Lesson 11: Image processing

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Generalities

Generalities:

- an *image* is represented as a two-dimensional array of *pixels* (picture elements)

- each pixel is either
  - a value in the *gray-scale* (usually, a number between 0 and 255)

  or it represents
  - a *color* (in rgb codding, i.e., the intensity of each primary red/green/blue color is specified; usually they are in the range 0-255)

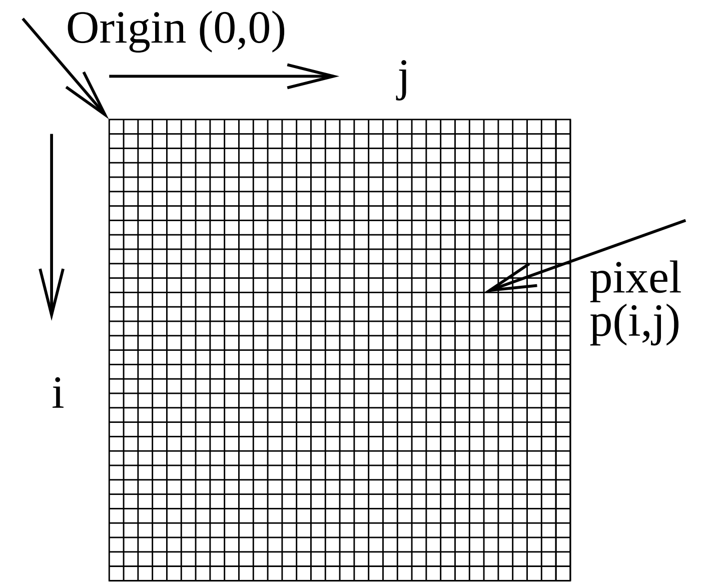


Image processing is computationally intensive

Estimation (the need for parallel processing):

- suppose an image has 1024x1024 8-bit pixels (1Mb)
- if each pixel requires one operation, then 10^6 operations are needed for one frame;
- with computers performing 10^8 operations/sec this would take 10^{-2}s (= 10ms)
- in real-time applications usually the rate is 60-80 frames/sec.; hence each frame need to be processed in 12-16ms
- however, many image processing operations are complex, requiring much more than 1 operation per pixel; hence parallel processing may be useful here

Image processing

Image processing methods: these methods use and modify the pixels; most of them are very suitable for parallel processing; examples:

- basic low-level image processing, including noise cleaning or noise reduction, contrast stretching, smoothing, etc.
- *edge detection*
- *matching* an image against a given template
- *Hugh transformation* (identify the pixels associated to straight lines or curves)
- *(fast) Fourier transform* - passing from an image to a frequency domain (and back)

Point processing

Point processing: methods acting on individual pixels (they are embarrassingly parallel)

- *Thresholding*: the pixels below a threshold are reduced to 0:

$$if(x < threshold) x = 0;\ else\ x = 1$$

- *Contrast stretching*: the range of gray level values is extended to make details more visible

$$x = (x - x_l)\left(\frac{x_H - x_L}{x_h - x_l}\right) + x_L$$

a pixel of value x within the range $[x_l, x_h]$ is stretched to the range $[x_L, x_H]$ (this is often used in medical images, either for soft tissue portions, or for dense bone-like structures)

Histogram

Histogram

- an histogram shows the number of pixels in the image for each gray level (say, between 0 and 255)

- sequential code:

```
for(i=0; i<height_max; x++)
  for(j=0; j<width_max; y++)
    hist[p[i][j]] = hist[p[i][j]] + 1;
```

  (`hist[k]` holds the number of pixels of gray level k)

- a parallel version may be easily developed (parallel addition of a set of numbers)

Smoothing, sharpening, noise reduction

Definitions:

- *smoothing* - suppress large fluctuations in intensity over the image area (can be achieved by reducing the high-frequency content)
- *sharpening* - accentuate the transitions, enhancing the detail (can be achieved in two ways: reduce low-frequency content or accentuate changes through differentiation)
- *noise reduction* - suppress a noise signal present in the image (not easy to distinguish noise from useful signal; a different method may be to capture the image more times and take the average on each pixel)

..Smoothing, sharpening, noise reduction

- the algorithms often require *local operations*, e.g., accessing all the pixels around the pixel to be updated

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

- Example: *mean*

$$x_4' = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}$$

  it is used as a smoothing technique

- a sequential code for mean requires 9 operations for each pixel; hence sequential time complexity is $O(n)$

Parallel mean computation

Horizontal directions:

1. each processor (i,j) receives the value $x_{i,j-1}$ from *left* and adds it to an accumulating sum (the original value $x_{i,j}$ is retained)
2. then it receives the original value $x_{i,j+1}$ from *right* and adds it to the accumulating sum;

Hence processor (i,j) holds $x_{i,j}+x_{i,j-1}+x_{i,j+1}$; replace original values by these sums and repeat 1,2 for the vertical directions

3. processor (i,j) receives $x_{i-1,j}+x_{i-1,j-1}+x_{i-1,j+1}$ from above and adds it (retaining its previous sum $x_{i,j}+x_{i,j-1}+x_{i,j+1}$)
4. then it receives the sum $x_{i+1,j}+x_{i+1,j-1}+x_{i+1,j+1}$ from below and adds it getting the final sum $x_{i-1,j}+x_{i-1,j-1}+x_{i-1,j+1}+x_{i,j}+x_{i,j-1}+x_{i,j+1}+x_{i+1,j}+x_{i+1,j-1}+x_{i+1,j+1}$

Median

- mean value method tends to blur edges or other sharp details
- an alternative method is to use the *median*:
  —order the values of the neighborhood pixels and
  —choose the center value (provided an odd number of cells are compared)
- for the described 3×3 structure $x_0, \ldots, x_8$, order the values as $v_0 \le v_1 \le \ldots \le v_8$ and choose v_4
- one may use bubble sort, but stop when the center value was found (after 5 steps); this requires $8+7+\ldots+4 = 30$ comparisons, hence $30n$ operations (for n pixels)

Parallel code / for median

—One may use a mash sorting algorithm, e.g., shear-sort;

—For greater speed an approximation method may be used:

- use compare-and-exchange to sort any row:
  Stage 1: $x_{i,j-1} \longleftrightarrow x_{i,j}$
  Stage 2: $x_{i,j} \longleftrightarrow x_{i,j+1}$
  Stage 3: $x_{i,j-1} \longleftrightarrow x_{i,j}$
- repeat for columns:
  Stage 1: $x_{i-1,j} \longleftrightarrow x_{i,j}$
  Stage 2: $x_{i,j} \longleftrightarrow x_{i+1,j}$
  Stage 3: $x_{i-1,j} \longleftrightarrow x_{i,j}$
- the value $x_{i,j}$ after these 6 steps may not be the median, but it is a good approximation

This is the basic code for cell (i,j); it interferes with the code for other cells.

Weighted masks

- mean method gives equal weights to all neighborhood pixels
- generally, a *weighted mask* may be used
- for our standard 3×3 structure $x_0, \ldots, x_8$ and weights $w_0, \ldots, w_8$, the new center pixel value is

$$x'_4 = \frac{w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8}{k}$$

- k is used to maintain a correct gray-scale balance; usually k is $\sum_{i=0}^{8} w_i$
- this operation may be seen as the "cross-correlation" of vectors x and w
- masks of other size may also be used; e.g., $5 \times 5, 9 \times 9$, etc.

..Weighted masks

Examples (of masks):

- $\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$ and $k = 9$ — may be used to compute mean

- $\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 8 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$ and $k = 16$ — a noise reduction mask

- $\begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$ and $k = 9$ — a sharpening filter mask

Edge detection

Edge detection:
—object identification often requires to find *edges*
—an "edge" is a *significant change* of the gray level intensity

Suppose a one variable function $f(x)$ is considered (e.g., corresponding to a row);

- if f is differentiable, then:
  —its derivative $\partial f/\partial x$ has a *spike* when f has a significant change
  —the polarity of this spike gives the sense of the changing: positive (resp. negative) $\Rightarrow$ increasing (resp. decreasing)
- if f is double differentiable, then its second derivative has a *zero* in the interval where f has such a significant change

..Edge detection

Suppose a (two-dimensional) image is considered; then the change in gray level have a

- *gradient magnitude* (number)

$$\nabla f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

- ... and a *gradient direction* (the angle with respect to the y-axis)

$$\phi(x,y) = tan^{-1}\left(\left(\frac{\partial f}{\partial y}\right) / \left(\frac{\partial f}{\partial x}\right)\right)$$

These formulas may be used to identify the edges in an image (the 1st one is approximated as $\nabla f = \left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial x}\right|$ *to simplify the computation)*

Edge detection masks

- as usual, for discrete functions derivatives are approximated by differences: for a 3×3 group, $\begin{array}{|c|c|c|}\hline x_0 & x_1 & x_2 \\ \hline x_3 & x_4 & x_5 \\ \hline x_6 & x_7 & x_8 \\ \hline \end{array}$ we have:

$$\frac{\partial f}{\partial x} \approx x_5 - x_3 \quad \text{and} \quad \frac{\partial f}{\partial y} \approx x_7 - x_1$$

hence

$$\bigtriangledown f = |x_5 - x_3| + |x_7 - x_1|$$

- to compute this,
—two masks may be used: one for $x_5 - x_3$, one for $x_7 - x_1$;
—then add the resulting absolute values
(the computation for each mask may be made in parallel)

Prewitt operator

Prewitt operator uses more points to approximate the gradient:

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + (x_5 - x_3) + (x_8 - x_6)$$

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + (x_7 - x_1) + (x_8 - x_2)$$

Then,

$$\nabla f = |x_2 - x_0 + x_5 - x_3 + x_8 - x_6| + |x_6 - x_0 + x_7 - x_1 + x_8 - x_2|$$

which, as above, requires two masks (one for each module)

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

Sobel operator

Sobel operator is a popular edge detection method; it uses a different approximation method for the gradient:

$$\frac{\partial f}{\partial x} \approx (x_2 - x_0) + 2(x_5 - x_3) + (x_8 - x_6)$$

$$\frac{\partial f}{\partial y} \approx (x_6 - x_0) + 2(x_7 - x_1) + (x_8 - x_2)$$

which, as above, requires two masks (one for each module)

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

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

Usually, the operators based of the 1st-order derivatives enhance noise*; but Sobel operator has also a* smoothing action.

Laplace operator

Laplace operator uses the 2nd-order derivatives

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

approximated to

$$\nabla^2 f = 4x_4 - (x_1 + x_3 + x_5 + x_7)$$

and computed using a single mask

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

Notice: We have studied this operator in other lectures, e.g., for heat distribution problem.

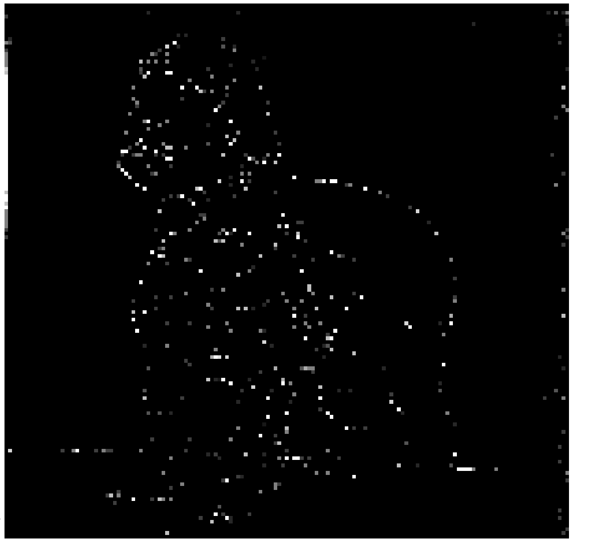
Edge detection

An original image

The effect of Sobel and Laplace operators

Sobel →

Laplace →

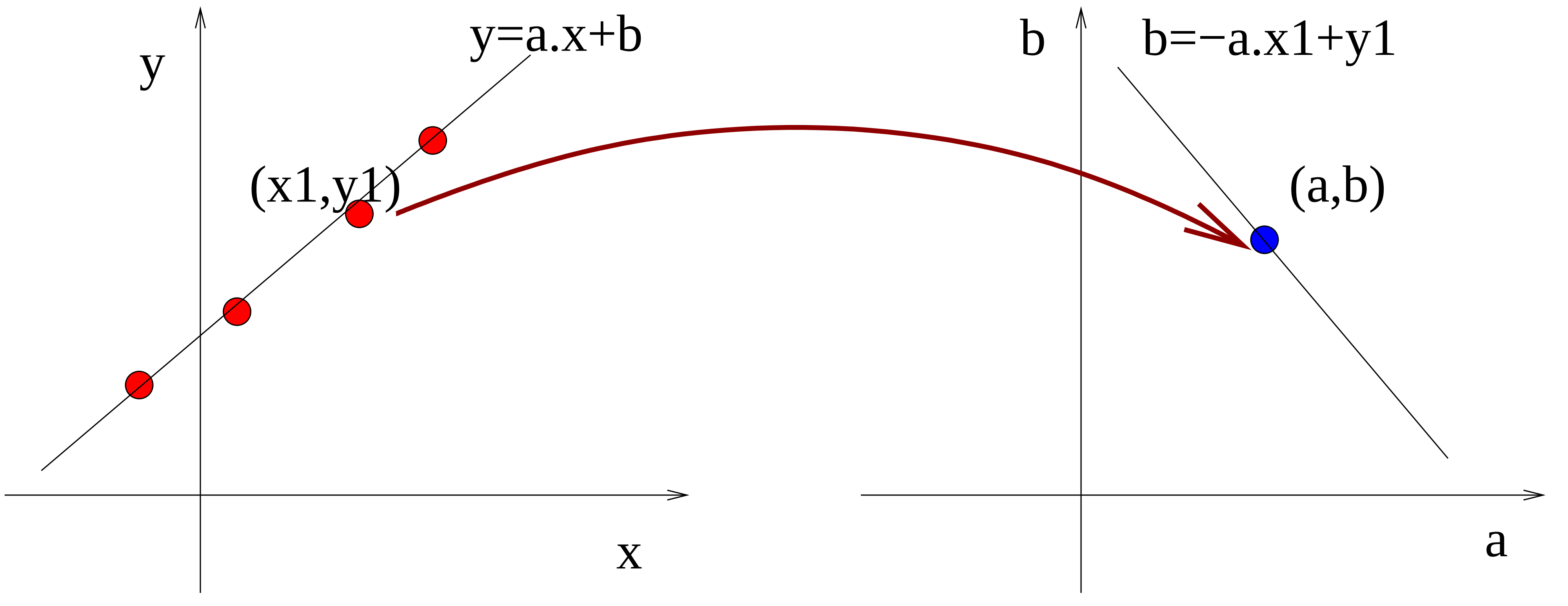


Hugh transform

Hugh transform:

- a method to *"find the parameters of equations of lines that most likely fit sets of pixels in an image"*
- may be used to fill in the gap between the points obtained in edge-detection result
- a line is described by the equation $y = ax + b$
- a direct search of the line including the largest number of pixels from a set of n pixels is expensive: $O(n^3)$ (for any 2 points check how many other points are on their line)
- rearrange the equation as $b = -xa + y$; then all points (x_i, y_i) on the line have the same associated (a, b) pair
- finally, count the number of points mapped to an (a, b) pair

..Hugh transform



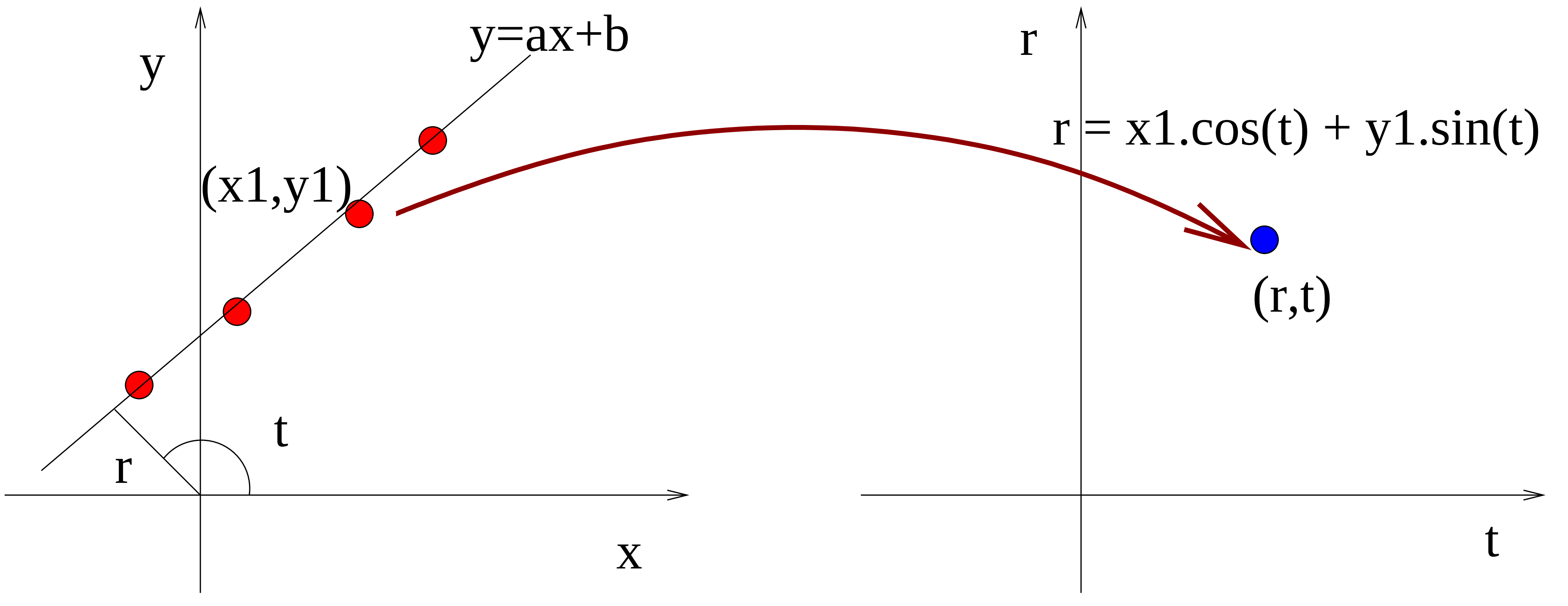
Find "most likely" lines

1st version: Cartesian coordinates

- a single originar point (x_1, y_1) is mapped into *all (a,b)-points* of the line
$$b = -x_1 \cdot a + y_1$$
- a discrete grid for pairs (a,b) is used; the computation is then rounded to the nearest possible (a,b) coordinate
- this is to be repeated for all given points (x_i, y_i)
- each pair (a,b) uses an accumulator to count the number of points mapped into it
- finally, the point (a,b) with the maximum value is chosen

Disadvantage: can not handle vertical lines.

..Hugh transform



..Find “most likely” lines

2nd version: Polar coordinates

- a single originar point (x_1, y_1) is mapped into *all* (r,θ)*-points* satisfying
$$r = x_1\ cos\theta + y_1\ sin\theta$$

- the rest of the procedure is as in the previous case:
—a grid of points (r,θ) is selected,
—the number of pixels mapped into each (r,θ)-point is counted, and
—the (r,θ)-point with the maximum value is selected

- it works well for all directions

Implementation

Implementation (using the standard image representation, i.e., origin = top/left corner):

- the parameter space is divided into small rectangular regions
- each region has an associated accumulator
- accumulators for the regions were a pixel maps into are incremented
- if k intervals are chosen for θ, then the complexity is $O(kn)$
- the complexity can be significantly reduced by limiting the range of lines for individual pixels using some criteria

..Implementation

Sequential code: (only one θ is used, based on the gradient function)

```
for (x=0; x<xmax; x++)
  for(y=0; y<ymax; y++) {
    sobel(&x,&y,dx,dx) /* find x,y gradients */
    magnitude = grad_max(dx,dy);
    if (magnitude > threshold) {
      theta = grad_dir(dx,dy) /* use atan() fn */
      theta = theta_quantize(theta);
      r = x * cos(theta) + y * sin(theta);
      r = r_quantize(r);
      acc[r][theta]++;
      append(r, theta, x, y); /* gather points */
    }
  }
```

..Implementation

- from the resulting matrix `acc[][]` the points `(r,theta)` realizing a (local) maximum are chosen (hence an optimization algorithm has to be used/implemented)

Parallel code:

- there is a lot of room for parallelization in the above sequential algorithm, e.g.:
  —the accumulators may be computed in parallel
  —they use the same image, hence a shared memory model may be selected
  —only read actions on the image are performed, hence no critical sections are necessary

Transformation into frequency domain

Fourier transform:

- very useful transformation, used in many areas of science and engineering
- in image processing it was successfully applied for *image enhancement*, *restoration*, and *compression*
- we start with the one-dimensional case:
  —a periodic function $x(t)$ (of time) can be decomposed into a series of sinusoidal waveforms of various frequencies and amplitudes;
  —for each frequency f one gets an associated value $X(f)$
- the resulting series is called *Fourier series*; the transform is called *Fourier transform*

Fourier series

Fourier series:

- a *Fourier series* is a summation of sine and cosine terms

$$x(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} \left(a_j cos \left(2\pi \frac{j}{T} t \right) + b_j sin \left(2\pi \frac{j}{T} t \right) \right)$$

- T is the period of $x(t)$; $1/T = f$ is the frequency
- a more convenient representation (using complex numbers) is

$$x(t) = \sum_{j=-\infty}^{\infty} X_j e^{2\pi i \frac{j}{T} t}$$

X_j is called the j-th Fourier coefficient; $i = \sqrt{-1}$

Fourier transform

Fourier transform (for continuous functions):

- (direct) *Fourier transform*: given a continuous function of time $x(t)$, the function on frequency $X(f)$, called the *spectrum* of $x(t)$, is defined by

$$X(f) = \sum_{-\infty}^{\infty} x(t) e^{-2\pi i f t} dt$$

- *inverse Fourier transform*: given a continuous function of frequency $X(f)$, the function on time $x(t)$ is defined by

$$x(t) = \sum_{-\infty}^{\infty} X(f) e^{2\pi i f t} df$$

- key result: direct and inverse Fourier transforms are mutually converse one to the other

Discrete Fourier transform

Fourier transform (for discrete functions): similar, but integrals are replaced by finite sums

- *discrete Fourier transform* (DFT):

$$X_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{-2\pi i \left(\frac{jk}{N}\right)}$$

- *inverse Fourier transform*:

$$x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j e^{2\pi i \left(\frac{jk}{N}\right)}$$

- $0 \leq k < N$; N (real) numbers $x_0, \ldots, x_{N-1}$ produce N (complex) numbers $X_0, \ldots, X_{N-1}$

The factor $1/N$ will be mostly omitted in the sequel; however, it has to be finally inserted to get a proper result

..Discrete Fourier transform

Example (16 points $x_0,\ldots,x_{15}$):

$$X_0 = \frac{1}{16}\left(x_0 e^{-2\pi i(0\frac{0}{16})} + x_1 e^{-2\pi i(1\frac{0}{16})} + \ldots + x_{15} e^{-2\pi i(15\frac{0}{16})}\right)$$

$$X_1 = \frac{1}{16}\left(x_0 e^{-2\pi i(0\frac{1}{16})} + x_1 e^{-2\pi i(1\frac{1}{16})} + \ldots + x_{15} e^{-2\pi i(15\frac{1}{16})}\right)$$

$$X_2 = \frac{1}{16}\left(x_0 e^{-2\pi i(0\frac{2}{16})} + x_1 e^{-2\pi i(1\frac{2}{16})} + \ldots + x_{15} e^{-2\pi i(15\frac{2}{16})}\right)$$

$$X_3 = \frac{1}{16}\left(x_0 e^{-2\pi i(0\frac{3}{16})} + x_1 e^{-2\pi i(1\frac{3}{16})} + \ldots + x_{15} e^{-2\pi i(15\frac{3}{16})}\right)$$

$$\vdots$$

$$X_{15} = \frac{1}{16}\left(x_0 e^{-2\pi i(0\frac{15}{16})} + x_1 e^{-2\pi i(1\frac{15}{16})} + \ldots + x_{15} e^{-2\pi i(15\frac{15}{16})}\right)$$

..Discrete Fourier transform

In a marticial form, this transformation may be written as follows (using $w = e^{-2\pi i(\frac{1}{16})}$):

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{15} \end{pmatrix} = \frac{1}{16} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & w & w^2 & w^3 & \dots & w^{15} \\ 1 & w^2 & w^4 & w^6 & \dots & w^{2\cdot 15} \\ 1 & w^3 & w^6 & w^9 & \dots & w^{3\cdot 15} \\ \vdots & & & & & \\ 1 & w^{15} & w^{15\cdot 2} & w^{15\cdot 3} & \dots & w^{15\cdot 15} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{15} \end{pmatrix}$$

Fourier transform in image processing

Two-dimensional Fourier transform (the factor $1/(NM)$ is omitted)

- a 2-dim DFT is

$$X_{lm} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left(\frac{jl}{N} + \frac{km}{M} \right)}$$

  where j (resp. k) is the row (resp. column) coordinate

- the formula may be rewritten as

$$X_{lm} = \sum_{j=0}^{N-1} \left[\sum_{k=0}^{M-1} x_{jk} e^{-2\pi i \left(\frac{km}{M} \right)} \right] e^{-2\pi i \left(\frac{jl}{N} \right)}$$

  showing that a 2-dim DFT may be obtained in two phases:
  —an (inner) 1-dim DFT operating on row, followed by
  —a 1-dim DFT operating on columns

..Fourier transform in image processing

Application of DFT in image processing:

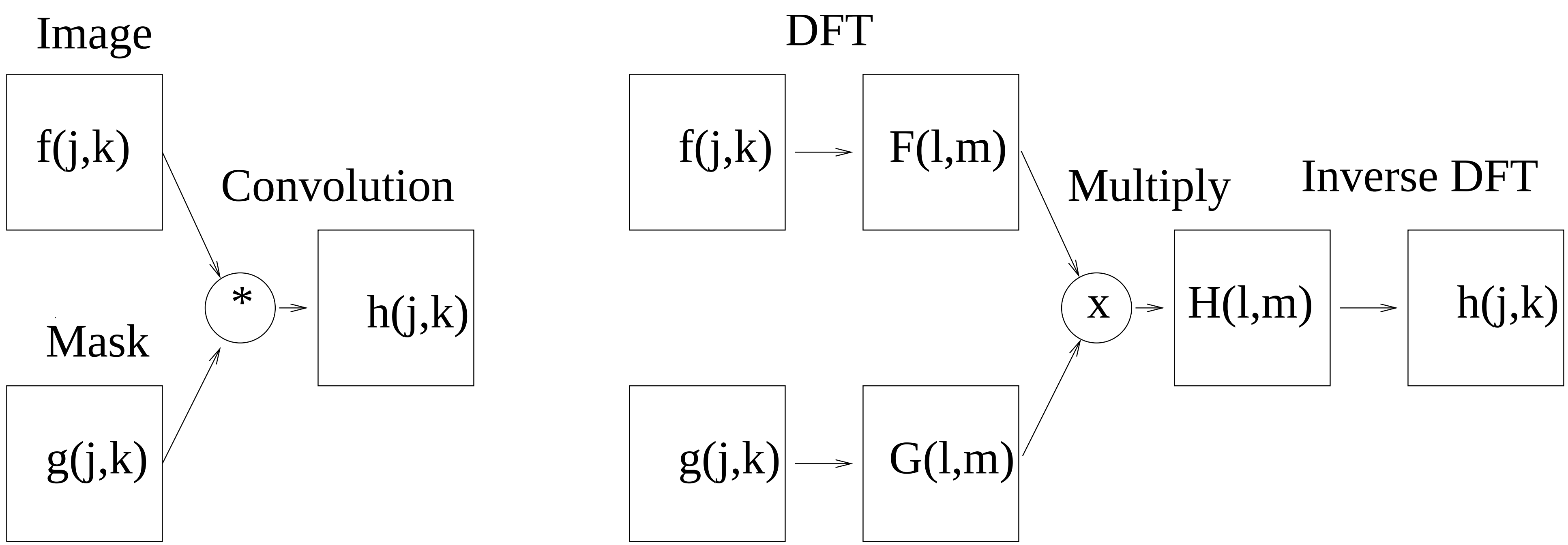
Frequency filtering is used for both smoothing and edge detection
—earlier we have used weighted masks for that purpose;
—but this is just a particular case of *convolution*

$$h(i,j) = g(j,k) * f(j,k)$$

where $g(j,k)$ describes the mask and $f(j,k)$ the image
—the convolution of functions corresponds to the product of their Fourier transform ("$\times$" denotes element-wise multiplication)

$$H(l,m) = G(l,m) \times F(l,m)$$

..Fourier transform in image processing



Sequential code for DFT

Sequential code: denote $w = e^{-2\pi i/N}$; then $X_k = \sum_{j=0}^{N-1} x_j (w^k)^j$

```
for (k=0; k<n; k++) {
  X[k] = 0;
  a = 1;
  for (j=0; j<N; j++) {
    X[k] = X[k] + a * x[j];
    a = a * w^k;
  }
}
```

Parallel DFT

Parallel implementation (direct approach):

- use a master/slave approach (one slave for computing each $X[k]$); with N processes this gives an $O(N)$ algorithm

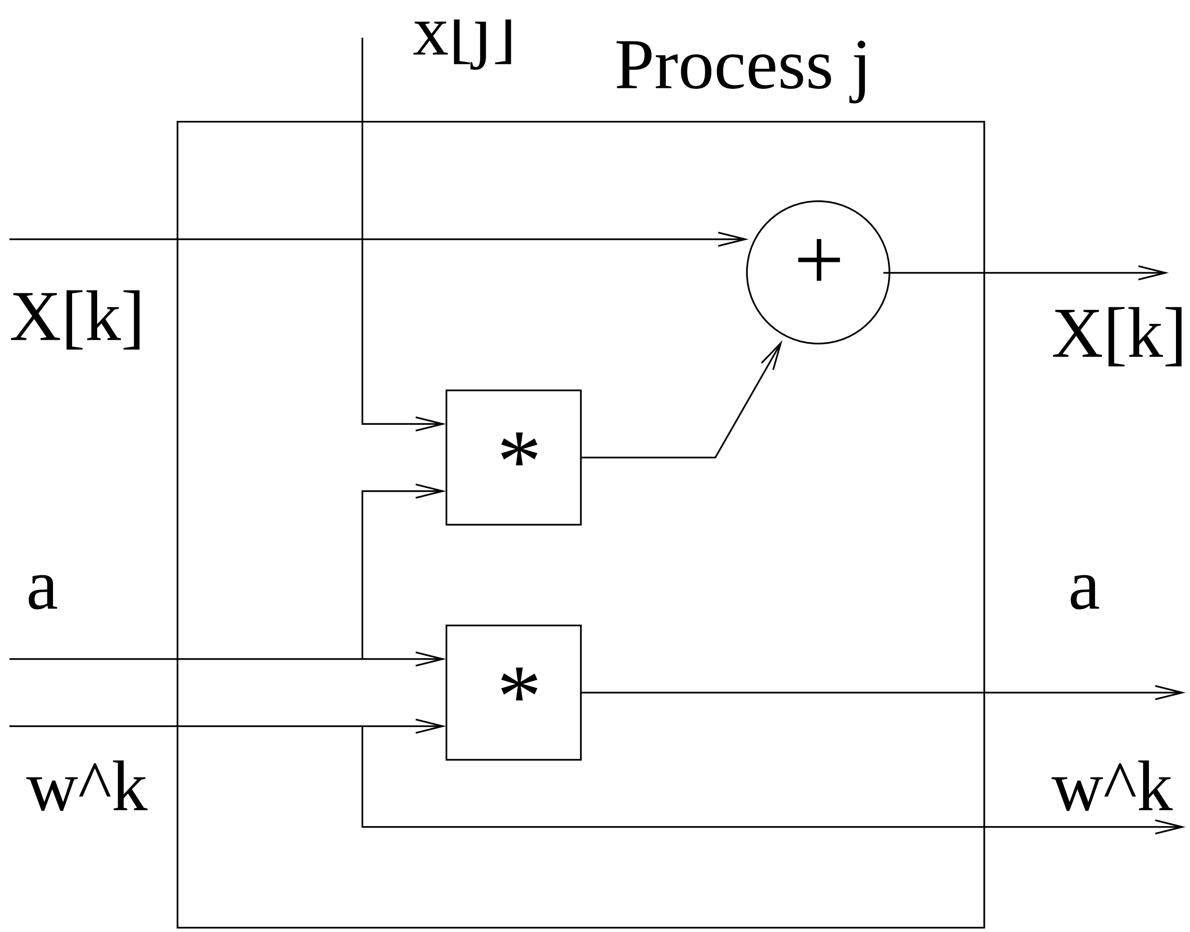
Pipeline implementation: Unfolding the inner loop we get

```
X[k] = 0;
a = 1;
X[k] = X[k] + a * x[0];
a = a * w^k
X[k] = X[k] + a * x[1];
a = a * w^k
⋮
```

..Parallel DFT

Pipeline implementation:

- one stage of the pipeline implementation is



Fast Fourier transform (FFT)

Fast Fourier transform: it's a method to reduce (sequential) time complexity form $O(N^2)$ to $O(N\,log(N))$

Based on a recursive procedure:

- start with

$$X_k = \frac{1}{N}\sum_{j=0}^{N-1} x_j w^{jk}$$

- divide the summation into two parts

$$X_k = \frac{1}{N}\left[\sum_{j=0}^{(N/2)-1} x_{2j} w^{2jk} + \sum_{j=0}^{(N/2)-1} x_{2j+1} w^{(2j+1)k}\right]$$

..FFT

- rewrite the sum as $X_k = (1/2)[X_{even} + w^k X_{odd}]$, namely

$$X_k = \frac{1}{2}\left[\frac{1}{N/2}\sum_{j=0}^{(N/2)-1} x_{2j}w^{2jk} + w^k \frac{1}{N/2}\sum_{j=0}^{(N/2)-1} x_{2j+1}w^{2jk}\right]$$

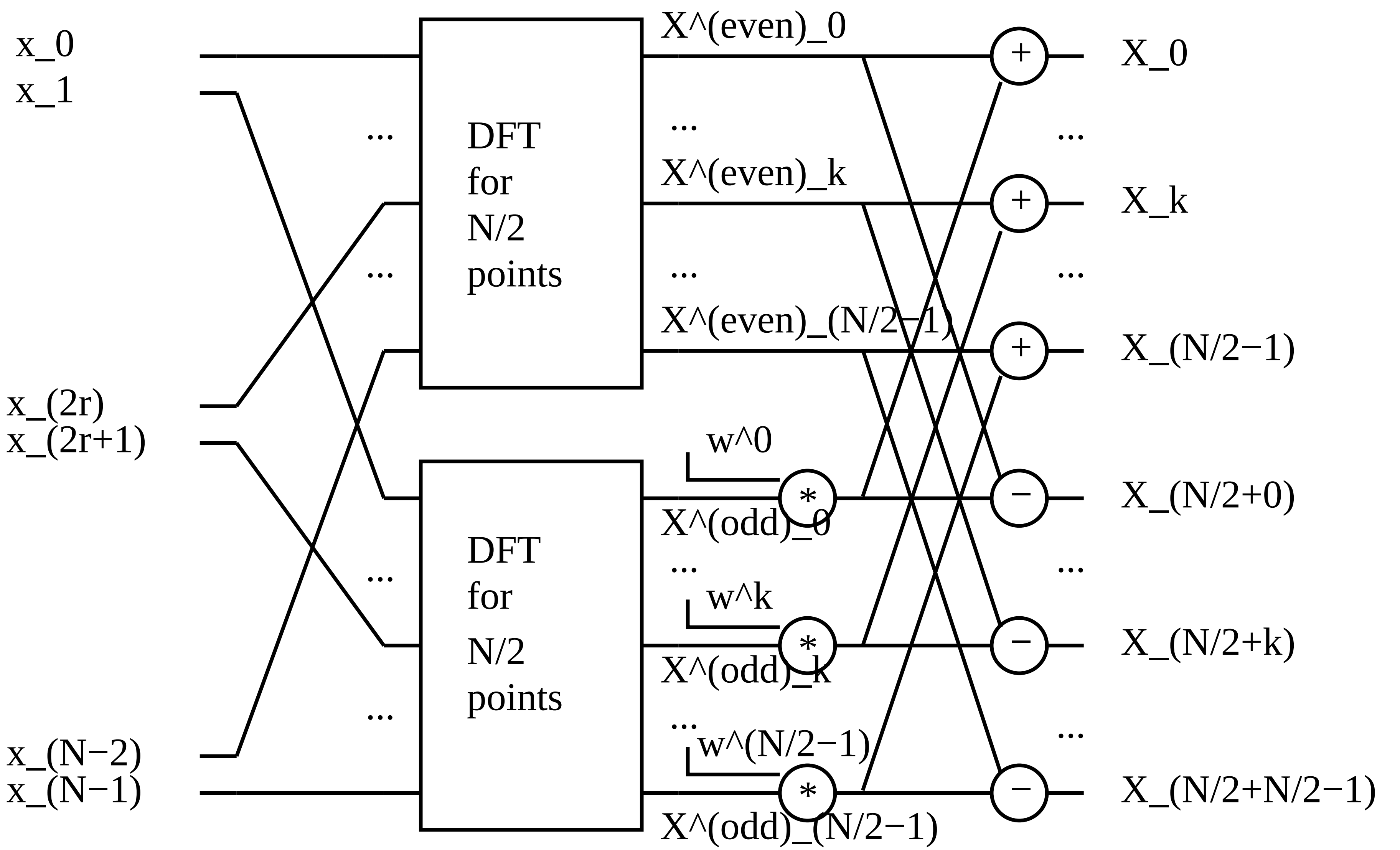
- notice that $w^{k+(N/2)} = e^{(-2\pi i/N)(k+(N/2))} = -e^{(-2\pi i/N)k} = -w^k$
  hence
  $$X_{k+(N/2)} = (1/2)[X_{even} - w^k X_{odd}]$$
  showing that X_k and $X_{k+(N/2)}$ could be computed using two transforms involving $N/2$ points

- the procedure can now be recursively applied

- a sequential code leads to an $O(N\,log(N))$ algorithm

..FFT

The recursive structure of *parallel FFT* is:



..FFT

Comments:

- notice that the terms w^k used in the figure are dependent on the number of points N, say w_N^k
- hence in the inner boxes of $N/2$ points different values have to be used $w_{N/2}^k$ for $k = 0, \ldots, N/2 - 1$
- ... but we can normalize them: $w_{N/2}^k = w_N^{2k}$

..Parallel FFT

Analysis:

- Computation: with p processes and N points, each process will compute N/p points; each points requires 2 operations; with $log(N)$ steps this gives $t_{comp} = 2(N/p)\,log(N) = O(N\,log(N))$
- Communication: if $p = N$, communication occurs at each step and one data is exchanges between pair of processors (a finer analysis may be give, depending on the network structure of the processors)