Chapter 11

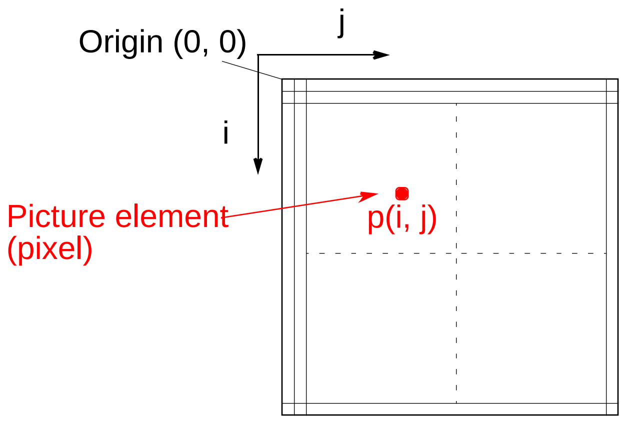
Image Processing

Application area chosen because it has very good parallelism and interesting output.

Low-level Image Processing

Operates directly on stored image to improve/enhance it.

Stored image consists of two-dimensional array of *pixels* (picture elements):



Many low-level image-processing operations assume monochrome images and refer to pixels as having *gray level* values or *intensities*.

Computational Requirements

Suppose a pixmap has 1024 × 1024 pixels and 8-bit pixels.

Storage requirement is 2^{20} bytes (1 Mbytes)

Suppose each pixel must be operated upon just once.

Then 2^{20} operations are needed in the time of one frame.

At 10^{-8} second/operation (10ns/operation), this would take 10 ms.

In real-time applications, the speed of computation must be at the frame rate (typically 60–85 frames/second).

All pixels in the image must be processed in the time of one frame; that is, in 12–16 ms.

Typically, many high-complexity operations must be performed, not just one operation.

Point Processing

Operations that produce output based upon value of a single pixel.

Thresholding

Pixels with values above predetermined threshold value kept and others below threshold reduced to 0. Given a pixel, x_i, operation on each pixel is

$$\text{if } (x_i < \text{threshold})\ x_i = 0;\ \text{else } x_i = 1;$$

Contrast Stretching

Range of gray level values extended to make details more visible. Given pixel of value x_i within range x_l and x_h, the contrast stretched to the range x_H to x_L by multiplying x_i by

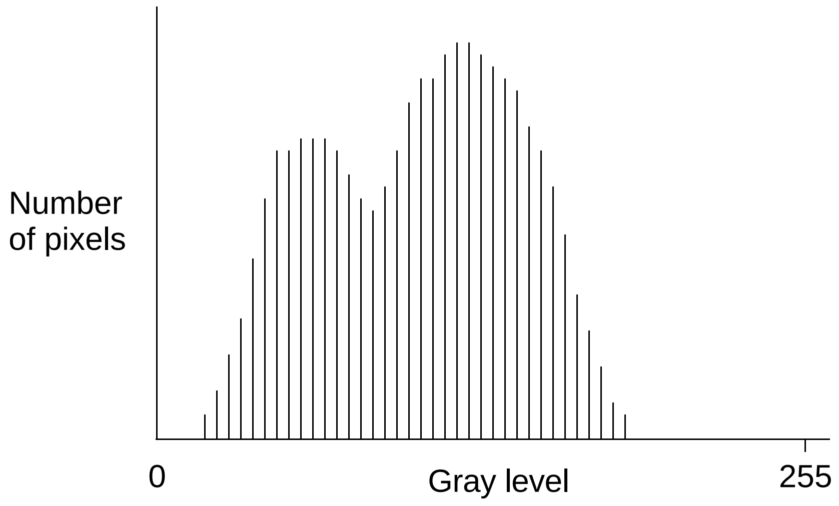
$$x_i = (x_i - x_l)\left(\frac{x_H - x_L}{x_h - x_l}\right) + x_L$$

Gray Level Reduction

Number of bits used to represent the gray level reduced. Simple method would be to truncate the lesser significant bits.

Histogram

Shows the number of pixels in the image at each gray level:



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;
```

where the pixels are contained in the array `p[][]` and `hist[k]` will hold the number of pixels having the *k*th gray level.

Similar to adding numbers to an accumulating sum and similar parallel solutions can be used for computing histograms.

Smoothing, Sharpening, and Noise Reduction

Smoothing suppresses large fluctuations in intensity over the image area and can be achieved by reducing the high-frequency content.

Sharpening accentuates the transitions, enhancing the detail, and can be achieved by two ways.

Noise reduction suppresses a noise signal present in the image.

Often requires a local operation with access to a group of pixels around the pixel to be updated. A common group size is 3 × 3:

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Mean

A simple smoothing technique is to take the *mean* or *average* of a group of pixels as the new value of the central pixel.

Given a 3 × 3 group, the computation is

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

where x_4' is the new value for x_4.

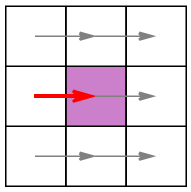
Sequential Code

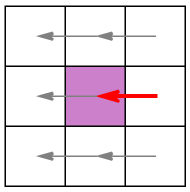
Nine steps to compute the average for each pixel, or 9*n* for *n* pixels.

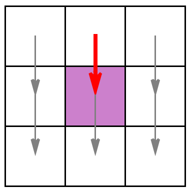
A sequential time complexity of O(*n*).

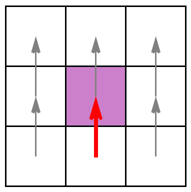
Parallel Code

Number of steps can be reduced by separating the computation into four data transfer steps in lock-step data-parallel fashion.

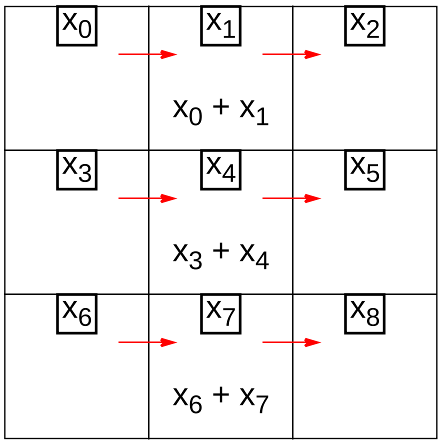

Step 1
Each pixel adds pixel from left


Step 2
Each pixel adds pixel from right

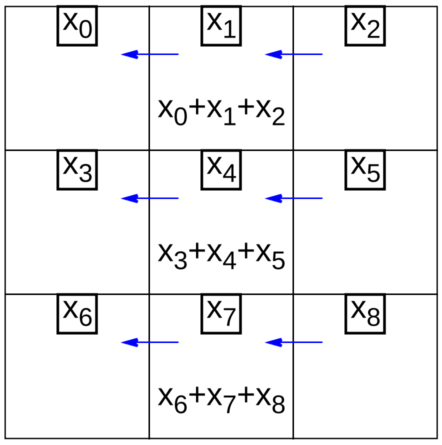

Step 3
Each pixel adds pixel from above


Step 4
Each pixel adds pixel from below

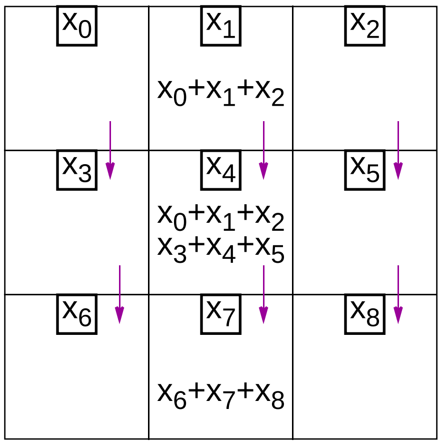
Parallel Mean Data Accumulation



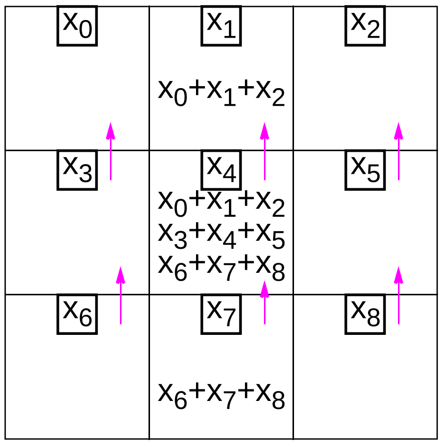
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

Median

Sequential Code

Median can be found by ordering pixel values from smallest to largest and choosing center pixel value (assuming an odd number of pixels).

With a 3 × 3 group, suppose values in ascending order are y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, and y_8. The median is y_4.

Suggests that all the values must first be sorted, and then fifth element taken.

Using bubble sort, in which the lesser values found first in order, sorting could, in fact, be terminated after fifth lowest value obtained.

Number of stepsgiven by 8 + 7 + 6 + 5 + 4 = 30 steps, or 30*n* for *n* pixels.

Parallel Code

An Approximate Sorting Algorithm

First, a compare-and-exchange operationperformed on each of the rows, requiring three steps. For the *i*th row, we have

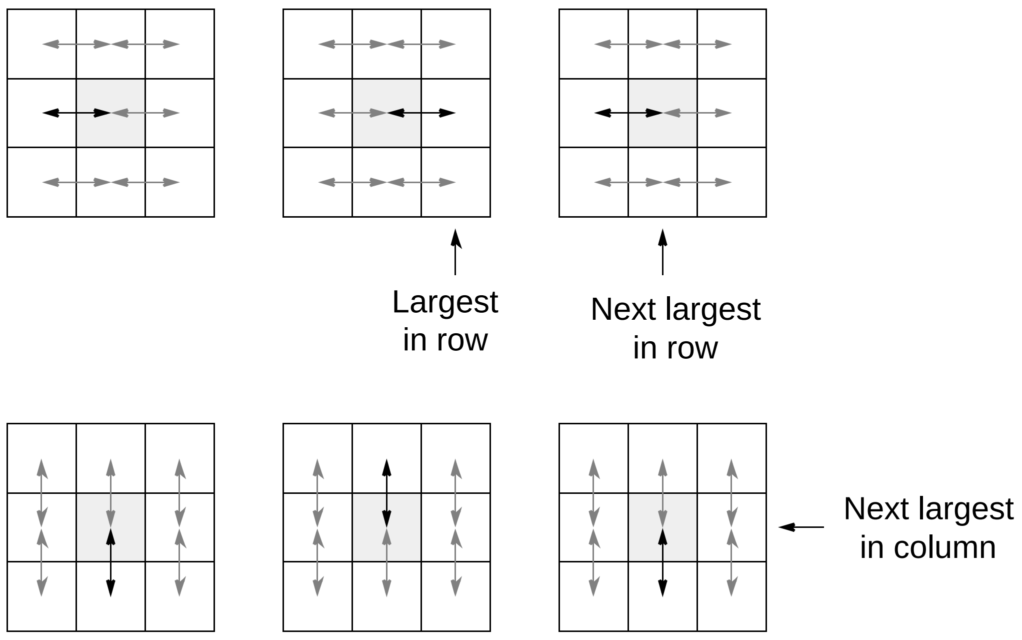
$$p_{i,j-1} \leftrightarrow p_{i,j}$$
$$p_{i,j} \leftrightarrow p_{i,j+1}$$
$$p_{i,j-1} \leftrightarrow p_{i,j}$$

where $\leftrightarrow$ means “compare and exchange if left gray level greater than right gray level”. Then done on columns:

$$p_{i-1,j} \leftrightarrow p_{i,j}$$
$$p_{i,j} \leftrightarrow p_{i+1,j}$$
$$p_{i-1,j} \leftrightarrow p_{i,j}$$

Value in $p_{i,j}$ taken to be fifth largest pixel value. Does not always select fifth largest value. Reasonable approximation. Six steps.

Approximate median algorithm requiring six steps



Weighted Masks

The mean method could be described by a weighted 3 × 3 mask.

Suppose the weights are w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7, and w_8, and pixel values are x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, and x_8.

The new center pixel value, x_4', is given by

$$x_4' = \frac{w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 + w_7x_7 + w_8x_8}{k}$$

Scale factor, $1/k$, set to maintain correct grayscale balance.

Often, k is given by $w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7$.

Using a 3 × 3 Weighted Mask

$$\begin{array}{c}\text{Mask}\\ \begin{array}{|c|c|c|}\hline w_0 & w_1 & w_2\\ \hline w_3 & w_4 & w_5\\ \hline w_6 & w_7 & w_8\\ \hline\end{array}\end{array}\quad \begin{array}{c}\text{Pixels}\\ \begin{array}{|c|c|c|}\hline x_0 & x_1 & x_2\\ \hline x_3 & x_4 & x_5\\ \hline x_6 & x_7 & x_8\\ \hline\end{array}\end{array}\quad = \quad \begin{array}{c}\text{Result}\\ \begin{array}{|c|c|c|}\hline & & \\ \hline & x_4' & \\ \hline & & \\ \hline\end{array}\end{array}$$

The summation of products, $w_i x_i$, from two functions w and x is the (discrete) *cross-correlation* of f with w (written as f w).

Mask to compute mean

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

k = 9

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 1\\ \hline 1 & 1 & 1\\ \hline 1 & 1 & 1\\ \hline\end{array}$$

A noise reduction mask

$$x_4' = \frac{8x_4 + x_0 + x_1 + x_2 + x_3 + x_5 + x_6 + x_7 + x_8}{16}$$

k =16

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 1\\ \hline 1 & 8 & 1\\ \hline 1 & 1 & 1\\ \hline\end{array}$$

High-pass sharpening filter mask

$$x_4' = \frac{8x_4 - x_0 - x_1 - x_2 - x_3 - x_5 - x_6 - x_7 - x_8}{9}$$

k = 9

$$\begin{array}{|c|c|c|}\hline -1 & -1 & -1\\ \hline -1 & 8 & -1\\ \hline -1 & -1 & -1\\ \hline\end{array}$$

Edge Detection

Highlighting edges of object where an edge is a significant change in gray level intensity.

Gradient and Magnitude

With a one-dimension gray level function, $f(x)$, first derivative, $\partial f/\partial x$, measures the gradient..

Edge recognized by a positive-going or negative-going spike at a transition.

Edge Detection using Differentiation

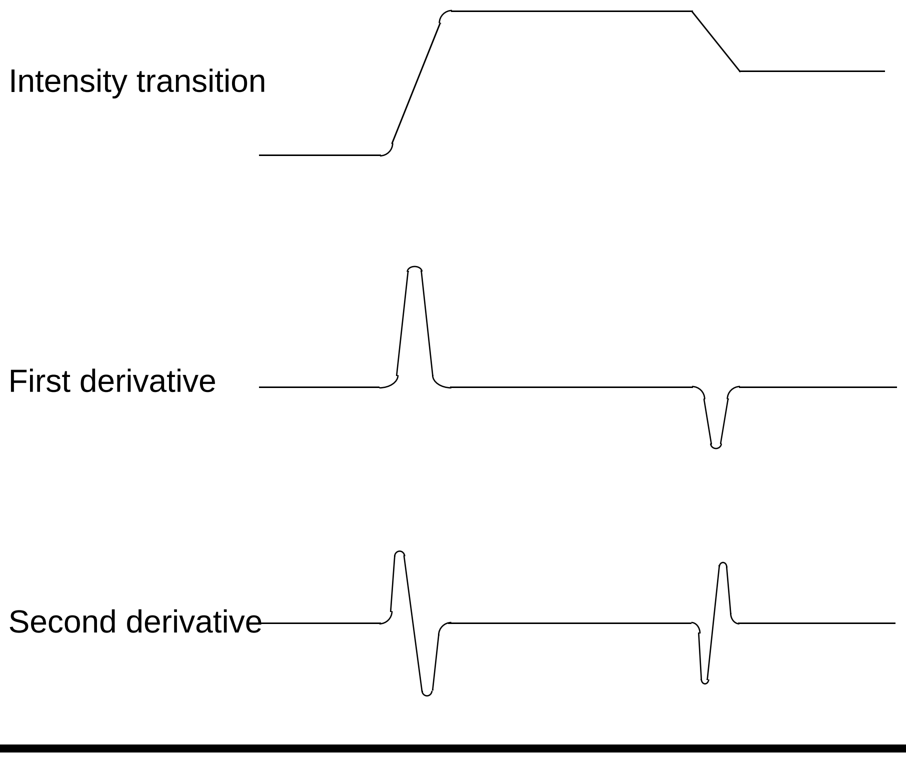


Image Function

A two-dimensional discretized gray level function, $f(x,y)$.

Gradient (magnitude)

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

Gradient Direction

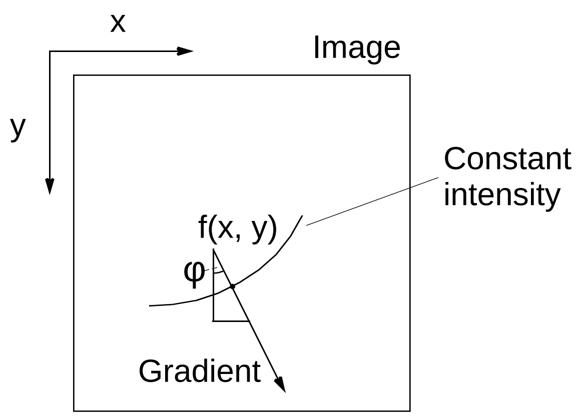
$$\phi(x, y) = \tan^{-1}\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right)$$

where φ is the angle with respect to the *y*-axis.
Gradient can be approximated to

$$\nabla f \approx \left|\frac{\partial f}{\partial y}\right| + \left|\frac{\partial f}{\partial x}\right|$$

for reduced computational effort

Gray Level Gradient and Direction



Edge Detection of Image Function

Image is a discrete two-dimensional function.

Derivative approximated by differences:

$\partial f/\partial x$ is difference in x-direction

$\partial f/\partial y$ is difference in y-direction

Edge Detection Masks

Might consider computing the approximate gradient using x_5 and x_3 (to get $\partial f/\partial x$) and x_7 and x_1 (to get $\partial f/\partial y$); i.e.,

$$\frac{\partial f}{\partial x} \approx x_5 - x_3$$

$$\frac{\partial f}{\partial y} \approx x_7 - x_1$$

so that

$$f \approx |x_7 - x_1| + |x_5 - x_3|$$

Two masks needed, one to obtain $x_7 - x_1$ and one to obtain $x_5 - x_3$.

The absolute values of results of each mask added together.

0	−1	0
0	0	0
0	1	0

0	0	0
−1	0	1
0	0	0

Prewitt Operator

The approximate gradient obtained from

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

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

Then

$$f \approx \left|x_6 - x_0 + x_7 - x_1 + x_8 - x_2\right| + \left|x_2 - x_0 + x_5 - x_3 + x_8 - x_6\right|$$

which requires using the two 3 × 3 masks.

Prewitt operator

−1	−1	−1
0	0	0
1	1	1

−1	0	1
−1	0	1
−1	0	1

Sobel Operator

Derivatives are approximated to

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

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

Operators implementing first derivatives will tend to enhance noise.

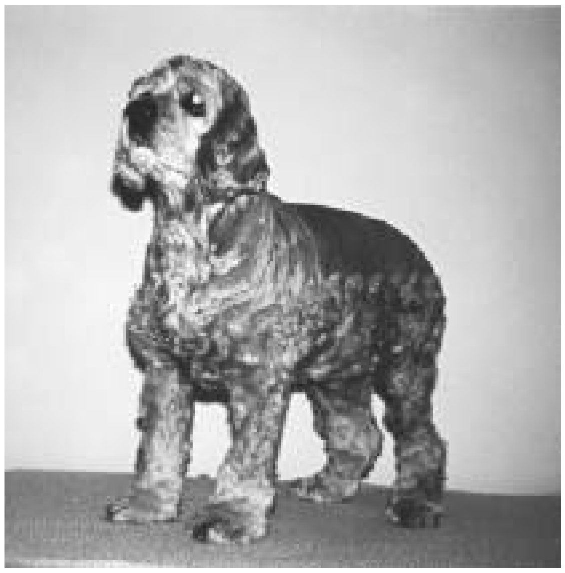
However, the Sobel operator also has a smoothing action.

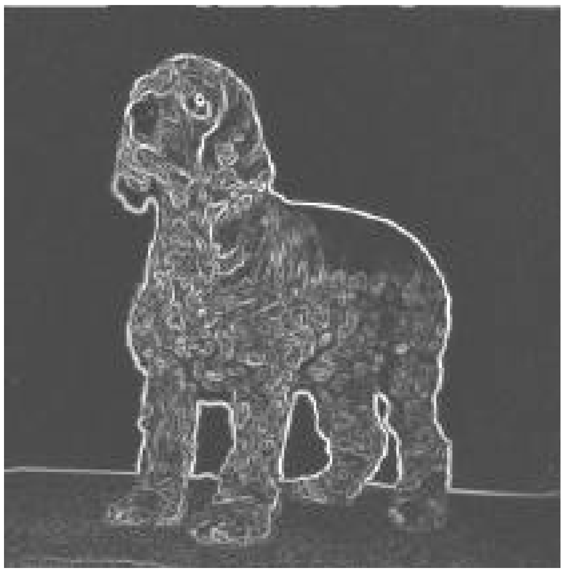
Sobel Operator

−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

Edge Detection with Sobel Operator


(a) Original image (Annabel)


(b) Effect of Sobel operator

Laplace Operator

The Laplace second-order derivative is defined as

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

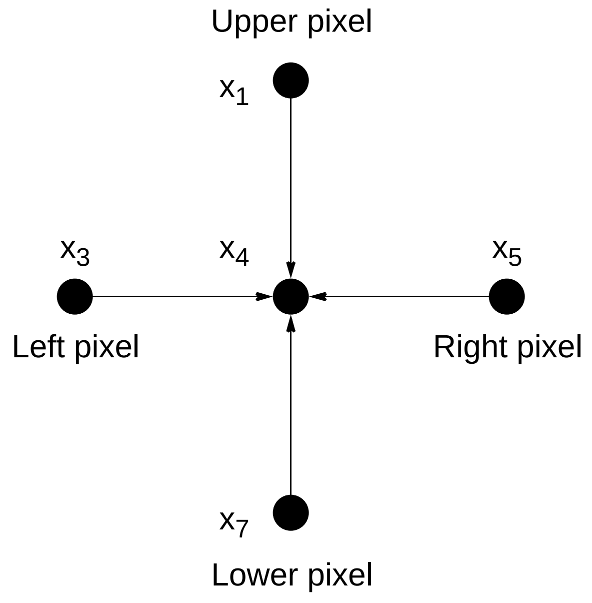
approximated to

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

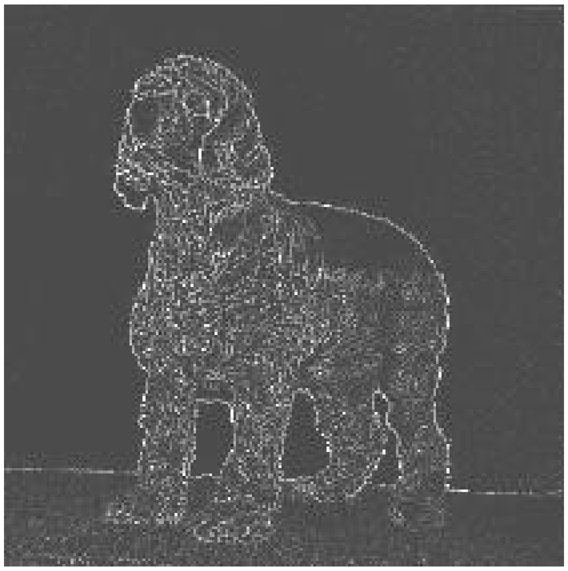
which can be obtained with the single mask:

0	−1	0
−1	4	−1
0	−1	0

Pixels used in Laplace operator



Effect of Laplace operator



Hough Transform

Purpose is to find the parameters of equations of lines that most likely fit sets of pixels in an image.

A line is described by the equation

$$y = ax + b$$

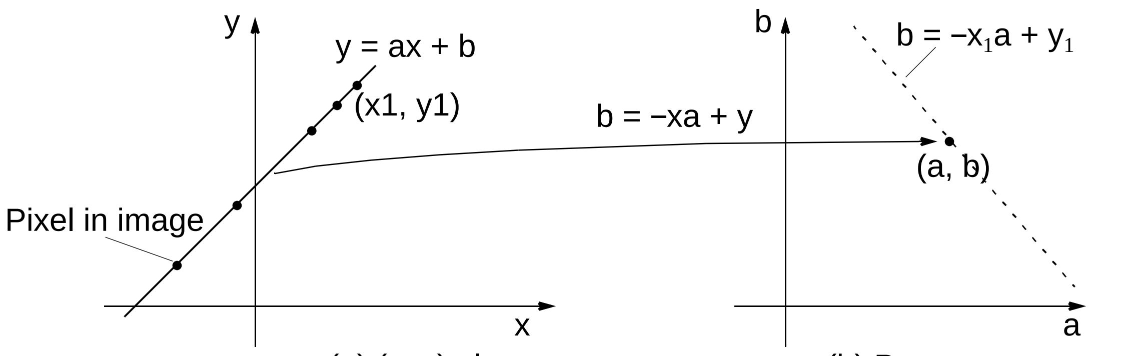
where the parameters, *a* and *b*, uniquely describe the particular line, *a* the slope and *b* the intercept on the *y*-axis.

A search for those lines with the most pixels mapped onto them would be computationally prohibitively expensive [$O(n^3)$].

Suppose the equation of the line is rearranged as:

$$b = -xa + y$$

Every point that lies on a specific line in the *x*-*y* space will map into same point in the a-b space (parameter space).



(a) (x, y) plane (b) Parameter space

Finding the Most Likely Lines

In the mapping process, discrete values will be used to a coarse prescribed precision and the computation is rounded to the nearest possible *a*-*b* coordinates.

The mapping process is done for every point in the *x*-*y* space.

A record is kept of those *a*-*b* points that have been obtained by incrementing the corresponding accumulator.

Each accumulator will have the number of pixels that map into a single point in the parameter space.

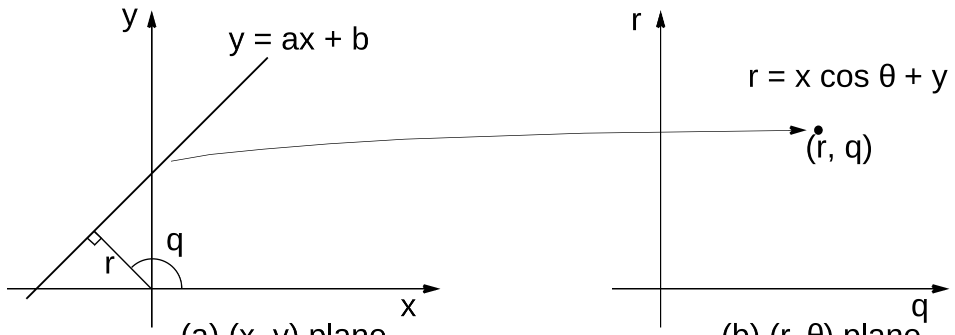
The points in the parameter space with locally maximum numbers of pixels are chosen as lines.

Unfortunately, this method will fail for vertical lines (i.e., with the slope, *a*, infinite and with the *y* intercept, *b*, infinite) and with lines that approach this extreme.

To avoid the problem, line equation rearranged to polar coordinates:

$$r = x \cos \theta + y \sin \theta$$

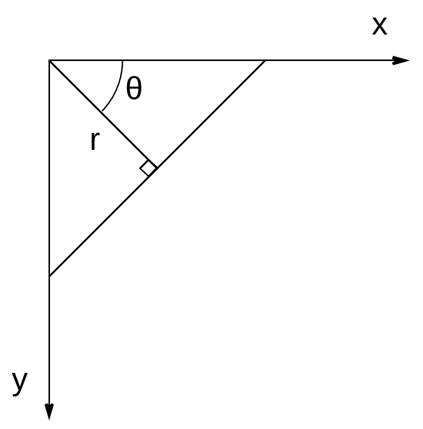
where r is the perpendicular distance to the origin in the original (x, y) coordinate system and θ is the angle between r and the x-axis. θ very conveniently the gradient angle of line (with respect to x-axis).



(a) (x, y) plane (b) (r, θ) plane

Implementation

Assume origin at the top left corner.



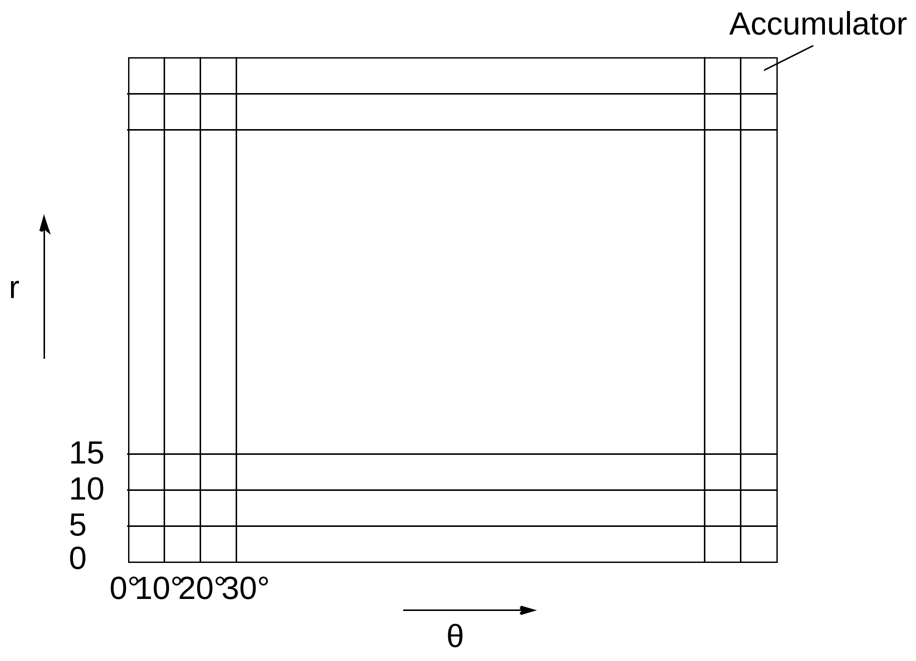
The parameter space divided into small rectangular regions. One accumulator for each region.

Accumulators of those regions that a pixel maps into incremented. Process done for all pixels in image.

If all values of θ were tried (i.e., incrementing θ through all its values), computational effort would be given by the number of discrete values of θ, say *k* intervals. With *n* pixels the complexity is O(*kn*).

Computational effort can be reduced significantly by limiting range of lines for individual pixels using some criteria. A single value of θ could be selected based upon the gradient of the line.

Accumulators, acc[*r*][θ], for Hough Transform



Sequential Code

Sequential code could be of the form

```
for (x = 0; x < xmax; x++)                /* for each pixel */
    for (y = 0; y < ymax; y++) {
      sobel(&x, &y, dx, dy);            /* find x and y gradients */
      magnitude = grad_mag(dx, dy);     /* find magnitude if needed */
      if (magnitude > threshold) {
        theta = grad_dir(dx, dy);       /* atan2() fn */
        theta = theta_quantize(theta);
        r = x * cos(theta) + y * sin(theta);
        r = r_quantize(r);
        acc[r][theta]++;                /* increment accumulator */
        append(r, theta, x, y);         /* append point to line */
      }
    }
```

Parallel Code

Since the computation for each accumulator is independent of the other accumulations, it could be performed simultaneously, although each requires read access to the whole image.

Left as an exercise.

Transformation into the Frequency Domain

Fourier Transform

Many applications in science and engineering. In image processing, Fourier transform used for image enhancement, restoration, and compression.

Image is a two-dimensional discretized function, $f(x, y)$, but first start with one-dimensional case.

For completeness, let us first review results of Fourier series and Fourier transform concepts from first principles.

Fourier Series

The Fourier series is a summation of sine and cosine terms:

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

T is the period ($1/T = f$, where f is a frequency).

By some mathematical manipulation:

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

where X_j is the jth Fourier coefficient in a complex form and $i = \sqrt{-1}$.
(Fourier coefficients can also be computed from specific integrals.)

Fourier Transform

Continuous Functions

The previous summation developed into an integral:

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

where $X(f)$ is a continuous function of frequency.

The function $X(f)$ can be obtained from

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

$X(f)$ is the *spectrum* of $x(t)$, or the *Fourier transform of $x(t)$*.

The original function, $x(t)$, can obtained from $X(f)$ using the first integral given, which is the *inverse Fourier transform*..

Discrete Functions

For functions having a set of N discrete values. Replace integral with summation, leading to the *discrete Fourier transform* (DFT):

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

and *inverse discrete Fourier transform* given by

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

for $0 \le k \le N - 1$. The N (real) input values, x_0, x_1, x_2, …, x_{N-1}, produce N (complex) transform values, X_0, X_1, X_2, …, X_{N-1}.

Fourier Transforms in Image Processing

A two-dimensional Fourier transform 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 $0 \le j \le N - 1$ and $0 \le k \le M - 1$.

Assume image is square, where $N = M$.

Equation can be rearranged into

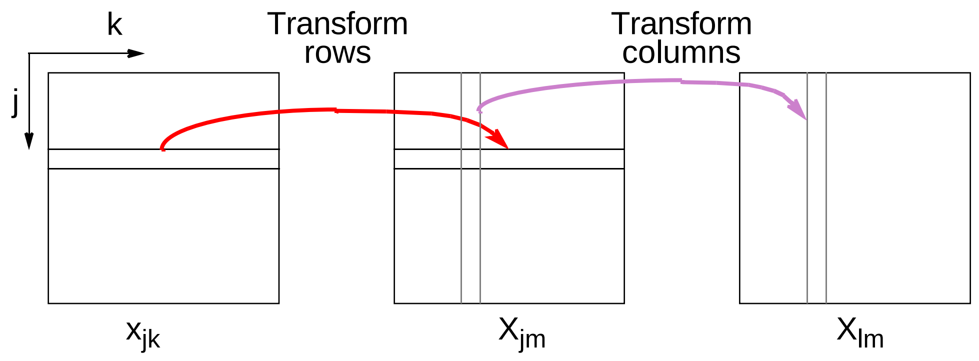
$$X_{lm} = \sum_{j=0}^{N-1}\left[\sum_{k=0}^{N-1} x_{jk} e^{-2\pi i\left(\frac{km}{N}\right)}\right] e^{-2\pi i\left(\frac{jl}{N}\right)}$$

Inner summation a one-dimensional DFT operating on *N* points of a row to produce a transformed row. Outer summation a one-dimensional DFT operating on *N* points of a column.

Can be divided into two sequential phases, one operating on rows of elements and one operating on columns:

$$X_{lm} = \sum_{j=0}^{N-1} X_{jm} e^{-2\pi i\left(\frac{jl}{N}\right)}$$

Two-Dimensional DFT



Applications

Frequency filtering can be described by the *convolution* operation:

$$h(j, k) = g(j, k) \quad f(j, k)$$

where $g(j, k)$ describes weighted mask (filter) and $f(j, k)$ the image.

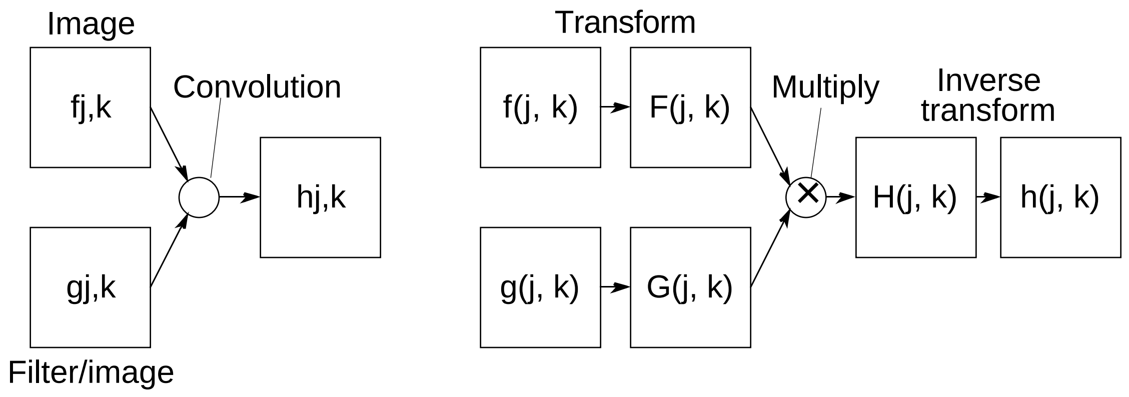
The Fourier transform of a product of functions is given by the convolution of the transforms of the individual functions.

Hence, convolution of two functions obtained by taking the Fourier transforms of each function, multiplying the transforms

$$H(j, k) = G(j, k) \times F(j, k)$$

(element by element multiplication), where $F(j, k)$ is the Fourier transform of $f(j, k)$ and $G(j, k)$ is the Fourier transform of $g(j, k)$, and then taking the inverse transform to return result into spatial domain.

Convolution using Fourier Transforms



(a) Direct convolution (b) Using Fourier transform

Parallelizing the Discrete Fourier Transform

Starting from

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

and using the notation $w = e^{-2\pi i/N}$,

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

w terms called *twiddle factors*. Each input multiplied by twiddle factor.

Inverse transform can be obtained by replacing *w* with w^{-1}.

Sequential Code

```
for (k = 0; k < N; k++) {    /* for every point */
  X[k] = 0;
  for (j = 0; j < N; j++)    /* compute summation */
    X[k] = X[k] + w^(j * k) * x[j];
}
```

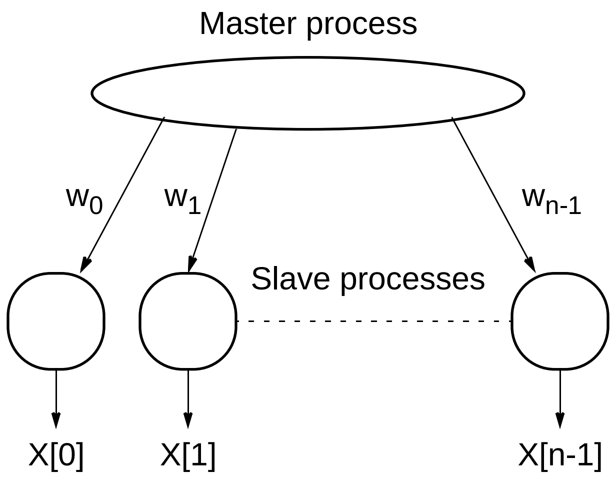
X[k] is *k*th transformed point, **x[k]** is *k*th input, **w = e^(-2pi/N)**. Summation requires complex number arithmetic. Can be rewritten:

```
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;
  }
}
```

where **a** is a temporary variable.

Elementary Master-Slave Implementation

One slave process of *N* slave processes assigned to produce one transformed value; i.e., *k*th slave process produces **`X[k]`**. Parallel time complexity with *N* (slave) processes is O(*N*).



Pipeline Implementation

Unfolding the inner loop for `X[k]`, we have

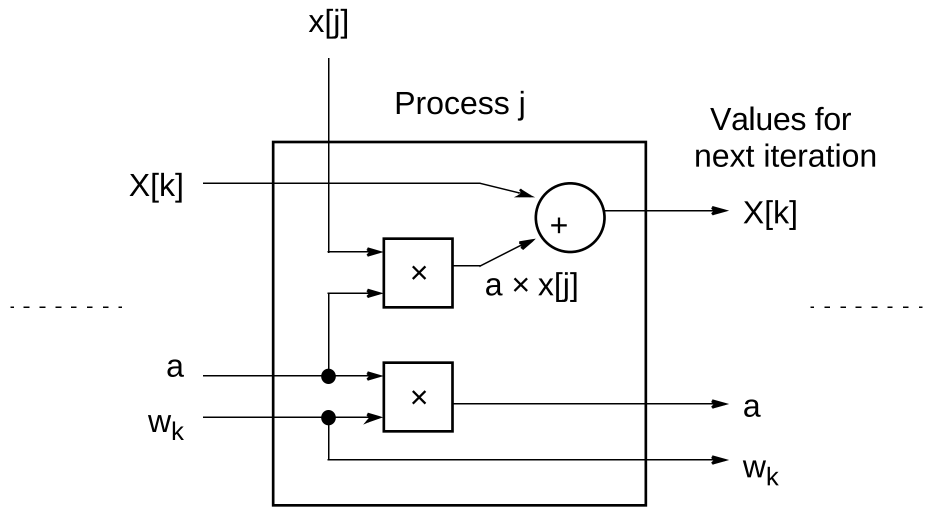
```
X[k] = 0;
a = 1;
X[k] = X[k] + a * x[0];
a = a * wk;
X[k] = X[k] + a * x[1];
a = a * wk;
X[k] = X[k] + a * x[2];
a = a * wk;
X[k] = X[k] + a * x[3];
a = a * wk;
  .
```

Each pair of statements

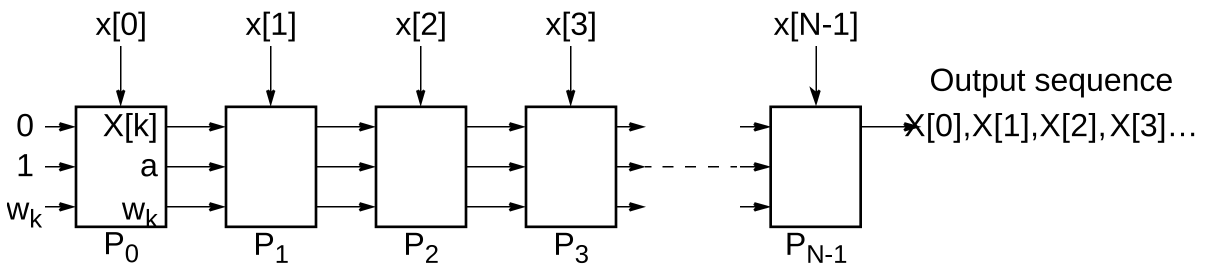
```
X[k] = X[k] + a * x[0];
a = a * wk;
```

could be performed by a separate pipeline stage.

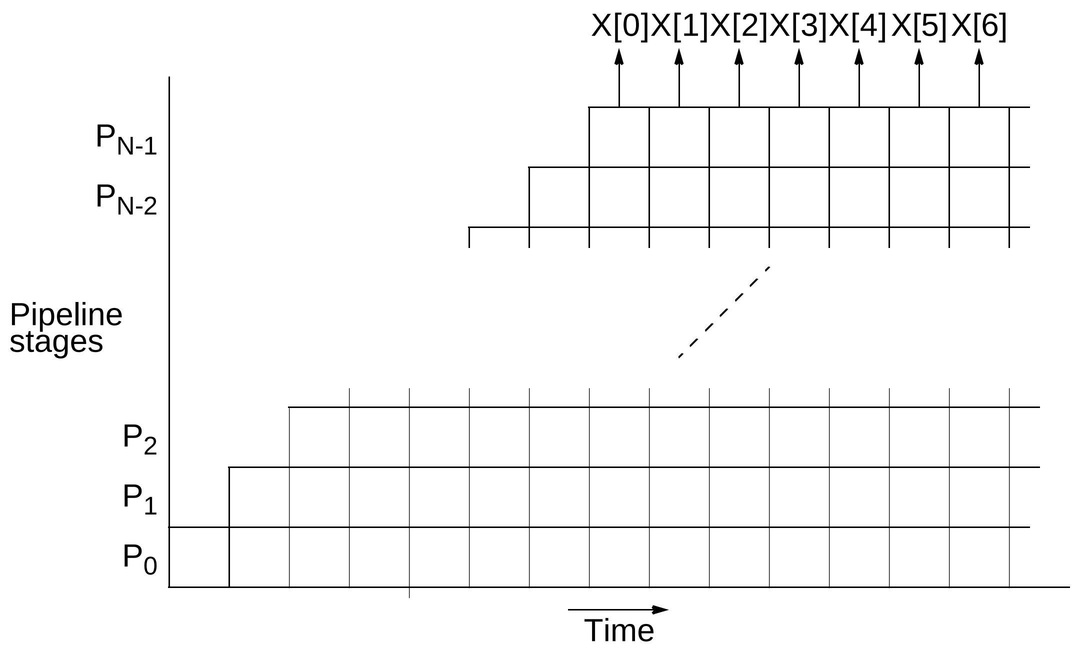
One stage of a pipeline implementation of DFT algorithm



Discrete Fourier transform with a pipeline



Timing diagram



DFT as a Matrix-Vector Product

The *k*th element of discrete Fourier transform given by

$$X_k = x_0w^0 + x_1w^1 + x_2w^2 + x_3w^3 + \dots x_{N-1}w^{N-1}$$

Whole transform can be described by a matrix-vector product:

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ . \\ X_k \\ . \\ X_{N-1} \end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & 1 & .\,. & 1 \\ 1 & w & w^2 & w^3 & .\,. & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & .\,. & w^{2(N-1)} \\ 1 & w^3 & w^6 & w^9 & .\,. & w^{3(N-1)} \\ . & . & . & . & .\,. & . \\ 1 & w^k & w^{2k} & w^{3k} & .\,. & w^{(N-1)k} \\ . & . & . & . & .\,. & . \\ 1 & w^{N-1} & w^{2(N-1)} & w^{3(N-1)} & .\,. & w^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ . \\ x_k \\ . \\ x_{N-1} \end{bmatrix}$$

(Note $w^0 = 1$.) Hence, parallel methods for matrix-vector product as described in Ch. 10 can be used for discrete Fourier transform.

Fast Fourier Transform

Method of obtaining discrete Fourier transform with a time complexity of O(N log N) instead of O(N^2).

Let us start with the discrete Fourier transform equation:

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

where $w = e^{-2\pi i/N}$.

Each summation an $N/2$ discrete Fourier transform operating on $N/2$ even points and $N/2$ odd points, respectively.

$$X_k = \frac{1}{2}\left[X_{\text{even}} + w^k X_{\text{odd}}\right]$$

for $k = 0, 1, \ldots N - 1$, where X_{even} is the $N/2$-point DFT of the numbers with even indices, $x_0, x_2, x_4, \ldots$, and X_{odd} is the $N/2$-point DFT of the numbers with odd indices, $x_1, x_3, x_5, \ldots$.

Now, suppose k is limited to 0, 1, … $N/2 - 1$, the first $N/2$ values of the total N values. Complete sequence divided into two parts:

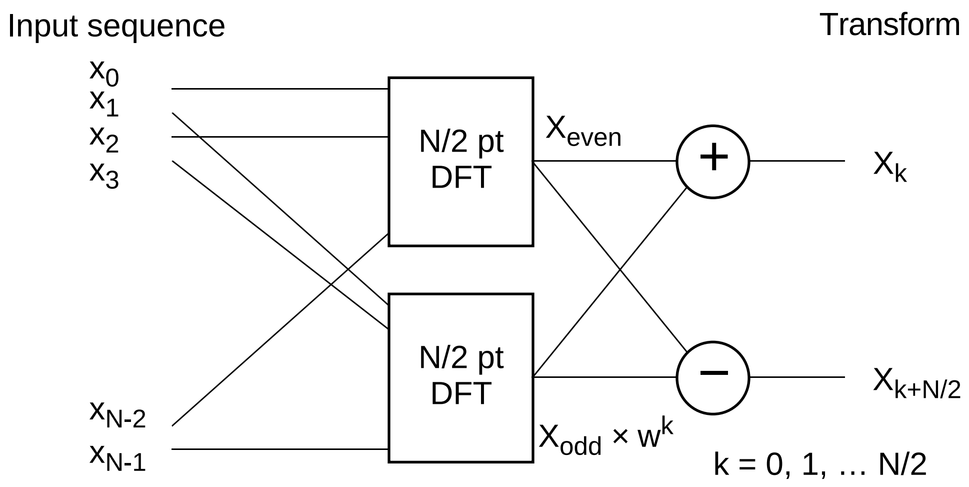
$$X_k = \frac{1}{2}\left[X_{even} + w^k X_{odd}\right]$$

and

$$X_{k+N/2} = \frac{1}{2}\left[X_{even} + w^{k+N/2} X_{odd}\right] = \frac{1}{2}\left[X_{even} - w^k X_{odd}\right]$$

since $w^{k+N/2} = -w^k$, where $0 \leq k < N/2$. Hence, we could compute X_k and $X_{k+N/2}$ using two $N/2$-point transforms:

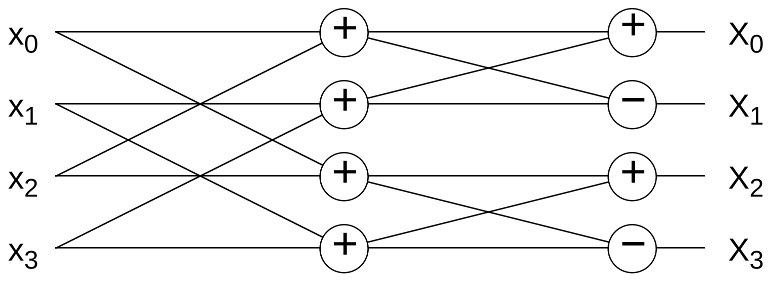
Decomposition of *N*-point DFT into two *N*/2-point DFTs



Each of the *N*/2-point DFTs can be decomposed into two *N*/4-point DFTs and the decomposition could be continued until single points are to be transformed.

A 1-point DFT is simply the value of the point.

Computation often depicted in the form:



Four-point discrete Fourier transform

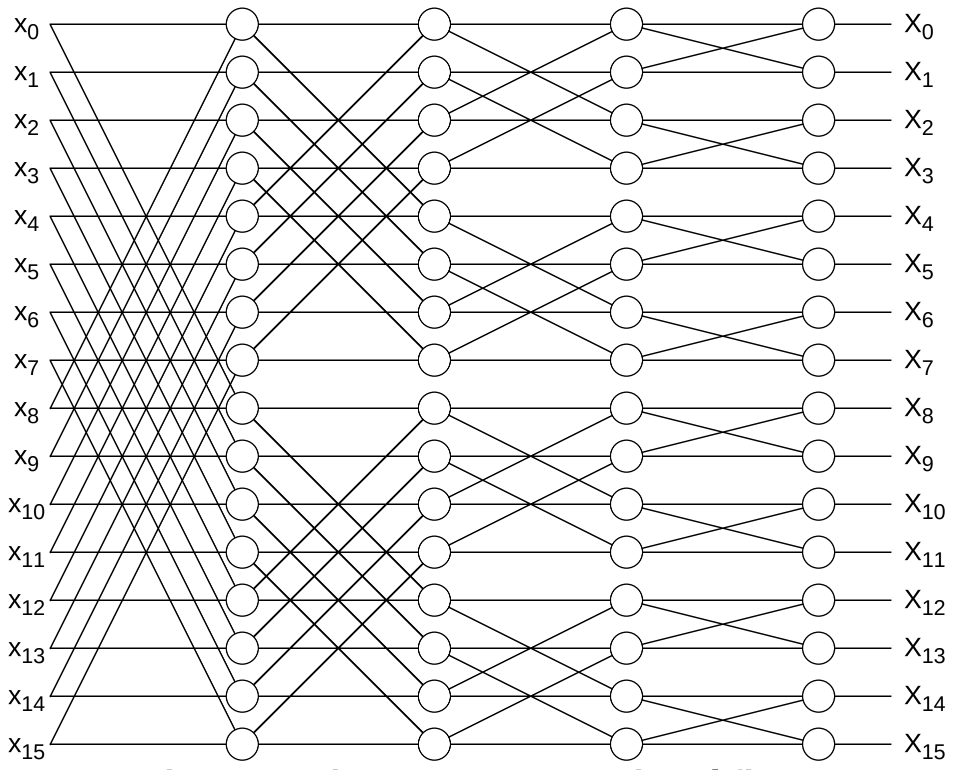
Sequential Code

Sequential time complexity is essentially O(*N* log *N*) since there are log *N* steps and each step requires a computation proportional to *N*, where there are *N* numbers.

The algorithm can be implemented recursively or iteratively.

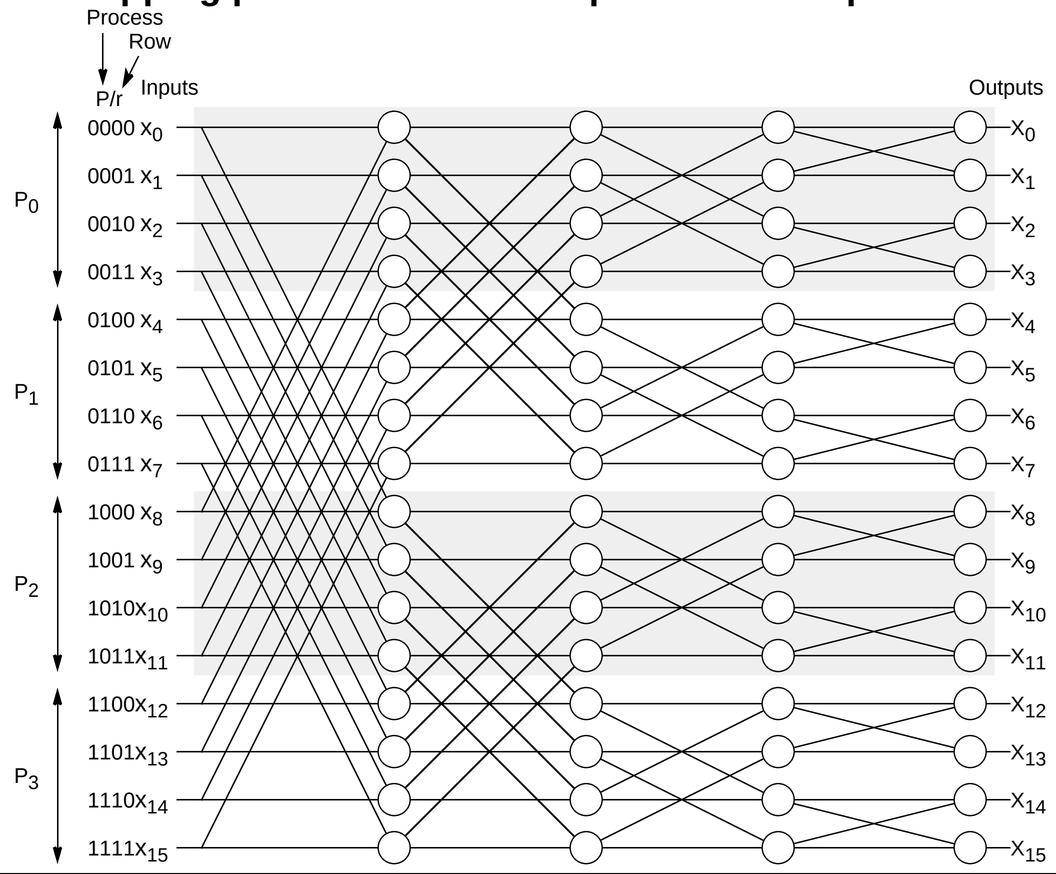
Parallelizing the FFT Algorithm

Binary Exchange Algorithm



Sixteen-point FFT computational flow

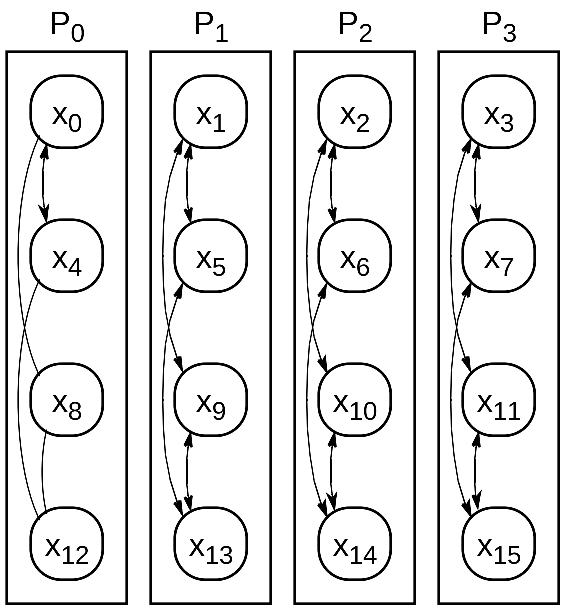
Mapping processors onto 16-point FFT computation



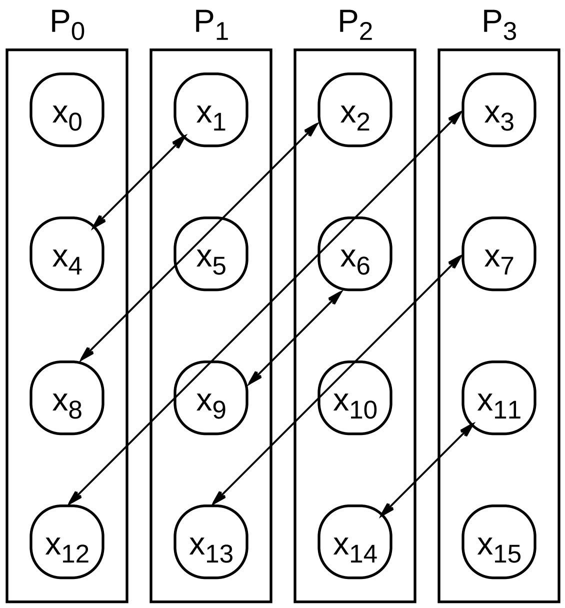
Transpose Algorithm

If processors organized as 2-dimensional array, communications first takes place between processors in each column, and then in each row:

FFT using transpose algorithm — first two steps.

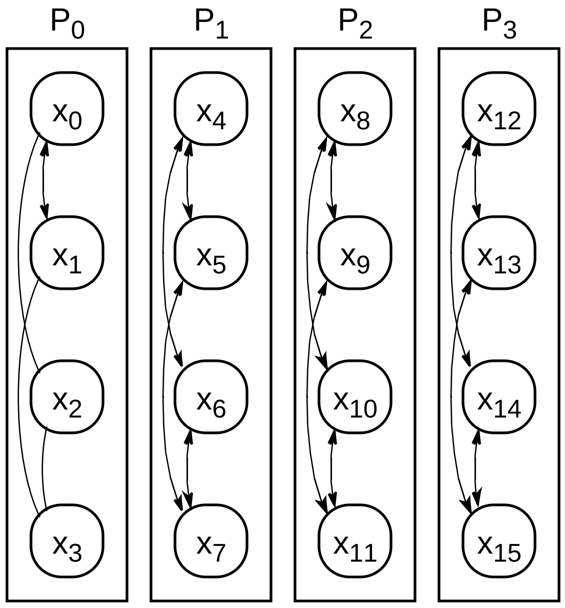


During the first two steps, all communication within a processor. Duringlast two steps, the communication between processors. Between the first two steps and the last two steps, the array elements transposed.



After transpose, last two steps proceed but now involve communication only within the processors. Only communication between processors is to transpose array.

FFT using transpose algorithm — last two steps


P_0
P_1
P_2
P_3
x_0
x_4
x_8
x_{12}
x_1
x_5
x_9
x_{13}
x_2
x_6
x_{10}
x_{14}
x_3
x_7
x_{11}
x_{15}