. However, as a fine-grain architecture executes $n \times n$ operations in each step, the total number of the executed operations are $2n^3$. However, the minimum number of requested operation to find the largest value is n^2 only. Therefore, the efficiency in this case is:

$$\eta = 1/2n \qquad (6)$$

The most frequently used operation in this category is *global OR*. To speed up this operation in the fine-grain arrays, a *global OR* net is implemented usually [84][78]. This $n \times n$ input OR gate requires minimal silicon space, and enables us to calculate *global OR* in a single step (a few microseconds).

However, in that case, when a fine-grain architecture is equipped with *global OR*, the global maximum can be calculated as a sequence of iterated *threshold* and *global OR* operations with interval halving (successive approximation) method applied in parallel to the whole array. This means that a global threshold is applied first for the whole image at level ½, and if there are pixels, which are larger than this, we will do the next global thresholding at ¾, and so on. Assuming 8 bit accuracy, this means that in 8 iterations (16 operations), the global maximum can be found. The efficiency is much better in this case:

$$\eta = 1/16$$

In **coarse-grain architectures**, each cell calculates the *global maximum* in its sub-array in $N \times N$ steps. Then n/N vertical steps come, and finally, n/N horizontal steps to find the largest values in the entire array. The total number of steps in this case is $N^2 + 2n/N$, and in each step, $(n/N)^2$ operations are executed. The efficiency is:

$$\eta = n^2 / (N^2 + 2n/N) * (n/N)^2 = 1/(1 + 2n/N^3) \qquad (7)$$

Since the sequence of the execution does not matter in this category, it can be solved with 100% efficiency in **pass-through** and the **DSP-memory architectures** and on the **Cell architecture**. We have to note that this task is memory bandwidth limited on the CELL architecture.

Area active operators.

The area active operators require some computation in each pixel in each update; hence, all the architectures work with 100% efficiency. Since the computational load is very high here, it is the most advantageous for the many-core architectures, because the speed advantage of the many processor can be efficiently utilized.

3.3. COMPARISON OF THE ARCHITECTURES

As we have stated in the previous section, front active wave operators run well under 100% efficiency on topographic architectures, since only the wave fronts need calculation, and the processors of the array in non-wave front positions do dummy cycles only or may be switched

off. On the other hand, the computational capability (GOPs) and the power efficiency (GOPs/W) of multi-core arrays are significantly higher than those of DSP-memory architectures. In this section, we show the efficiency figures of these architectures in different categories. To make fair comparison with relevant industrial devices we have selected three market-leader, video processing units, a DaVinci video processing DSP from Texas Instruments (TMS320DM6443) [93], and a Spartan 3 DSP FPGA from Xilinx (XC3SD3400A) [103], and the GTX280 from NVIDIA [104]. All three of these products' functionalities, capabilities and prices were optimized to efficiently perform embedded video analytics.
Table I summarizes the basic parameters of the different architectures, and indicates the processing time of a 3×3 convolution, and a 3×3 erosion. To make the comparison easier, values are calculated for images of 128×128 resolution. For this purpose, we considered 128×128 Xenon and Q-Eye chips. Some of these data are from catalogues, other ones are from measurements, or estimation. As fine-grain architecture examples, we included both the SCAMP and Q-Eye architectures.

We can see from Table I, the DSP was implemented on 90nm, while the FPGA the GPU and the CELL on 65 nm technologies. In contrast Xenon, Q-Eye, and SCAMP were implemented on more conservative technologies (180nm, 180nm, and 350nm respectively) and their power budget is an order of magnitude smaller compared to DSP and FPGA, and two orders of magnitude smaller than CELL and GPU. When we compare the computational power figures, we also have to take these parameters into consideration.

Table I shows the speed advantages of the different architectures, compared to DSP-memory architecture both in 3×3 neighborhood arithmetic (8 bit/pixel) and morphologic (1 bit/pixel) cases. This indicates the speed advantage of the area active single step, and the front active content-dependent execution-sequence-variant operators. In Table II, we summarize the speed relations of the rest of the wave type operations. The table indicates the computed values, using the formulas that we have derived in the previous section. In some cases, however, the coarse- and especially the fine-grain arrays contain some special accelerator circuits, which takes the advantage of the topographic arrangement and the data representation (e.g., global OR network, mean network, diffusion network). These are marked by notes, and the real speed-up with the special hardware is shown in parenthesis.
Among the low-power multi-core processor architectures, the pass-through is the only one that can handle both high-resolution and low resolution images too, due to the relatively small memory demand. While the coarse- and fine-grain architectures require the storage of 6-8 entire frames, the pass-through architecture needs only a few lines for each processor. In case of a mega-pixel image, it can be less than one third of the frame. This means that as opposed to the coarse- and fine-grain architectures, the pass-through architecture can **trade speed for resolution**. This is very important, because the main criticism of the topographic architectures is that they cannot handle large images, and many of the users do not need their 1000+ FPS. The price what the pass-through architectures pay for this trade-off is their rigidity. Once the architecture is downloaded to an FPGA (or an ASIC is fabricated), it cannot be flexibly reprogrammed, only the computational parameters can be varied. It is very difficult to introduce conditional branching, unless all the passes of the branching are implemented on silicon (multi-thread pipeline), or significant delay or latency is introduced.

Table I Computational parameters of the different architectures for arithmetic (3×3 convolution) and logic (3×3 binary erosion) operations.

	DSP (DaVinci+)	**Pass-through (FPGA++)**	**Coarse-grain (Xenon)**	**Fine-grain (SCAMP/Q-Eye)**	**Cell Architecture**	**GPU GTX280**
Silicon technology	90nm	65nm	180nm	350/180nm	65nm	65nm
Silicon area mm²			100	100/50		576
Power consumption	1.25 W	2-3W	0.08 W	0.20 W	86 W	236 W (board)
Arithmetic proc. clock speed	600 MHz	250 MHz	100 MHz	1,2 / 2.5 MHz	3200 MHz	1300 MHz
Number of arithmetic proc.	8	120	256	16384	8x4	240
Nominal arithmetic comp. power	4.8 GMAC (8 bit int)	30 GMAC (8 bit int)	25.6GMAC (8 bit int)	19GOPS****	102GMAC (32 bit float)	324GMAC (32 bit float)
Reached arithmetic comp. power	3.5 GMAC (8 bit int)	30 GMAC (8 bit int)	12.2GMAC (8 bit int)	6.7GOPS****	48GMAC (32 bit float)	14GMAC (32 bit float)
Efficiency of arithmetic calc.	73% *	100%	48% ***	41% **	47%	4%
3×3 convolution time (128x128 pixel)	42.3 µs*****	4.9 µs	12.1 µs	22 µs ****	3.1µs	14 µs
Arithmetic speed-up	1	8.6	3.5	1.9	13.7	4
Morph. proc. clock speed	600 MHz	83 MHz	100 MHz	1,2 / 5 MHz	3200 MHz	1300 MHz
Number of morphologic proc.	64	864	2048	147456	1024	7680
Morphologic processor kernel type	2 × 32 bit	96 × 9 bit	256 × 8 bit	16384 × 9 bit	8x128	240x32
Nominal morphologic comp. power	38GLOPS#	216GLOPS	205GLOPS	737GLOPS	3280GLOPS	10400GLOPS
Reached morphologic comp. power	10GLOPS	72GLOPS	134GLOPS	737GLOPS	1540GLOPS	
Efficiency of morphological calc.	28% *	33%	65% ***	100%	47%	
3×3 morphologic operation time	13.6 µs*****	2.05 µs	1.1 µs	0.2 µs	0.1µs	
Morphologic speed-up	1	6.6	12.4	68.0	142	

+ Texas Instrument DaVinci video processor (TMS320DM64x)

++ Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* processors are faster than cache access

** data access from neighboring cell is an additional clock cycle

*** due to pass-through stages in the processor kernel, (no effective calculation in each clock cycle)

**** no multiplication, scaling with few discrete values

***** these data-intensive operators slow down to 1/3rd or even 1/5th when the image does not fit to the internal memory (typically above 128×128 with a DaVinci, which has 64kByte internal memory)

LOPS: Logic operation per second (1 bit)

The CELL and the GPU can also handle high-resolution images due to their relatively high external memory bandwidth. As it can be seen from the comparison, the efficiency of the GPU is very low. It is due to the small size convolution kernel. In convolutions with larger kernels or in other local operations, the GPU is much more efficient.

In our comparison tables, we have represented a typical FPGA as a vehicle to implement the pass-through architectures. The only reason is that all the currently available pass-through architectures are implemented in FPGAs is mainly attributed to much lower costs and quicker time-to-market development cycles. However, they could also be certainly implemented in ASIC,

which would significantly reduce their power consumption, and decrease their large-volume prices making it possible to process even multi-mega pixel images at a video rate.

Table II Speed relations in the different function groups calculated for 128×128 sized images. The notes indicate the functionalities by which the topographic arrays are speeded up with special purpose devices.

	DSP (DaVinci[+])	**Pass-through (FPGA[++])**	**Coarse-grain (Xenon)**	**Fine-grain discrete time (SCAMP/ Q-Eye)**	**Fine-grain continuous time (ACLA)**
1D content-independent front active operators e.g. (shadow)					
processor util. Efficiency	100%	100%	N/n: 6.25%	1/n: 0.8%	1/n: 0.8%
speed-up in advantageous direction (vertical)	1	6.6	0.77	0.53	188
speed-up in disadvantageous direction (horizontal)	1	1	2	10.6	3750
2D content-independent front active operators					
processor util. Efficiency	100%	100%	*$1/(1+2n/N^3)$: 66%*	1/2n: 0.4%	-
speed-up (*global OR*)	1	6.6	8.2 (13*)	0.27 (20*)	n/a
speed-up (*global max*)	1	8.6	2.3	n/a	n/a
speed-up (*average*)	1	8.6	2.3	n/a (2.5)**	n/a
Execution-sequence-invariant content-dependent front active operators					
hole finder with k=10 sized small objects	4 updates	k+1 updates (11)	n/N+k (26)	n/2+k updates (74)	n/2+k updates (74)
Speedup	1	2.4	1.9	3.7	1500
Area active					
processor util. Efficiency	100%	100%	100%	100%	
Speedup	1	8.6	3.5	1.9 (210***)	n/a
Multi-scale					
1:4 scaling	1	8.6	3.5	0.1	n/a

[+] Texas Instrument DaVinci video processor (TMS320DM64x)

[++] Xilinx Spartan 3ADSP FPGA (XC3SD3400A)

* Hard wired global OR device speeds up this function (<1 μs concerning the whole array)

** Hard wired mean calculator device makes this function available (~2 μs concerning the whole array)

*** Diffusion calculated on resistive network (<2 μs concerning the whole array)

Table III shows the computational power, the consumed power and the power efficiency of the selected architectures. As we can see, the three topographic arrays have over hundred times power efficiency advantage compared to DSP-memory architectures. This is due to their local data access, and relatively low clock frequency. In case of ASIC implementation, the power efficiency of the pass-through architecture would also be increased with a similar factor.

Table III Computational power, and the consumed electronic power, and their proportion in different architectures for convolution operations.

	GOPs	W	GOPs/W	Accuracy
DaVinci	3.6	1.25	2.88	1/8 int
Pass-through (FPGA)	30	3	10	1/8 int
Xenon (64x64)	10	0.02	500	1/8 int
SCAMP (128x128)	20	0.2	100	6-7 analog
Q-Eye	25	0.2	125	6-7 analog
Cell multiprocessor	225	85	2.6	32 float
GPU	324	236	1.37	32 float

Figure 12 shows the relation between the frame-rate and the resolution in a video analysis task. Each of the processors had to calculate 20 *convolutions*, 2 *diffusions*, 3 *means*, 40 *morphologies* and 10 *global OR*s. Only the DSP-memory and pass-through architectures support trading between resolution and frame-rate. The characteristics of these architectures form lines. The chart shows the performance of the three discussed chips too. The chips are represented here with their physical sizes. Naturally, this chart belongs to this particular task with the given operations "basket". By using a different one, different chart would come out.

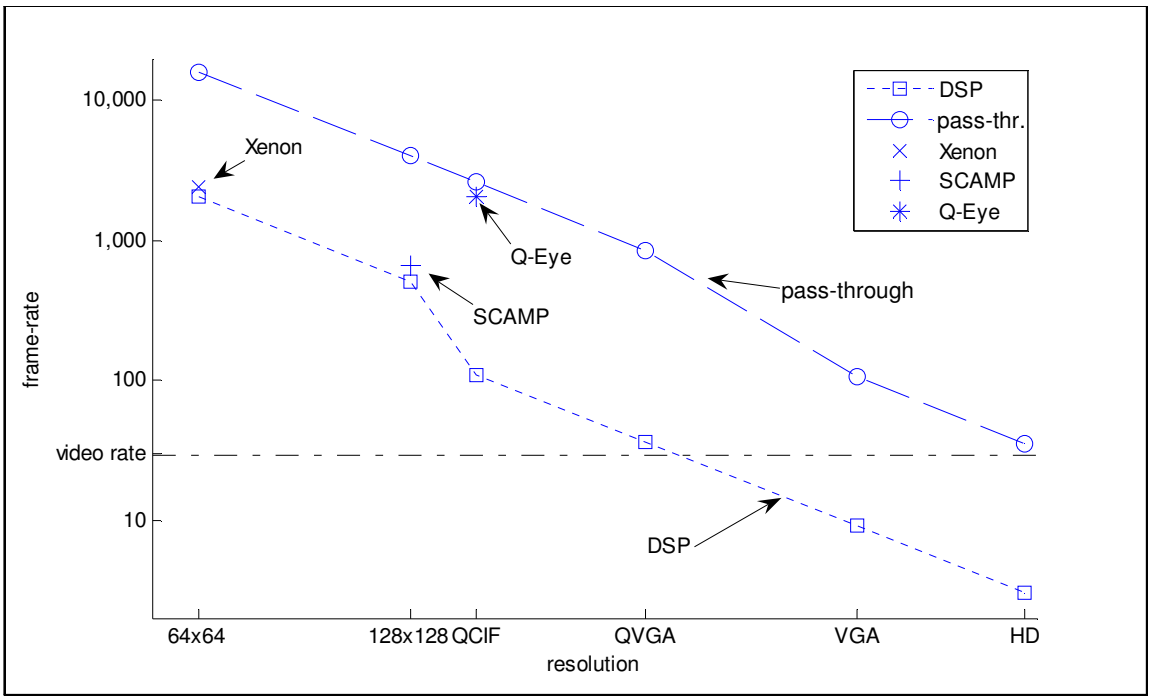


Figure 12. Frame-rate versus resolution in a typical image analysis task. Both of the axes are in logarithmic scale.

As it can be seen in Figure 12, both SCAMP and Xenon have the same speed as the DSP. In the case of Xenon, this is so, because its array size is 64×64 only. In the case of SCAMP, the processor was designed for very accurate low power calculation by using a conservative technology. In this particular task, the Q-Eye chip was almost as fast as the pass-through architectures, thanks to its integrated diffusion circuitry and support of binary morphology.

3.4. EFFICEINT ARCHITECTURE SELECTION

So far, we have studied how to implement the different wave type operators on different architectures, identified constrains and bottlenecks, and analyzed the efficiency of these implementations. After having these results in our hand, we can define rules for optimal image processing architecture selection for topographic problems. This section considers the low power devices, where the embedded operation is a viable option.

Image processing devices are usually special purpose architectures, optimized for solving specific problems or a family of similar algorithms. Figure 13 shows a method of special purpose processor architecture selection. It always starts with the understanding of the problem in all aspect. Then, different algorithms suitable for solving the problem are derived. The algorithms are described with flowchart, with the list of the used operations, and with the specification of the most important parameters. In this way, a set of formal data describes the algorithms, which are as follows: ***resolution, frame-rate, pixel clock, latency, computational demand*** (type and number of operators), ***and flowchart***. Other application-specific (secondary) parameters are also given: maximal ***power consumption***, maximal ***volume***, ***economy*** etc. The algorithm derivation is a human activity supported by various simulators for evaluation and verification purposes.

The next step is the architecture selection. By using the previously compiled data, we can define a methodology for the architecture selection step. As we will see, based on the formal specifications, we can derive the possible architectures. There might not be any, there might be exactly one, or there might be several, according to the demands of the specification of the algorithm.

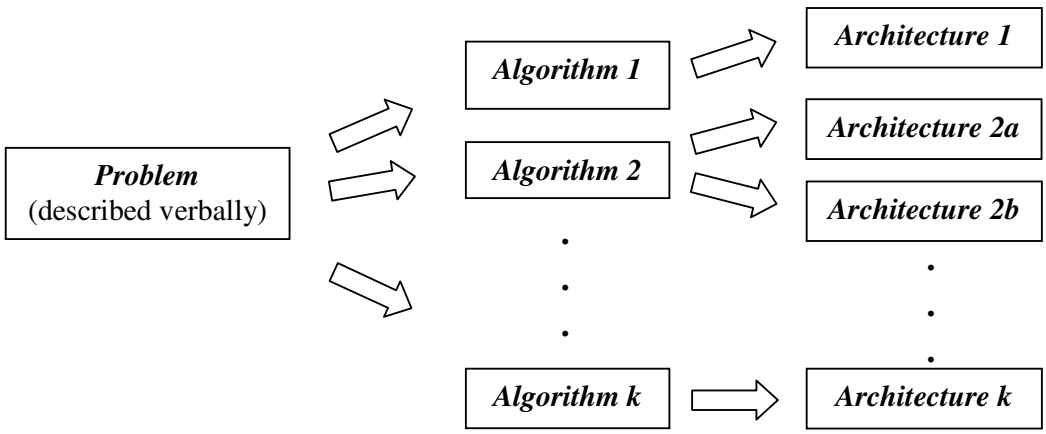


Figure 13. Methodology of special purpose processor architecture selection

The first step of the method is the comprehensive analysis of the parameter set. Fortunately, in many cases it immediately leads to a single possible architecture. If it does not lead to any architecture, in a second step, we have to seek for options, how to fulfill the original specification demands. If it leads to multiple architectures, a ranking is needed based on secondary parameters. The three most important parameters are the ***frame-rate***, the ***resolution***, and their product, the minimal value of the ***pixel clock***.[1] In many cases, especially in challenging applications, these parameters determine the available solutions. Figure 14 shows frame-rate – resolution matrix.

[1] The minimal value of the pixel clock is equivalent to the product of the frame-rate and the number of pixels (resolution). If the image source is a sensor, the pixel clock of the processor is defined by the pixel clock of the sensor. Since there are short blank periods in the sensor readout protocol for synchronization purposes, the pixel clock is slightly higher then the minimal pixel clock even in those cases, when the integration is done parallel with the readout (CMOS or CCD rolling shutter mode). However, in low light applications, the integration time is much longer than the readout time. In these cases, the sensor pixel clock can be orders of magnitude higher than the minimal pixel clock.

The matrix is divided into 16 segments, and each segment indicates the potential architectures that can operate in that particular parameter environment. The matrix shows the minimal pixel clock figures (red) in the grid points also.
In Figure 14, the pass-through and the DSP can be positioned freely between frame-rate and resolution without constrains. Thus they appear everywhere, under a certain pixel clock rate. The digital coarse-grain sensor-processor arrays appear in the low resolution part (left column), while the analog (mixed-signal) fine-grain sensor-processor arrays appear in both the low and medium resolution columns.
The next important parameter is the ***latency***. Latency is critical when the vision device is in a control loop, because large delays might make the control loops instable. It is worth to distinguish three latency requirement regions:

- very low latency (*latency* <2ms; e.g. missile, UAV, high speed robot controlling);
- low latency (2ms < *latency* <50ms; e.g. robotic, automotive);
- high latency (50ms < *latency; e.g.* security, industrial quality check).

Latency has two components. The first is the readout time of the sensor, and the second is the completion of the processing on the entire frame. The readout time is negligible in the fine-grain mixed-signal architectures, since the analog sensor readout should be transferred to an analog memory through a fully parallel bus. The readout time is also very small (~100µs) in the coarse-grain digital processor array, because there is an embedded AD converter array to do conversion in parallel. The DSPs and the pass-through processor arrays use external image sensors, in which the readout time usually is in the millisecond range. Therefore, in case of very low latency requirements, the mixed-signal and the digital focal plane arrays can be used. (There are some ultra-high frame-rate sensors with high speed readout, which can be combined with pass-through processors. However, these can be applied in very special applications only due to their high complexities and costs.)

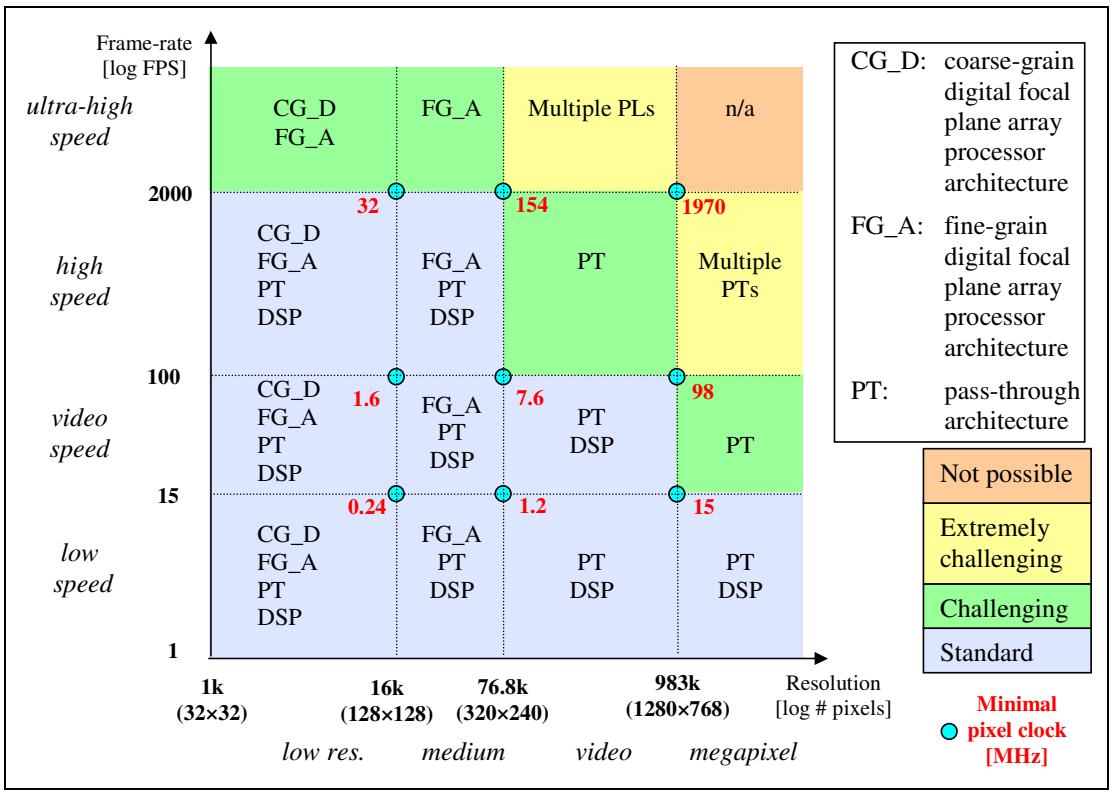


Figure 14. Feasible architectures in the frame-rate – resolution matrix

In the low latency category, those architectures can be used only, in which the sensor readout time plus the processing time is smaller than the latency requirements. In the high latency region, the latency does not mean any bottleneck.
The next descriptor of the algorithms is the ***computational demand***. It is a list of the applied operations. Using the execution time figures that we calculated for different operations on the examined architectures, we can simply calculate the total execution time. (In case of the pass-through architecture the delay of the individual stages should be summed up.) The total processing time should satisfy the following two relations:

$$t_{total\_processing} < t_{latency} - t_{readout}$$
$$t_{total\_processing} < 1/frame\_rate$$

The last primary parameter is the ***program flow***. The array processors and the DSP are not sensitive for branches in program flow. However, the pass-through architectures are challenged by the conditional program flow branches, because the branching should happen at the calculation of the first pixel of the frame, but at that time, the branching condition is not yet calculated. (The condition is calculated during the processing of the entire frame.) Since that, before the branching, the application of a frame-buffer is required, which generates significant hardware overhead and latency increase.
There are three secondary design parameters. The first is the ***power consumption***. Generally, the ASIC solutions need much less power than the FPGA or DSP solutions. The second is the cubature of the circuit. Smaller cubature can be achieved with sensor-processor arrays, because the combination of these two functionalities reduces the chip count. The third parameter is the economy. In case of low volume, the DSP is the cheapest, because the invested engineering cost is the smaller there. In case of medium volume, the FPGA is the most economical, while in case of high volume, the ASIC solutions are the cheapest.

Chapter 4. PDE SOLVERS

In this chapter after two simple PDE (Laplace and Poison Equations) some emulated digital CNN based PDE problems are discussed. The models (by partial differential equations) are defined in each problem first. Than discretizing in space and in time the problems can be mapped to an emulated digital CNN model. The final architectures were further optimized, and evaluated in terms of time, space, accuracy and dissipated power. The considered problems are:

(i) Tactile sensor model,

(ii) Geothermal model,

(iii) Ocean stream model,

(iv) Fast fluid flow model.

LaplacePDESolver: Solves the Laplace PDE: $\nabla^2 \mathbf{x} = 0$

Old names: LAPLACE

A =

0	ν/h^2	0
ν/h^2	$-4*\nu/h^2+1/R$	ν/h^2
0	ν/h^2	0

B =

0	0	0
0	0	0
0	0	0

z = 0

where ν is the diffusion constant,
h is the spatial step
R is the value of the resistor of the CNN cell.

If $\nu = h = R = 1$, our template is as follows:

A =

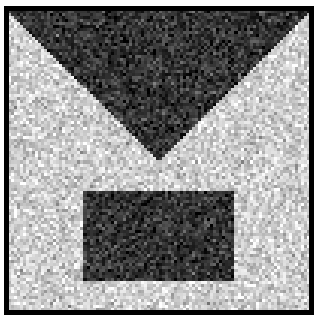
0	1	0
1	-3	1
0	1	0

B =

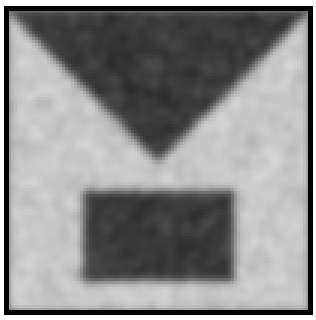
0	0	0
0	0	0
0	0	0

z = 0

Example : image name: laplace.bmp, image size: 100x100; template name: laplace.tem .



initial state



output (t=T)

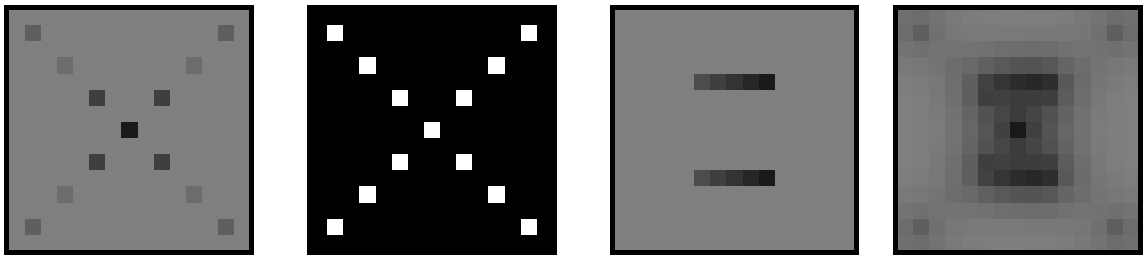
PoissonPDESolver: Solves the Poisson PDE: $\nabla^2 x = f(x)$

Old names: POISSON

$$\mathbf{A} = \begin{bmatrix} 0 & \nu/h^2 & 0 \\ \nu/h^2 & -4*\nu/h^2+1/R & \nu/h^2 \\ 0 & \nu/h^2 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad z = \boxed{0}$$

where -f(x) is filled into the Bias map.

Example: $\nu = h = R = 1$. Image names: poisson1.bmp, poisson2.bmp, poisson3.bmp; image size: 15x15; template name: poisson.tem. Torus frame style is used.



initial state fixed state bias map output

4.1. TACTILE SENSOR MODELING BY USING EMULATED DIGITAL CNN-UM

Model of the tactile sensor

Tactile sensors are usually composed of a central shuttle plate, which is suspended by four bridges over a pit. In our case the bridges are located on the center of each edge as shown on Figure 1. Each bridge contains an embedded piezoresistor. The suspension of the whole structure allows deformation of the bridges as normal and shear stress is applied to the central plate.

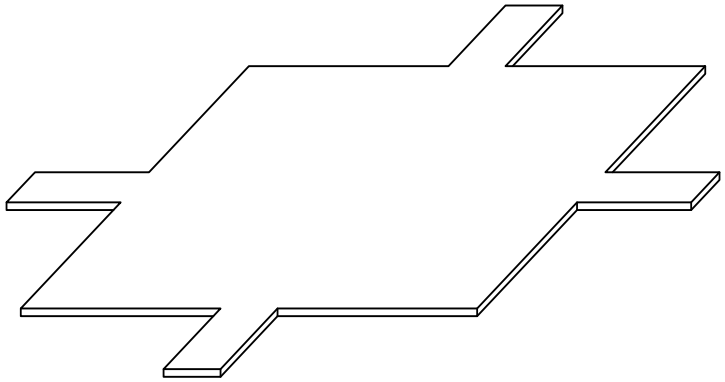


Figure 1. Structure of the tactile sensor

The transient response of the central plate due to an applied normal pressure can be described by the following partial differential equation:

$$h\rho\frac{\partial^2 w}{\partial t^2} = p - D\left(\frac{\partial^4 w}{\partial x^4} + 2\frac{\partial^4 w}{\partial x^2 \partial y^2} + \frac{\partial^4 w}{\partial y^4}\right) \tag{1}$$

where w is the displacement of the plate, p is the applied pressure, h is the thickness of the plate and ρ is the density of the plate. The flexure rigidity D can be computed by the following expression:

$$D = \frac{Eh^3}{12\left(1-\nu^2\right)} \tag{2}$$

where E is the Young's modulus and ν is the Poisson's ratio.

The dimensions of the plate is 100μm×100μm and the thickness is 2.85μm. The width of the suspension bridges is 12.5μm. The tactile sensor is made from silicon so the material constants are the following: E=47GPa, ν=0.278 and ρ=2330kg/m^3.

Emulated digital solution

To solve (1) on a CNN-UM the plate should be spatially discretized and each finite element is assigned to one CNN cell. Equation (1) is second order in time so two coupled CNN layers are required where the displacement of the plate is computed by the first layer while the velocity is computed by the second. The approximation of the spatial derivatives requires the following 5×5 sized template:

$$D\nabla^4 w = D\left(\frac{\partial^4 w}{\partial x^4} + 2\frac{\partial^4 w}{\partial x^2 \partial y^2} + \frac{\partial^4 w}{\partial y^4}\right) \approx \frac{D}{\Delta x^4}\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & -8 & 2 & 0 \\ 1 & -8 & 20 & -8 & 1 \\ 0 & 2 & -8 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{3}$$

where Δx is the distance between the grid points. Equation (1) can not be solved on the current analog VLSI chips because 5×5 sized templates are not supported in these architectures. Using the Falcon configurable emulated digital CNN-UM architecture the limitations of the analog VLSI chips can be solved. The Falcon architecture can be configured to support two CNN layers and 5×5 sized templates. To achieve better numerical stability the leapfrog

method is used instead of the forward Euler method during the computation of the new cell value. The leapfrog method computes the new cell value using the data from the previous time step according to the following equation:

$$w^{n+1} = w^{n-1} + \Delta t f\left(w^{n}\right) \tag{4}$$

where Δt is the time step value, w^{n-1}, w^{n}, w^{n+1} are the cell values at the previous, current and the next time step respectively. f(.) is the derivative computed by using template (3) at the given point. The implementation of the leapfrog method requires additional memory elements and doubles the required bandwidth of the processor but these modifications are worthwhile because much larger time step can be used. Additionally the symmetry of the required template operator makes it possible to optimize the arithmetic unit of the Falcon architecture. Using the original Falcon arithmetic unit the template operation is computed in 5 clock cycles in row-wise order and 5 multipliers are used. However multiplication with 2, -8 and 20 can be done by shifts in a radix 2 number system. Multiplication by 20 can be computed by multiplying the value by 16 and 4 and sum the partial results. After this optimization just one clock cycle and only one multiplier is required during the computation. The required resources to implement one processor which can compute (1) on a 512×512 sized grid with different precisions are summarized on Table I.

Table I. Resource requirements of one optimized Falcon processor core

	Required resources		Available resources		Resource utilization			
			XC2V1000	XC2VP125	XC2V1000		XC2VP125	
Precision	18 bit	35 bit			18 bit	35 bit	18 bit	35 bit
Mult18x18*	1	4	40	556	2.5%	10%	0.18%	0.72%
Slice**	650	1200	5120	55616	12.7%	23.4%	1.17%	2.16%
BRAM***	5	10	40	556	12.5%	25%	0.9%	1.8%

*Mult18x18 is an 18bit by 18bit signed multiplier
**Each slice contains two 4 input look-up tables, two registers and carry logic for two full adders
***BRAM is an 18kbit dual-ported SRAM

Results, performance

The proposed architecture will be implemented on our RC200 prototyping board from Celoxica Ltd. The Virtex-II 1000 (XC2V1000) FPGA on this card can host four Falcon processor cores using 35bit precision, which makes it possible to compute four iterations in one clock cycle. The performance of the system is limited by the speed of the on board memory resulting in a maximum clock frequency of 90MHz. The theoretical performance of the four processor cores are 360 million cell update/s. Unfortunately the board has 72bit wide data bus, so 4 clock cycles are required to read a new cell value and to store the results this reduces the achievable performance to 90 million cell update/s. The size of the memory is also a limiting factor because the state values must fit into the 4Mbyte memory of the board.

By using the new Virtex-II Pro devices with larger and faster memory the performance of the architecture can reach 230MHz clock rate and can compute a new cell value in each clock cycle. Additionally the huge amount of on-chip memory and multipliers on the largest XC2VP125 FPGA makes it possible to implement 45 processor cores resulting in 10,350 million cell update/s computing performance. On the other hand the large number of arithmetic units makes it possible to implement higher order and more accurate numerical methods. The achievable performance and speedup compared to conventional microprocessors are summarized on Table II. The results show that even the limited implementation of the modified Falcon processor on our RC200 prototyping board can outperform a high performance desktop PC. If adequate memory bandwidth (288 bit wide memory bus running on 230MHz clock frequency) is provided the performance of the emulated digital solution is 1400 times faster!

Table II. Performance comparison

	RC200	XC2V1000	XC2VP125	Athlon XP	Pentium IV
Clock freq. (MHz)	90	190	230	1833	3000
Performance (million cell iteration/s)	90	760	10,350	3.92	7.42
Iteration time on 512×512 array (ms)	2.9127	0.3449	0.0253	66.8734	35.3293
Speedup	12.12	102.42	1394.87	0.5283	1

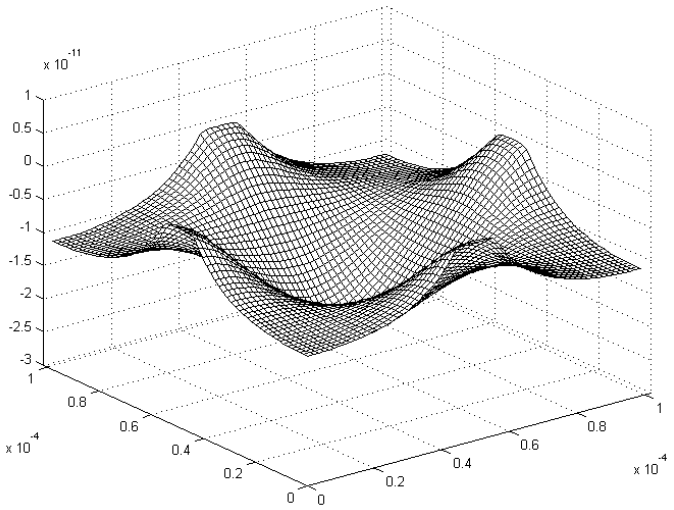


Figure 2. Displacement of the plate after 3μs.

A simple test case was used to determine the accuracy of the fixed-point solution. The input function was a step function, which applied to the center of the plate. The transient response was computed using 64bit floating-point numbers. This result was compared to the results of the 18 and 32bit fixed-point computations. On Figure 2. the displacement of the plate is shown after 3.81μs (131,072 steps).The transient response of the center of the plate computed by 64bit floating-point and different fixed-point precisions is shown on Figure 3.

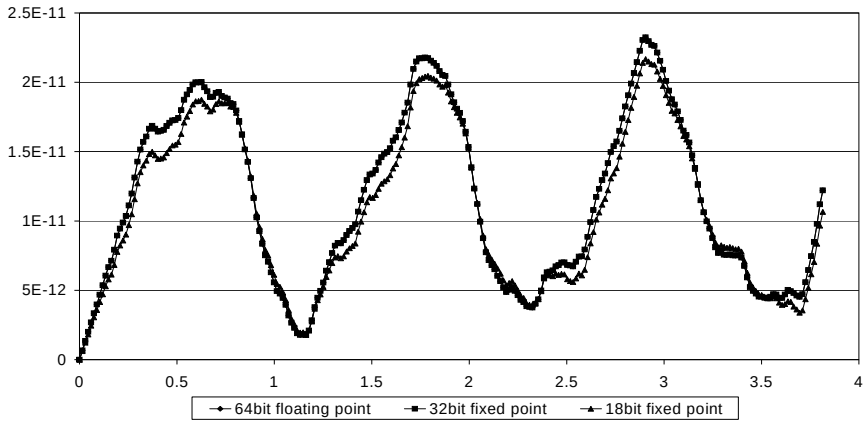


Figure 3. Transient response of the center of the plate

The result shows that even the 18bit solution is very similar to the 64bit floating point solution.

Reference

Z. Nagy, Zs. Szolgay, P. Szolgay, "Tactile Sensor Modeling by Using Emulated Digital CNN-UM", *Proc. of CNNA04, Budapest,2004, pp. 399-404*

4.2. ARRAY COMPUTING BASED IMPLEMENTATION OF WATER REINJECTION IN GEOTHERMAL STRUCTURe

The geothermal model

A mathematical model of filtration and thermal processes of the surveyed region, the "Kazichene-Ravno pole" has been developed with a view to producing geothermal energy. The mechanism of thermodynamic processes is strictly defined by Darcy's law of filtration and Fourier's law of heat transfer [4], so it is expressed by differential equations, supplemented with initial and boundary conditions, conformable to the specific problem.

Boundary value problem of filtration process

For underground water movement the main differential equation referring to the filtration in the surveyed stratum, is

$$\frac{\partial}{\partial x}\left(T\frac{\partial H}{\partial x}\right)+\frac{\partial}{\partial y}\left(T\frac{\partial H}{\partial y}\right)=0 \qquad (1)$$

where H is water pressure measured from unspecified reference plane, T is stratum conductivity $T = k_f m$ which is function of co-ordinates x and y, k_f is filtration coefficient and m is stratum thickness which is a function of the co-ordinates x and y. The filtration rate in each point of the filtration field is:

$$V_x = -k_f \frac{\partial H}{\partial x}, \quad V_y = -k_f \frac{\partial H}{\partial y} \qquad (2)$$

Boundary value problem of heat transfer

As the heat transfer takes place in three layers of different hydro-geological and thermo-physical characteristics, the model is described by three differential equations. The structure of the layers is shown in Figure 1.

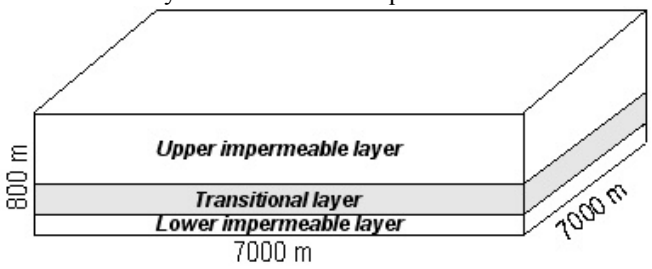


Figure 1. *Model of the examined region*

In conformity with the key prerequisites of the heat transfer model, the main differential equations can be expressed in the following way:

$$\frac{\partial}{\partial x}\left(\lambda_1\frac{\partial t_1}{\partial x}\right)+\frac{\partial}{\partial y}\left(\lambda_1\frac{\partial t_1}{\partial y}\right)+\frac{\partial}{\partial z}\left(\lambda_1\frac{\partial t_1}{\partial z}\right)=\rho_1 c_1\frac{\partial t_1}{\partial \tau} \qquad (3)$$

$$\frac{\partial}{\partial x}\left(\lambda_2\frac{\partial t_2}{\partial x}\right)+\frac{\partial}{\partial y}\left(\lambda_2\frac{\partial t_2}{\partial y}\right)+\frac{\partial}{\partial z}\left(\lambda_2\frac{\partial t_2}{\partial z}\right)-m\rho c V_x\frac{\partial t_2}{\partial x} -m\rho c V_y\frac{\partial t_2}{\partial y}=m\rho_2 c_2\frac{\partial t_2}{\partial \tau} \qquad (4)$$

$$\frac{\partial}{\partial x}\left(\lambda_3\frac{\partial t_3}{\partial x}\right)+\frac{\partial}{\partial y}\left(\lambda_3\frac{\partial t_3}{\partial y}\right)+\frac{\partial}{\partial z}\left(\lambda_3\frac{\partial t_3}{\partial z}\right)=\rho_3 c_3\frac{\partial t_3}{\partial \tau} \qquad (5)$$

where the symbols are as follows:

- t_1, t_2 and t_3 denote temperatures in the 1st, 2nd and 3rd area and are functions of x, y and z.
- λ_1, λ_2 *and* λ_3 are the coefficients of conductivity in the same areas and are functions of x, y and z.
- c_1, c_2 and c_3 mean heat capacities of the rocks.
- ρ_1, ρ_2 and ρ_3 is the rock thickness in the respective areas.
- ρ and c are density and heat capacity of water.

The equation (3) and (5) describe the heat transfer in the upper and lower argillaceous, impermeable layer, while the equation (4) denotes the process in the transitional, water saturated calcareous layer.

Discretisation of PDEs in time and space

The process described by the governing equation of filtration is a truly boundary value problem, which does not depend from the time and although by our computations the filtration terms are space dependent, but constant values in time.

In implementation of the governing equations of heat transfer on different hardware units, it is necessary to discretise the system of equations both in accordance with space and time. The first order forward Euler formulation has been used to perform this operation and result wise a set of explicit, coupled finite-difference equations have been derived corresponding to equation (3)-(5) and can be described by the following formulas:

$$t_{1_{x,y,z}}^{k+1} = t_{1_{x,y,z}}^{k} + \frac{\Delta\tau}{\rho_1 c_1}\frac{1}{\Delta x^2}\left(\lambda_{1_{x-1,y,z}} t_{1_{x-1,y,z}}^{k} - \left(\lambda_{1_{x-1,y,z}} + \lambda_{1_{x,y,z}}\right) t_{1_{x,y,z}}^{k} + \lambda_{1_{x,y,z}} t_{1_{x+1,y,z}}^{k}\right)$$
$$+\frac{\Delta\tau}{\rho_1 c_1}\frac{1}{\Delta y^2}\left(\lambda_{1_{x,y-1,z}} t_{1_{x,y-1,z}}^{k} - \left(\lambda_{1_{x,y-1,z}} + \lambda_{1_{x,y,z}}\right) t_{1_{x,y,z}}^{k} + \lambda_{1_{x,y,z}} t_{1_{x,y+1,z}}^{k}\right) \qquad (6)$$
$$+\frac{\Delta\tau}{\rho_1 c_1}\frac{1}{\Delta z^2}\left(\lambda_{1_{x,y,z-1}} t_{1_{x,y,z-1}}^{k} - \left(\lambda_{1_{x,y,z-1}} + \lambda_{1_{x,y,z}}\right) t_{1_{x,y,z}}^{k} + \lambda_{1_{x,y,z}} t_{1_{x,y,z+1}}^{k}\right),$$

$$t_{2_{x,y,z}}^{k+1} = t_{2_{x,y,z}}^{k} + \frac{\Delta\tau}{m_{x,y}\rho_2 c_2}\frac{1}{\Delta x^2}\left(\lambda_{2_{x-1,y,z}} t_{2_{x-1,y,z}}^{k} - \left(\lambda_{2_{x-1,y,z}} + \lambda_{2_{x,y,z}}\right) t_{2_{x,y,z}}^{k} + \lambda_{2_{x,y,z}} t_{2_{x+1,y,z}}^{k}\right)$$
$$+\frac{\Delta\tau}{m_{x,y}\rho_2 c_2}\frac{1}{\Delta y^2}\left(\lambda_{2_{x,y-1,z}} t_{2_{x,y-1,z}}^{k} - \left(\lambda_{2_{x,y-1,z}} + \lambda_{2_{x,y,z}}\right) t_{2_{x,y,z}}^{k} + \lambda_{2_{x,y,z}} t_{2_{x,y+1,z}}^{k}\right)$$
$$+\frac{\Delta\tau}{m_{x,y}\rho_2 c_2}\frac{1}{\Delta z^2}\left(\lambda_{2_{x,y,z-1}} t_{2_{x,y,z-1}}^{k} - \left(\lambda_{2_{x,y,z-1}} + \lambda_{2_{x,y,z}}\right) t_{2_{x,y,z}}^{k} + \lambda_{2_{x,y,z}} t_{2_{x,y,z+1}}^{k}\right) \qquad (7)$$
$$-\frac{\Delta\tau}{\rho_2 c_2}\frac{\rho c}{\Delta x} V_{x_{x,y}}\left(t_{2_{x,y,z}}^{k} - t_{2_{x+1,y,z}}^{k}\right)$$
$$-\frac{\Delta\tau}{\rho_2 c_2}\frac{\rho c}{\Delta y} V_{y_{x,y}}\left(t_{2_{x,y,z}}^{k} - t_{2_{x,y+1,z}}^{k}\right)$$

and

$$t_{3_{x,y,z}}^{k+1} = t_{3_{x,y,z}}^{k} + \frac{\Delta\tau}{\rho_3 c_3}\frac{1}{\Delta x^2}\left(\lambda_{3_{x-1,y,z}} t_{3_{x-1,y,z}}^{k} - \left(\lambda_{3_{x-1,y,z}} + \lambda_{3_{x,y,z}}\right) t_{3_{x,y,z}}^{k} + \lambda_{3_{x,y,z}} t_{3_{x+1,y,z}}^{k}\right)$$
$$+\frac{\Delta\tau}{\rho_3 c_3}\frac{1}{\Delta y^2}\left(\lambda_{3_{x,y-1,z}} t_{3_{x,y-1,z}}^{k} - \left(\lambda_{3_{x,y-1,z}} + \lambda_{3_{x,y,z}}\right) t_{3_{x,y,z}}^{k} + \lambda_{3_{x,y,z}} t_{3_{x,y+1,z}}^{k}\right) \qquad (8)$$
$$+\frac{\Delta\tau}{\rho_3 c_3}\frac{1}{\Delta z^2}\left(\lambda_{3_{x,y,z-1}} t_{3_{x,y,z-1}}^{k} - \left(\lambda_{3_{x,y,z-1}} + \lambda_{3_{x,y,z}}\right) t_{3_{x,y,z}}^{k} + \lambda_{3_{x,y,z}} t_{3_{x,y,z+1}}^{k}\right)$$

where $\Delta\tau$ is the time step, Δx, Δy and Δz are the distance of grid points in direction x,y and z respectively.

Cell Blade Systems

The third generation blade system is the IBM Blade Center QS22 equipped with new generation PowerXCell 8i processors manufactured by using 65nm technology. Double precision performance of the SPEs are significantly improved providing extraordinary computing density – up to 6.4 TFLOPS single precision and up to 3.0 TFLOPS double precision in a single Blade Center house. These blades are the main building blocks of the world's fastest supercomputer at Los Alamos National Laboratory which first breaks through the "petaflop barrier" of 1,000 trillion operations per second.

Solution on a CNN Architecture

To model the process of reinjection on emulated digital CNN architecture [8] a space-variant CNN model has been developed based on equation (6)-(8), which is operating with 3,5 dimensional templates. The second equation which describes the behavior of the water saturated transitional layer contains two additional parts which were derived from the time-independent filtration equation and make the connection between the process of filtration and heat transfer.

By the process of filtration only the temperature values have to calculated and updated during the iterations, so it can be used zero-input CNN templates using the given initial values as initial state of the template running. To design space-variant, non-linear template for the three-dimensional medium we have designed 3 coupled 2D templates using an r=1 neighborhood for every three physical layers, so every feedback template-threefold is containing 27 elements.

The structure of the coupled templates for one physical layer can be seen in Figure 3., where n denotes the described physical layer.

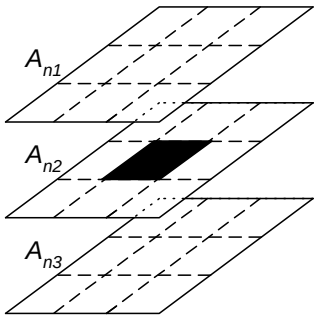


Figure 2. *Structure of coupled template for the 3D heat transfer (r=1)*

The coupled templates of the second layer which was determined from equation (7) are as follows:

$$A_{21}=\begin{bmatrix}0 & 0 & 0\\ 0 & \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta z^2}\lambda_{x,y,z-1} & 0\\ 0 & 0 & 0\end{bmatrix} \qquad A_{23}=\begin{bmatrix}0 & 0 & 0\\ 0 & \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta z^2}\lambda_{x,y,z} & 0\\ 0 & 0 & 0\end{bmatrix}$$

$$A_{22}=\begin{bmatrix}0 & \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta x^2}\lambda_{x-1,y,z} & 0\\ \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta y^2}\lambda_{x,y-1,z} & 1-\dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta x^2}\left(\lambda_{x-1,y,z}+\lambda_{x,y,z}\right) & \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta y^2}\lambda_{x,y,z}\\ & -\dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta y^2}\left(\lambda_{x,y-1,z}+\lambda_{x,y,z}\right) & \\ & -\dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta z^2}\left(\lambda_{x,y,z-1}+\lambda_{x,y,z}\right) & +\dfrac{\Delta\tau}{\rho_2 c_2}\dfrac{\rho c}{\Delta y}V_{y_{x,y}}\\ & -\dfrac{\Delta\tau}{\rho_2 c_2}\dfrac{\rho c}{\Delta x}V_{x_{x,y}}-\dfrac{\Delta\tau}{\rho_2 c_2}\dfrac{\rho c}{\Delta y}V_{y_{x,y}} & \\ 0 & \dfrac{\Delta\tau}{m_{x,y}\rho_2 c_2}\dfrac{1}{\Delta x^2}\lambda_{x,y,z}+\dfrac{\Delta\tau}{\rho_2 c_2}\dfrac{\rho c}{\Delta x}V_{x_{x,y}} & 0\end{bmatrix}$$

The space-variant templates for the first and third physical layers can be determined similarly, there only need to be used the appropriate ρ and c multiplier coefficients.

By using the previously described discretization method a C based solver is developed which is optimized for the SPEs of the Cell architecture.

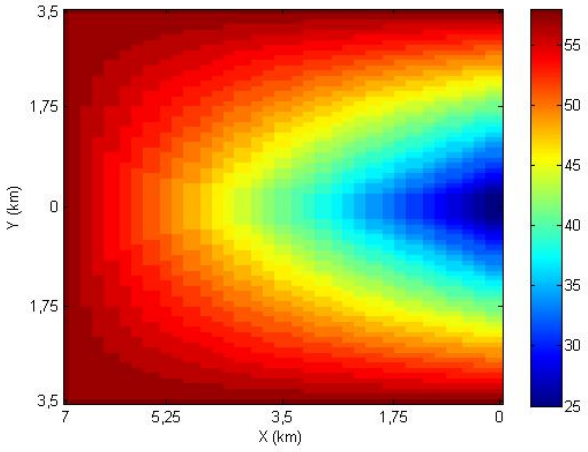


Figure 3. *Input map for transitional layer*

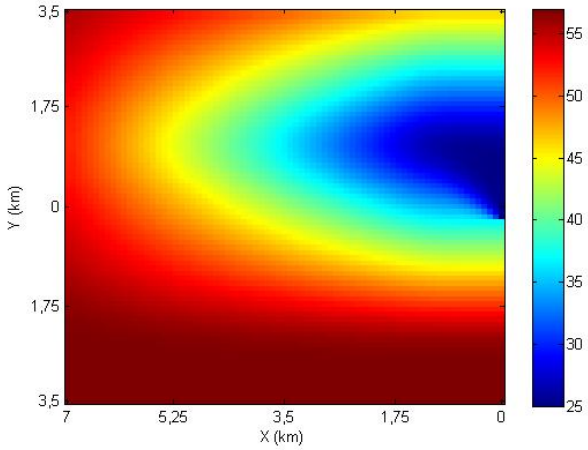


Figure 4. *Temperature map after simulation*

Reference

[1] S Kocsárdi, Z. Nagy, S Kostianev, P. Szolgay, "FPGA based implementation of water reijection in geothermal structure", *Proc. of CNNA2006, pp. 323-327, 2006,Istanbul*

4.3. EMULATED DIGITAL CNN-UM BASED OCEAN MODEL

The ocean model

Building a universal ocean model that can accurately describe the state of the ocean on all spatial and temporal scales is very difficult. Thus ocean modeling efforts can be diversified into different classes, some concerned only with the turbulent surface boundary layers, some with continental shelves and some with the circulation in the whole ocean basin. Fine resolution models can be used to provide real-time weather forecasts for several weeks. These forecasts are very important to the fishing industry, ship routing and search and rescue operations. The more coarse resolution models are very efficient in long term global climate simulations such as simulating El Nino effects of the Pacific Ocean.

In general, ocean models describe the response of the variable density ocean to atmospheric momentum and heat forcing. In the simplest barotropic ocean model a region of the ocean's water column is vertically integrated to obtain one value for the vertically different horizontal currents. The more accurate models use several horizontal layers to describe the motion in the deeper regions of the ocean. Though these models are more accurate investigation of the barotropic ocean model is not worthless because it is relatively simple, easy to implement and it provides a good basis for the more sophisticated 3-D layered models.

The governing equations of the barotropic ocean model on a rotating Earth can be derived from the Navier-Stokes equations of incompressible fluids. Using Cartesian coordinates these equations have the following form:

$$\frac{du_x}{dt} = \underbrace{2\Omega \sin\theta u_y}_{Coriolis} - \underbrace{gH\frac{\partial \eta}{\partial x}}_{Pressure} + \underbrace{\tau_{wx} - \tau_{bx}}_{Lateral\ viscosity} + A\nabla^2 u_x \underbrace{- \frac{u_x}{H}\frac{\partial u_x}{\partial x} - \frac{u_y}{H}\frac{\partial u_x}{\partial y}}_{Advection} \quad \text{a)}$$

$$\frac{du_y}{dt} = \underbrace{-2\Omega \sin\theta u_x}_{Coriolis} - \underbrace{gH\frac{\partial \eta}{\partial y}}_{Pressure} + \underbrace{\tau_{wy} - \tau_{by}}_{Lateral\ viscosity} + A\nabla^2 u_y \underbrace{- \frac{u_x}{H}\frac{\partial u_y}{\partial x} - \frac{u_y}{H}\frac{\partial u_y}{\partial y}}_{Advection} \quad \text{b)}$$

$$\frac{d\eta}{dt} = -\frac{\partial u_x}{\partial x} - \frac{\partial u_y}{\partial y} \quad \text{c)}$$

Where η is the height above mean sea level, u_x and u_y are volume transports in the x and y directions respectively. In the Coriolis term Ω is the angular rotation of the Earth and θ is the latitude. The pressure term contains H(x,y), the depth of the ocean and the gravitational acceleration g. The wind and bottom stress components in both x and y directions are represented by τ_{wx}, τ_{wy}, τ_{bx} and τ_{by} respectively. The lateral viscosity is denoted by A.

The bottom stress components can be linearly approximated by multiplying the u_x and u_y by a constant value σ' the recommended value of this parameter is in the range $1\text{-}5\cdot10^{-7}$. The wind stress components can be computed from the wind speed above the sea surface by the following approximation:

$$\tau_w = \rho C_d U_{10}^2 \quad \text{d)}$$

Where ρ is the density of the air, U_{10} is the speed of the wind at 10 meters above the surface and C_d is the drag coefficient. One possible approximation of the drag coefficient is the following:

$$1000C_d = 0.29 + \frac{3.1}{U_{10}} + \frac{7.7}{U_{10}^2} \quad (3 \le U_{10} \le 6\ m/s) \quad \text{e)}$$

$$1000C_d = 0.5 + 0.071 \cdot U_{10} \quad (6 \le U_{10} \le 26\ m/s) \quad \text{f)}$$

The horizontal friction parameter A can be computed from the mesh-box Reynolds number R_c:

$$R_c = \frac{\Delta x U}{A} \quad \text{g)}$$

Where Δx is the mesh size and U is the magnitude of the velocity in the mesh-box. By approximating U with $\sqrt{gH}$ and setting R_c=4 which is generally considered in nonlinear flow simulations the lateral friction can be computed by the following equation:

$$A = \frac{\Delta x \sqrt{gH}}{R_c} \qquad \text{h)}$$

At the edges of the model closed boundary conditions are used e.g. there is no mass transport across the boundaries. In this case u_x and u_y are both zero at the edges of the CNN cell array.

The circulation in the barotropic ocean is generally the result of the wind stress at the ocean's surface and the source sink mass flows at the basin boundaries. In this paper we use steady wind to force our model. In this case the ocean will generally arrive at a steady circulation after an initial transient behavior.

CNN-UM solution

Solution of equations a)-c) on a CNN-UM architecture requires finite difference approximation on a uniform square grid. The spatial derivatives can be approximated by the following well known finite difference schemes and CNN templates:

$$\frac{\partial}{\partial x} \approx \frac{1}{2\Delta x}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} = A_{dx} \qquad \text{i)}$$

$$\frac{\partial}{\partial y} \approx \frac{1}{2\Delta x}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} = A_{dy} \qquad \text{j)}$$

$$\nabla^2 \approx \frac{1}{\Delta x^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = A_n \qquad \text{k)}$$

Using these templates the pressure and lateral viscosity terms can be easily computed on a CNN-UM architecture. However the computation of the advection terms requires the following non-linear CNN template which can not be implemented on the present analog CNN-UM architectures.

$$u_{x,ij}\frac{\partial}{\partial x} \approx \frac{u_{x,ij}}{2\Delta x}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} = A_{x,x,ij} \qquad \text{l)}$$

Most ocean models arrange the time dependent variables u_x, u_y and η on a staggered grid called C-grid. In this case the pressure p and height H variables are located at the center of the mesh boxes, and mass transports u_x and u_y at the center of the box boundaries facing the x and y directions respectively. In this case the state equation of the ocean model can be solved by a one layer CNN but the required template size is 5×5 and space variant templates should be used. Another approach to use 3 layers for the 3 time dependent variables. In this case the CNN-UM solution can be described by the following equations:

$$\frac{du_{x,ij}}{dt} = f_{ij}u_{y,ij} - gH_{ij}\sum A_{dx}\eta + \tau_{wx,ij} - \sigma' u_{x,ij} + A_{ij}\sum A_n u_x - \frac{1}{H_{ij}}\left(\sum A_{x,x,ij}u_x + \sum A_{x,y,ij}u_y\right) \qquad \text{m)}$$

$$\frac{du_{y,ij}}{dt} = -f_{ij}u_{x,ij} - gH_{ij}\sum A_{dy}\eta + \tau_{wy,ij} - \sigma' u_{y,ij} + A_{ij}\sum A_n u_y - \frac{1}{H_{ij}}\left(\sum A_{y,x,ij}u_x + \sum A_{y,y,ij}u_y\right) \qquad \text{n)}$$

$$\frac{d\eta_{ij}}{dt} = -\sum A_{dx}u_x - \sum A_{dy}u_y \qquad \text{o)}$$

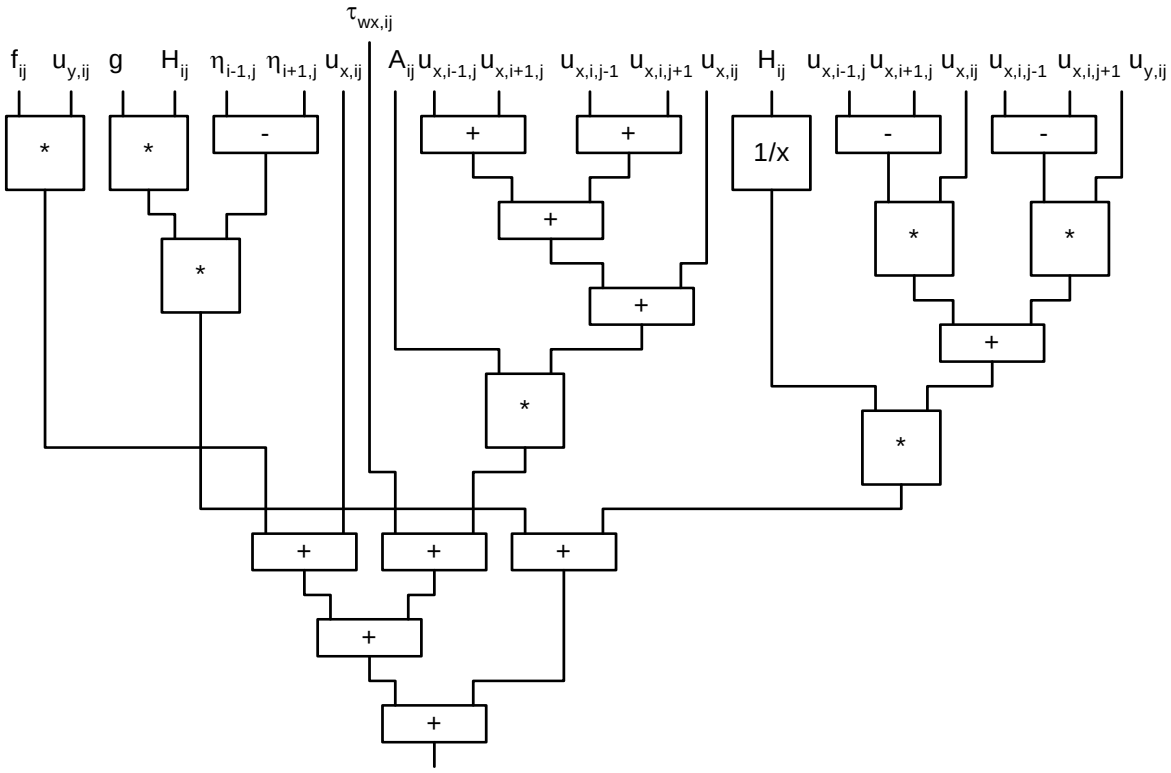


Fig. 1. The proposed arithmetic unit to compute the derivative of u_x.

(Multiplication with -4 in template A_n and division with $2\Delta x$ and Δx^2 is done by shifts. For simplicity these shifts are not shown on the figure.)

The emulated digital solution

Using equation m)-o) and templates i)-l) an analogic algorithm can be constructed to solve the state equation of the barotropic ocean model. However the non-linear advection terms do not allow us to implement our algorithm on the present analog CNN-UM chips. The non-linear behavior can be modeled by using software simulation but this solution does not differ from the traditional approach and does not provide any performance advantage.

Some previous results show that configurable emulated digital architectures can be very efficiently used in the computation of the CNN dynamics and in the solution of simple PDEs. The Falcon emulated digital CNN-UM architecture can be modified to handle the non-linear templates required in the advection terms of the ocean model. The required blocks are an additional multiplier to compute the non-linearity and a memory unit to store the required $u_{x,ij}$ or $u_{y,ij}$ values. Of course this modification requires the redesign of the whole control unit of the processor.

However this modified Falcon architecture can run the analogic algorithm of the ocean model its performance would not be significant because six templates should be run to compute the next step of the ocean model. The performance can be greatly improved by designing a specialized arithmetic unit which can compute these templates fully parallel. Instead building a general CNN-UM architecture which can handle non-linear templates an array of specialized cells is designed which can directly solve the state equation of the discretized ocean model.

To emulate the behavior of the specialized cells the continuous state equations m)-o) must be discretized in time. In the solution the simple forward Euler formula is used but in this case we have an upper limit on the Δt timestep. The maximal value of the timestep can be computed by using the Courant-Friedrichs-Levy (CFL) stability condition.

$$\Delta t < \Delta x / c_w \qquad \text{p)}$$

Where Δt is the timestep, Δx is the distance between the grid points and c_w is the speed of the surface gravity waves typically $c_w = \sqrt{gH}$.

Computation of the derivatives of u_x and u_y is the most complicated part of the arithmetic unit. The proposed structure to compute the derivative of u_x is shown on Fig. 1. Similar circuit is required to compute the derivative of u_y fortunately the results of the multiplication of g and H_{ij} and the computation of the reciprocal of H_{ij} can be used in this part too. This complex arithmetic unit can compute the derivatives and update the cell's state value in one clock cycle but it requires pipelining to achieve high clock speeds. The values of Δx and Δt is restricted to be integer powers of two. In this case multiplication and division by these values can be done by shifts. This simple

trick makes it possible to eliminate several multipliers from the arithmetic unit and greatly reduces the area requirements.

The arithmetic unit requires 20 input values in each clock cycle to compute a new cell value. It is obvious that these values cannot be provided from a separate memory. To solve this I/O bottleneck a belt of the cell array should be stored on the chip. In the case of u_x, u_y and η two lines should be stored because these lines are required in the computation of the spatial derivatives. From the remaining values such as f, H, A, τ_{wx} and τ_{wy} only one line should be stored for synchronization purposes.

Inside the arithmetic unit fixed point number representation is used because fixed point arithmetic requires much smaller area than floating point arithmetic. The bit width of the different input values can be configured independently before the synthesis. The width of the values are computed using simple rules and heuristics. For example the deepest point of the Pacific Ocean is about 11,000 meters deep. So 14 bits is required to represent this depth H with one meter accuracy. This is far more accurate than the available data of the ocean bottom so the last or the last two bits can be safely removed. In this case we always have to multiply H by 2 or 4 if it is used in the computations. Fortunately this multiplication can be implemented by shifts. Similar considerations can be used to determine the required bit width and the displacement of the radix point for the remaining constant values such as f, A, g, τ_{wx} and τ_{wy}. Using fixed point numbers with various width and displacement values makes it harder to design the arithmetic unit but it is worthwhile because the area can be reduced and clock speed is increased.

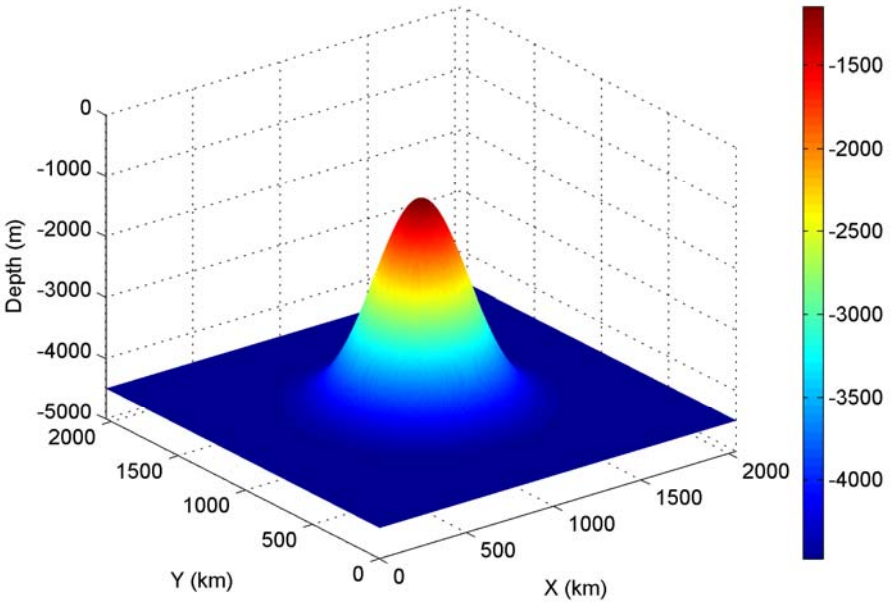
In our recent implementation predefined configurable multipliers from Xilinx are used to simplify circuit design. The maximum input width of this IP core is 64 bit thus the bit width of the u and η values can not be larger than 31 and 34 bits to avoid rounding errors inside the arithmetic unit. In this case 41 dedicated 18 bit by 18 bit signed multipliers are required to implement the arithmetic unit. Of course the bit width can be further increased by using custom multipliers. The required resources to implement this arithmetic unit on Xilinx Virtex series FPGAs is summarized in column General on Table I.

Results

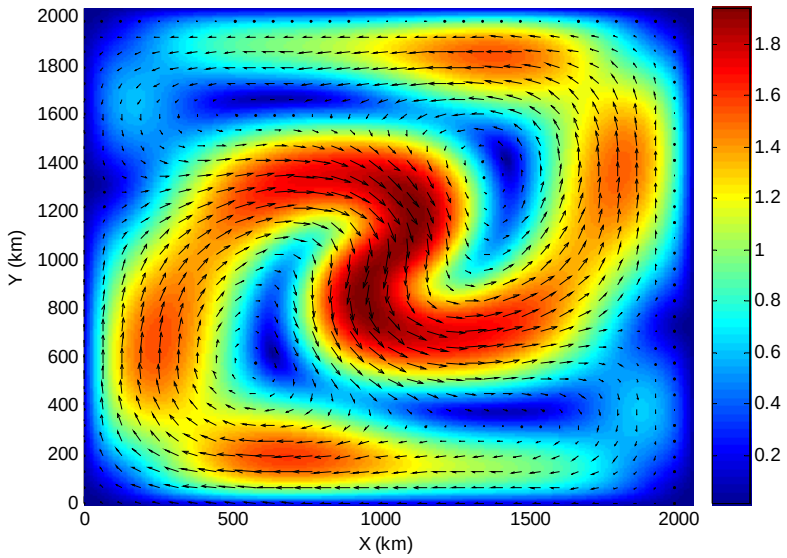
The proposed architecture will be implemented on our RC200 prototyping board from Celoxica Ltd.. The XC2V1000 FPGA on this card can host one arithmetic unit which makes it possible to compute a new cell value in one clock cycle. Unfortunately the board has 72 bit wide data bus, so 5 clock cycles are required to read a new cell value and to store the results. The performance can be increased by slightly lowering the precision as shown in column RC200 on Table I. and implementing three memory units which use the arithmetic unit alternately. In this case 4 clock cycles are required to compute 3 new cell values and the utilization of the arithmetic unit is 75%. The performance of the system is limited by the speed of the on board memory resulting in a maximum clock frequency of 90MHz. In this case the performance of the chip is 67.5 million cell update/s. The size of the memory is also a limiting factor because the input and state values must fit into the 4Mbyte memory of the board. The cell array is restricted to 512×512 cells by the limited amount of on-board memory however the XC2V1000 FPGA can be used to work with 1024 or even 2048 cell wide arrays.

By using the new Virtex-II Pro devices with larger and faster memory the performance of the architecture can reach 200MHz clock rate and can compute a new cell value in each clock cycle. Additionally the huge amount of on-chip memory and multipliers on the largest XC2VP125 FPGA makes it possible to implement 14 separate arithmetic units. These arithmetic units work in parallel and the cumulative performance is 2800 million cell update/s. On the other hand the large number of arithmetic units makes it possible to implement more accurate numerical methods.

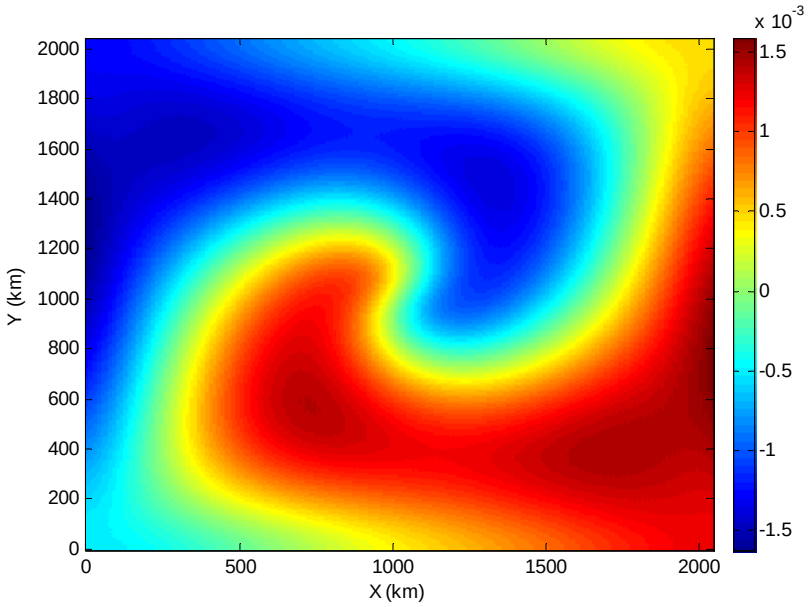
The results of the different fixed point computation are compared to the 64 bit floating point results. To evaluate our solution a simple model is used. The size of the modeled ocean is 2097km, the boundaries are closed, the grid size is 512×512 and the grid resolution is 4096m. The model is forced by a wind blowing from west and its value is constant along the x direction. The wind speed in the y direction is described by a reversed parabolic curve where the speed is zero at the edges and 8m/s in the center. The results of the 36 bit fixed-point computations after 72 hours of simulation time using 32s timestep are shown in Figure 3.


Depth (m)
0
-1000
-2000
-3000
-4000
-5000
2000
1500
1000
500
0
Y (km)
0
500
1000
1500
2000
X (km)
-1500
-2000
-2500
-3000
-3500
-4000

(a)


Y (km)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
500
1000
1500
2000
X (km)
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2

(b)



(c)

Fig. 3. Results after 72 hours simulation (a) Bottom topography: seamount, (b) flow direction, (c) elevation

TABLE I
RESOURCE REQUIREMENTS AND DEVICE UTILIZATION OF THE ARITHMETIC UNIT

	Bit width		
Variable	General	RC200	
f	18	10	
H	17	10	
A	17	10	
τ_w	18	12	
η	34	30	
u	31	30	
I/O bus width (bit)	184	144	
Required resources			
18x18 bit multiplier	41	41	
18kbit Block RAM	16	13	
Part number	18x18 bit multiplier utilization		Available resources
XC2V1000	103%	103%	40
XC2V8000	24%	24%	168
XC2VP125	7%	7%	556
Part number	18kbit Block RAM utilization		Available resources
XC2V1000	40%	33%	40
XC2V8000	10%	8%	168
XC2VP125	3%	2%	556

Reference

Z. Nagy, Zs. Vörösházi,P. Szolgay, “Emulated Digital CNN-UM Solution of Partial Differential Equations”, *Int. J. CTA, Vol. 34, No. 4, pp. 445-470 (2006)*

4.4. 2D COMPRESSIBLE FLOW SIMULATION ON EMULATED DIGITAL CNN-UM

For engineering applications in the topic of flow simulation the basic of developments is the well-known Navier-Stokes system of equations. This system was determined from the fundamental laws of mass conservation, momentum conservation and energy conservation.

In this paper we will concentrate on inviscid, compressible fluids where the dissipative, transport phenomena of viscosity, mass diffusion and thermal conductivity can be neglected. Therefore if we simply drop all the terms involving friction and thermal conduction from the Navier-Stokes equations, than we have the Euler equations, so the governing equations for inviscid flows .

Euler equations

The resulting equations according to the above mentioned physical principles for an unsteady, two-dimensional, compressible inviscid flow are displayed below without external sources. In order to take the compressibility and variations of density in high-speed flows into account, we utilize the conservation form of the governing equations, using the density-based formulation.

Continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathrm{V}) = 0 \tag{1}$$

Momentum equations

x component:
$$\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho u \mathrm{V}) = -\frac{\partial p}{\partial x} \tag{2}$$

y component:
$$\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho v \mathrm{V}) = -\frac{\partial p}{\partial y} \tag{3}$$

where t denotes time, x and y are the space coordinates, furthermore in two-dimensional Cartesian coordinates the vector operator ∇ is defined as

$$\nabla \equiv \mathrm{i}\frac{\partial}{\partial x} + \mathrm{j}\frac{\partial}{\partial y} \tag{4}$$

The dependent variables are ρ, $V(u, v)$ and p and they denote the density, the velocity vector field and the scalar field of pressure, respectively. We can see in the above-mentioned equations that we have three equations in terms of four unknown flow-field variables. In aerodynamics, it is generally reasonable to assume that the gas is perfect gas so the equation of state can be written in the following form:

$$p = \rho R T \tag{5}$$

where R is the specific gas constant, and its value in case of air is $286,9 \ \frac{J}{\mathrm{kg} \cdot \mathrm{K}}$ and T is the absolute temperature and the temperature value can be defined as an initial condition because we made isothermal system consideration. For this reason the fourth governing equation, the energy equation, can be neglected.

Discretization of PDE's in time and space

In implementation of Euler equations on different hardware units it is necessary to discretize the system of equations both in accordance with space and time and it is advantageous to chose finite difference approximation method using explicit integrating formula because of the regular, rectangular arrangement of the processing elements in the CNN-UM architecture. So according to this we have tried different explicit approximations such as the

- forward Euler method,
- Lax method and
- Lax-Wendroff method .

In the next sections we will examine these methods applied to two-dimensional first order hyperbolic Euler equations from the point of view of their stability, realizability with CNN templates, usability in different engineering applications and their hardware utilization on an emulated digital CNN-UM solution.

During the discretization of Euler equations we are interested in replacing the different partial derivatives with suitable algebraic difference quotients.

Euler's forward time central space method

Euler's forward time and central space (FTCS) approximation method computes the partial derivative of a given two-dimensional function *u* around a given point approximating the solutions using only the first two terms of the Taylor expansion of the function. For the temporal derivatives the FTCS applies (6) the first order accurate forward difference formula, while the spatial derivatives are computed using (7) the central difference formula having second order accuracy.

$$\left(\frac{\partial u}{\partial t}\right)_{i,j} = \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} - \frac{\Delta t}{2}\frac{\partial^2 u}{\partial x^2} \tag{6}$$

$$\left(\frac{\partial u}{\partial x}\right)_{i,j} = \frac{u_{i+1,j}^{n} - u_{i,j}^{n}}{2\Delta x} - \frac{(\Delta t)^2}{6}\frac{\partial^3 u}{\partial x^3} \tag{7}$$

Substituting (6) and (7) into (1)-(3) the following discretized formulas can be derived:

$$\rho_{i,j}^{n+1} = \rho_{i,j}^{n} - \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}v_{i,j-1}^{n}\right) \tag{8}$$

$$\begin{aligned}\rho_{i,j}^{n+1}u_{i,j}^{n+1} &= \rho_{i,j}^{n}u_{i,j}^{n} - RT\frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n} - \rho_{i-1,j}^{n}\right)\\ &- \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n}u_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}u_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}u_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}u_{i,j-1}^{n}v_{i,j-1}^{n}\right)\end{aligned} \tag{9}$$

$$\begin{aligned}\rho_{i,j}^{n+1}v_{i,j}^{n+1} &= \rho_{i,j}^{n}v_{i,j}^{n} - RT\frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n} - \rho_{i,j-1}^{n}\right)\\ &- \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n}v_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}v_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}v_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}v_{i,j-1}^{n}v_{i,j-1}^{n}\right)\end{aligned} \tag{10}$$

where Δt is time-step, while Δx and Δy denote differences between grid points in directions *x* and *y*.

The von Neumann stability analysis for hyperbolic equations shows that the solutions applying the FCTS methods will be unconditionally unstable.

Lax and Lax-Wendroff methods

In the Lax method the FTCS differencing scheme is used but to maintain stability we replace $u_{i,j}^{n}$ by its average so the following discretized equations can be determined:

$$\begin{aligned}\rho_{i,j}^{n+1} &= \frac{1}{4}\left(\rho_{i,j+1}^{n} + \rho_{i,j-1}^{n} + \rho_{i+1,j}^{n} + \rho_{i-1,j}^{n}\right)\\ &- \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}v_{i,j-1}^{n}\right)\end{aligned} \tag{11}$$

$$\begin{aligned}\rho_{i,j}^{n+1}u_{i,j}^{n+1} &= \frac{1}{4}\left(\rho_{i,j+1}^{n}u_{i,j+1}^{n} + \rho_{i,j-1}^{n}u_{i,j-1}^{n} + \rho_{i+1,j}^{n}u_{i+1,j}^{n} + \rho_{i-1,j}^{n}u_{i-1,j}^{n}\right) - RT\frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n} - \rho_{i-1,j}^{n}\right)\\ &- \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n}u_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}u_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}u_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}u_{i,j-1}^{n}v_{i,j-1}^{n}\right)\end{aligned} \tag{12}$$

$$\begin{aligned}\rho_{i,j}^{n+1}v_{i,j}^{n+1} &= \frac{1}{4}\left(\rho_{i,j+1}^{n}v_{i,j+1}^{n} + \rho_{i,j-1}^{n}v_{i,j-1}^{n} + \rho_{i+1,j}^{n}v_{i+1,j}^{n} + \rho_{i-1,j}^{n}v_{i-1,j}^{n}\right) - RT\frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n} - \rho_{i,j-1}^{n}\right)\\ &- \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n}v_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}v_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}v_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}v_{i,j-1}^{n}v_{i,j-1}^{n}\right)\end{aligned} \tag{13}$$

The Lax-Wendroff method computes new values of dependent variables in two steps. First it evaluates the values at half-step time using the Lax method and in second step it applies leapfrog method with half-step. The computation formula for (2) can be seen below.

First step:

$$\rho_{i,j}^{n+1/2}u_{i,j}^{n+1/2} = \frac{1}{4}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n} + \rho_{i,j+1}^{n}u_{i,j+1}^{n} + \rho_{i-1,j}^{n}u_{i-1,j}^{n} + \rho_{i,j-1}^{n}u_{i,j-1}^{n}\right) - \frac{\Delta t}{2\Delta x}RT\left(\rho_{i+1,j}^{n} - \rho_{i-1,j}^{n}\right)$$
$$-\frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n}u_{i+1,j}^{n}u_{i+1,j}^{n} - \rho_{i-1,j}^{n}u_{i-1,j}^{n}u_{i-1,j}^{n}\right) - \frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n}u_{i,j+1}^{n}v_{i,j+1}^{n} - \rho_{i,j-1}^{n}u_{i,j-1}^{n}v_{i,j-1}^{n}\right) \tag{14}$$

Second step:

$$\rho_{i,j}^{n+1}u_{i,j}^{n+1} = \rho_{i,j}^{n}u_{i,j}^{n} - \frac{\Delta t}{2\Delta x}\left(\rho_{i+1,j}^{n+1/2}u_{i+1,j}^{n+1/2}u_{i+1,j}^{n+1/2} - \rho_{i-1,j}^{n+1/2}u_{i-1,j}^{n+1/2}u_{i-1,j}^{n+1/2}\right)$$
$$-\frac{\Delta t}{2\Delta y}\left(\rho_{i,j+1}^{n+1/2}u_{i,j+1}^{n+1/2}v_{i,j+1}^{n+1/2} - \rho_{i,j-1}^{n+1/2}u_{i,j-1}^{n+1/2}v_{i,j-1}^{n+1/2}\right) - \frac{\Delta t}{2\Delta x}RT\left(\rho_{i+1,j}^{n+1/2} - \rho_{i-1,j}^{n+1/2}\right) \tag{15}$$

The partial differential equations of density and the *y* directional velocity component can be discretized in similar way.

In equations (11) through (15) the notations are the same as they were in the discrete form of FTCS.

The von Neumann stability analysis shows in both cases of Lax and Lax-Wendroff methods that these schemes are conditionally stable in case of ordinary hyperbolic equations and although there is no analytical stability analysis to determine limiting time step requirements because of the nonlinear nature of the Euler equations, the following empirical formula can be applied.

$$\Delta t \le \frac{\sigma\left(\Delta t\right)_{CFL}}{1 + 2/\mathrm{Re}_{\Delta}} \tag{16}$$

where $\sigma \cong 0.7 - 0.9$, $\left(\Delta t\right)_{CFL}$ can be determined using the Courant-Friedrichs-Lewy condition and $\mathrm{Re}_{\Delta} = \min\left(\mathrm{Re}_{\Delta x}, \mathrm{Re}_{\Delta y}\right) \ge 0$, where Re denotes the Reynolds number.

Considering the above mentioned methods it is very remarkable that during the computation regular, rectangular grids can be used, so we have the possibility to implement these solutions on CNN-UM architecture. Furthermore the last two examined methods have some valuable advantages compared to the FTCS in case of hyperbolic partial differential equations:

- they conserve the mass in a closed system,
- the computational accuracy of Lax-Wendroff method is second order in time and space,
- the hardware utilization of the implemented Lax formula will be very similar compared to the case using FTCS method and
- oscillations caused by square points – like corners in the space – exist in case of Lax-Wendroff method, or the solution of Lax method is dissipative but the effect of these properties does not blow up like FTCS scheme if the time step is defined properly.

In 1995 a former CNN based Navier-Stokes solver was published, by which it was possible to divide both momentum equations with density because that solution was designed to model the behavior of incompressible fluids but in our solution it is necessary to use dividers due to compressible property of medium in order to get the *u* and *v* values after steps.

The Euler equations were solved by a modified Falcon processor array in which the arithmetic unit has been changed according to the discretized continuity and momentum equations. For the momentum equations the values of *R* and *T* were defined as an initial condition because we supposed slow motions and so isothermal system condition in the fluid flow.

Since each CNN cell has only one real output value, three layers are needed to represent the variables *u*, *v* and ρ in case of FTCS and Lax approximations. In these cases the CNN templates acting on the *u* layer can easily be taken from (9) and (12) in conformity with using the FTCS or the Lax scheme. Equations (17)-(19) show templates, in

which cells of different layers at positions (k, l) are connected to the cell of layer *u* at position (i, j). The terms in (9) and (12) including only ρ are realized by

$$A^{\rho,u} = -\frac{RT}{2\Delta x}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}\left[\rho_{kl}\right]. \tag{17}$$

The nonlinear terms are

$$A^{\rho u,u} = -\frac{1}{2\Delta x}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}\left[\rho_{kl} \cdot u_{kl}^2\right], \tag{18}$$

$$A^{\rho uv,u} = -\frac{1}{2\Delta x}\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}\left[\rho_{kl} \cdot u_{kl} \cdot v_{kl}\right]. \tag{19}$$

Of course the value of *u* can get only after the division with the density value. The templates for the ρ and *v* layers can be defined analogously.

In case of Lax-Wendroff scheme *u, v* and ρ denote the state variables of the first step, while *u', v'* and ρ' the state variables of the second computational step so the number of required layers is six. The linear and nonlinear templates can be determined in a very similar way as in case of FTCS or Lax methods. The only difference will be that the layers without comma will be interconnected with layers having comma notation. For example the nonlinear connection between the cell of layer *u* at position (i, j) and cells of ρu^2 at positions (k, l) in (15) can be described by the following template:

$$A^{\rho' u',u} = -\frac{1}{2\Delta x}\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}\left[\rho_{kl}' \cdot \left(u_{kl}'\right)^2\right]. \tag{20}$$

In accordance with the different discretized equation systems we have designed three complex circuits which are able to update state values of a cell in every clock cycle in emulated digital CNN-UM architecture.

The proposed arithmetic unit to compute the derivative of *u* layer using Lax method is shown in Fig. 1. In order that the different hardware units of this arithmetic could work with maximal parallelism and so achieve the highest possible clock speed during the computation, pipelining technique has been used.

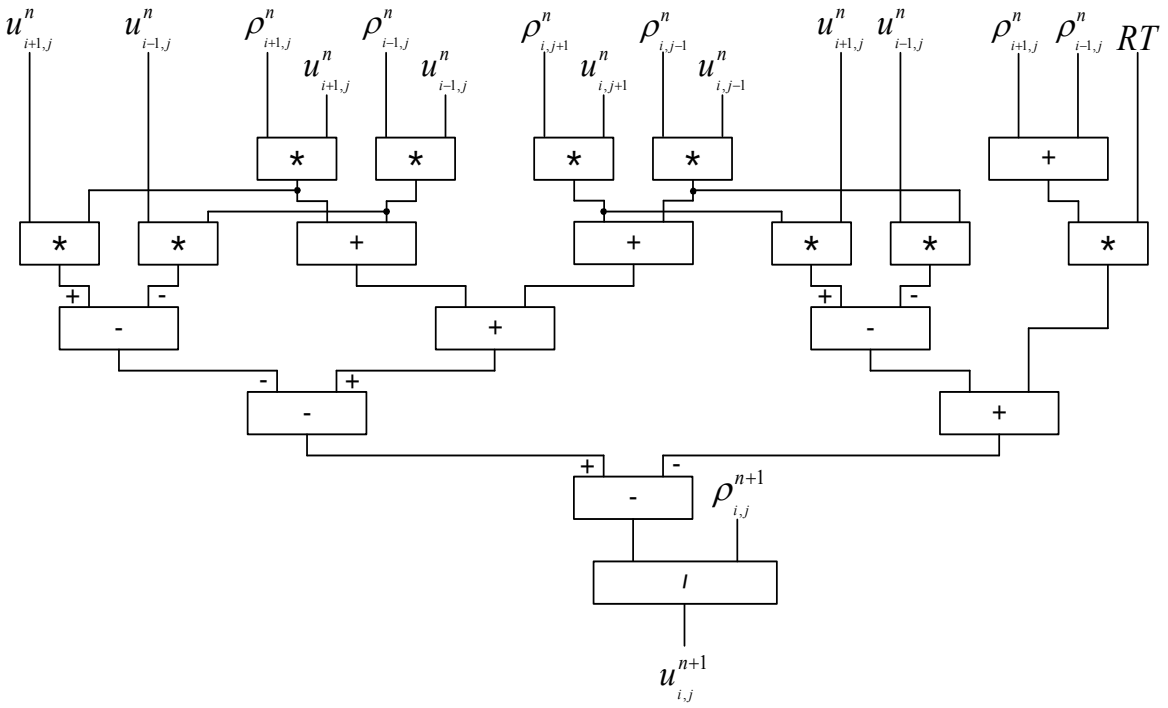


Figure 1. The proposed arithmetic unit to compute the derivative of u layer in the solution using Lax approximation method

Other trick can be applied if we choose the value of Δt, Δx and Δy to be integer power of two because the multiplication with these values can be done by shifts so we can eliminate several multipliers from the hardware and additionally the area requirements will be greatly reduced.

Comparing the equation systems of (8)-(10) and (11)-(13) one can see that the only modification in case of solution applying FTCS, which need to be executed, will be that two additional multipliers need to be build in but some adders can be eliminated.

The implementation of Euler equations discretized by the Lax-Wendroff scheme requires about two times larger hardware area than the previous solutions because in this case actually both the arithmetic units of Lax and FTCS solutions need to be implemented in one circuit.

The input and some result images of 32 bit precision simulation computing 1 million iterations with 2^{-10} second time step on a model having 128x128 grid points can be seen in Fig.10. The red area denotes larger, while the blue area means smaller density and the arrows show the direction of the flow. Using the Lax method the computational time was 1772.7 seconds using Athlon 64 3500+ processor, and 1098.9 seconds on one core of an Intel Core2Duo T7200 applying floating point numbers. This is equivalent to approximately 9.3 million and 14.7 million cell update/s, respectively. The Lax-Wendroff method is about 60% slower as the Lax method. In this case the computation can be carried out in 2852.4 s and 1803.5 s on the AMD and Intel processors, respectively.

Using the Lax method the previous simulation takes approximately just 123.18 s using our XC2V3000 FPGA and 65.5 s using the XUP2Pro so the computation has been accelerated approximately by 8.9 and 16.7 times compared to the performance of Core2Duo. Using the high performance XC4SX55 FPGA the simulation lasts 8.19 seconds, so the computation is 134-fold faster. The expected computing time of the Lax-Wendroff method on the XC4SX55 FPGA is 20.9 s which is 86.3 times faster than the Core2Duo microprocessor.

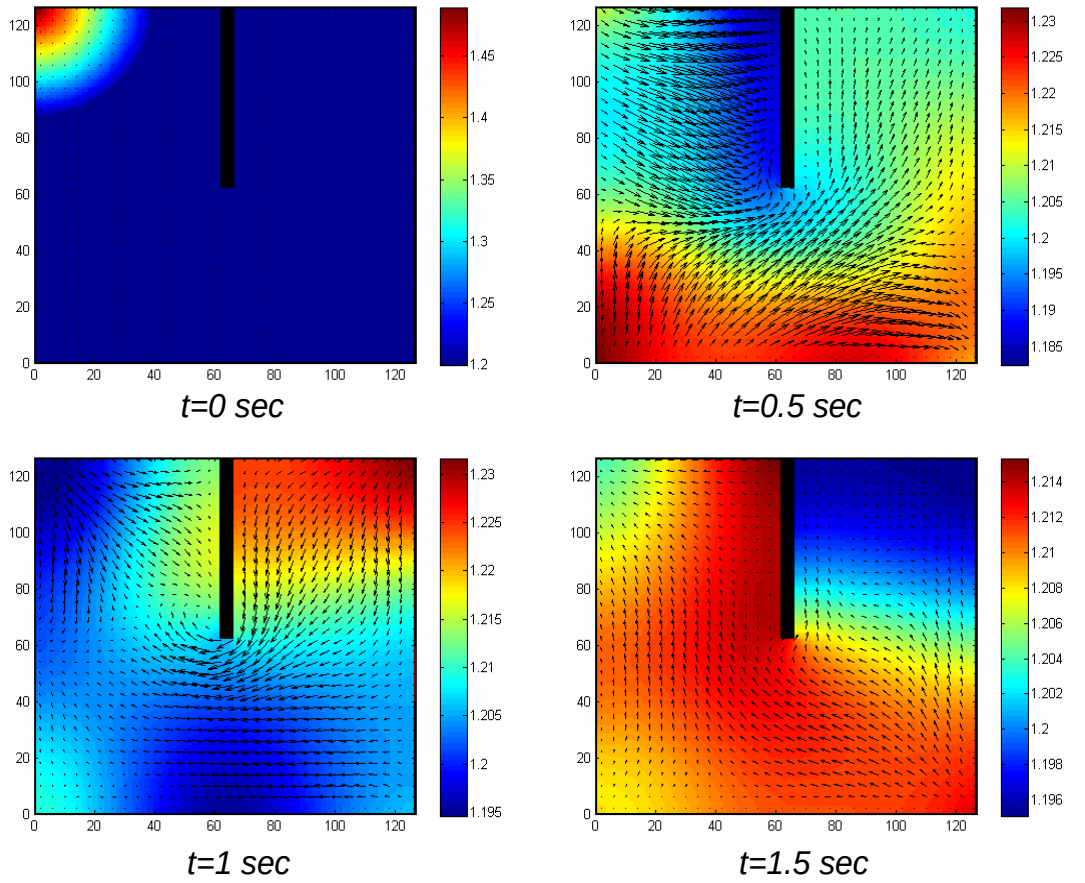


Figure 10. Image sequence about the fluid in a closed area inside with a wall computed with 32 bit fixed point numbers using the Lax method

Reference

S. Kocsárdi, Z. Nagy, Á. Csik, P. Szolgay,"Two-dimensional compressible flow simulation on emulated digital CNN-UM", *Int. J. CTA, Vol. 37, No.4, pp.569-585(2009)*

Chapter 5. SIMULATORS

In this section, two freely downloadable simulator is introduced. The first is the MATCNN simulator, which enables the simulation of any linear or non-linear CNN, and CNN template sequence. The second is the CI simulator, which supports the simulation of cellular automations.

5.1. MATCNN SIMULATOR

Linear templates specification

|

EDGE_I = -1.5;

Example 2: **DIFFUS** - performs linear diffusion (the linear *B* term is zero)

```
DIFFUS_A = [    0.1         0.15    0.1;
                            0.15    0           0.15;
                            0.1             0.15    0.1 ];
DIFFUS_I = 0;
```

Nonlinear "AB-type" template: [A B $\hat{A}$ $\hat{B}$ I]

The associated CNN state equation:

$$\frac{d}{dt}v_{x_{ij}}(t) = -v_{x_{ij}}(t) + \sum_{kl\in N_r} A_{ij,kl}v_{y_{kl}}(t) + \sum_{kl\in N_r} B_{ij,kl}v_{u_{kl}}(t) + I_{ij} + \sum_{kl\in N_r} \hat{A}_{ij,kl}(\Delta v_{yy}) + \sum_{kl\in N_r} \hat{B}_{ij,kl}(\Delta v_{uu}) \tag{3}$$

MATCNN format:

Example 1: **GRADIENT** - performs magnitude gradient thresholding
(the linear *B* term and nonlinear $\hat{A}$ term are zero)

```
GRADT_A = [    0       0       0;
                       0       2       0;
                       0       0       0 ];
GRADT_Bb = [   1       1       1;
                       1       0       1;
                       1       1       1 ];
GRADT_b = [ 1 3                -3 3   0 0   3 3 ]; (function of the nonlinear interaction, see later)
GARDT_I = -1.8;
```

Nonlinear "D-type" template: [A B $\hat{D}$ I]

The associated CNN state equation:

$$\frac{d}{dt}v_{x_{ij}}(t) = -v_{x_{ij}}(t) + \sum_{kl\in N_r} A_{ij,kl}v_{y_{kl}}(t) + \sum_{kl\in N_r} B_{ij,kl}v_{u_{kl}}(t) + I_{ij} + \sum_{kl\in N_r} \hat{D}_{ij,kl}(\Delta v) \tag{4}$$

MATCNN format:

Example 1: **MEDIAN** - performs median filtering
(the linear *B* term and the cell current *I* is zero)

```
MEDIAN_A = [  0       0       0;
                      0       1       0;
```

```
                                0       0       0 ];
MEDIAN_Dd = 0.5 *[      1       1       1;
                                                1       0       1;
                                                1       1       1 ];
MEDIAN_d = [ 0 2                0 -1    2 1     12 ]; (function of the nonlinear interaction, see later)
```

Example 2: **NLINDIFF** - performs nonlinear diffusion
(the linear *B* term and the cell current *I* is zero)

```
NLINDIFF_A = [          0       0       0;
                                        0       1       0;
                                        0       0       0 ];
NLINDIFF_Dd = [         0.5     1       0.5;
                                        1       0       1;
                                        0.5     1       0.5 ];
NLINDIFF_d = [ 1 5      -2 0    -0.1 0    0 1    0.1 0    2 0    122 ]
```

Nonlinear function specification in "AB-type" and "D-type" nonlinear templates

A unique nonlinear function *a*, *b*, and *d* is assigned to each nonlinear operator $\hat{A}$, $\hat{B}$, and $\hat{D}$, respectively. These nonlinear functions determine the nonlinear cell interaction and allowed to be piecewise-constant (pwc) or piecewise-linear (pwl). Their specification in *MATCNN* is as follows:

For *a* and *b*: [interp p_num x_1 y_1 x_2 y_2 … x_n y_n]

Where:
- interp - interpolation method (0 - pwc; 1 - pwl)
- p_num - number of points
- x1 y1 - first point ordinate and abscissa
- xn yn - last point ordinate and abscissa

Example (from **GRAD** template):
GRADT_b = [1 3 -3 3 0 0 3 3];
Pwl interpolation between three points: (-3.3), (0,0), (3,3), the absolute value "radial-basis" function.
Remark: *a* and *b* are always applied to the output and input difference values of the neighboring cells, respectively (see formula (1)).

For *d*: [interp p_num x_1 y_1 x_2 y_2 … x_n y_n intspec]

Where:
- interp - interpolation method (0 - pwc; 1 - pwl)
- p_num - number of points
- x1 y1 - first point ordinate and abscissa
- xn yn - last point ordinate and abscissa
- intspec - interaction specification

In case of nonlinearity *d* the interaction should also be specified, the valid codes are as follows:
For the interaction: $d(\Delta v)$

11 – $\Delta v = v_{u2kl} - v_{u1ij}$

12 – $\Delta v = v_{ukl} - v_{x_{ij}}$

13 – $\Delta v = v_{ukl} - v_{y_{ij}}$

21 – $\Delta v = v_{xkl} - v_{uij}$

22 – $\Delta v = v_{xkl} - v_{x_{ij}}$

23 – $\Delta v = v_{xkl} - v_{y_{ij}}$

31 – $\Delta v = v_{ykl} - v_{uij}$

32 – $\Delta v = v_{ykl} - v_{x_{ij}}$

33 – $\Delta v = v_{ykl} - v_{y_{ij}}$

For the interaction: $d(\Delta v)(\Delta v)$ 100 should be added to the above codes.

Remark: it can be seen that in case of the code 11 and code 33 the interaction is exactly that of the *a* and *b*, respectively (this also explains why is $\hat{D}$ called the generalized nonlinear operator since it includes both $\hat{A}$ and $\hat{B}$). The above specification of the interactions makes it possible to formulate the operators of the gray-scale morphology, statistical filtering and nonlinear diffusion as simple CNN templates.

Example 1 (from **MEDIAN** template):
MEDIAN_d = [0 2 0 -1 2 1 12]
Pwc interpolation, two points: (0,-1), (2,1) - a thresholding-type nonlinear function applied to the input and state difference of the neighboring cells (code: 12).
Modification to a sigmoid-type nolinearity (pwl interpolation, 4 points):
MEDIAN_d = [1 2 -2 -1 -0.5 -1 0.5 1 2 1 12]

Example 2 (from **NLINDIFF** template):
NLINDIFF_d = [1 5 -2 0 -0.1 0 0 1 0.1 0 2 0 122]
Pwl interpolation, five points: (-2,0), (-0.1, 0), (0, 1), (0.1, 0), (2,0) - a pwl radial-basis function applied to the state difference values. In the interaction the function output is also multiplied by the state difference value (code: 122=100 +22).

CNN Template Library

The *MATCNN* Toolbox includes a default CNN template library (temlib.m) where a number of tested CNN templates (linear, nonlinear "AB-type" and nonlinear "D-type") are given in the format discussed in subsection 1.3. The demonstration examples of 2.2 are using this template library. The user can define a similar library assigned to the algorithms.

Images Assigned to the CNN Models

The name of the images (global variables) assigned to all CNN models in *MATCNN* are as follows:
INPUT1 - ***U*** or $\boldsymbol{U_1}$ (primary input image of the CNN model)
INPUT2 - $\boldsymbol{U_2}$ (secondary input image of the CNN model)
STATE - ***X*** (state image of the CNN model)
OUTPUT - ***Y*** (output of the CNN model)
BIAS - ***B*** (bias image of the CNN model: "bias map")
MASK - ***M*** (mask image of the CNN model: "fixed-state map")

Remark: the sum of the constant cell current (*I*) specified in the CNN template and the space variant bias value ($\boldsymbol{B}_{ij}$) is the space variant current of the CNN model ($I_{ij} = I + \boldsymbol{B}_{ij}$). The default bias is zero. The mask is a binary map ($\boldsymbol{M}_{ij}$ = 1 or -1) and determines whether a CNN cell is active (operating) or inactive (fixed). The default mask value is 1 (all cells are active). (see also footnotes 1 and 2) Arithmetical, logical and $\hat{D}$ - type operators can have two input values ($\boldsymbol{U}=\boldsymbol{U_1}$ and $\boldsymbol{U_2}$).

Global Variables

Scripts and functions of the *MATCNN* assume that certain global variables exist and are initialized in the environment.

The global variables that should be modified by the user when simulating a CNN model are as follows:

- UseMask
- UseBiasMap
- Boundary
- TemGroup
- TimeStep
- IterNum

Subsection 1.7 explains the role of the above variables, more details and the initial setting can be found in subsection 2.1.

Remark: There exists another group of global environmental variables used and modified by *MATCNN* scripts and functions that should not be modified by the user. These global variables are listed and explained in subsection 2.1.

Running a CNN Simulation

To run a CNN simulation and visualize the result in *MATCNN* the following steps should be followed:

- ***set the CNN environment and template library***

  e.g. Cnn_SetEnv; - set the CNN environment
  TemGroup = 'MyTemLib'; - the template library is stored in mytemlib.m file

  Remark: the environment always has to be set, if the template library is not specified the default library of the *MATCNN* is temlib.m.

- ***initialize the input and state images assigned to the CNN model***

  e.g. INPUT1 = LBmp2CNN('Road'); - road.bmp is loaded to the input
  STATE = zeros(size(INPUT1)); - all initial state values are set to zero

  Remark: the input initialization is optional (depends on the template), but the state layer should always be initialized. In this phase noise can also be added to the images for test purposes (see CImNoise).

- ***specify whether the bias and mask image will be used***

  e.g. UseBiasMap = 1; - use the bias image
  UseMask = 0; - do not use the mask image

  Remark: if a the UseBiasMap global is set to 1 the BIAS image should be initialized, similarly when the UseMask global is set to 1 the MASK image should also be initialized. The Cnn_SetEnv *MATCNN* script sets these global variables to zero.

- ***specify CNN boundary condition***

  e.g. Boundary = -1; - the boundary is set to a constant value (-1)

  Remark: the boundary condition specified can be constant ($-1 \leq Boundary \leq 1$), zero flux (Boundary = 2) and torus (Boundary = 3).

- ***set the simulation parameters (time step and number of iteration steps)***

  e.g. TimeStep = 0.1; - the time step of the simulation is 0.1
  IterNum = 100; - the number of iteration steps is 100

Remark: the default values for TimeStep = 0.2 and IterNum = 25, that correspond approximately to 5τ (when $R = 1$ and $C = 1$) analog transient length, guarantee that in a non-propagating CNN transient all cells will reach the steady state value. However, for a number of templates different settings should be used.

- ***load the template that completely determines the CNN model***

  e.g. LoadTem('EDGE'); - load the **EDGE** template from the specified library

  Remark: All templates of a project or algorithm can be stored in a common library (M-file). The LoadTem function activates the given one from the specified library that will determine the CNN model of the simulation.

- ***run the simulation with the specified template***

  e.g. RunTem; - run the CNN simulation with the specified template

  Remark: the ODE of the CNN model is simulated using the forward Euler formula.

- ***visualize the result***

e.g. CNNShow(OUTPUT)- show the output of the CNN simulation

Remark: since the CNN is primarily referred to as a visual microprocessor, and the inputs and outputs are images, in most cases it might be helpful to visualize these images using different magnification rates, palettes etc. The user can exploit the capabilities of MATLAB's graphical interface and the Image Processing Toolbox when evaluating the CNN performance.

A simple example using the **EDGE** template from the default template library to perform edge detection on a black and white image (road.bmp) stored in user's directory. The output is visualized and saved as roadout.bmp.

```
-----------------------------------------
Cnn_SetEnv;
INPUT1 = LBmp2CNN('Road');
STATE = INPUT1;
Boundary = -1;
TimeStep = 0.1;
IterNum = 50;
LoadTem('EDGE');
RunTem;
CNNShow(OUTPUT);
SCNN2Bmp('RoadOut', OUTPUT);
-----------------------------------------
```

Subsection 2.1 gives an example for the linear, nonlinear "AB-type" and nonlinear "D-type" template. It is also shown how the template operations can be combined to built up an analogic CNN algorithm.

Running an analog-and-logic (analogic) CNN algorithm

Given an algorithm, defined – for example – by an UMF diagram. This algorithm contains elementary instructions defined by various templates and local logic operators. Demonstration algorithms are included in the MATCNN simulation program. See subsections 1.8.1 – 1.8.4 for details on running these algorithms made up of elementary instructions.

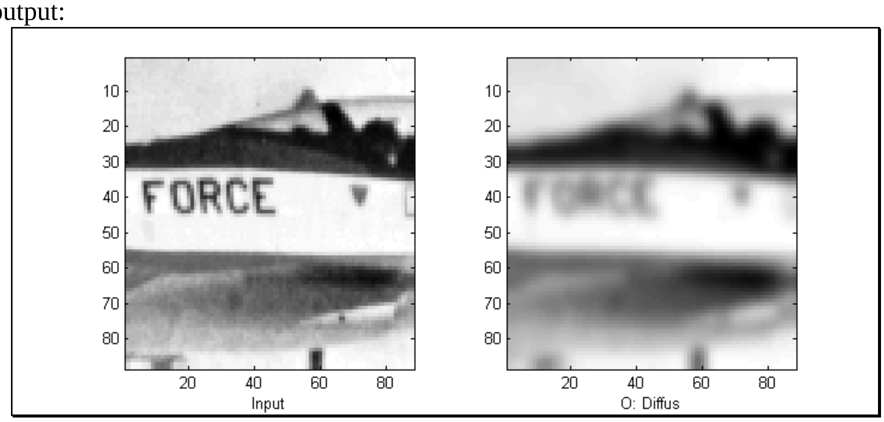
Sample CNN Simulation with a Linear Template

The first sample algorithm, demonstrates how to run a simulation with a linear DIFFUSE template using zero-flux boundary condition.

The *MATCNN* program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% D_LIN  Sample CNN operation with a linear template
 %
% set CNN environment
    Cnn_SetEnv                          % default environment
    TemGroup='TemLib';          % default template library
% load images, initialize layers
    load pic2;                                  % loads the image from pic2.mat to the INPUT1
    STATE = INPUT1;
% set boundary condition
    Boundary = 2;                               % zero flux boundary condition
% run linear template
    LoadTem('DIFFUS');          % loads the DIFFUS template (linear)
    TimeStep = 0.2;
    IterNum = 50;
    RunTem;                                     % runs the CNN simulation
% show result
    subplot(121); CNNShow(INPUT1);      % displays the input
     xlabel('Input');
    subplot(122); CNNShow(OUTPUT);      % displays the output
     xlabel('O: Diffus');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The MATLAB output:



Sample CNN Simulation with a Nonlinear "AB - type" Template

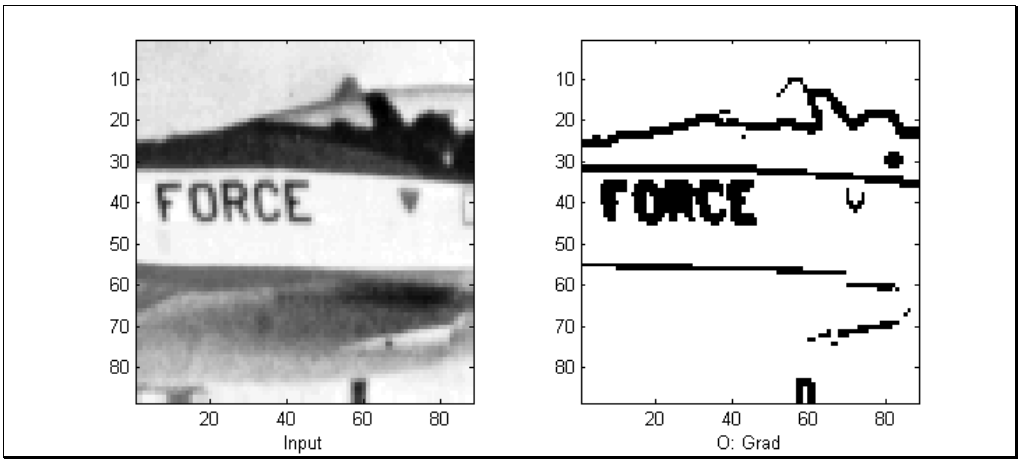
The second algorithm, demonstrates a non-linear gradient template using zero-flux boundary condition.

The *MATCNN* program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% D_NLINAB  Sample CNN operation with a nonlinear AB template
 %
% set CNN environment
    Cnn_SetEnv                      % default environment
    TemGroup = 'TemLib';% default template library
% load images, initialize layers
    load pic2;                      % loads the image from pic2.mat to the INPUT1
    STATE = INPUT1;
% set boundary condition
    Boundary = 2;                   % zero flux boundary condition
% run nonlinear AB template
    LoadTem('GRADT');   % loads the GRADT template (nonlinear "AB-type")
    TimeStep = 0.4;
    IterNum = 15;
    RunTem;                         % runs the CNN simulation
% show result
    subplot(211); cnnshow(INPUT1);          % displays the input
     xlabel('Input');
    subplot(212); cnnshow(OUTPUT);          % displays the output
     xlabel('O: Grad');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The MATLAB output:



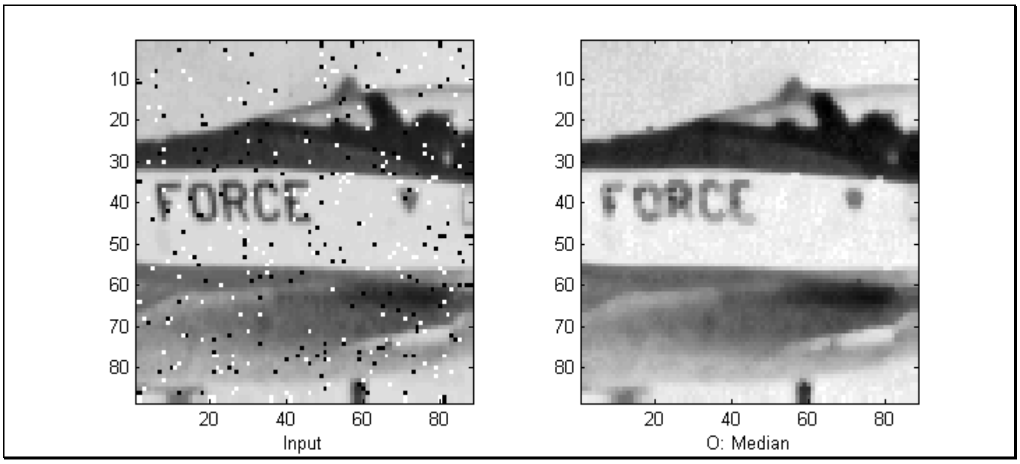
Sample CNN Simulation with a Nonlinear "D - type" Template

The third algorithm, demonstrates how to filter a noisy image with a D-type non-linear medain template using zero-flux boundary condition.

The *MATCNN* program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% D_NLIND  Sample CNN operation with a nonlinear D template
 %
% set CNN environment
    Cnn_SetEnv                              % default environment
    TemGroup = 'TemLib';          % default template library
% load images, initialize layers
    load pic2;                              % loads the image from pic2.mat to the INPUT1
    STATE = INPUT1;
% put noise in the image
    STATE1 = cimnoise(STATE1, 'salt & pepper',0.05);
    INPUT1 = STATE1;    % 1st input
    INPUT2 = STATE1;    % 2nd input
% set boundary condition
    Boundary = 2;                           % zero flux boundary condition
% run nonlinear D template
    LoadTem('MEDIAN');            % loads the MEDIAN template (nonlinear "D-type")
    TimeStep = 0.02;
    IterNum = 50;
    RunTem;                                 % runs the CNN simulation
% show result
    subplot(211); cnnshow(INPUT1);          % displays the input
     xlabel('Input');
    subplot(212); cnnshow(OUTPUT1);      % displays the output
     xlabel('O: Median');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The MATLAB output:



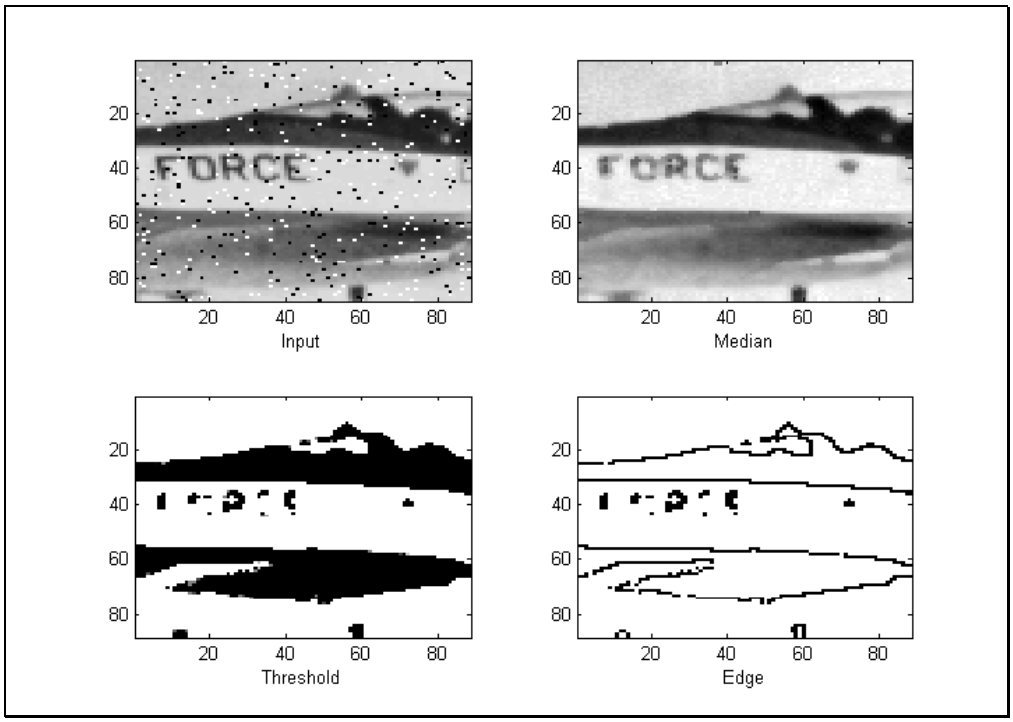
Sample Analogic CNN Algorithm

The fourth algorithm is an example for a simulation using multiple templates. Noise is added to the original image, and a series of templates: threshold, median, edge templates are executed.

The *MATCNN* program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% D_ALGO  Sample ANALOGIC CNN ALGHORITHM with linear and nonlinear templates
 %
% set CNN environment
    Cnn_SetEnv                          % default environment
    TemGroup = 'TemLib';                % default template library
% load images, initialize layers
    load pic2;                                  % loads the image from pic2.mat to the INPUT1
    STATE = INPUT1;
% put noise in the image
    STATE = cimnoise(STATE, 'salt & pepper',0.05);
    INPUT1 = STATE;                     % 1st input
    INPUT2 = STATE;                     % 2nd input
    LAM1 = STATE;                       % the nosy original image is also stored in LAM1
% set boundary condition
    Boundary = 2;                               % zero flux boundary condition
% run nonlinear “D-type” template (filtering)
    LoadTem('MEDIAN');          % loads the MEDIAN template
    TimeStep = 0.02;
    IterNum = 50;
    RunTem;                                     % runs the CNN simulation
    LAM2 = OUTPUT;                      % the CNN output is stored in LAM2
% run linear templates (thresholding and edge detection)
    LoadTem('THRES');                   % loads the THRES template
    TimeStep = 0.4;
    IterNum = 15;
    RunTem;                                     % runs the CNN simulation
    INPUT1 = OUTPUT;
    STATE = OUTPUT;
    LoadTem('EDGE');                    % loads the EDGE template
    RunTem;                                     % runs the CNN simulation
% show results
    subplot(2,2,1); cnnshow(LAM1);              % displays the noisy original image
     xlabel(‘Input’);
    subplot(2,2,2); cnnshow(LAM2);              % displays the result of noise filtering
     xlabel(‘1. O: Median’);
    subplot(2,2,3); cnnshow(INPUT1);            % displays the result of thresholding
     xlabel(‘2. O: Threshold’);
    subplot(2,2,4); cnnshow(OUTPUT);    % displays the result of edge detection
     xlabel(‘3. O: Edge’);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The MATLAB output:



MATCNN SIMULATOR REFERENCES

In this section a short reference is given to the *MATCNN* scripts, functions and global variables and some CNN examples are shown.

List of All *MATCNN* Scripts, Functions and Global Variables

MATCNN M-files (scripts and functions):

Basic:

Cnn_SetEnv	-	set CNN environment and initialize global variables
ShowEnv	-	show global variables of the CNN environment
LoadTem	-	load the specified CNN template
ShowTem	-	show the actual template loaded into the CNN environment
RunTem	-	run the specified CNN template
CNNShow	-	show a CNN-type intensity image
TemLib	-	default CNN template library

Miscellaneous:

CNN2Gray	-	convert a CNN-type image to a gray-scale intensity image
Gray2CNN	-	convert a gray-scale intensity image to a CNN-type image
CBound	-	add a specified boundary to a CNN-type image
CImNoise	-	put noise in a CNN-type image
LBmp2CNN	-	load a BMP file from disk and convert it to a CNN-type image
SCNN2Bmp	-	save a CNN-type image to disk in BMP format

MATCNN MEX-files:

Basic MEX-files:

tlinear	-	linear CNN template simulation
tnlinab	-	nonlinear “AB-type” CNN template simulation
tnlind	-	nonlinear “D-type” CNN template simulation

Special MEX-files implementing different nonlinear filters:
(no upper level support from the *MATCNN* environment)

tmedian	-	median (ranked order) CNN template simulation
tmedianh	-	median (ranked order) CNN template simulation (for switch analysis)
tanisod	-	anisotropic (nonlinear) diffusion CNN template simulation
ordstat	-	order statistic (OS) filters
modfilt	-	mode filters

MATCNN demos:

D_Lin	-	linear CNN template demo (**EDGE**)
D_NLinAB	-	nonlinear “AB-type” template demo (**GRADT**)
D_NLinD	-	nonlinear “D-type” template demo (**MEDIAN**)
D_Algo	-	analogic CNN algorithm demo (edge detection)
ShowDPic	-	show demo pictures of the environment (pic1-pic8)

MATCNN global variables (complete list):

Global variables to be set externally:

UseBiasMap	-	determines whether the CNN uses a space-variant current
UseMask	-	determines whether the CNN operates in B&W fixed-state mode
Boundary	-	boundary condition for the CNN layer (and boundary value)

(-1<= constant <=1, 2: duplicated - zero flux, 3: torus)
TemGroup - name of the actual template group
TimeStep - time step of the simulation
IterNum - number of iteration steps in simulation

Global variables - initial setting (to be modified during the simulations):
UseBiasMap = 0; - no bias map
UseMask = 0; - no mask
Boundary = 2; - zero-flux boundary condition
TemGroup = 'TemLib'; - the default template library is TemLib
TimeStep = 0.2; - default time step
IterNum = 25; - default number of simulation steps

Global variables modified by MATCNN scripts and functions /default values/:
TemName - name of the actual template /””/
TemNum - order number of the actual template used in the algorithm /0/
TemType - type of the actual template /0/
(0: linear, 1; nonlinear AB, 2: nonlinear D)
Atem - linear feedback /0/
Btem - linear control /0/
At_n - nonlinear feedback /0/
nlin_a - nonlinear function in feedback /0/
Bt_n - nonlinear control /0/
nlin_b - nonlinear function in control /0/
Dt_n - generalized nonlinear interaction /0/
nlin_d - nonlinear function in generalized term /0/
I - cell current /0/
RunText - display template name, order number,
time step, number of steps and report
module call before a CNN template execution /0/
CNNEnv - status variable (1: CNN environment is loaded, 0: not loaded) /0/

INPUT1 - primary input image of the CNN model
INPUT2 - secondary input image of the CNN model
STATE - state image of the CNN model
OUTPUT - output image of the CNN model
MASK - mask image of the CNN model
BIAS - bias image of the CNN model

The default template library is specified in “TemLib” template group where the syntax of linear and nonlinear CNN templates is also given.

Type HELP "FName" in MATLAB environment to learn the details on the *MATCNN* M-files (scripts and functions) and MEX-files!

Remarks:

The installation guide and detailed information about the available UNIX script files (slink - creates a soft link to the existing M-files to deal with the case sensitivity problem, cunix - compiles all *.c source files and creates the MEX-kernels in UNIX environment) and the Windows batch file (cwindows.bat - compiles all *.c source files and creates the DLL MEX-kernels in Windows environment) can be found in the file readme.txt where the known bug reports are also included.

5.2. 1D Cellular Automata SIMULATOR

User guide
version 1.0

Brief notes about 1D binary Cellular Automata

A one-dimensional binary Cellular Automaton consist of an array of length *L* of dynamical systems called *cells*, which can take only two states (e.g., 0 and 1). Cells are iteratively updated synchronously and the state of each cell at iteration *n+1* depends on the states of its nearest neighbors at iteration *n*. This dependence can be conveniently represented by means of a truth table that indicates which output state β_i, either 0 or 1, corresponds to each of the eight possible inputs, from 000 to 111.

Input	Output
000	β_0
001	β_1
010	β_2
...	
111	β_7

Usually, the bit string is considered to be an “unrolled” ring, in the sense that the right neighbor of the last bit is the first bit of the string, while the left neighbor of the first string is the last neighbor.

The 8-tuple (β_7 $\beta_{6...}$ β_0) defines the dynamics of the Cellular Automaton, and hence it is called *CA rule*. There are $2^8 = 256$ possible *rules,* which can be numbered from 0 to 255 through the formula:

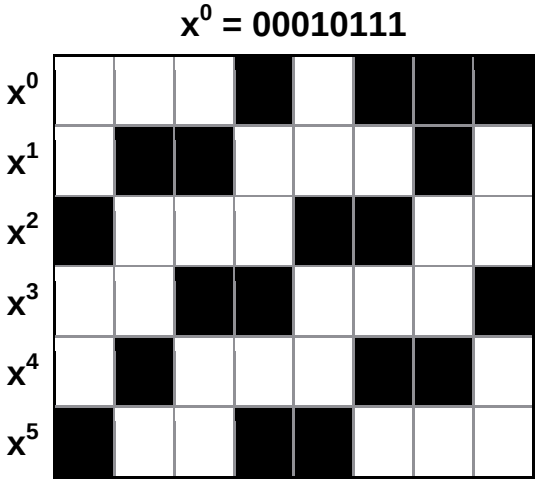
$$N = \sum_{i=0}^{7} \beta_i \cdot 2^i$$

Therefore, a CA rule transforms a given bit string x^n into another bit string x^{n+1} at each iteration; given an initial state x^0, the union of all x^n, for every *n* up to infinite, is called *space-time pattern* obtained from x^0.

Even though Cellular Automata have been introduced long before Cellular Neural/Nonlinear Networks, CA are actually a special case of CNNs. Indeed, it is possible to prove that every local rule is a code for attractors of a dynamical system which can written as a Universal CNN cell.

Example

Rule 131 can be expressed as (β_7 $\beta_{6...}$ β_0)=(10000011). Its *space-time pattern* starting from the initial condition x^0=00101110 for L=8, is



It is now evident that the *space-time pattern* provides a direct representation of the evolution of a given bit string throughout time.

1D CA Simulator

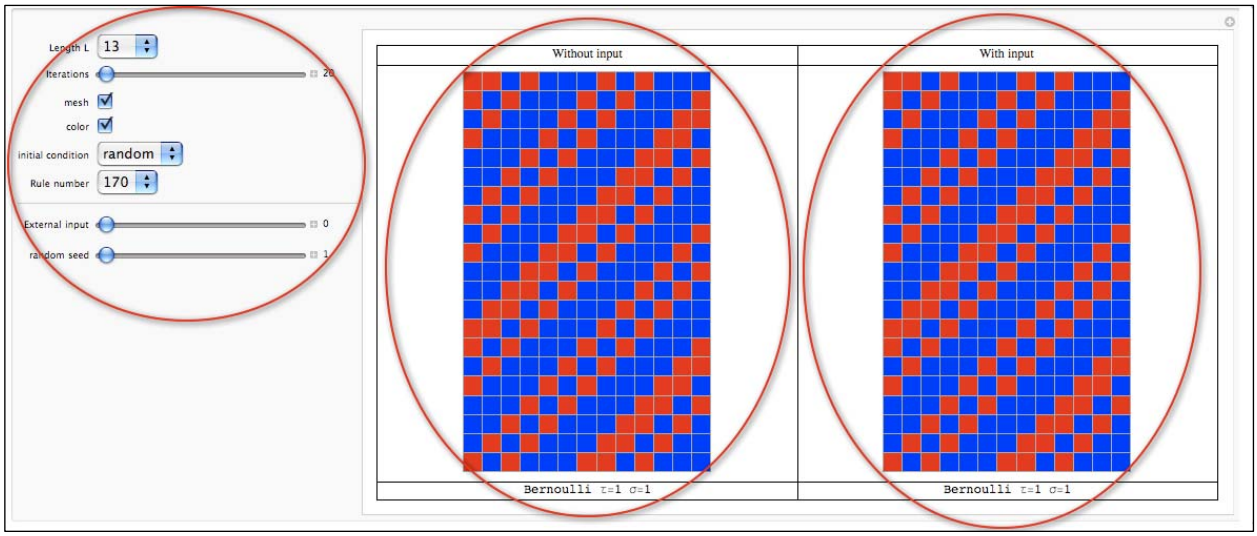
There exist numerous solid results about CA which have been found through a rigorous theoretical approach. However, empirical experiments can help to understand what kind of dynamics may arise in CA and under what conditions certain phenomena can be verified.

For this reason, here we describe the working principle of a simulator which can be downloaded from the webpage

http://sztaki.hu/~gpazienza/CAsimulator.nbp

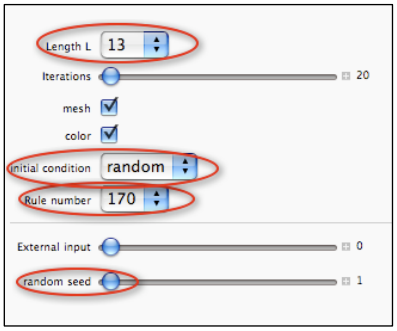
The simulator is written in Mathematica; hence, you need either Mathematica or Mathematica Player to run it.

The main window of the simulator is divided into three parts (from left to right):1) controls; 2) space-time pattern of the CA without input; 3) space-time pattern of the CA with input.



Control panel

The control panel includes the fundamental features needed to simulate a Cellular Automaton: rule number; length of the bit string; initial condition; number of iterations.



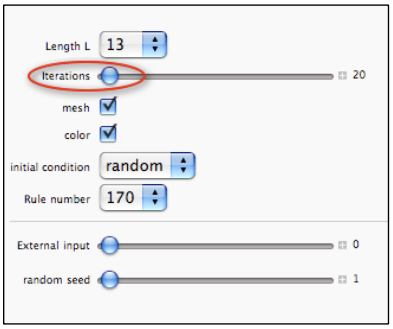
There are only 88 fundamental 1D binary CA rules, whose numbers are specified in (Chua & Pazienza, 2009). All other rules are equivalent to one of them, and the equivalence tables can be found in (Chua, 2009)

The length of the bit string can vary from 3 to 99 in this version, even though it should be kept under 25, otherwise the resolution of the single cell may become too small to be appreciated.

The initial bit string can be set either randomly or by the user (only the first option is active in the current version). The random initial bit string can be changed by modifying the random seed.

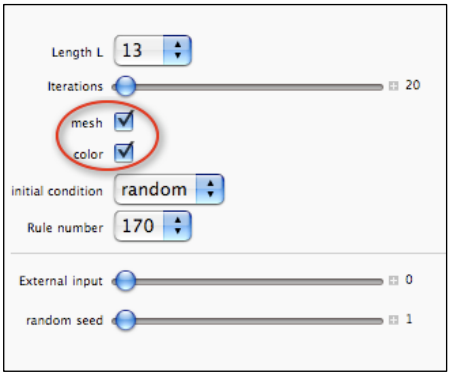
The number of iterations for which the given rule should be run is controlled by the slide ‘iterations’. Given any deterministic CA rule, at least one bit string must repeat itself after at most 2^L iterations. When such repetition occurs, it is possible to determine the ω-limit orbit the initial bit string belongs to, according to the definition given in. In

practice, when L is equal to or lower than 20, it is sufficient to set the number of iterations to 30 to find out the basin of attraction.

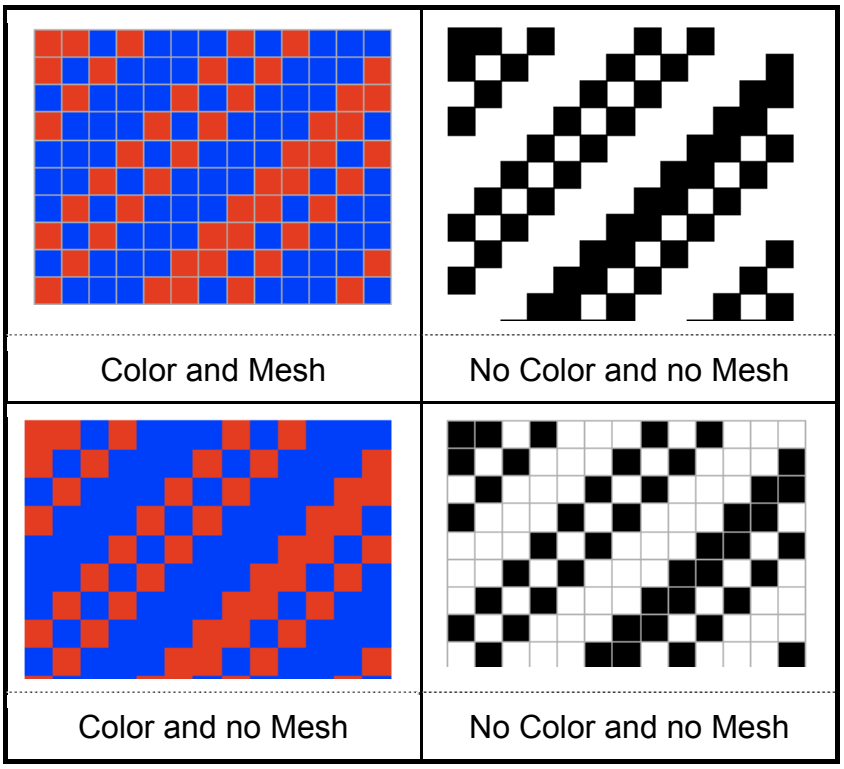


Visualization

The states of a binary Cellular Automaton can be represented by only two color: either white (for 0) and black (for 1), as Wolfram proposes, or blue (for 0) and red (for 1), as Chua proposes. It is possible to select either color combination by selecting the color box in the simulator.



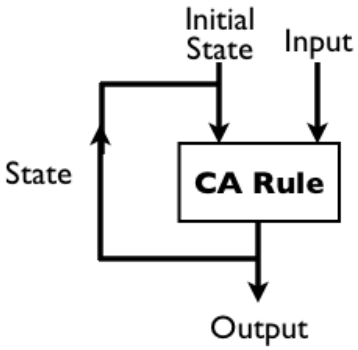
Furthermore, the cells appear as divided by thin lines when the option *mash* is selected. The following figures show the four possible combinations of *color* and *mesh*.



Space-time patterns

In the space-time pattern, the initial bit string is displayed on the first row, while the following rows contain the successive iterations. In the bottom part, it is shown the kind of the ω-limit orbit the initial bit string belongs to.

The standard model of Cellular Automaton, extensively studied by Wolfram and Chua among others, does not include an input, but the evolution is made on the the initial state. However, it can be interesting to see what happens when an input is introduced into the system, according to the scheme

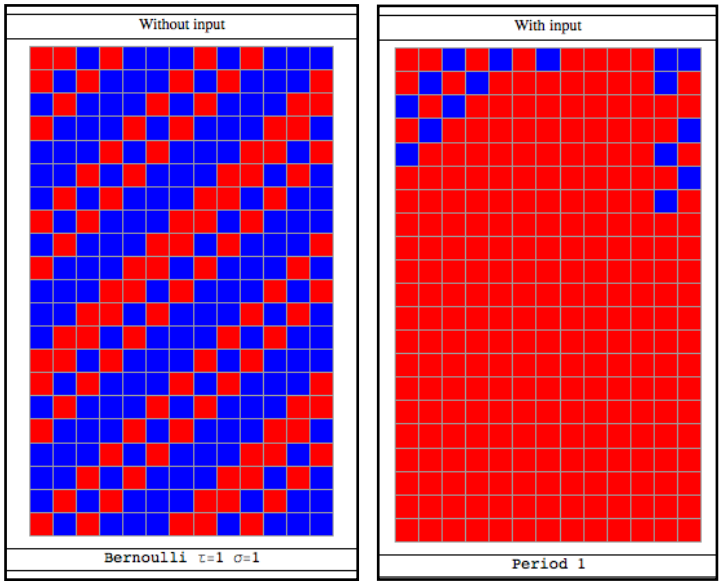


In which the rule is executed on the pattern resulting from the logic OR between the state and the input.

An emblematic example of the dramatic changes that the introduction of an input may imply is the simulation of rule 170 with an arbitrary input bit string. Rule 170 acts as

a bit shift, in the sense that at each iteration it shift all bit left by one position. This kind of ω-limit orbit is called Bernoulli-shift with σ=1 and τ=1.

When an *arbitrary* input is introduced the left shift operation performed by rule 170 implies that the input, which is constant both in space and time, is quickly extended to the whole bit string. In other words, after a few iterations (at most *L*), we obtain necessarily a constant output.



In the next versions of the simulator, it will be possible to introduce an input variable in space and/or time, so that even more exotic dynamics can be explored.

References

L. O. Chua, "*A Nonlinear Dynamics Perspective of Wolfram's New Kind of Science Vol. I, II, III* ", World Scientific, 2005–2009.

S. Wolfram, "*A New Kind of Science*", Wolfram Media, 2002.

L. O. Chua, G. E. Pazienza, L. Orzo, V. Sbitnev, and J. Shin, "*A Nonlinear Dynamics Perspective of Wolfram's New Kind of Science, Part IX: Quasi-Ergodicity*", International Journal of Bifurcation and Chaos, 9:18, pag. 2487-2642, 2008.

L. O. Chua, G. E. Pazienza, and J. Shin, "*A Nonlinear Dynamics Perspective of Wolfram's New Kind of Science, Part X: Period-1 rules*", International Journal of Bifurcation and Chaos, 5:19, pag. 1425-1655, 2009.

L. O. Chua, G. E. Pazienza, and J. Shin, "*A Nonlinear Dynamics Perspective of Wolfram's New Kind of Science, Part XI: Period-2 rules*", International Journal of Bifurcation and Chaos, 6:19, pag. 1751-1931, 2009.

L. O. Chua and G. E. Pazienza, "*A Nonlinear Dynamics Perspective of Wolfram's New Kind of Science, Part XII: Period-3, Period-6 and Permutive rules*", International Journal of Bifurcation and Chaos, 9:19, 2009.

Appendix 1: UMF Algorithm Description

Version 1.3

Elementary instruction

An elementary CNN wave instruction is defined on a 2 dimensional grid (ij) by the constitutive standard CNN dynamics. The simplest PDDE (partial differential difference equation) defining it with a cloning template {A,B, z} is:

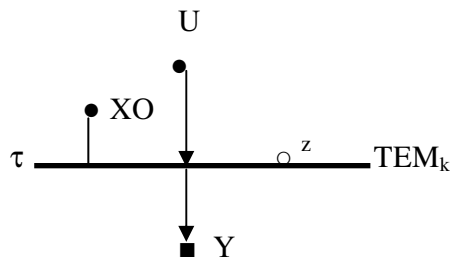
$$\tau\, dx_{ij}/dt = -x_{ij} + \sum A_{kl}\, y_{kl} + \sum B_{kl}\, u_{kl} + z_{ij}$$

(kl is in the r neighborhood of ij)

$$y_{ij} = f(x_{ij})$$

$\{x_{ij}(0)\}=X0$, $\{u_{ij}\}= U$,
$\{y_{ij}\}= Y$, $\{z_{ij}\}= Z$,
if z_{ij} is the same for all ij then $z_{ij} = z, t \in T$

The CNN Software Library contains many templates {A, B, z} implemen–ting simple and exotic waves, with standard and more complex templates and first order as well as higher order (complex) cells for various image processing tasks.



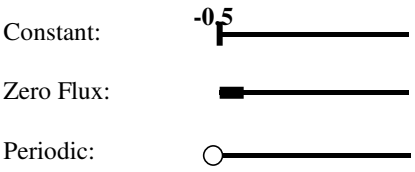
Note: Template oprators, TEMk, can also begiven by other forms (e.g. logic truth table, or nonlinear operators of the x_{nl} or y_{nl} variables, without the matrix form.

Signals, variables

■	logic array
■■	vector of logic arrays
□	logic value
●	analog array
●●	vector of analog arrays
○	analog value

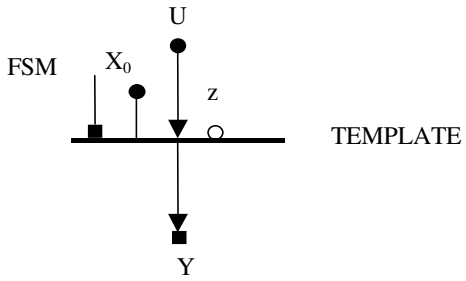
Boundary conditions

Left side: Input boundary condition
Right side: Output boundary condition

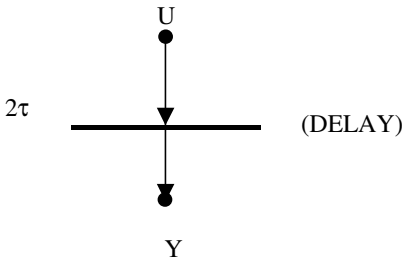


Boundary conditions are optional, if not given, it means "don't care"

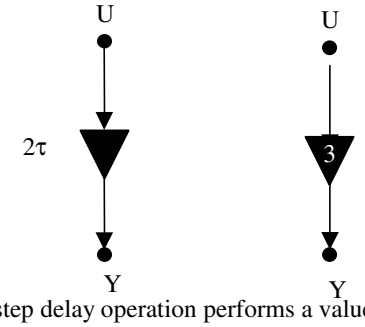
Fixed State Map



Continuous delay

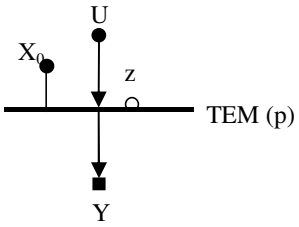


Step delay



A step delay operation performs a value delay. The time can be specified either in τ or in GAPU instruction steps. If neither is given, delay time defaults to a single GAPU instruction step.

Parametric templates



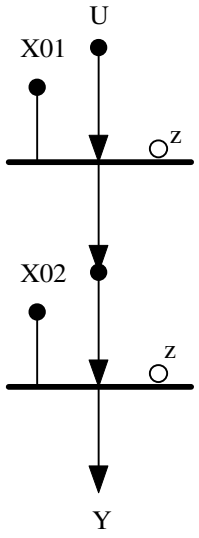
Example: Multiplication with constant

MULT(p)

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & p & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

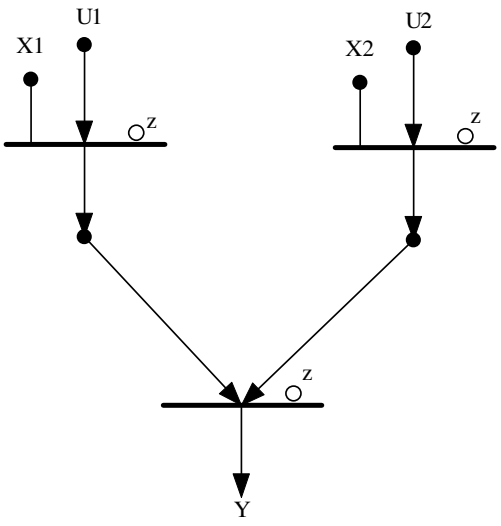
Algorithmic structures

Cascade

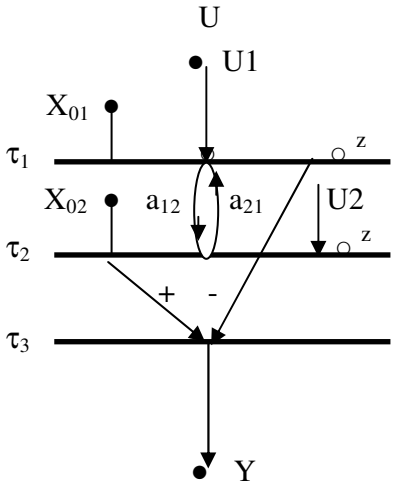


Parallel

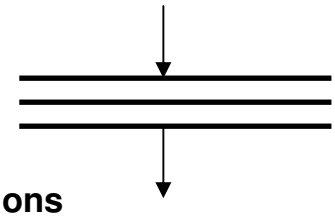
A typical parallel structure with two parallel flows is shown below, by combining them in the final



Three layer complex cell



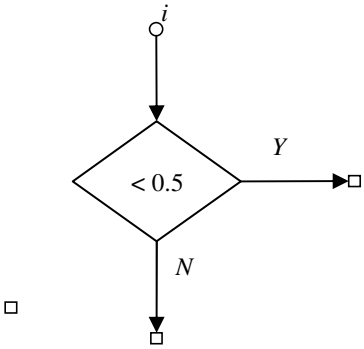
We can use a compact symbol below for the compact cell and use it in cascade, parallel and combined cascade-parallel structures.



Decisions

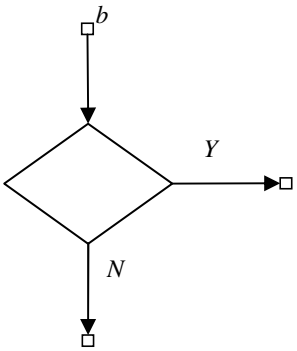
On global analog parameter

Is the value of variable *i* less then 0.5?



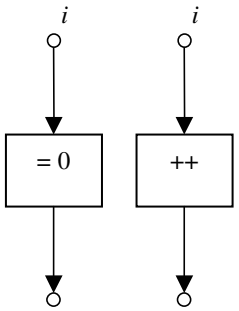
On global Logic parameter set, including global Fluctuation

Does the logic value of variable *b* refers to white?

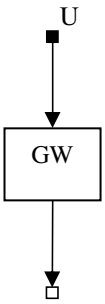


Operators

C-like imperative operators



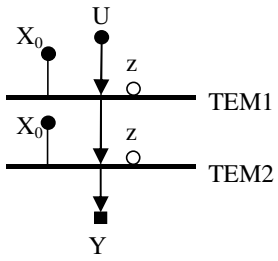
Global White operator



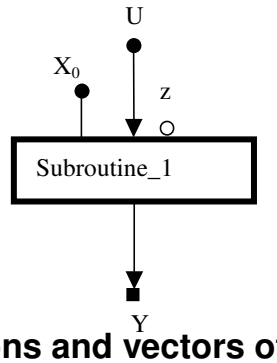
Subroutines

Definition

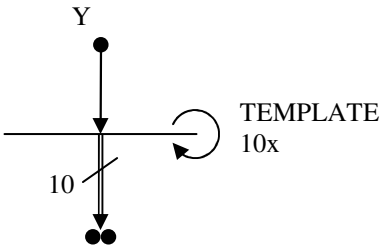
Subroutine_1



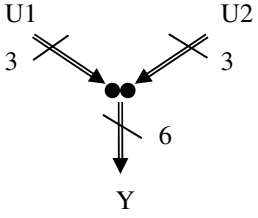
Usage



Iterations and vectors of arrays

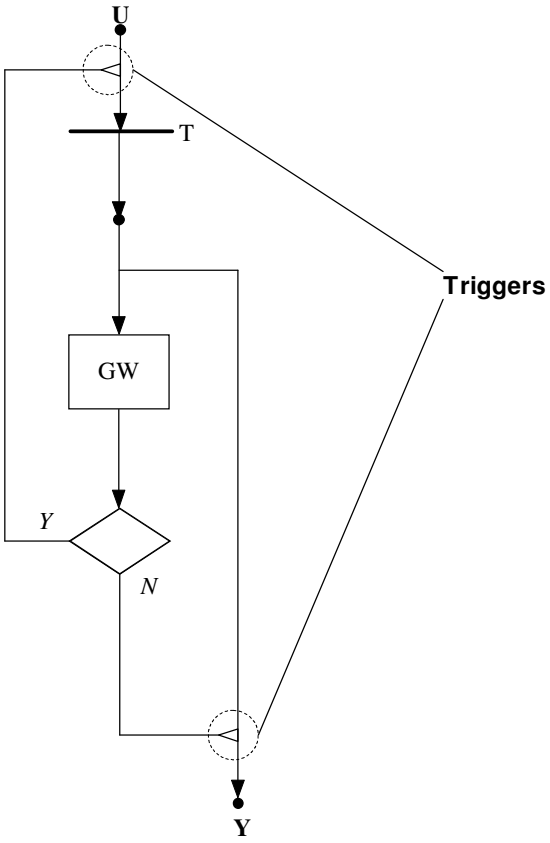


Merging arrays



Triggers, Cycles

A trigger makes a dataflow to continue



Depending on the output of the decision (that can be either yes or no) the proper dataflow continues.

```
flow=U;
repeat
       flow=T(flow);
until GW==1;
Y=flow;
```

Appendix 2: Virtual and Physical Cellular Machines

Virtual Cellular Machine

1. Notations and definitions

Core=Cell

Core or cell will be used as synonyms, it is defined as a unit implementing a well defined operator (with input, output, state) on binary, real or string variables (also defined as logic, arithmetic/analog or symbolic variables, respectively). Cores/cells are used typically in arrays, mostly with well defined interaction patterns with their neighbor core/cells, although sparse longer wires/communications/interactions are also allowed.

Core is used if we emphasize the digital implementation, cell is used if it is more general.

Elementary array instructions

A **Logic (L), Arithmetic/analog (A) or Symbolic (S) elementary array instruction** is defined via r input (u(t)), m output (y(t)) and n state (x(t)) variables (t is the time instant), operating on binary, real, or symbol variables, respectively. Each dynamic cell is connected mainly locally, in the simplest case, to their neighbor cells.

- **L**: A typical *logic elementary array instruction* might be a binary logic function on n or nxn (2D) binary variables, (special cases: a disjunctive normal form, a memory look-up table array, a binary state machine, an integer machine),
- **A**: a typical *arithmetic/analog elementary array* instruction is a multiply and accumulate (add) term (MAC) core/cell array or a dynamic cell array generating a spatial-temporal wave, and
- **S**: a typical *symbolic elementary array instruction* might be a string manipulation core/cell array, mainly locally connected .

Mainly local connectedness means that the local connection has a speed preference compared to a global connection via a crossbar path.

A classical 8, 16, or 32 bit microprocessor could be considered as well as an elementary array instruction with iterative or multi-thread implementation on the three types of data. However, the main issue is that we have elementary array instructions, as the protagonist instructions.

2. Physical Implementation types of elementary core/cell array instructions (A, B, C)

We have three *elementary cell processor* (cell core) array implementation types:

D: A digital algorithm with input, state and output vectors of **r**eal/ arithmetic (finite precision analog), binary/**d**igital logic, and **s**ymbolic variables (typically implemented via digital circuits).

R: A real valued dynamical system cell with **a**nalog/continuous or **a**rithmetic variables (typically implemented via mixed mode/**a**nalog-and-**l**ogic circuits and digital control processors), placed in a mainly locally connected array

G: A physical dynamic entity with well defined **G**eometric *Layout* and I/O ports (function in layout) – (typical implementations are CMOS and/or nanoscale designs, or optical architectures with programmable control), placed in a mainly locally connected array.

3. Physical parameters of array processor units (typically a chip or a part of a chip) and interconnections

Each of these array units is characterized by its

- **g,** geometric area,
- **e,** energy,
- **f,** operating frequency,
- **w = e f** local power dissipation**,** and
- the signals are traveling on a wire with length **l** , width **q,** and with speed $\mathbf{v_q}$ introducing a delay of **D** $= \mathbf{l}\, \mathbf{v_q}$

Ω *cores/cells* can be placed **on a single Chip** , typically in a square grid, with input and output physical connectors typically at the corners (sometimes at the bottom and top "corners" in a 3D packaging) of the Chip, altogether there are **K** input/output connectors. The maximal value of dissipation of the Chip is **W**. The physics is represented by the maximal values of **Ω, K, and W** (as well as the operating frequency). The operating frequency might be global for the whole Chip **Fo**, or could be local within the Chip, $\mathbf{f_i}$ (some parts might be switched off, $\mathbf{f_i = 0}$), may be a *partially local frequency* **fo > Fo**

The *interconnection pathways* between the arrays and other major building blocks are characterized by the delay and the bandwidth (B).

4. Virtual and Physical Cellular Machine architectures and their building blocks

A **Virtual Cellular Machine** is composed of five types of building blocks:

(i) *cellular processor arrays/layers* with simple (L, or A, or S type) or complex cells and their local memories), these are the protagonist building blocks,

(ii) *classical* digital stored program *computers* (microprocessors),

(iii) multimodal topographic or non-topographic *inputs* (e.g. scalar, vector, and matrix signals),

(iv) *memories* of different data types, organizations and qualitatively different sizes and access times (e.g. in clock cycles), and

(v) *interconnection pathways* (busses).

The **tasks**, the algorithms to be implemented, are defined on the Data of the Virtual Cellular Machines.

We consider two types of Virtual Cellular Machines: Single- or multi-cellular array/layer machines, also called homogeneous and heterogeneous cellular machines.

In the ***homogeneous Virtual Cellular Machine***, the basic problem is to execute a task, for example a Cellular Wave Computer algorithm, on a bigger topographic Virtual Cellular Array using a smaller size of physical cellular array. Four different types of algorithms have already been developed (Zarándy, 2008)

Among the many different, sometimes problem oriented ***heterogeneous Virtual Cellular Machine*** architectures we define two typical ones. Their *five building block*s, are as follows.

(i) Cellular processor arrays of one dimensional , CP1, and two dimensional , CP2, ones

(ii) P - classical digital computer with memory & I/O, for example a classical microprocessor

(iii) T - topographic fully parallel 2D (or 1D) input

(iv) M - memory with high speed I/O, single port or dual port (L1, L2, L3 parts as cache and/or local memories with different access times)

(v) B - data bus with different speed ranges (B1, B2, …)

The CP1 and CP2 types of cellular arrays may be composed of cell/core arrays of simple and complex cells. In the CNN Universal Machine, each complex cell contains logic and analog/arithmetic components, as well as local memories, plus local communication and control units. Each array has its own controlling processor; we called it in the CNN Universal Machine as Global Analog/arithmetic-and-logic Programming Unit (GAPU).

The size of the arrays in the Virtual Cellular Machines are typically large enough to handle all the practical problems that might encounter in the minds of the designers. In the physical implementation, however, we confront the finite, reasonable, cost effective sizes and other physical parameters.

The **Physical Cellular Machine** architecture is defined by the same kind of five building blocks , however, with well defined physical parameters, either in a similar architecture like that of the Virtual Cellular Machine or a different one.

A building block could be physically implemented as a separate chip or as a part of a chip. The *geometry of the architecture* is reflecting the physical layout within a chip and the chips within the Machine (multi-chip machine).

This architectural geometry defines also the communication (interacting) speed ranges, as well. Hence physical closeness means higher speed ranges and smaller delays.

The *spatial location or* ***topographic address*** of each elementary cell or core, within a building block, as well as that of each building block within a chip, and each chip, within the Virtual Cellular Machine (Machine) architecture, plays a crucial role. *This is one of the most dramatic difference compared to classical computer science.*

In the Physical Cellular Machine models we can use exact, typical or qualitative values for size, speed, delay, power, and other physical parameters. The simulators can use these values for performance evaluation.

We are not considering here the problems and design issues within the building blocks, it was fairly well studied in the Cellular Wave Computing or CNN Technology literature, as well as implementing a virtual 1D or 2D Cellular Wave Computer on a smaller physical machine. The decomposition of bigger memories on smaller physical memories are the subject of the extensively used virtual memory concept.

We mention that sometimes a heterogeneous machine can be implemented on a single chip by using the different areas for different building blocks (Rekeczky et. al., 2008)

The architecture of the Virtual Cellular Machine and the Physical Cellular Machine might be the same, though the latter might have completely different physical parameters. On the other hand they might have completely different architectures.

The *internal functional operation* of the cellular building blocks are not considered here. On one hand, they are well studied in the recent Cellular Wave Computer literature, as well as in the recent implementations (ACE 16k, ACE 25k = Q-Eye, XENON), etc.), on the other hand, they can be modeled based on the Graphics Processing Units (GPU) and FPGA literature. Their functional models are described elsewhere (see also the Open CL language description).

The *two basic types of multi-cellular heterogeneous Virtual Machine architectures* are defined next.

I. Global system control and memory architecture is defined in Figure 1.

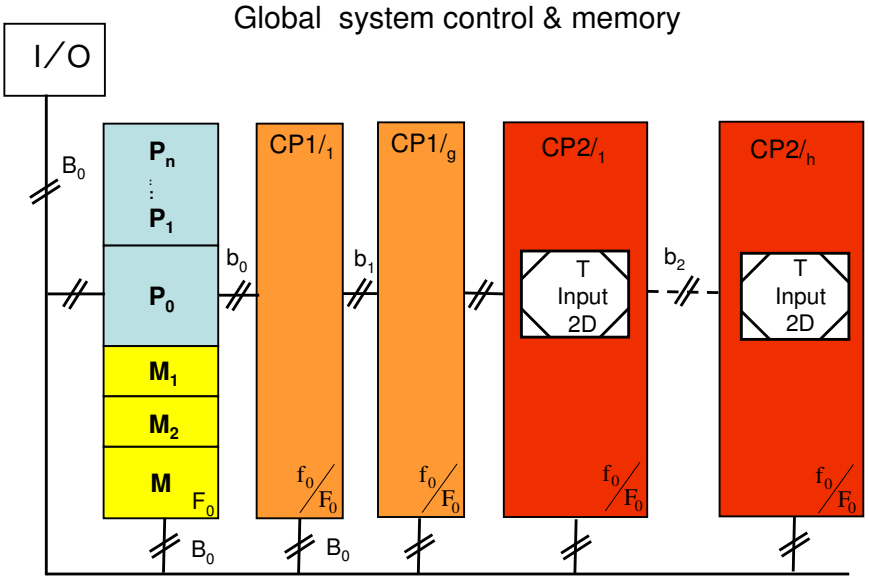


Fig. 1.

II *Distributed system control and memory* architecture is shown in Figure 2.
The thick buses are "equi-speed" with much higher speed than the connecting thin buses.

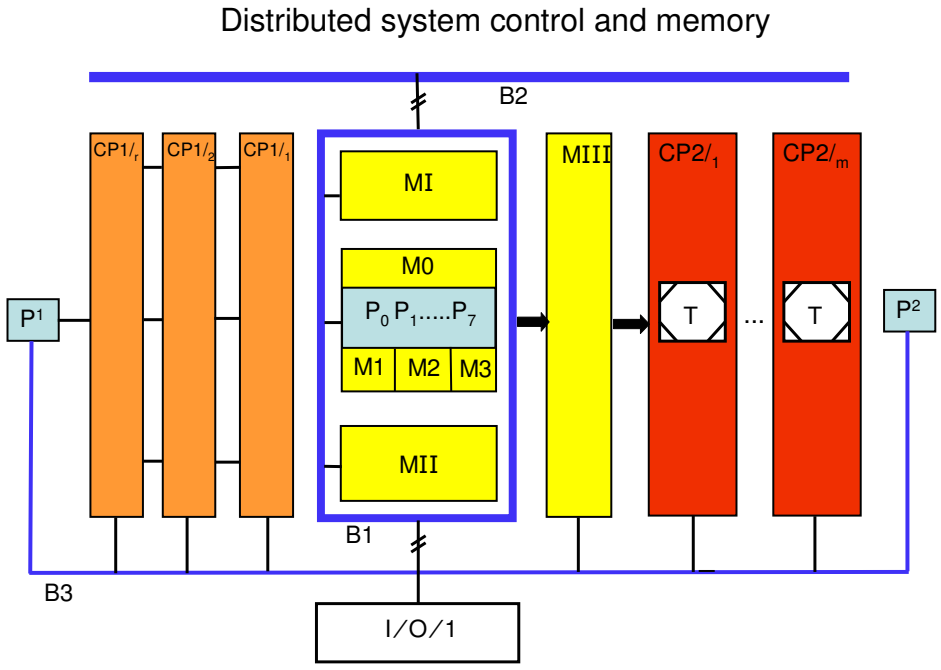


Fig. 2.

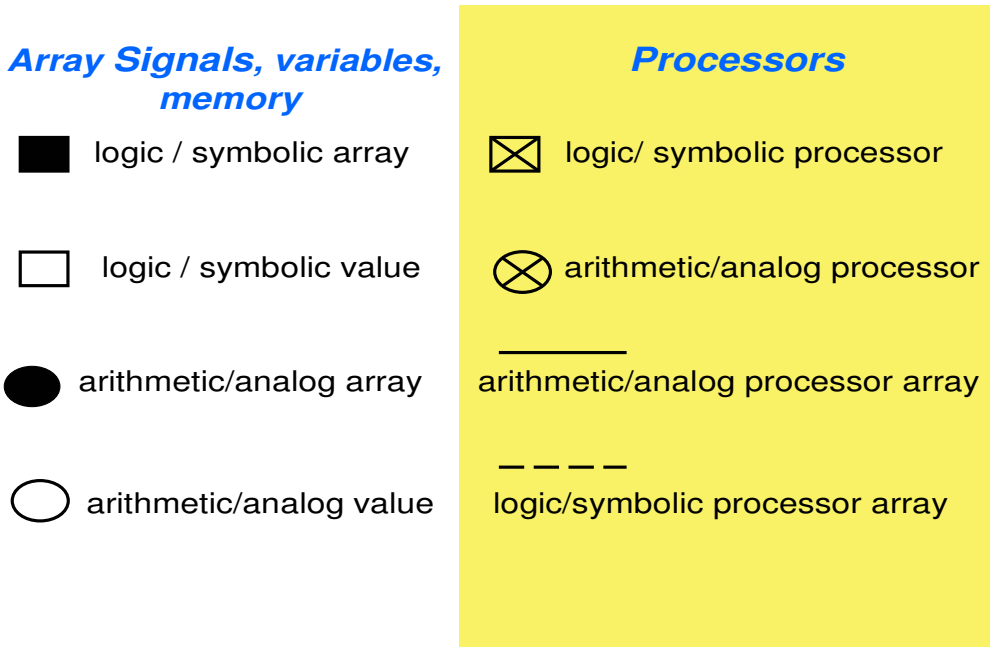
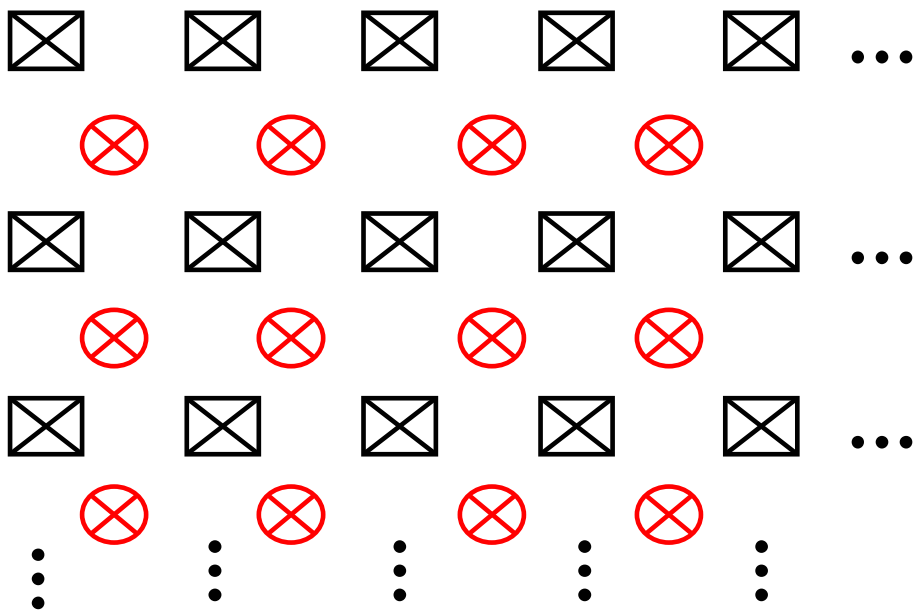


Fig.3.



Tiled-in cellular processor arrays

Fig. 4.

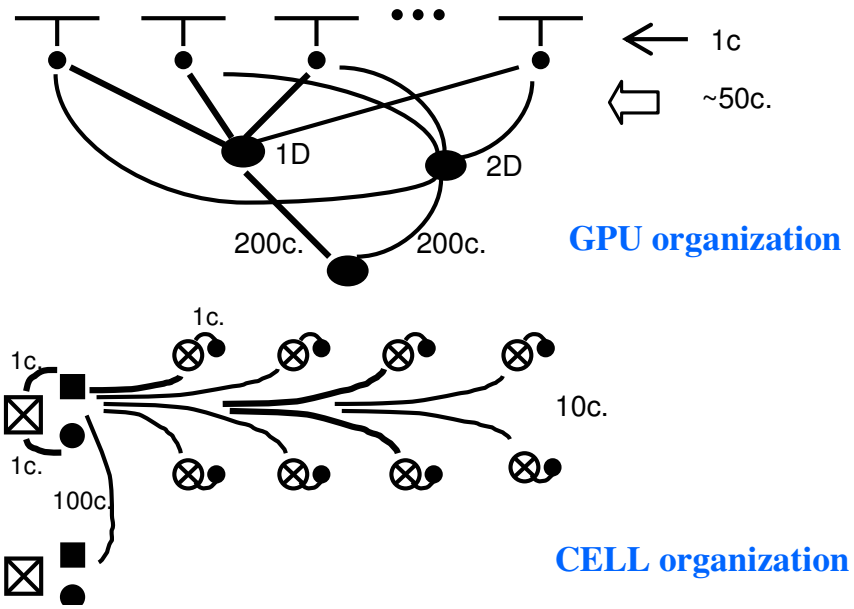


Fig. 5.

5. *The Design Scenario*

There are three domains in the design scenario :

- The *Virtual Cellular Machine* architecture based on the data/object and operator relationship architecture of the problem (topographic or non-topographic),
- The physical processor/memory topography of the *Physical Cellular Machine*, and the
- *Algorithmic domain* connecting the preceding two domains.

The design task is the is to map the algorithm defined on the Virtual Cellular Machine into the Physical Cellular Machine. For example the decomposition of bigger virtual machine architectures into smaller physical ones, as well as to transform non-topographic data architectures into topographic processor and memory architectures.

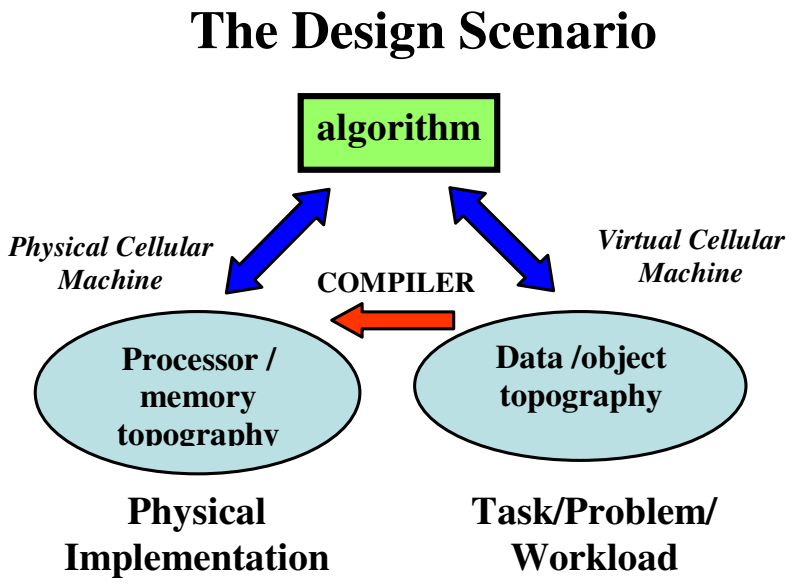


Fig.

6. The dynamic operational graph and its use for acyclic UMF diagrams

Extending the UMF diagrams (Roska, 2003) describing Virtual Cellular Machines leads to digraphs, with processor array and memory nodes, and signal array pathways as branches with bandwidth weights. These graphs with the dissipation side-constraint define optimization problems representing the design task, under well defined equivalent transformations.

In some well defined cases, especially within a 1D or 2D homogeneous array, the recently introduced method via Genetic Programming with Indexed Memory (GP-IM) using UMF diagrams with Directed Acyclic Graphs (DAG) seems a promising tool showing good results in simpler cases (Pazienza, 2008).

Appendix 3: Template Robustness

TEMPLATE ROBUSTNESS

Here we will give the definition of template robustness for the case of uncoupled binary input/output templates.
It is known that the set of uncoupled binary input/output templates is isomorphic to the set of linearly separable Boolean functions of 9 variables [44]. Such functions can be described by 9-dimensional hyper-cubes [45]. If the function is linearly separable, a hyper-plane exists which separates the set of -1s from the set of 1s.
Definition: The robustness of the template T, denoted by ρ, is defined as the minimal distance of the hyper-plane, which separates the set of -1s from the set of 1s, and from the vertices of the hyper-cube (see Figure 1a for an illustration in 2 dimensions).

The robustness of T can be increased by choosing the optimal template T^{opt}, for which the minimal distance of the separating hyper-plane from the vertices is maximal [45] (see Figure 1b).

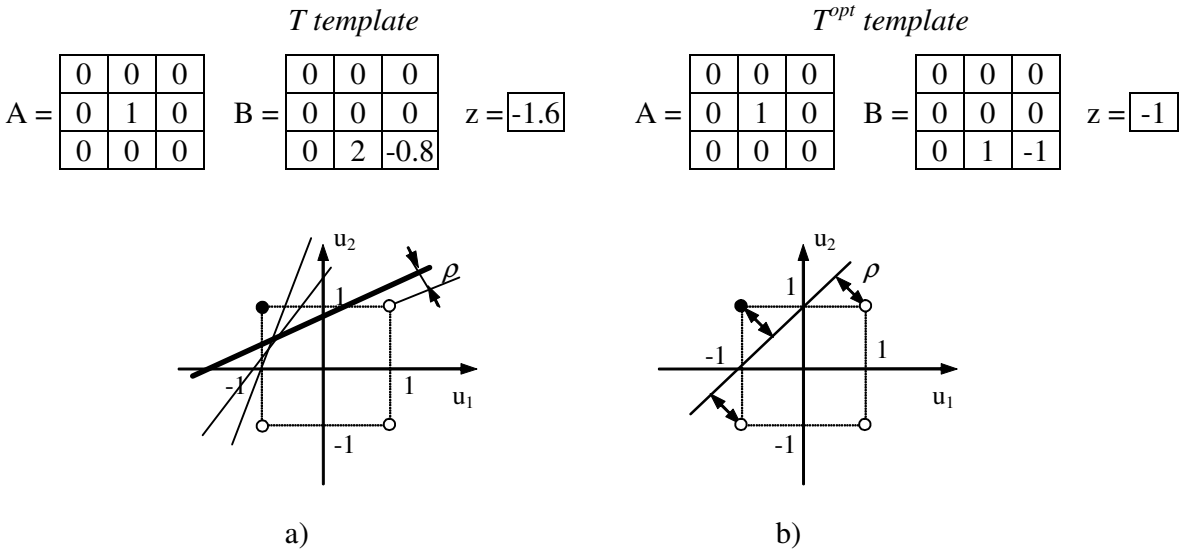


Figure 1. These diagrams illustrate the separation of vertices for the 2 dimensional logic function $F(u_1, u_2) = \bar{u}_1 u_2$. ***Logic TRUE and FALSE are represented by filled and empty circles, respectively. The concept of robustness*** ρ ***is also illustrated. Figure a) shows a few possible separating lines. The thick line corresponds to the template T with robustness*** $\rho = 0.18$. ***Figure b) depicts the optimal separation line corresponding to the template*** T^{opt}***; its robustness is*** $\rho = 0.71$.

REFERENCES

[1] L. O. Chua and L. Yang, “Cellular neural networks: Theory and Applications”, *IEEE Transactions on Circuits and Systems*, Vol. 35, pp. 1257-1290, October 1988.

[2] L. O. Chua and L. Yang, “The CNN Paradigm”, *IEEE Transactions on Circuits and Systems–I: Fundamental Theory and Applications*, Vol. 40, pp. 147-156, March 1993.

[3] T. Roska and L. O. Chua, “The CNN Universal Machine: An Analogic Array Computer”, *IEEE Transactions on Circuits and Systems–II: Analog and Digital Signal Processing*, Vol. 40, pp. 163-173, March 1993.

[4] The CNN Workstation Toolkit, Version 6.0, MTA SzTAKI, Budapest, 1994.

[5] P. L. Venetianer, A. Radványi, and T. Roska, "ACL (an Analogical CNN Language), Version 2.0, *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-3-1994, Budapest, 1994.

[6] T. Matsumoto, T. Yokohama, H. Suzuki, R. Furukawa, A. Oshimoto, T. Shimmi, Y. Matsushita, T. Seo and L. O. Chua, "Several Image Processing Examples by CNN", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-90)*, pp. 100-112, Budapest, 1990.

[7] T. Roska, T. Boros, A. Radványi, P. Thiran, L. O. Chua, "Detecting Moving and Standing Objects Using Cellular Neural Networks", *International Journal of Circuit Theory and Applications,* October 1992, and *Cellular Neural Networks*, edited by T. Roska and J. Vandewalle, 1993.

[8] T. Boros, K. Lotz, A. Radványi, and T. Roska, "Some Useful New Nonlinear and Delay-type Templates", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-1-1991, Budapest, 1991.

[9] S. Fukuda, T. Boros, and T. Roska, "A New Efficient Analysis of Thermographic Images by using Cellular Neural Networks", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-11-1991, Budapest, 1991.

[10] L. O. Chua, T. Roska, P. L. Venetianer, and Á. Zarándy, "Some Novel Capabilities of CNN: Game of Life and Examples of Multipath Algorithms", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-3-1992, Budapest, 1992.

[11] L. O. Chua, T. Roska, P. L. Venetianer, and Á. Zarándy, "Some Novel Capabilities of CNN: Game of Life and Examples of Multipath Algorithms", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-92)*, pp. 276-281, Munich, 1992.

[12] T. Roska, K. Lotz, J. Hámori, E. Lábos, and J. Takács, "The CNN Model in the Visual Pathway - Part I: The CNN-Retina and some Direction- and Length-selective Mechanisms", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-8-1991, Budapest, 1991.

[13] T. Roska, J. Hámori, E. Lábos, K. Lotz, L. Orzó, J. Takács, P. L. Venetianer, Z. Vidnyánszky, and Á. Zarándy, "The Use of CNN Models in the Subcortical Visual Pathway", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-16-1992, Budapest, 1992.

[14] P. Szolgay, I. Kispál, and T. Kozek, "An Experimental System for Optical Detection of Layout Errors of Printed Circuit Boards Using Learned CNN Templates", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-92)*, pp. 203-209, Munich, 1992.

[15] K. R. Crounse, T. Roska, and L. O. Chua, "Image halftoning with Cellular Neural Networks", *IEEE Transactions on Circuits and Systems–II: Analog and Digital Signal Processing*, Vol. 40, No. 4, pp. 267-283, 1993.

[16] H. Harrer and J. A. Nossek, "Discrete-Time Cellular Neural Networks", TUM-LNS-TR-91-7, Technical University of Munich, Institute for Network Theory and Circuit Design, March 1991.

[17] T.Sziranyi and M.Csapodi, "Texture classification and Segmentation by Cellular Neural Network using Genetic Learning", *Computer Vision and Image Understanding*, Vol. 71, No. 3, pp. 255-270, September 1998.

[18] A. Schultz, I. Szatmári, Cs. Rekeczky, T. Roska, and L. O. Chua, "Bubble-debris classification via binary morphology and autowave metric on CNN", *International Symposium on Nonlinear Theory and its Applications*, Hawaii, 1997

[19] P. L. Venetianer, F. Werblin, T. Roska, and L. O. Chua, "Analogic CNN Algorithms for some Image Compression and Restoration Tasks", *IEEE Transactions on Circuits and Systems*, Vol. 42, No.5, 1995.

[20] P. L. Venetianer, K. R. Crounse, P. Szolgay, T. Roska, and L. O. Chua, "Analog Combinatorics and Cellular Automata - Key Algorithms and Layout Design using CNN", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-94)*, pp. 249-256, Rome, 1994.

[21] H. Harrer, P. L. Venetianer, J. A. Nossek, T. Roska, and L. O. Chua, "Some Examples of Preprocessing Analog Images with Discrete-Time Cellular Neural Networks", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-94)*, pp. 201-206, Rome, 1994.

[22] Á. Zarándy, F. Werblin, T. Roska, and L. O. Chua, "Novel Types of Analogic CNN Algorithms for Recognizing Bank-notes", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-94)*, pp. 273-278, Rome, 1994.

[23] E. R. Kandel and J. H. Schwartz, "Principles of Neural Science", Elsevier, New York, 1985.

[24] A. Radványi, "Using Cellular Neural Network to 'See' Random-Dot Stereograms" in *Computer Analysis of Images and Patterns*, Lecture Notes in Computer Science 719, Springer Verlag, 1993.

[25] M. Csapodi, Diploma Thesis, Technical University of Budapest, 1994.

[26] K. Lotz, Z. Vidnyánszky, T. Roska, and J. Hámori, "The receptive field ATLAS for the visual pathway", Report NIT-4-1994, Neuromorphic Information Technology, Graduate Center, Budapest, 1994.

[27] G. Tóth, Diploma Thesis, Technical University of Budapest, 1994.

[28] T. Boros, K. Lotz, A. Radványi, and T.Roska, "Some useful, new, nonlinear and delay-type templates", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI)*, DNS-1-1991, Budapest, 1991.

[29] G. Tóth, "Analogic CNN Algorithm for 3D Interpolation-Approximation", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI)*, DNS-2-1995, Budapest, 1995.

[30] P. Perona and J. Malik, "Scale space and edge detection using anisotropic diffusion", *Proceedings of the IEEE Computer Society Workshop on Computer Vision*, 1987.

[31] F. Werblin, T. Roska, and L. O. Chua, "The Analogic Cellular Neural Network as a Bionic Eye", *International Journal of Circuit Theory and Applications,* Vol. 23, No. 6, pp. 541-569, 1995.

[32] R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image Analysis Using Mathematical Morphology", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 532-550, Vol. PAMI-9, No. 4, July 1987.

[33] L. O. Chua, T. Roska, T. Kozek, and Á. Zarándy, "The CNN Paradigm – A Short Tutorial", *Cellular Neural Networks*, T. Roska and J. Vandewalle, editors, John Wiley & Sons, New York, 1993, pp. 1-14.

[34] Cs. Rekeczky, Y. Nishio, A. Ushida, and T. Roska, "CNN Based Adaptive Smoothing and Some Novel Types of Nonlinear Operators for Grey-Scale Image Processing", in *proceedings of NOLTA'95*, Las Vegas, December 1995.

[35] T. Szirányi, "Robustness of Cellular Neural Networks in image deblurring and texture segmentation", *International Journal of Circuit Theory and Applications*, Vol. 24, pp. 381-396, May 1996.

[36] Á. Zarándy, "The Art of CNN Template Design", *International Journal of Circuit Theory and Applications,* Vol. 27, No. 1, pp. 5-23, 1999.

[37] M. Csapodi, J. Vandewalle, and T. Roska, “Applications of CNN-UM chips in multimedia authentication”, ESAT-COSIC Report / TR 97-1, Department of Electrical Engineering, Katholieke Universiteit Leuven, 1997.

[38] L. Nemes, L. O. Chua, “**TemMaster** Template Design and Optimization Tool for Binary Input-Output CNNs, User’s Guide”, *Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA-SzTAKI)*, Budapest, 1997.

[39] P. Szolgay, K. Tömördi**,** "Optical detection of breaks and short circuits on the layouts of printed circuit boards using CNN", *Proceedings of the International Workshop on Cellular Neural Networks and their Applications (CNNA-96)*, pp. 87-91, Seville, 1996.

[40] Hvilsted, S.; Ramanujam, P.S., “Side-chain liquid crystalline azobenzene polyesters with unique reversible optical storage properties”. *Curr. Trends Pol. Sci.* (1996) v.1, pp. 53-63.

[41] S. Espejo, A. Rodriguez-Vázquez, R. A. Carmona, P. Földesy, Á. Zarándy, P. Szolgay, T. Szirányi, and T. Roska, “0.8µm CMOS Two Dimensional Programmable Mixed-Signal Focal-Plane Array Processor with On-Chip Binary Imaging and Instruction Storage”, *IEEE Journal on Solid State Circuits*, Vol. 32., No. 7., pp. 1013-1026,. July 1997.

[42] G. Liñán, S. Espejo, R. Domínguez-Castro, E. Roca, and A. Rodriguez-Vázquez, “CNNUC3: A Mixed-Signal 64x64 CNN Universal Chip”, *Proceedings of the International Conference on Microelectronics for Neural, Fuzzy and Bio-inspired Systems (MicroNeuro’99)*, pp. 61-68, Granada, Spain, 1999.

[43] S. Ando, "Consistent Gradient Operations", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 22., No. 3., pp. 252-265,. March 2000.

[44] L. O. Chua, "CNN: a paradigm for complexity", *World Scientific Series On Nonlinear Science*, Series A, Vol. 31, 1998.

[45] L. Nemes, L.O. Chua, and T. Roska, “Implementation of Arbitrary Boolean Functions on the CNN Universal Machine”, *International Journal of Circuit Theory and Applications - Special Issue: Theory, Design and Applications of Cellular Neural Networks: Part I: Theory*, (CTA Special Issue - I), Vol. 26. No. 6, pp. 593-610, 1998.

[46] I. Szatmári, Cs. Rekeczky, and T. Roska, "A Nonlinear Wave Metric and its CNN Implementation for Object Classification", *Journal of VLSI Signal Processing, Special Issue: Spatiotemporal Signal Signal Processing with Analogic CNN Visual Microprocessors*, Vol.23, No.2/3, pp. 437-448, Kluwer, 1999.

[47] I. Szatmári, "The implementation of a Nonlinear Wave Metric for Image Analysis and Classification on the 64x64 I/O CNN-UM Chip", CNNA 2000, 6th *IEEE International Workshop on Cellular Neural Networks and their Applications*, May 23-25, 2000, University of Catania, Italy.

[48] I. Szatmári, A. Schultz, Cs. Rekeczky, T. Roska, and L. O. Chua, "Bubble-Debris Classification via Binary Morphology and Autowave Metric on CNN", *IEEE Trans. on Neural Networks*, Vol. 11, No. 6, pp.1385-1393, November 2000.

[49] P. Földesy, L. Kék, T. Roska, Á. Zarándy, and G. Bártfai, “Fault Tolerant CNN Template Design and Optimization Based on Chip Measurements”, *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA’98)*, pp. 404-409, London, 1998.

[50] P. Földesy, L. Kék, Á. Zarándy, T. Roska, and G. Bártfai, “Fault Tolerant Design of Analogic CNN Templates and Algorithms − Part I: The Binary Output Case”, *IEEE Transactions on Circuits and Systems* special issue on *Bio-Inspired Processors and Cellular Neural Networks for Vision,* Vol. 46, No. 2, pp. 312-322, February 1999.

[51] Á. Zarándy, T. Roska, P. Szolgay, S. Zöld, P. Földesy and I. Petrás, "CNN Chip Prototyping and Development Systems", *European Conference on Circuit Theory and Design - ECCTD'99*, Design Automation Day proceedings, (ECCTD'99-DAD), Stresa, Italy, 1999.

[52] I. Petrás, T. Roska, "Application of Direction Constrained and Bipolar Waves for Pattern Recognition", *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA’2000)*, pp. 3-8, Catania, Italy, 23-25 May, 2000.

[53] B. E. Shi, "Gabor-type filtering in space and time with cellular neural networks," *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications*, vol. 45, pp. 121-132, 1998.

[54] G. Tímár, K. Karacs, and Cs. Rekeczky: Analogic Preprocessing and Segmentation Algorithms for Off-line Handwriting Recognition., *IEEE Journal on Circuits, Systems and Computers*, Vol. 12(6), pp. 783-804, Dec. 2003.

[55] L. Orzó, T. Roska, "A CNN image-compression algorithm for improved utilization of on-chip resources", *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA'2004)*, pp. 297-302, Budapest, Hungary, 22-24 July, 2004.

[56] I. Szatmari, "Synchronization Mechanism in Oscillatory Cellular Neural Networks", *Research report of the Analogical and Neural Computing Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SzTAKI),* DNS-1-2006, Budapest 2006.

[57] I. Petrás, T. Roska, and L. O. Chua "New Spatial-Temporal Patterns and The First Programmable On-Chip Bifurcation Test-Bed", *IEEE Trans. on Circuits and Systems I, (TCAS I.)*, Vol. 50(5), pp. 619-633, May 2003.

[58] A. Gacsádi and P. Szolgay, "Image Inpainting Methods by Using Cellular Neural Networks", *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA'2005)*, ISBN:0780391853, pp. 198-201, Hsinchu, Taiwan, 2005

[59] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variational based noise removal algorithms", *Physica D*, Vol. 60, pp. 259–268, 1992.

[60] A. Gacsádi, P. Szolgay, "A variational method for image denoising by using cellular neural networks", *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA'2004)*, ISBN 963-311-357-1, pp. 213-218, Budapest, Hungary, 2004.

[61] R. Matei, "New Image Processing Tasks On Binary Images Using Standard CNNs", *Proceedings of the International Symposium on Signals, Circuits and Systems, SCS'2001,* pp.305-308, July 10-11, 2001, Iaşi, Romania

[62] R. Matei, "Design Method for Orientation-Selective CNN Filters", *Proceedings of the IEEE International Symposium on Circuits and Systems ISCAS'2004*, May 23-26, 2004, Vancouver, Canada

[63] CNN Young Researcher Contest, Analogic CNN Algorithm Design, *7th IEEE International Workshop on Cellular Neural Networks and their Applications*, Frankfurt-Germany, July 2002.

[64] D. Bálya: "CNN Universal Machine as Classification Platform: an ART-like Clustering Algorithm", *Int. Journal of Neural Systems*, 2003, Vol. 13(6), pp. 415-425.

[65] B. Roska, F. Werblin, "Rapid global shifts in natural scenes block spiking in specific ganglion cell types", *Nature Neuroscience*, 2003

[66] D. Bálya "Sudden Global Spatial-Temporal Change Detection and its Applications", *Journal of Circuits,* Systems*, and* Computers *(JCSC)*, Vol. 12(5), Aug-Dec 2003.

[67] Gy. Cserey, Cs. Rekeczky and P. Földesy "PDE Based Histogram Modification With Embedded Morphological Processing of the Level-Sets", *Journal of Circuits, System and Computers* (JCSC 2002).

[68] K. Karacs, G. Prószéky and T. Roska, "Intimate Integration of Shape Codes and Linguistic Framework in Handwriting Recognition via Wave Computers", *European Conference on Circuit Theory and Design*, Kraków, Poland, Sept. 2003.

[69] Z. Szlávik, T. Szirányi, "Face Identification with CNN-UM", *European Conference on Circuit Theory and Design*, Kraków, Poland, Sept. 2003.

[70] Cs. Rekeczky, G. Tímár, and Gy. Cserey "Multi-Target Tracking With Stored Program Adaptive CNN Universal Machines" in *Proc. 7th IEEE International Workshop on Cellular Neural Networks and their Applications*, Frankfurt am Main, Germany, July 22-24, 2002., pp. 299-306.

[71] L. Török, Á. Zarándy "CNN Optimal Multiscale Bayesian Optical Flow Calculation", *European Conference on Circuit Theory and Design*, Kraków, Poland, Sept. 2003.

[72] Z. Fodróczi, A. Radványi "Computational Auditory Scene Analysis in Cellular Wave Computing Framework" *International Journal of Circuit Theory and Applications,* Vol: 34(4) pp: 489-515, ISSN:0098-9886 (July 2006)

[73] L. Kék and Á. Zarándy, "Implementation of Large-Neighborhood Nonlinear Templates on the CNN Universal Machine", *International Journal of Circuit Theory and Applications,* Vol. 26, No. 6, pp. 551-566, 1998.

[74] G. Constantini, D. Casali, M. Carota, and R. Perfetti, Translation and Rotation of Grey-Scale Images by means of Analogic Cellular Neural Network, *Proceedings of the IEEE International Workshop on Cellular Neural Networks and their Applications (CNNA'2004)*, ISBN 963-311-357-1, pp. 213-218, Budapest, Hungary, 2004.

[75] M. Radványi, G. E. Pazienza, and K. Karacs, "Crosswalks Recognition through CNNs for the Bionic Camera: Manual vs. Automatic Design", in Proc. of the 19th European Conference on Circuit Theory and Design, Antalya, Turkey, 2009.

[76] L.O. Chua and L. Yang, "Cellular Neural Networks: Theory and Applications", IEEE Transactions on Circuits and Systems, vol. 35, no. 10, October 1988, pp. 1257-1290, 1988.

[77] L.O. Chua and T. Roska, "The CNN Paradigm", IEEE Transactions on Circuits and Systems - I, vol. 40, no. 3, March 1993, pp. 147-156, 1993.

[78] T. Roska and L.O. Chua, "The CNN Universal Machine: An Analogic Array Computer", IEEE Transactions on Circuits and Systems - II, vol. 40, March 1993, pp. 163-173. 1993.

[79] S. Espejo, R. Carmona, R. Domínguez-Castro and A. Rodríguez-Vázquez "A VLSI-Oriented Continuous-Time CNN Model", International Journal of Circuit Theory and Applications, Vol. 24, pp. 341-356, May-June 1996.

[80] Cs. Rekeczky and L. O. Chua, "Computing with Front Propagation: Active Contour and Skeleton Models in Continuous-time CNN", Journal of VLSI Signal Processing Systems, Vol. 23, No. 2/3, pp. 373-402, November-December 1999.

[81] J.M.Cruz, L.O.Chua, and T.Roska, "A Fast, Complex and Efficient Test Implementation of the CNN Universal Machine", Proc. of the third IEEE Int. Workshop on Cellular Neural Networks and their Application (CNNA-94), pp. 61-66, Rome Dec. 1994.

[82] H.Harrer, J.A.Nossek, T.Roska, L.O.Chua, "A Current-mode DTCNN Universal Chip", Proc. of IEEE Intl. Symposium on Circuits and Systems, pp135-138, 1994.

[83] A. Paasio, A. Dawindzuk, K. Halonen, V. Porra, "Minimum Size 0.5 Micron CMOS Programmable 48x48 CNN Test Chip" European Conference on Circuit Theory and Design, Budapest, pp. 154-15, 1997.

[84] Gustavo Liñan Cembrano, Ángel Rodríguez-Vázquez, Servando Espejo-Meana, Rafael Domínguez-Castro: ACE16k: A 128x128 Focal Plane Analog Processor with Digital I/O. Int. J. Neural Syst. 13(6): 427-434 (2003)

[85] S. Espejo, R. Carmona, R. Domingúez-Castro, and A. Rodrigúez-Vázquez, "CNN Universal Chip in CMOS Technology", Int. J. of Circuit Theory & Appl., Vol. 24, pp. 93-111, 1996

[86] S. Espejo, R. Carmona, R. Domínguez-Castro and A. Rodríguez-Vázquez "A VLSI-Oriented Continuous-Time CNN Model", International Journal of Circuit Theory and Applications, Vol. 24, pp. 341-356, May-June 1996.

[87] P.Dudek "An asynchronous cellular logic network for trigger-wave image processing on fine-grain massively parallel arrays", IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,. 53 (5): pp. 354-358, 2006.

[88] A. Lopich, P. Dudek, "Implementation of an Asynchronous Cellular Logic Network As a Co-Processor for a General-Purpose Massively Parallel Array", ECCTD 2007, Seville, Spain.

[89] A. Lopich, P. Dudek., " Architecture of asynchronous cellular processor array for image skeletonization", Circuit Theory and Design, Volume: 3, On page(s): 81-84, 2005.

[90] P.Dudek and S.J.Carey, "A General-Purpose 128x128 SIMD Processor Array with Integrated Image Sensor", Electronics Letters, vol.42, no.12, pp.678-679, June 2006

[91] Z. Nagy, P. Szolgay "Configurable Multi-Layer CNN-UM Emulator on FPGA" IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 50, pp. 774-778, 2003

[92] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy "Introduction to the Cell multiprocessor" IBM J. Res. & Dev. vol. 49 no. 4/5 July/September 2005

[93] www.ti.com

[94] 176x144 Q-Eye chip, www.anafocus.com

[95] Video security application: http://www.objectvideo.com/ efficient

[96] Cs. Rekeczky, J. Mallett, A. Zarandy, „Security Video Analitics on Xilinx Spartan -3A DSP", Xcell Journal, Issue 66, fourth quarter 2008, pp: 28-32

[97] Á. Zarándy, "The Art of CNN Template Design", Int. J. Circuit Theory and Applications - Special Issue: Theory, Design and Applications of Cellular Neural Networks: Part II: Design and Applications, (CTA Special Issue - II), Vol.17, No.1, pp.5-24, 1999

[98] Á. Zarándy, P. Keresztes, T. Roska, and P. Szolgay, "CASTLE: An emulated digital architecture; design issues, new results", Proceedings of 5th IEEE International Conference on Electronics, Circuits and Systems, (ICECS'98), Vol. 1, pp. 199-202, Lisboa, 1998

[99] P. Keresztes, Á. Zarándy, T. Roska, P. Szolgay, T. Bezák, T. Hídvégi, P. Jónás, A. Katona, "An emulated digital CNN implementation", Journal of VLSI Signal Processing Special Issue: Spatiotemporal Signal Processing with Analogic CNN Visual Microprocessors, (JVSP Special Issue), Kluwer, 1999 November-December

[100] P. Földesy, Á. Zarándy, Cs. Rekeczky, and T. Roska „Configurable 3D integrated focal-plane sensor-processor array architecture", Int. J. Circuit Theory and Applications (CTA), pp: 573-588, 2008

[101] L.O. Chua, T. Roska, T. Kozek, Á. Zarándy "CNN Universal Chips Crank up the Computing Power", IEEE Circuits and Devices, July 1996, pp. 18-28, 1996.

[102] T. Roska, L. Kék, L. Nemes, Á. Zarándy, M. Brendel and P. Szolgay, "CNN Software Library (Templates and Algorithms) Version 7.2", (DNS-1-1998), Budapest, MTA SZTAKI, 1998, http://cnn-technology.itk.ppke.hu/Library_v2.1b.pdf

[103] http://www.xilinx.com/support/documentation/data_sheets/ds706.pdf

INDEX

INDEX (OLD NAMES)