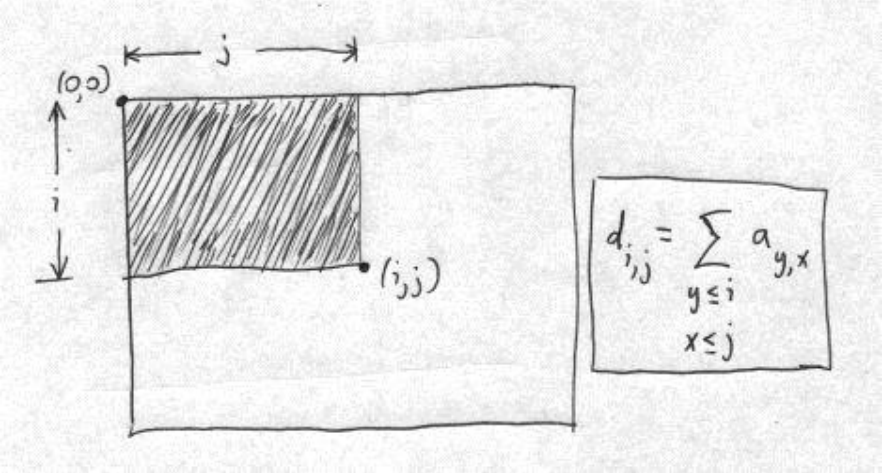
GRAYSCALE CONVOLUTION USING AN ACCUMULATOR

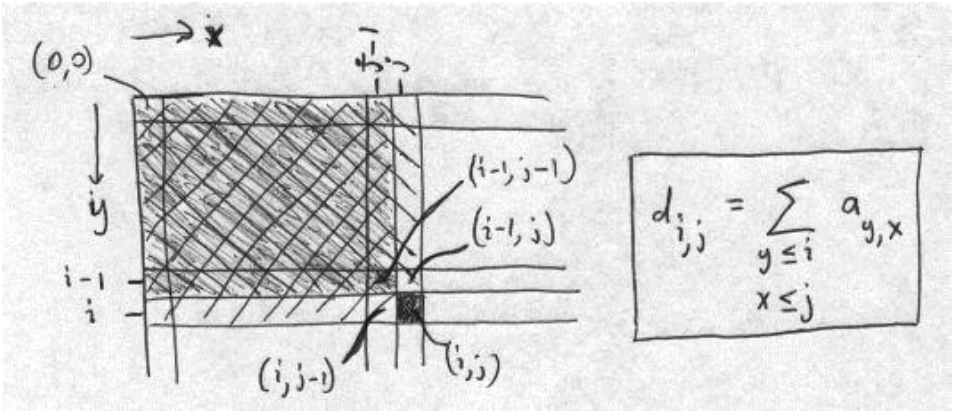
The algorithm for convolution with a flat, rectangular filter was described in a classic 1984 graphics paper paper by Frank Crow. The definition of the accumulator sum array is given in this figure:



The accumulator is a 32 bit/pixel array, where the value of each "pixel" in the accumulator is the sum of all pixel values in the source that are in the rectangle that is above and to the left of the pixel; i.e., the rectangle for which that pixel is in the lower-right corner. Fortunately, the accumulator array di,j can be found recursively from the source pixels a, by computing the accumulator values in row-major order (left-to right fast; top-to-bottom slow) using

di,j = di,j-1 + di-1,j + ai,j - di-1,j-1 (accumulator)

This recursion relation can be seen geometrically from the following diagram:

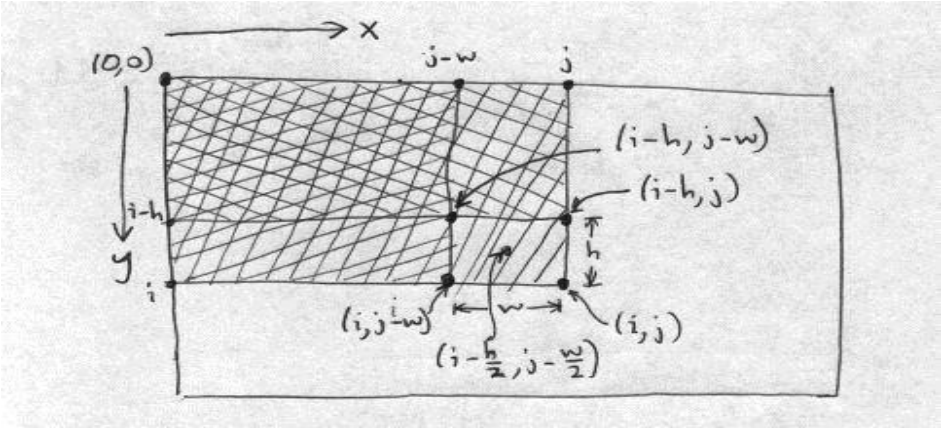


The gray-shaded rectangle labeled by di-1,j-1 must be subtracted because the other two (hatched) accumulator rectangles each include it, and it must only be included once.

The convolution with a flat rectangular kernel of width w and height h is then found using the accumulator in the following way:

bi,j = [di,j + di-h,j-w - di,j-w - di-h,j] / wh (convolution using accumulator)

This is the "inner loop" of the convolution, and it can be visualized as:



The convolution is being performed over the w by h rectangle, using accumulator values at the four corners of this rectangle, given in the equation above. It is evident that once the accumulator is found, a convolution for any size filter takes only a sum of four terms (and a multiplication if the normalization by filter area is not included in the accumulator values) for each dest pixel.

The equation for the convolution using the accumulator, as written above, has an unnecessary asymmetry in that the pixels used for the convolution are in the rectangle above and to the left of it. The average should instead be taken over pixels on all sides, so that the dest pixel is as close as possible to the center of the rectangle of source pixels used. We change parameters and let the full width and height of the convolution filter be 2w+1 and 2h+1, respectively. We can then write the convolution inner loop as

$$b_{i,j} = [d_{i+h,j+w} + d_{i-w-1,j-h-1} - d_{i+h,j-h-1} - d_{i-h-1,j+w}] / (2w+1)(2h+1)$$

(symmetric convolution using accumulator)

With reference to the figure above, we are finding the dest pixel value in the center of the rectangle covered by the filter. In the figure this pixel was labelled by (i-h/2,j-w/2), but we are now labelling the dest pixel by (i,j). Note also that in the figure, the filter area is given as wh, not as (2w + 1)(2h + 1) in the above equation.

In the equation, we still have a slight lack of symmetry in the convolution between the positive direction (e.g., i+h) and the negative direction (e.g., i-h-1). This is discussed in the source code in convolvelow.c.

We are not yet finished, because the boundary conditions must be handled properly. The accumulator is found recursively, looking up one row and left one column, so special cases need to be used for the top row and leftmost column of the array. Boundary effects on the convolution are more difficult. We have three choices: (1) ignore pixels where the convolution filter goes beyond the edge of the image, (2) do the best job you can with the pixels near the boundary, and (3) use mirrored pixels to compute near the boundary. For pixBlockconv(), we choose the second method. For pixels on the corners of the image, for example, we only have about 1/4 of the neighbors to use in the convolution that we have for pixels where the full filter can be used. The sum over those neighboring pixels should be normalized by the actual number of pixels used in the sum. The method we use is (1) for every pixel, use all possible pixels in the convolution, staying within the source image, but normalize as if we had used the entire filter, and (2) then make a second pass for the boundary pixels, adjusting the normalization upwards by the inverse of the fraction of the filter pixels that were actually used at each dest pixel.

The first part gives values that are too small for convolutions near the boundary; the second pass increases these pixel values to their correct normalization, depending on exactly which row and column the pixel is in. Doing the normalization this way avoids overflow in the destination pixels. The result has no visible boundary pixel artifacts in the convolution for typical grayscale images.

An alternative approach, mentioned above, is to add mirrored border pixels, of sufficient size so that the accumulator array can be used at all points in the interior, corresponding to the original image, and without special-casing any pixels. This is implemented in pixBlockconvTiled(), where we also allow the convolution to be done independently in an arbitrary set of tiles of the image. The functions to generate the tiles and put the result back together after separate convolution, are found inpixtiling.c. If there is a single tile, the tiled convolution defaults to the original pixBlockconv. Otherwise, it is verified that the result of block tiling is identical to that given by the generic function pixConvolve(), for any tiling and kernel size. (Well, almost any: the constraint on the convolution size is that the convolution width must not exceed the tile width, and similiarly for the heights.)

Breaking the convolution up into tiles is useful in two situations. First, if the grayscale image has more than 16M pixels, the accumulator array, stored as a 32-bit unsigned integer, can overflow. For such images, using tiles with less than 16M pixels is required.

In addition, because each tile is computed independently, the convolution can be carried out in parallel, making use of multiple cores to provide a linear speedup.

Source: http://www.leptonica.com/convolution.html