# Program Optimization Methodology

E. Dokladalova

ESIEE, November 2011

# Outline

- Objectives
- Basic notions
- Practical example 1: Deriche filter
- Practical example 2: Hough transform
- Project introduction
- Conclusions

# Objectives

- Learn and experience software optimization methodology

*Donald E. Knuth (1974):*

*…We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil…" [1]*

- Apply previously acquired notions of program optimization techniques

# Basic notions

- Software optimization

  *Application of a collection of methods/techniques allowing to improve the software performances in terms of*

  - Execution time
  - Memory occupation (data, code)
  - Power budget
  - …

- Software optimization methodology

  *Study of methods (and their relations) that have been applied within the software optimization domain*;

  - Defines general guidelines for the software optimization

# Methodology

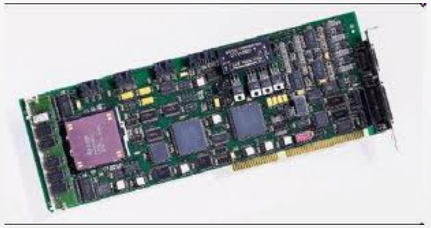
- Algorithm – Architecture Matching (Adéquation)
  - *Aims to study simultaneously both algorithmic and architectural issues*
  - *Takes into account multiple implementation constraints, as well as algorithm and architecture optimizations, that couldn't be achieved otherwise if considered separately.*

# Methodology (II)

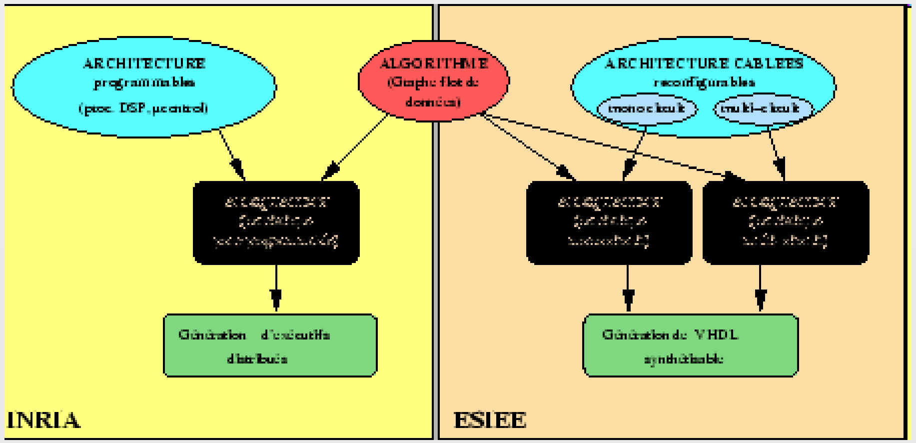
- Algorithm – Architecture  Matching (Adéquation)
  - Allows solving the problem of optimized software implementation on existing hardware (RISC, DSP, VLIW, …)

Application
with constraints

X



  - Proposes improved and automated design flows for specialized architectures, optimized for given application field



# Levels of software optimization

- Algorithm design level
  - Choice of algorithm
- Source code level
  - Produce the good quality of code (Critical parts of code in assembler)

with respect to the features of hardware architecture resources

- Compiler level
  - Automatic optimization using the compiler capabilities

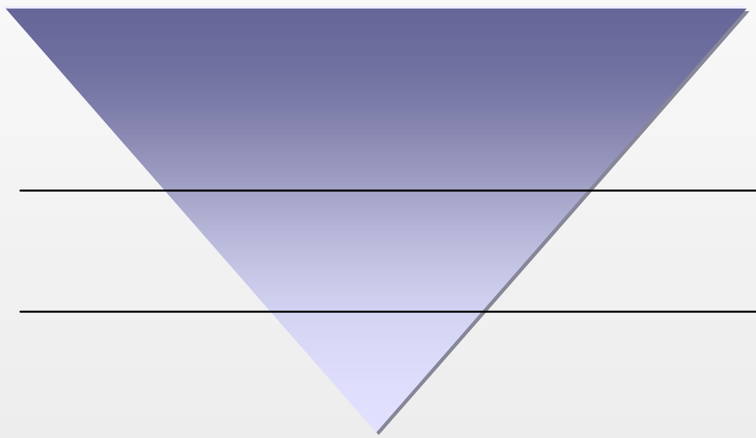
# Source program and compiler level

- Source program level
  - Register rotation
  - Loop unrolling
  - Software pipeline
  - Data locality exploitation
  - Respect of the hardware architecture features
    - RISC
    - SIMD, VLIW
    - DMA
    - Hiérarchie mémoire
- Compiler level
  - Automatic loop unrolling, optimization of memory accesses, local code optimization, pipeline optimization …

Used in practical session on winDLX

# Practical considerations

- Contribution of each optimization level to the final gain of performances



- Algorithm
- Source code
- Compiler

# Algorithm design level optimization

- Choice of algorithm
  - Example: some of N integers

```
int i, sum;
sum = 0;
for (i = 1; i <= N; ++i)
        sum += i;
```

```
int i sum;
sum = N * (1 + N) / 2;
```

- Data type

| Processor/instruction (32 bits) | Pentium III/IV | |
|---|---|---|
| | Latency | Pipelined ? |
| Integer Add | 1 clk | 3 |
| Integer Multiply | 10 clk | 1 |
| Float Multiply and add | 6 clk | 1 |
| Float Div | 38 clk | no |

# Algorithm design level optimization

- Design of algorithm defines:
  - Data dependency
    - Locally dependent data, globally dependent data
  - Data structures and amount of occupied memory
    - Tables, lists, unions, trees …
  - Data access
    - Random access, streaming, parallel access…
  - Data types/precision
    - Integer, floating point, double, …
  - Number of operations
    - Data load/store operations, arithmetic operations, …

- Trade-off: optimization of one or two parameters at expense of some others
  - Some examples :
    - Execution time x amount of occupied memory
    - Execution time x numerical precision
    - Execution time x energy budget

# Software optimization process

1. Choice of algorithms and first “basic” implementation

2. Identification of bottlenecks
   - Profiling techniques - collection of tools for estimation of metrics used for optimization
     - Execution time
     - Memory access
     - Number and type of instructions
     - Numbers of function calls
     - …
   - Available tools
     - Gprof, Valgrind, …
     - CCstudio, VTune

Repeat until the given implementation constraints are satisfied

3. Algorithm and Source code optimization
   - Optimization starts by the most demanding task (80-20 rule !)

# Software optimization process (II)

- **Guidelines** (time and memory optimization)

  1. For given algorithm, estimate the required performances with respect to the given implementation constraints
     - Number of operations per second and memory bandwidth

  2. For present implementation, estimate manually or by profiling tools
     - Number of operations per second (MOps),
     - Number of floating points operations (Flops)
     - Memory access bandwidth (Bytes per second)

  3. Considering the results of 1. and 2., analyse the efficiency of hardware resources utilization
     - Specialized computing units: FPU, SIMD, …
     - Memory organization and hierarchy, DMA access
     - Load balance of different computing units

  4. Modify the algorithm or source code in order to exploit better the available hardware
     - Reduce number of operations, modify data dependency
     - Use DMA data transfer (increases the available memory bandwidth)
     - Exploit data and instruction parallelism

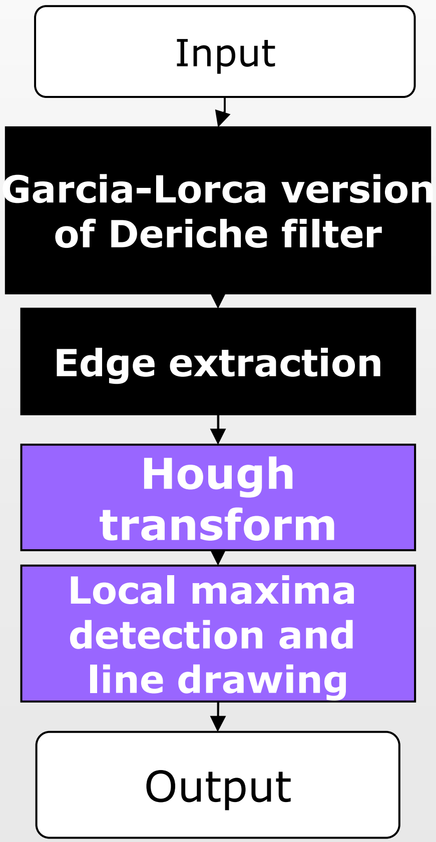
# Application example

- Automotive security: lane departure warning system
  - Strict real time constraints
  - Embedded system affected also by space occupation and power budget constraints

- Lane detection by Hough transform
  - Prototyping methodology
    - Simulink prototype (demonstration)
    - Application design in Matlab (demonstration and profiling example)
    - Application transfer on DSP platform (objective of your project)
    - Optimization of the execution (objective of your project)

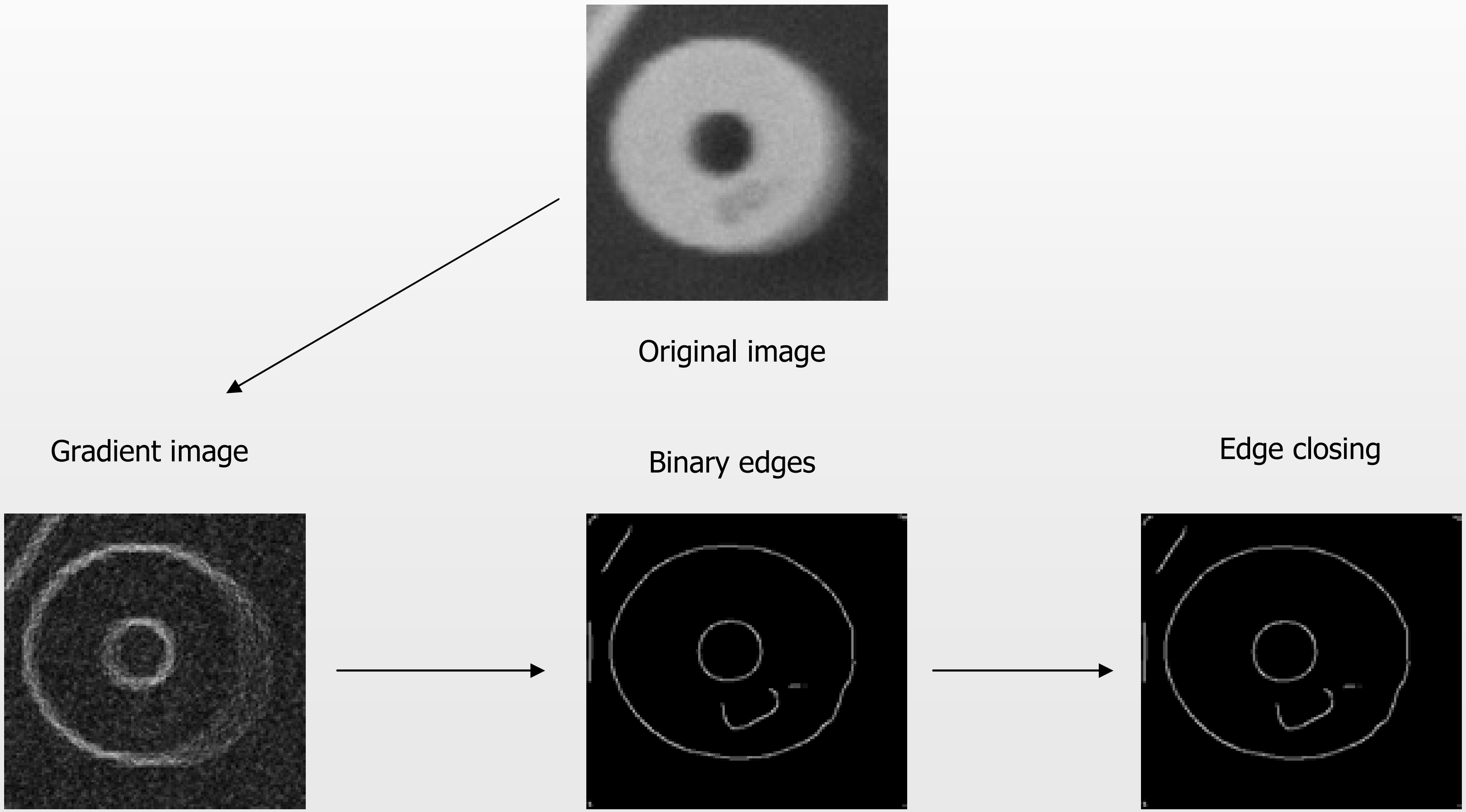
# Application profiling example

- Matlab implementation of line detection by Hough transform

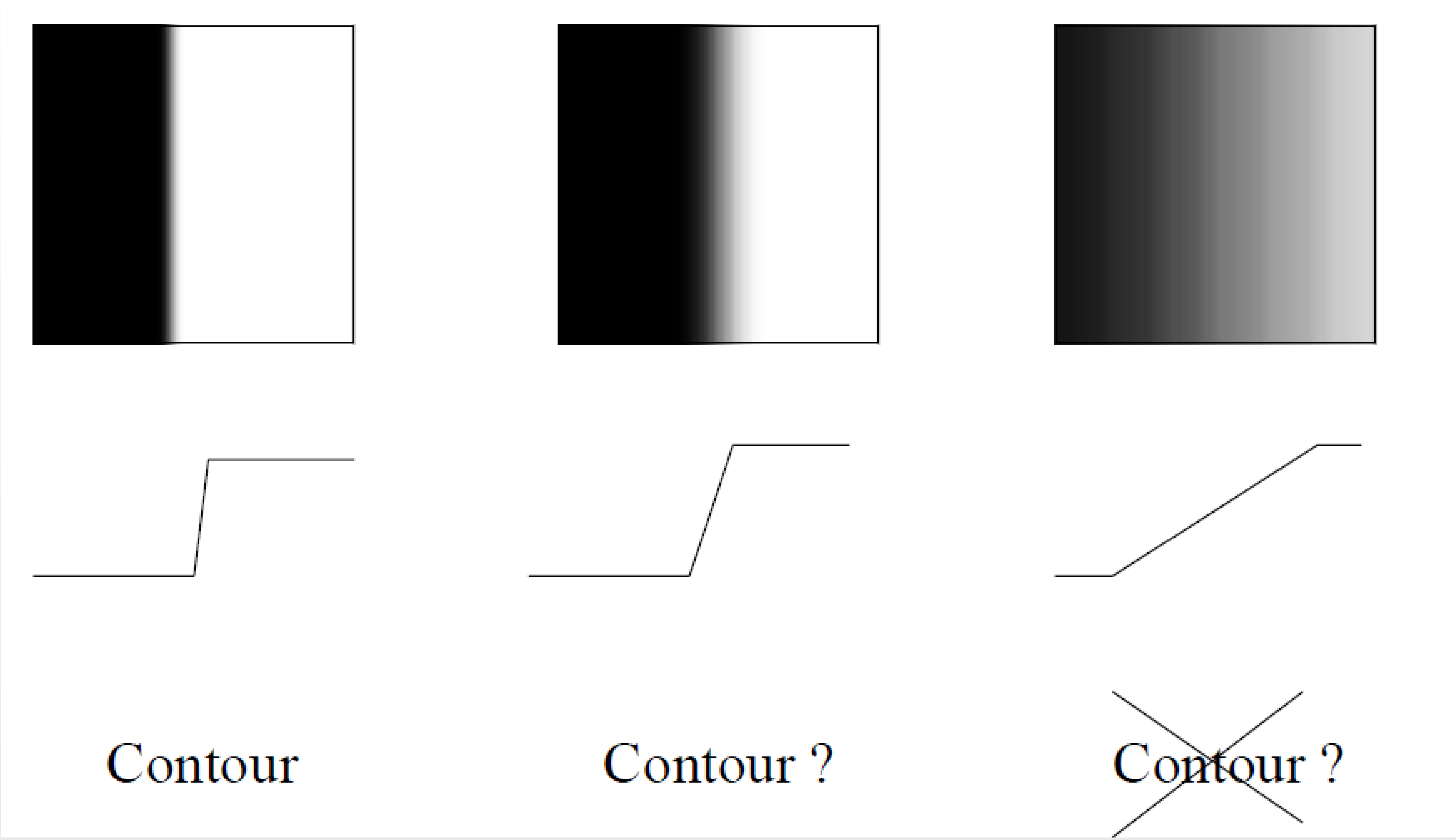



| Function Name | Calls | % | **Total Time** | Self Time* |
|---|---|---|---|---|
| lf4arch | 1 | 100 | 29.046 s | 0.684 s |
| Hough | 1 | 75 | 22.074 s | 2.978 s |
| Garcia-Lorca | 1 | 3 | 0.929 s | 0.929 s |
| Local max and lines | 1 | 1,9 | 0.555 s | 0.134 s |
| imread | 1 | 1,7 | 0.493 s | 0.122 s |
| Edge extraction | 1 | <1 | 0,087 s | 0,087 s |

- Principle of edge detection operator chain

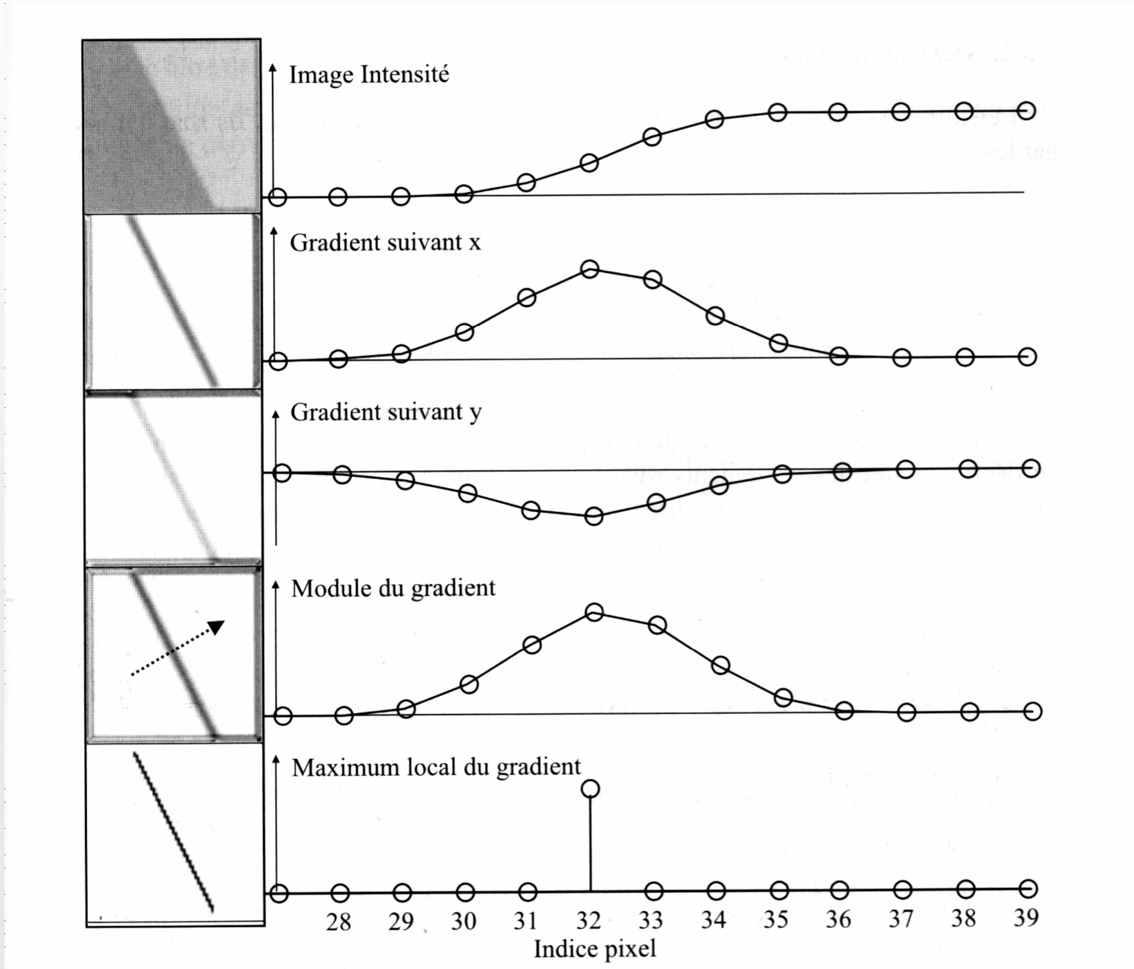


# Edge detection problem



- Edge detection – gradient
- Digital image (2D)

$$P : Z^2 \rightarrow R$$



# Sobel gradient

- 2 Masques :

Vertical

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

Horizontal

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

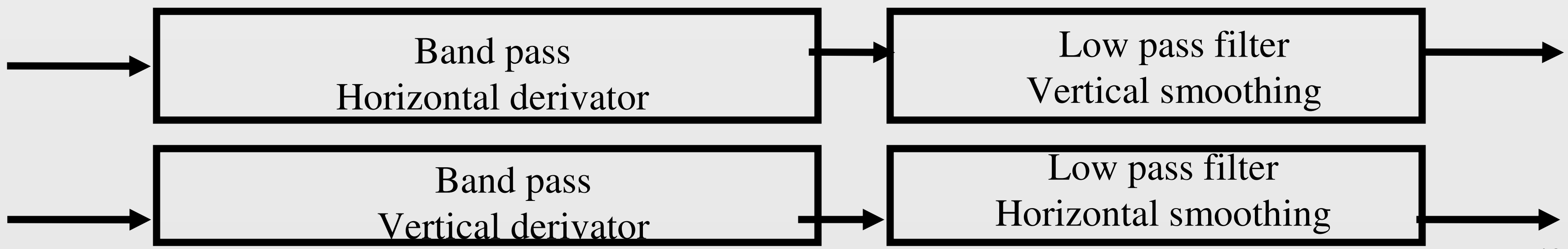
- Finite impulse response filter
  - Other examples: Prewitt, Roberts

- Horizontal filter
  - *p(k) denotes pixel p at position k, N is image width in pixels*

$$g_h(k) = p(k-N+1) - p(k-N-1) + 2p(k+1) - 2p(k-1) + p(k+N+1) - p(k+N-1)$$

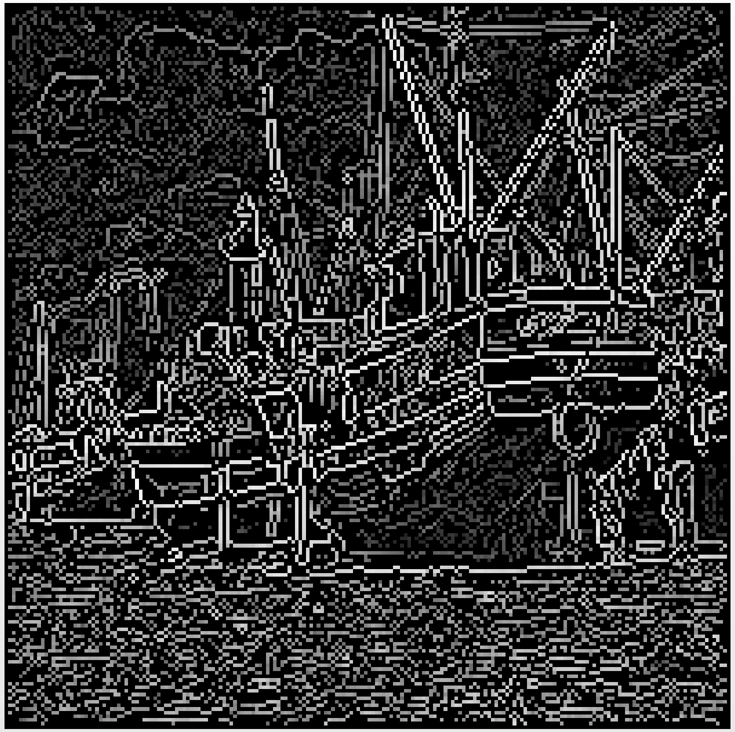
- Transfer function

$$SH(z) = Gh(z) / (Z^{-N+1} - z^{-N-1} + 2z - 2z^{-1} + z^{N+1} - z^{N-1}) ) = 1 / [(z-z^{-1}) (z^{-N}+2+z^{N})]$$



- Example:

Original image



Gradient image

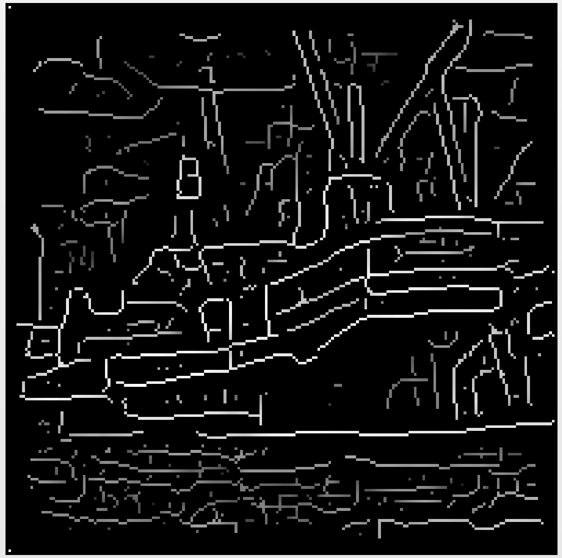
# Deriche optimal edge detector

- Recursive filter (Infinite Impulse Response)
  - Any filter width obtained in constant time !
- Parameter α  defining the « **width** » of filter,
  - trade-off between the quality of detection and the precision of edge localization
    - For larger α we obtain more edges
    - For smaller α we delete the less significant details

Original image

*alpha = 1*

*alpha = 0.5*



# Deriche edge detector (II)

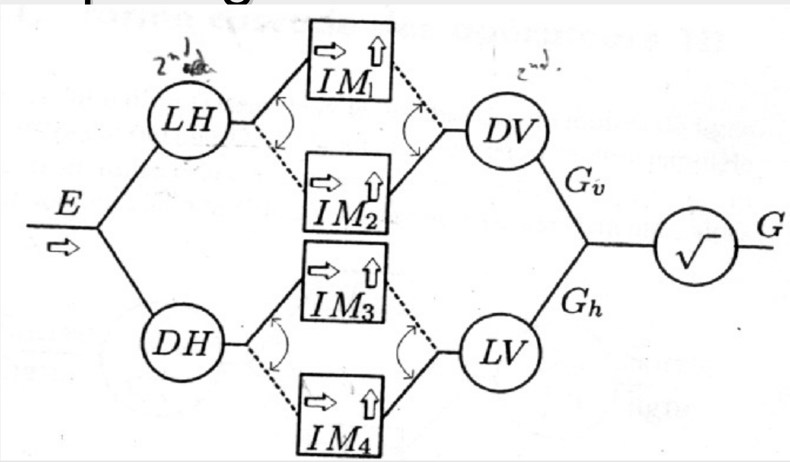
- Deriche filter in Z transform
  - Smoother

$$L(z) = k_L \left[ \frac{(\alpha+1)e^{-\alpha}z - e^{-2\alpha}z^2}{(1-e^{-\alpha}z)^2} + \frac{1+(\alpha-1)e^{-\alpha}z^{-1}}{(1-e^{-\alpha}z^{-1})^2} \right]$$

  - Derivator

$$D(z) = k_D \left( \frac{z}{(1-e^{-\alpha}z)^2} - \frac{z^{-1}}{(1-e^{-\alpha}z^{-1})^2} \right)$$

- Standard computing scheme




H – horizontal direction

V – vertical direction

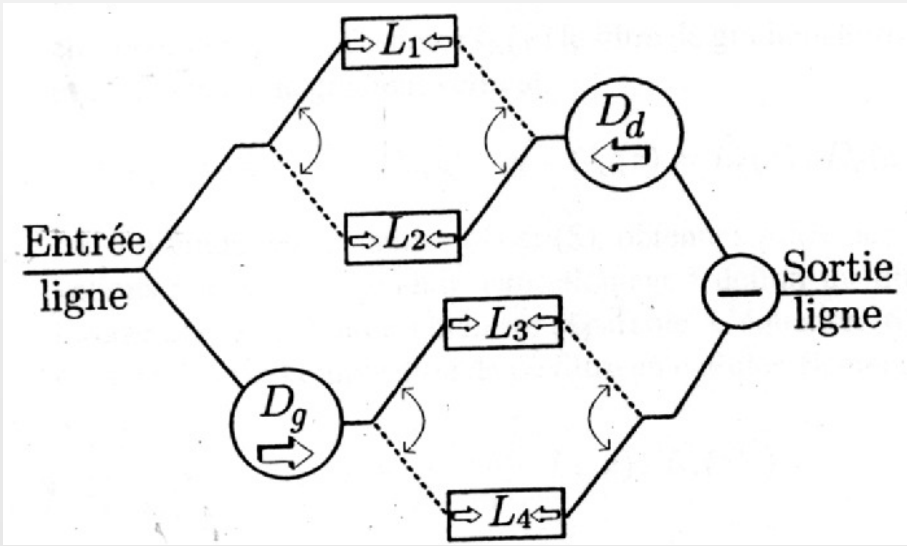
$IM_x$ – allocated image

# Parallel organization of derivator

- 2 poles → 2 processing directions: causal et anticausal

$$D(z) = k_D \left( \underbrace{\frac{z}{(1 - e^{-\alpha} z)^2}}_{Dg} - \underbrace{\frac{z^{-1}}{(1 - e^{-\alpha} z^{-1})^2}}_{Dd} \right)$$

- Anticausal pixel reading requires entire image line in memory -> 4 memory lines used in « ping pong »



# Derivator algorithm evaluation

- Algorithm (1 ligne, 1 direction)

**Number of operations per pixel**

**Pour** $k$ **de** $0$ **à** $N-1$

$$y_m(k) = p(k-1) + 2e^{-\alpha} y_m(k-1) - e^{-2\alpha} y_m(k-2)$$

**Fin pour** $k$

**2 +**
**2 x**

**Pour** $k$ **de** $N-1$ **à** $0$

$$y_p(k) = p(k+1) + 2e^{-\alpha} y_p(k+1) - e^{-2\alpha} y_p(k+2)$$

$$d(k) = (1 - e^{-\alpha})^2 (y_p(k) - y_m(k))$$

**Fin pour** $k$

**3 +**
**3 x**

---

**5 ADD, 5 MUL**

- Smoother has the same resolution as derivator

$$L(z) = k_L \left[ \frac{(\alpha+1)e^{-\alpha}z - e^{-2\alpha}z^2}{(1-e^{-\alpha}z)^2} + \frac{1+(\alpha-1)e^{-\alpha}z^{-1}}{(1-e^{-\alpha}z^{-1})^2} \right]$$

  - 2 poles: causal en anticausal processing direction

- Algorithme

$y_m(-2) = y_m(-1) = p(-1) = 0$

**Pour** $k$ de 0 à $N-1$

$$y_m(k) = p(k) + (\alpha-1)e^{-\alpha}p(k-1) + 2e^{-\alpha}y_m(k-1) - e^{-2\alpha}y_m(k-2)$$

**Fin pour** $k$

$y_p(N+1) = y_p(N) = a\, y_m(N-1)$ *et* $p(N) = p(N+1) = b\, y_m(N-1)$

**Pour** $k$ de $N-1$ à 0

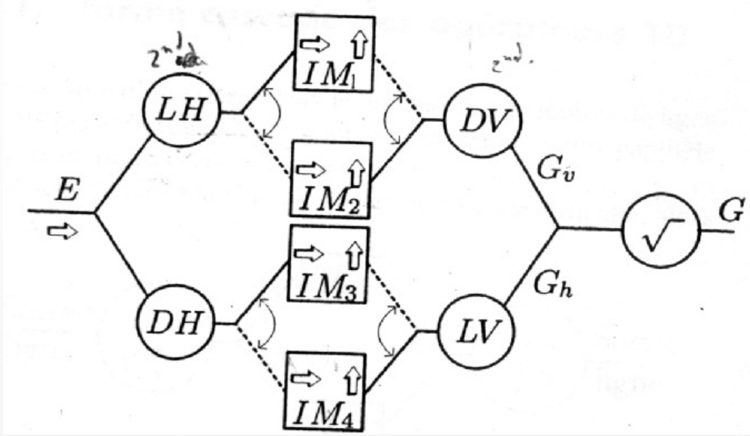
$$y_p(k) = (\alpha+1)e^{-\alpha}p(k+1) - e^{-2\alpha}p(k+2) + 2e^{-\alpha}y_p(k+1) - e^{-2\alpha}y_p(k+2)$$

$$l(k) = \frac{(1-e^{-\alpha})^2}{1+2\alpha e^{-\alpha} - e^{-2\alpha}} (y_p(k) + y_m(k))$$

**Fin pour** $k$

**7 ADD, 8 MUL**

# Deriche edge detector in 2D




- Transfer functions
  - $GH(z)=D(z)L(z^N)P(z)$
  - $GV(z)=D(z^N)L(z)P(z)$

➔ order of operator computing has no importance

➔ Intermediate storage has to ensure the data validity : i.e. finish D before L

- Number of operations per pixel: 26 MUL et 24 ADD

- Memory occupation
  - 4 image memories + 16 memory lines
- Data type
  - Computing in floating point because of alpha

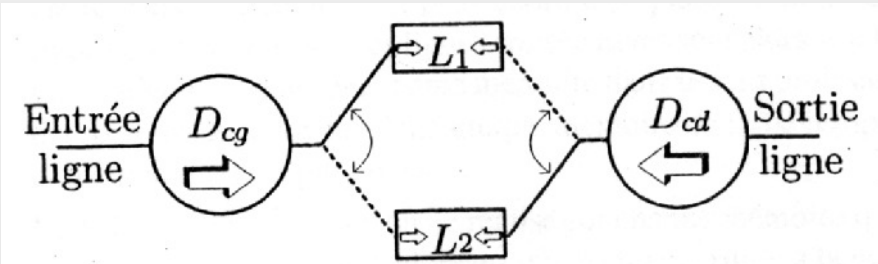
# Optimization of Deriche edge detector

- Called also Garcia-Lorca optimization
- Objective: reduction of occupied memory and number of operations

$$\gamma = e^{-\alpha} \quad D(z) = k_D \left( \frac{z}{(1-\gamma z)^2} - \frac{z^{-1}}{(1-\gamma z^{-1})^2} \right)$$

$$D(z) = k'_D \left( z - z^{-1} \right) \frac{1}{(1-\gamma z)^2} \frac{1}{(1-\gamma z^{-1})^2}$$

- Cascade of basic 1-D operators
  - Preserves complexity but reduces memory occupation



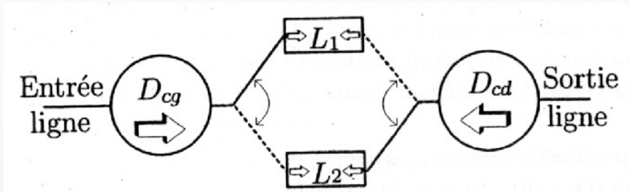
# Optimisation Garcia-Lorca

- New derivator formulation

$$D_c(z) = D_{cg}(z) \cdot D_{cd}(z)$$

$$D_{cg}(z) = \frac{(1-\gamma)^2 \cdot z^{-1}}{(1-\gamma z^{-1})^2}$$

$$D_{cd}(z) = \frac{(1-\gamma^2) \cdot (z^2 - 1)}{(1-\gamma z)^2}$$



# Garcia Lorca optimization

- Derivator

$$\gamma = e^{-\alpha} \quad D(z) = k_D \left( \frac{z}{(1-\gamma z)^2} - \frac{z^{-1}}{(1-\gamma z^{-1})^2} \right)$$

$$D(z) = k'_D \, (z - z^{-1}) \, \frac{1}{(1-\gamma z)^2} \, \frac{1}{(1-\gamma z^{-1})^2}$$

**Pour** $k$ de 0 à $N-1$

$$y_m(k) = p(k) + 2\gamma \, y_m(k-1) - \gamma^2 \, y_m(k-2)$$

**Fin pour** $k$

**Pour** $k$ de $N-1$ à 0

$$y_p(k) = y_m(k) + 2\gamma \, y_p(k+1) - \gamma^2 \, y_p(k+2)$$

$$d(k+1) = (1-\gamma)^2 \, (1-\gamma^2) \, (y_p(k+2) - y_p(k))$$

**Fin pour** $k$

# Garcia Lorca optimization

- New smoother formulation (trapèzes method)

$$L'(z) = (z + 2 + z^{-1})\left(\frac{1}{(1-\gamma z)^2}\,\frac{1}{(1-\gamma z^{-1})^2}\right)$$

Recursive part is equivalent to derivator

- Derivator non recursive part = Sobel derivator
- Smoother non recursive part = Sobel smoother

- Factorization of non recursive parts

$$\begin{aligned} (z - z^{-1}) &= (1 - z^{-1})\,\cancel{(1 + z)} \\ (z + 2 + z^{-1}) &= (1 + z^{-1})\,\cancel{(1 + z)} \end{aligned}$$

- New smoother and derivator of Garcia Lorca

$$Dgl(z) = kgl_D\,(1 - z^{-1})\,\frac{1}{(1-\gamma z)^2}\,\frac{1}{(1-\gamma z^{-1})^2}$$

$$Lgl(z) = kgl_L\,(1 + z^{-1})\,\frac{1}{(1-\gamma z)^2}\,\frac{1}{(1-\gamma z^{-1})^2}$$

LGL(z)

# Garcia Lorca : gradient 2D

- $GH(z)=Dgl(z)Lgl(z^N)P(z)$
- $GV(z)=Dgl(z^N)Lgl(z)P(z)$

- Impose :

$$LGL(z) = \frac{1}{(1-\gamma z)^2}\,\frac{1}{(1-\gamma z^{-1})^2} \qquad kgl = \frac{(1-\gamma)^4(1-\gamma^2)^3}{2\,(1+\gamma^2)}$$

- Result :

$$\begin{aligned} GH(z) &= kgl\; LGL(z)\; LGL(z^N)\,\boxed{(1-z^{-1})\,(1+z^{-N})}\,P(z) \\ GV(z) &= kgl\; LGL(z)\; LGL(z^N)\,\boxed{(1-z^{-N})\,(1+z^{-1})}\,P(z) \end{aligned}$$

- Impose :

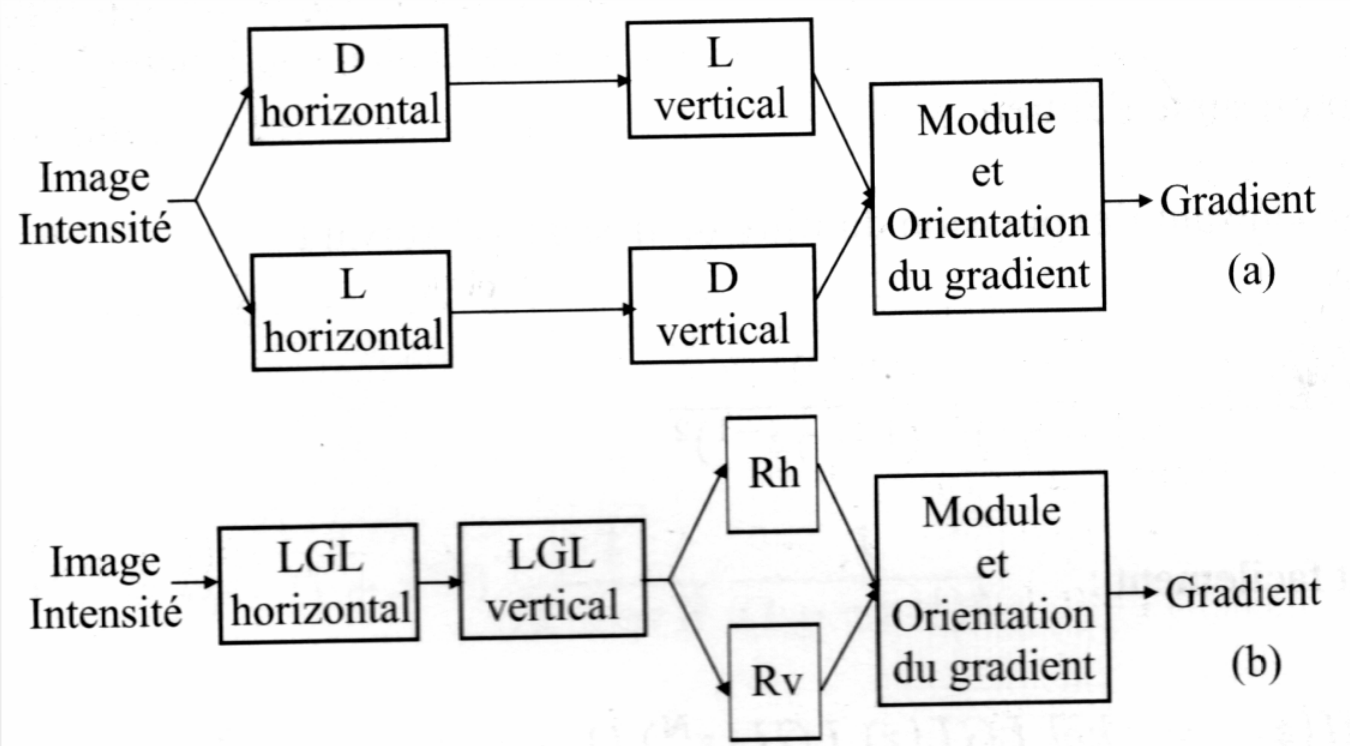
$$\begin{aligned} R_h(z) &= (1-z^{-1})\,(1+z^{-N}) \\ R_v(z) &= (1-z^{-N})\,(1+z^{-1}) \end{aligned}$$

- Rh et Rv = local filters with masks

| | |
|---|---|
| -1 | 1 |
| -1 | 1 |

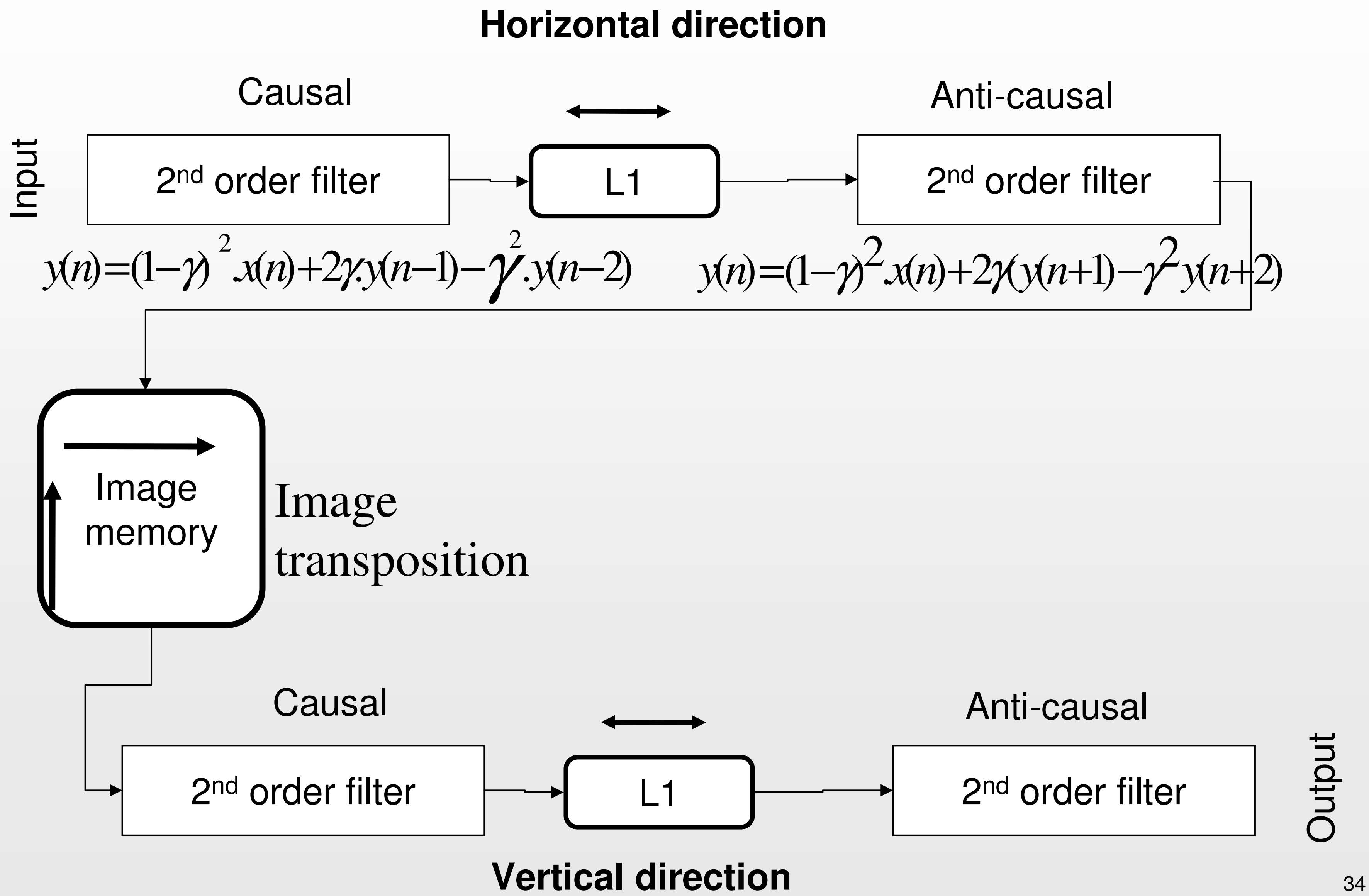
| | |
|---|---|
| -1 | –1 |
| 1 | 1 |

# Garcia Lorca : new organization of gradient 2D




Equivalent complexity of D and L; equivalent to LGL
-> gain of 50%

# Garcia Lorca smoother filter organization



# Garcia Lorca 2D edge detector evaluation

Horizontal smoothing : **4 +, 4 x**

**Pour** $i$ de 0 à $N-1$ *(chaque ligne)*

$lh(i,N+1) = lh(i,N) = lh(i,N-1)$
**Pour** $k$ de $N-1$ à 0
$lh(i,k) = p(i,k) + 2\gamma\, lh(i,k+1) - \gamma^2\, lh(i,k+2)$
**Fin pour** $k$
$lh(i,-2) = lh(i,-1) = lh(i,0)$
**Pour** $k$ de 0 à $N-1$
$lh(i,k) = lh(i,k) + 2\gamma\, lh(i,k-1) - \gamma^2\, lh(i,k-2)$
**Fin pour** $k$
**Fin pour** $i$

Vertical smoothing : **4 +, 5 x**

**Pour** $j$ de 0 à $N-1$ *(chaque colonne)*

**Pour** $k$ de $N-1$ à 0
$lv(k) = lh(k) + 2\gamma\, lv(k+1) - \gamma^2\, lv(k+2)$
**Fin pour** $k$
$lv(-2) = lv(-1) = lv(0)$
**Pour** $k$ de 0 à $N-1$
$lv(k) = kgl\, lv(k) + 2\gamma\, lv(k-1) - \gamma^2\, lv(k-2)$
**Fin pour** $k$
**Fin pour** $j$

Local derivation: **4 +**

**Pour** $j$ de 1 à $N-1$ *(chaque colonne)*
**Pour** $i$ de 1 à $N-1$ *(chaque ligne)*
$th(i,j) = lv(i,j) - lv(i,j-1)$
$tv(i,j) = lv(i,j) + lv(i,j-1)$
$gh(i,j) = th(i,j) + th(i-1,j)$
$gv(i,j) = tv(i,j) - tv(i-1,j)$
**Fin pour** $i$
**Fin pour** $j$

⇒ 9 MUL (if normalization constant applied)
12 ADD

# Comparaison

- Original Deriche filter
  - Number of operations per pixel: 26 MUL et 24 ADD
  - Number of operations for 25 frames per second (fps):
    - 25 fps 640x480 (VGA) => 384 MOPS
    - 25 fps 1920x1080 (HD) => 2,6 GOPS
  - Memory occupation
    - 4 image memories + 16 memory lines
  - Data type
    - Computing in floating point because of alpha

- Garcia Lorca optimization
  - Number of operations per pixel: 9 MUL et 12 ADD
  - Number of operations for 25 frames per second (fps):
    - 25 fps 640x480 (VGA) => 161 MOPS
    - 25 fps 1920x1080 (HD) => 1,1 GOPS
  - Memory occupation
    - 2 image memories + 4 memory lines
  - Data type
    - Computing in floating point because of gamma

- Unique constant : $\gamma$
  - Coding of $\gamma$ defines the number of possible filters
    - i.e. 3 bits : $\gamma = \gamma_1 2^{-1} + \gamma_2^{-2} + \gamma_3^{-3}$ res different filters

| $\gamma$ | $\alpha$ | résolution |
|---|---|---|
| 0.125 | 2.08 | 1 |
| 0.250 | 1.38 | 2 |
| 0.375 | 0.98 | 3 |
| 0.500 | 0.69 | 4 |
| 0.625 | 0.47 | 5 |
| 0.750 | 0.29 | 8 |
| 0.875 | 0.13 | 18 |

  - Computing precision
    - To ensure the stability of system:

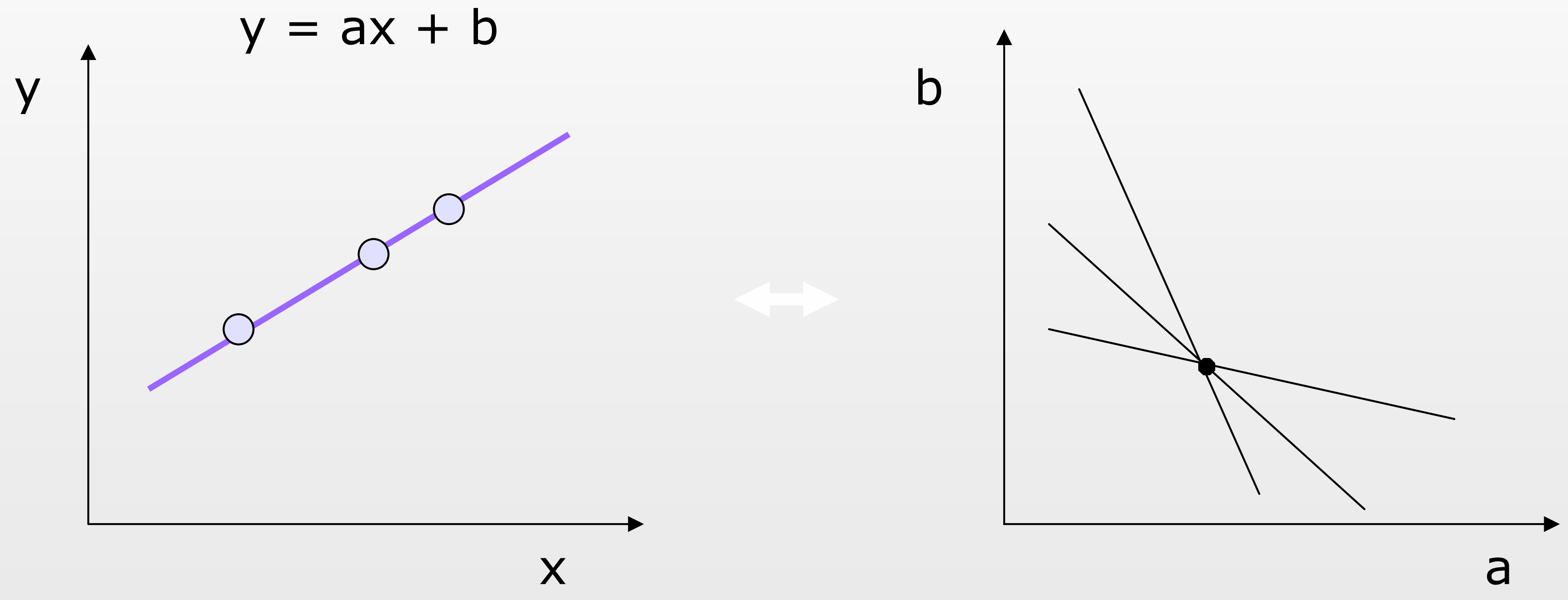
      21 bits for computing, storage in 12 bits

# Hough transform

- Object recognition technique (1962)
- Parametric description : lines, circles, ellipses
- Good noise robustness
- Allows detecting the partially overlapping objects

# Hough transform principle

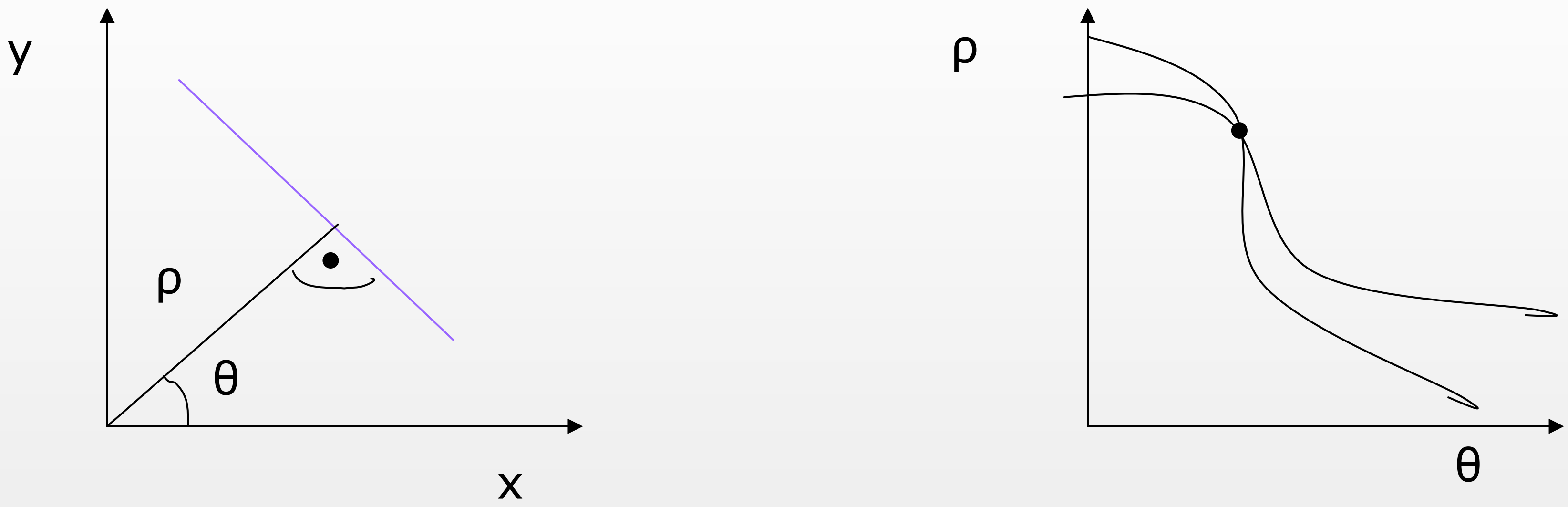
- Lines detection




"image" space

"parameters" space

# Hough transform: space (ρ, θ)

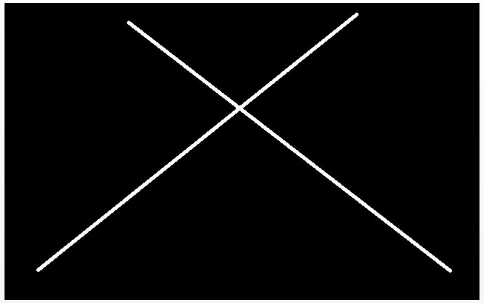



$$x = \rho \cos(\theta)$$
$$y = \rho \sin(\theta)$$

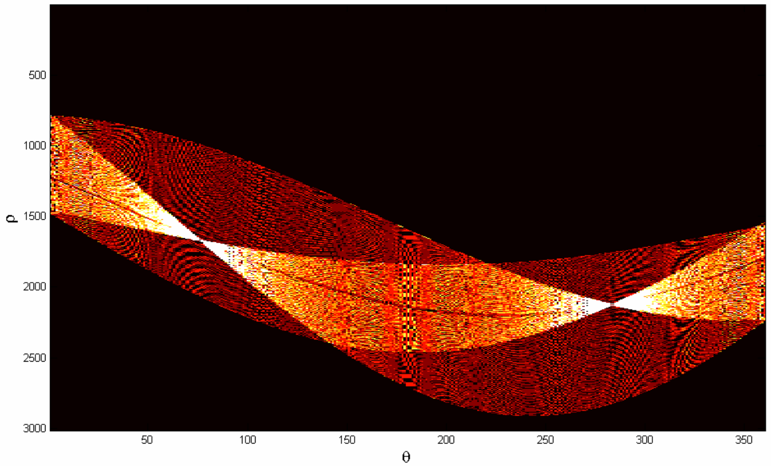
$$x \cos(\theta) + y \sin(\theta) - \rho = 0$$

Sinusoids, corresponding to one point on the line, have an intersection at the parameters representing the line.

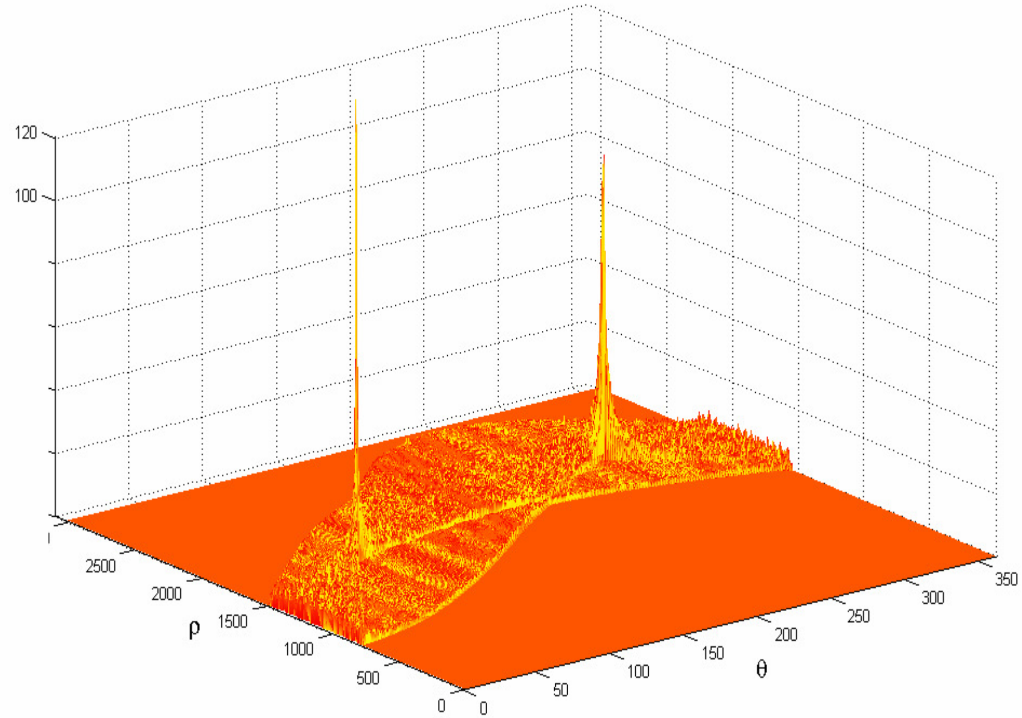
# Hough transform



Original image



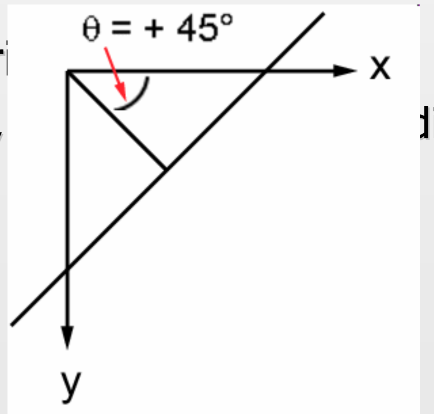

2D view of parameters space
Hough accumulator




3D view of Hough accumulator

## Standard algorithm: initialization

- Allocate and intialize to 0 Hough accumulator space A(ρ, θ)
  - Define the precision of parameters ρ, θ
  - Define the intervals of parameters
    - Angle :
      - θ ∈ [0°, 180°]
    - Distance to ori
      - ρ ∈ [0, d], diagonal



# Standard algorithm

- Input: binary image P, containing object contours
- For all points p(x,y)>0 :
  - For all $\theta \in [0, 180]$ compute $\rho$ :
    - $\rho = x \cos(\theta) + y \sin(\theta)$
    - $A(\rho, \theta) = A(\rho, \theta) + 1$
  - End for
- End for

```
Ib = P; [m,n]=size (Ib);
m2 = round(m/2);
n2 = round(n/2);
maxrho = (ceil(sqrt(m^2 + n^2)));
% Accumulator initialization
A = zeros (180,maxrho);
% Compute Hough transform for every Ib(x,y)>0
for i=2:m-1,
    for j=2:n-1,
        if Ib(i,j) > 0,
            for angle = 1:180,
               rho = (j)*cosd(angle) + (i)*sind(angle);
               rho_idx = round(rho/2 + maxrho/2);
               if rho_idx > 0,
               A(angle,rho_idx) = A(angle,rho_idx) + 1;
               end;
            end;
        end;
    end;
end;
```

# Standard algorithm evaluation

- Arithmetic complexity
  - Assumption : θ considered with precision Δ = 1

    → for ∀ p(x,y) > 0 compute 180 times $\rho = x \cos(\theta) + y \sin(\theta)$

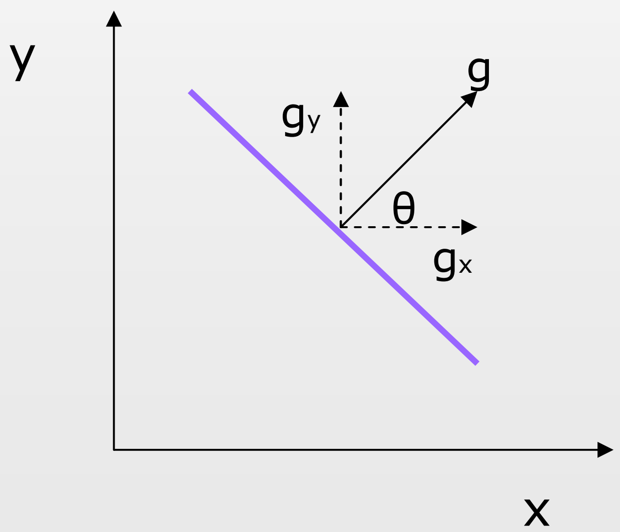
    → if 20% pixels > 0

    25 fps 640*480 => 829 MOPS + 276x10^6 sin,cos

    25 fps 1920*1080 => 5,6 GOPS + 1,7x10^9 sin,cos

  - Remarks :
    - 180 random access per pixel
    - 180 sine and cosine function call per pixel

- Principle : use local gradient information to reduce the number of angles to evaluate
  - Local gradient direction gives a good estimation of θ



# O'Gorman et Clowes Optimization

- For all p(x,y) > 0
  - $\theta = \arctan(g_y/g_x)$
  - $\rho = x \cos(\theta) + y \sin(\theta)$
  - $A(\rho, \theta) = A(\rho, \theta) + 1$

- Arithmetic complexity
  - 1 computation of sine and cosine per pixel
  - Trade-off: 1 division and 1 arctan

  - si 20% pixels de l'image à calculer :
    25 fps 640*480 : 6,1 MOPS + 1,5x10^6 sin,cos,arctan
    25 fps 1920*1080 : 5,6 GOPS + 10,4x10^6 sin,cos,arctan

- Remarks :
  - 1 memory access per pixel
  - Better identification of Hough « peaks »

- Trigonometric functions
  - Mathematical libraries
    - Not always optimum in the terms of execution time
    - Not always matching with required precision
  - Some other possibilities
    - LUT
    - CORDIC

# LUT

- LUT is a table of correspondence associating one output with one input
- LUT allows replacing complex computation by only 1 memory access
- Principle
  - Pre-compute the function values with given precision for given interval of values
  - Create a memory structure containing these values
  - Could be done automatically at the beginning of the program



- Trade-off between LUT size and precision

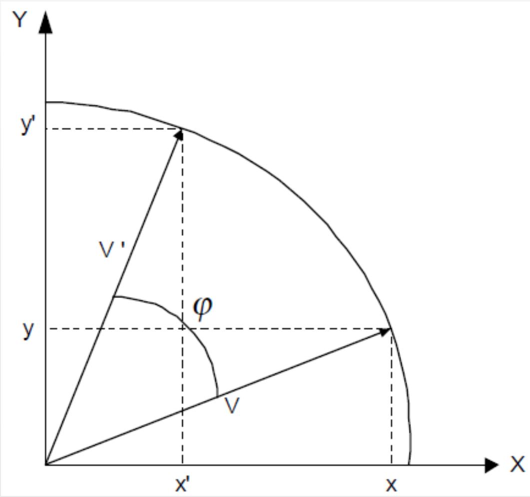
# LUT

- Discretize given interval 0 <= x <= 180 with defined precision
  - If precision Δ = 1, the values of degrees can represent the table index
- Define values precision (int, float, …)
- Allocate LUT memory [180/ Δ]
- Initialize the table
  - Pour i=0, i<=360, i=i+ Δ faire
    - LUT[i] = sin(i)

- CORDIC = COordinate Rotation DIgital Computer (1971)
  - Popular in FPGA implementation

- Principle : recursive computing, the number of iterations determines result precision
  - By successive rotations of vector v, we search its coordinates x,y on unitary circle

$$x' = \cos(\varphi)\ [x - y\tan(\varphi)]$$
$$y' = \cos(\varphi)\ [y + x\tan(\varphi)]$$




- Advantages : minimizes memory occupation

- Example (C)

```
// Initialisation des variables
a = 25;                              // Angle initiale
x=0.607252951;
y=0;
d2=2;                                // Diviseur

    for(i=0; i<=10; i++)
    {
        d2/=2;                       // Multiple de 2^-i
        dx=x*d2;
        dy=y*d2;
        da=atan(d2);
        da= 180*da/PI;               // Pour une valeur en degré

        if(a<0)
        {
            x += dy;
            y  -= dx;
            a += da;
        }
        else
        {
            x -= dy;
            y += dx;
            a -= da;
        }
    }
```

# References


1. Donald E. Knuth, Structured Programming with go to Statement, Computing Surveys, Vol. 6, No. 4, December 1974 (http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf)
2. R. Deriche, Recursively implementing the Gaussian and its derivatives, Technical report, INRIA, N° RR-1893, 1993 (http://hal.inria.fr/docs/00/07/47/78/PDF/RR-1893.pdf)
3. T. Ea, L. Lacassagne, P. Garda, Execution temps reel des detecteurs de contours de Deriche par des processeurs RISC, Congrès Adéquation Algorithme Architecture, 1998, France (http://www.ief.u-psud.fr/~lacas/Publications/AAA98.pdf)
4. F. Lohier, L. Lacassagne, P. Garda, Programming techniques for real time software implementation of optimal edge detectors: a comparison between state of the Art DSP and RISC architectures", DSP World, 1998 (http://www.ief.u-psud.fr/~lacas/Publications/DSPWorld98.pdf)
5. D. Demigny, F. G. Lorca, L Kemal et J.P Cocquerez, Conception nouvelle du détecteur de contours de Deriche, Symposium on signal and Image Processing, GRETSI, 1995 (http://documents.irevues.inist.fr/bitstream/handle/2042/2030/006.PDF%20TEXTE.pdf?sequence=1 )
6. Gen, M.; Cheng, R. (2002), Genetic Algorithms and Engineering Optimization, New York: Wiley
7. http://users.ecs.soton.ac.uk/msn/book/new_demo/hough/
8. http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm
9. Feng Zhou , Peter Kornerup, A High Speed Hough Transform Using CORDIC (1995), In International Conference on Digital Signal Processing (DSP'95) (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.3209)
10. F O'Gorman, M Clowes, Finding picture edges through collinearity of feature points (1973), Third International Joint Conference on Artificial Intelligence (http://www.ijcai.org/Past%20Proceedings/IJCAI-73/PDF/058.pdf)