Chapter 3

Intensity and Affine Transformations

3.1 Intensity Transformations

Transformations applied directly to the pixels of an image can be easily implemented and have useful practical applications. A general expression for these transformations is:

$$g(x, y) = T[i(x, y)] \tag{3.1}$$

For example one can apply a transformation on a greyscale image to achieve some improvement or to focus on some aspect of the image. To invert an image the following expression can be used:

$$g(x, y) = 255 - i(x, y) \tag{3.2}$$

The effect is shown in the figure 3.1.

Figure 3.1: Inverted image.

Thresholding is used to segment images and it can also be used to binarise images. The following example shows the effect of a simple threshold (see figure 3.2).

$$g(x, y) = \begin{cases} 0 & \text{if } i(x, y) <= 128 \\ 255 & \text{if } i(x, y) > 128 \end{cases} \tag{3.3}$$

In OpenCV the function to apply a threshold is: `threshold(greyscaleimage, threshimage, t, 255, THRESH_BINARY);`

Figure 3.2: Thresholding.

where: the `greyscaleimage` is the original image, `threshimage` is the result, `t` is the threshold and 255 is the maximum value (in this case we have pixels with either 0 or 255, a binary image). The last parameter is the type of thresholding, in this case it is a plain application of equation 3.3. Next we will explore another more sophisticated form of threshold approach.

3.1.1 Otsu's thresholding

Histograms of greyscale images are very simple to compute. Histograms do not hold any information about location, but often they have interesting information about the differences between objects and backgrounds. Otsu's approach takes into consideration that images can be bimodal, i.e., the foreground and background hold completely different intervals for their pixel values. If that is the case, Otsu's approach can be used (after Nobuyuki Otsu [1]). However this approach should be used with caution, specially in cases where the illumination changes randomly.

A bimodal image has a histogram with clear two peaks. Each peak refers to either the foreground or the background. In order to find the optimal threshold, Otsu applied a simple principle: among all possible thresholds (256 of them in a greyscale image with 1 byte depth) choose the one that minimises the variance within the variances of the two classes (background or foreground). The within class variance for two classes is:

$$\sigma_w^2(t) = p_f(t)\,\sigma_f^2(t) + p_b(t)\,\sigma_b^2(t) \tag{3.4}$$

where: σ^2 is the variance and p is the portion of the pixels in one class or another and t is the threshold.

The only way to get the portions on each class is by trying, by assuming a certain threshold. Once we know p, it is easy to compute the two variances, and compute $V_w(t)$ for that threshold. After the set of all possible thresholds are computed, we choose the minimum and set Otsu's threshold to segment the image.

In OpenCV the implementation of Otsu's method is:

```
threshold(greyscaleimage,threshimage,0,255,CV_THRESH_BINARY | CV_THRESH_OTSU);
```

Note that the threshold parameter is zero, indicating that the threshold will be computed by the function instead.

3.2 Contrast Stretching

Another form of simple transformation, yet a powerful one, is contrast stretching. A simple approach is to use the current statistics of the image to transform all pixels. The results can be dramatic, see some examples in figure 3.3.

There are different equations for contrast stretching, but the one used here is:

$$\bar{i}(x,y) = \frac{255(i(x,y) - \mu + c\sigma^2)}{2c\sigma^2}, c \in \Re^+, and \quad 0 \leq \bar{i}(x,y) \leq 255 \tag{3.5}$$

Where $i(x,y)$ is the image before the transformation, $\bar{i}(x,y)$ is the image after the transformation. μ is the mean value for image $i(x,y)$, σ^2 is the variance for image $i(x,y)$, and c is a constant. This equation is similar, yet not the same, to the one commonly used for contrast stretching in the literature (e.g. see [2]). Common values for c lie between 1 and 2. Contrast stretching of this form can be implemented efficiently using Integral Images (presented in section 3.5).

Equation 3.5 is, in principle, a linear transformation of the image $i(x,y)$. However, as pixels are limited in range, any new values for image $\bar{i}(x,y)$ that are outside the expected range are flattened to 0 or 255. If these cut-offs are used, some details of the original image might be lost.

3.3 Histogram Equalisation

Histogram equalisation is also often used to enhance contrast. For example when some features of the image are almost indistinguishable from the background, an equalisation transformation might be enough to sharpen details. However, depending on the initial histogram, the contrast may improve over some areas of the image while over other areas it may flatten the details.

Algorithm 2 ***Histogram equalisation***$((image, image2)$

The input is a grey-scale *image* and the output is a grey-scale *image*2.
Auxiliary datastructures are two histograms $H[i]$, $HC[j]$ and a transform $T[k]$.

1. Create an array $H[r]$. (Where r is the range of possible values, e.g., grey-scale would use 256).

2. Compute the histogram H. For each pixel p of value i:

   $H[i] = H[i] + 1;$

3. Compute a *cumulative histogram* HC:

   $HC[0] = H[0]$

   $HC[i] = HC[i-1] + H[i]$ for i=1,2,3...255 for grey-scale images

4. Set a discrete transformation function:

   $T[i] = round(\frac{255}{width.height}.HC[i])$ for i=1,2,3...255

5. Create image2 scanning all pixels p' adopting values according to:

   $p' = T[p']$

The histogram equalisation can be seen in figure 3.4. The objective is to assign new values to pixels in such a way that we get a flat area in the new histogram. Even using a discrete histogram, using properties of a probability density function we can write:

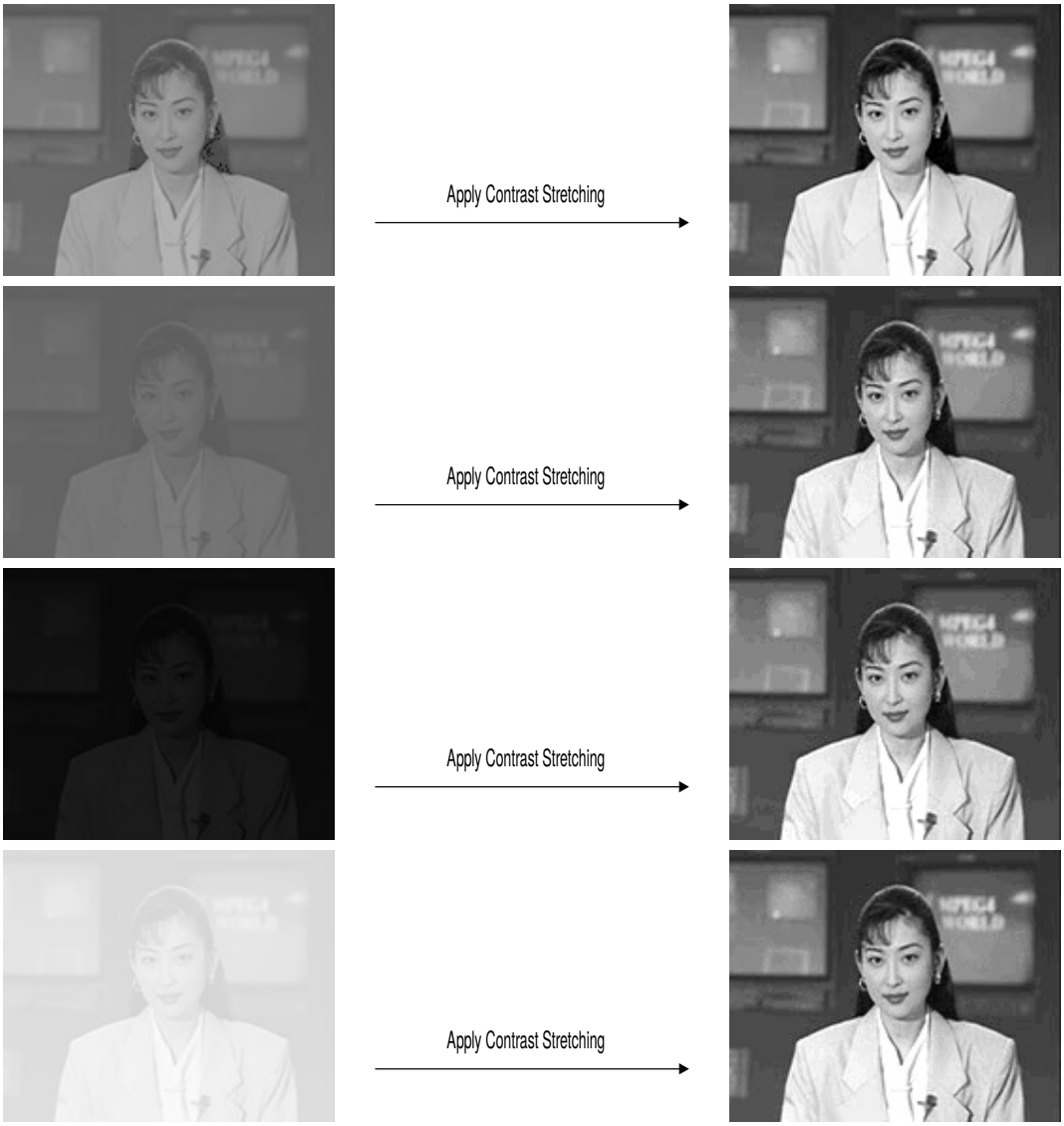

Figure 3.3: The images on the left column were artificially altered in such a way that the pixel values are distributed within a narrow band. On the right column are the results of the tranformation using equation 3.5 with $c = 1.8$. The resulting images in the right columns are actually different then each other, though with a very similar histogram.

$$\sum_{i=0}^{k} G(q_i) = \sum_{i=0}^{k} H(p_i) \tag{3.6}$$

A simple algorithm can be written for a general histogram equalisation (algorithm 2). Because the equalisation of grey-scale image necessarily involves rounding up to integers, there is a degree of uncertainty on the areas below the histogram line, so the equalised area is often not perfectly rectangular. It is interesting to implement the following algorithm and verify the shape of the histogram after the equalisation.

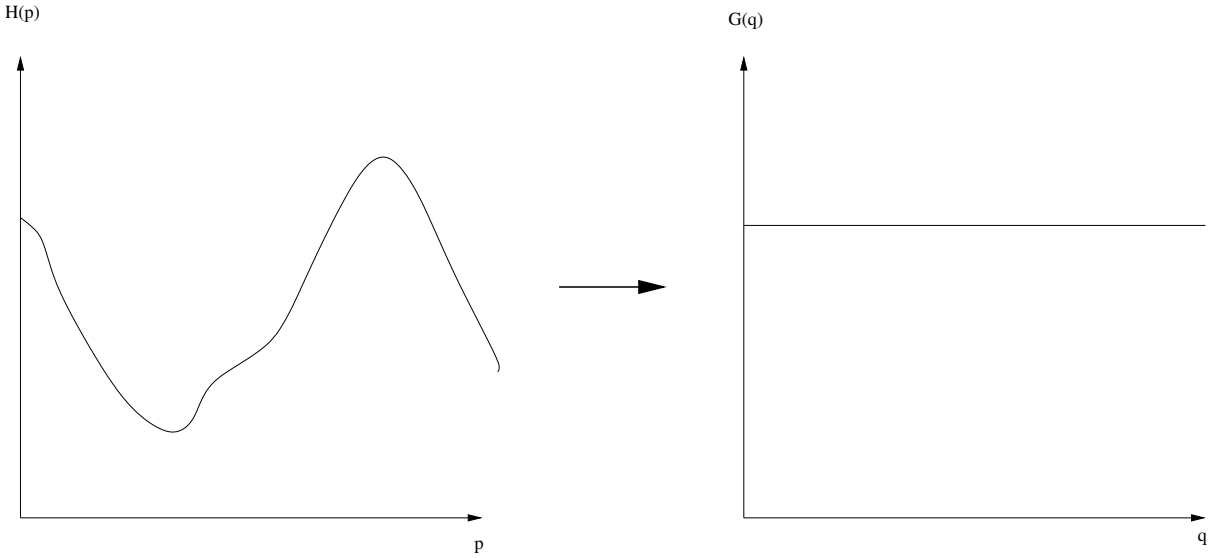

Figure 3.4: Histogram equalisation.

3.3.1 Histogram specification

Histogram specification modifies the image in a similar way to histogram equalisation. The difference is that the final histogram should be constructed using the histogram of another image.

Sometimes it is convenient to specify the new histogram to be as the equalised histogram of a sub-window of the image. Algorithm 2 can be used with minor modifications for this task. Once the histogram equalisation is completed, the new image is reconstructed based on the characteristics of the sub-window rather than the entire image.

3.4 Spatial Filtering

Spatial filtering or *Neighbourhood processing* can be done by defining a centre point and performing operations that involve the pixels that are in the neighbourhood. The neighbourhood can be defined for example as a square or oblong region of the image. The nature of the mathematical operation over the pixels is either linear or non-linear.

One can use the so called kernel (also called mask, filter, window etc) to define the centre and the neighbourhood. The kernel is swiped over the image and each pixel receives a new value based on the operations defined in the kernel. This process usually requires that a new image is created, so new pixel values are not mixed with the previous values that should be used for the computations. Figure 3.5 shows a simple example. The kernel can be also seen as a matrix in the case where a linear operation is involved.

A simple example of a kernel is one used to obtain a weighted sum of the pixels in the neighbourhood. This is sometimes called a convolution operation[1]. An associated equation defines the new value for the centre of the kernel (equation 3.7).

[1]There is actually a distinction between a true convolution and the spatial filter we are using here. For more details refer to [2]

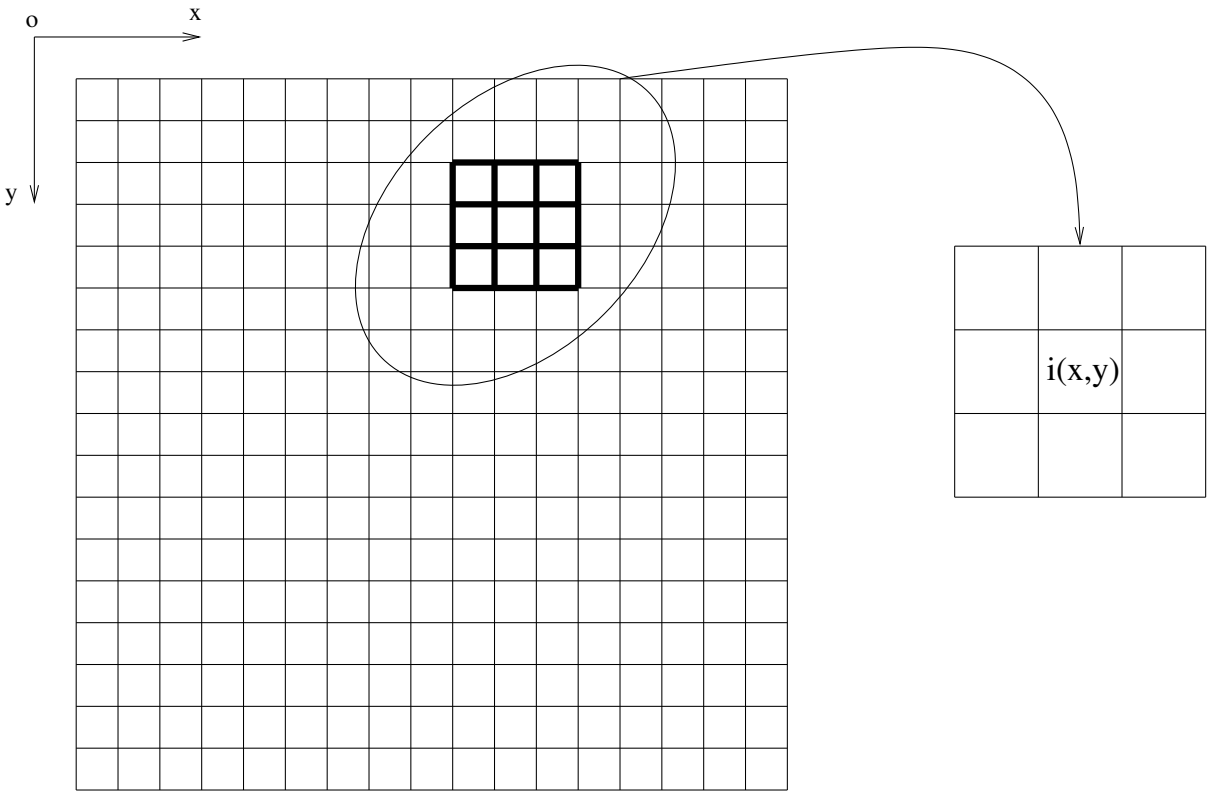

Figure 3.5: A 3x3 kernel.

$$j(x,y) = i(x,y) * M(m,n) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} M(s,t) i(x+s, y+t) \tag{3.7}$$

Where: $j(x,y)$ is the resulting image, $i(x,y)$ is the original image and M is a convolution mask of size m x n, $m = 2a+1$ and $n = 2b+1$. Common sizes for convolution masks are 3x3, 5x5 and 7x7.

3.4.1 Smoothing masks

If all the elements in the mask are positives, then the image is smoothened (or blurred). One can eliminate some types of noises from images using this mask. If the mask is an approximation of a Gaussian distribution then better results regarding noise can be obtained. See figure 3.6.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Figure 3.6: Two smoothing 3x3 kernels.

3.4.2 Sharpening masks

The opposite of smoothing an image is to sharpen its edges by applying a convolution mask that is equivalent to the derivative of the image. An image is in fact a two-dimensional function. The first and the second derivatives have specific uses for image processing.

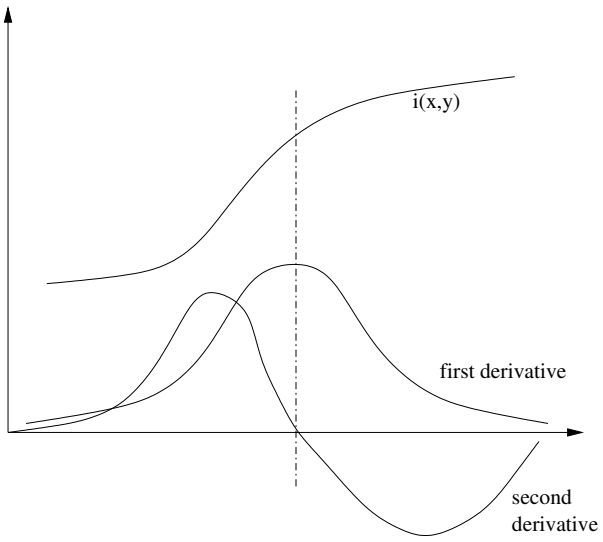

Figure 3.7: Derivatives of the image in one direction.

Gradient

The first derivative can give information about the magnitude and direction of the gradient of the image. Unfortunately in order to find the magnitude non-linear operators have to be used. In order to avoid that and keep the convolution mask feasible an approximation is used. For example, for a 3x3 mask the following equation applies:

$$\nabla i \approx (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3) + (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7) \tag{3.8}$$

Two convolution masks can express equation 3.8, see figure 3.8. These are known as *Sobel operators* and are used to find edges in images.

−1	−2	−1
0	0	0
1	2	1

−1	0	1
−2	0	2
−1	0	1

Figure 3.8: Sobel operators.

Laplacian

The second derivative (Laplacian) is also used to identify edges, but the results can be used to enhance the image directly. The Laplacian is defined as:

$$\nabla^2 i = \frac{\delta^2 i}{\delta x^2} + \frac{\delta^2 i}{\delta y^2} \tag{3.9}$$

An approximation of the implementation of the Laplacian is:

$$\nabla^2 i = z_2 + z_4 + z_6 + z_8 - 4z_5 \tag{3.10}$$

A mask can be created from equation 3.10. Different masks can also be used considering that the diagonals are used (see figure 3.9). Applying the masks in figure 3.9 result in an image with the edges of objects in a dark background. Homogeneous areas of the image will yield pixels with value zero.

0	1	0
1	−4	1
0	1	0

1	1	1
1	−8	1
1	1	1

Figure 3.9: Two different kernel implementations of the Laplacian.

If the edges of the image are subtracted from the original image, the result is to sharpen the edges. Simple masks can be devised for that (see figure 3.10).

0	−1	0
−1	5	−1
0	−1	0

−1	−1	−1
−1	9	−1
−1	−1	−1

Figure 3.10: Sharpening masks.

3.4.3 Median filter

The median filter is a non-linear operator. The pixel in the centre is replaced by the median value obtained from the kernel. This filter changes pixels with values that are too dissimilar from the neighbour pixels while preserving the edges. It eliminates some types of noise. The implementation of median filters require sorting the pixels in the kernel followed by the substitution of the centre pixel by the value found in the middle of the sorted set.

For example, suppose that the kernel contains 9 pixels, and that its values are: 1,2,1,3,4,5,1,2,3

By sorting these values we get: 1,1,1,2,2,3,3,4,5

The value in the middle is this case is 2, and this should replace the original pixel.

One of the best applications for the Median filter is to eliminate the so-called salt and pepper noise. This type of noise was common on old photographs and negatives, as the chemicals in them would slowly combine with other substances or naturally erode. Figure 3.11 shows an example of an image with salt and pepper noise that had gone through the Median filter. Note that as long as the noise pixels are small, the noise can be completely eliminated.

In OpenCV there is a function for that: `medianBlur(image,image,kernelsize);`

where: image is a Mat image, and the last parameter is the kernel size (the size of the matrix for the convolution kernel).

Figure 3.11: Median filter example.

3.4.4 Kuwahara filter

The Kuwahara filter [3] is also a non-linear operator. It is more demanding than a median filter because more operations are required and the square of the pixels are needed to compute the variance of areas (sub-windows) of the image. The Kuwahara filter is capable of smoothing entire areas of the image, while still preserving the more important edges. A variety of shapes are possible. The filter requires that the kernel is divided in four different regions that intersect in the centre pixel. The variance (equation 3.11 represents the sample variance) for each of these areas is computed. The centre pixel assumes the value of the mean brightness of the area where the variance was the smallest. Figure 3.12 shows the implementation for a 5x5 kernel.

$$\sigma_a^2 = \frac{1}{N}\sum(i(x,y) - \mu_a)^2 \tag{3.11}$$

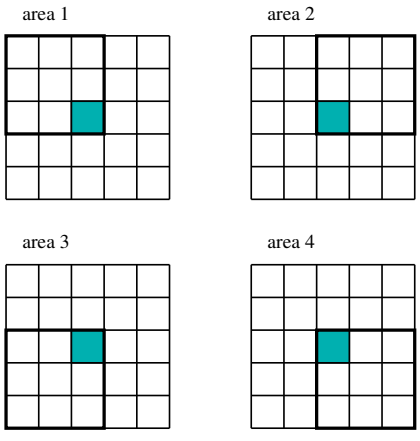

Figure 3.12: The Kuwahara filter.

One possible implementation approach is to use *integral images* (also known as *summed-area tables*) to rapidly compute the mean and the variance.

3.5 Integral Images (or Summed-area Tables)

Given an image $i(x_i, y_i)$, the integral image $I(x, y)$ is:

$$I(x, y) = \sum_{x \leq x_i, y \leq y_i} i(x_i, y_i) \tag{3.12}$$

After computing the integral image for one frame, sum of rectangular areas over the image can be computed with just 4 look-ups (figure 3.13).

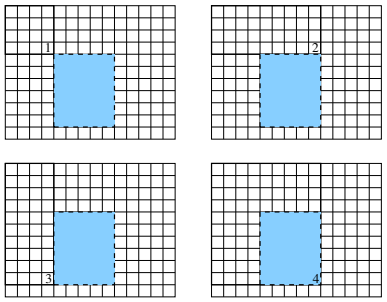

Figure 3.13: The integral image approach.

In order to compute a square area, four lookups in the integral image are needed:

$$Area = pt_4 - pt_2 - pt_3 + pt_1 \tag{3.13}$$

The implementation of integral images are also tricky for many reasons. Due to the discrete nature of the structure, the indexes need to be carefully calculated to correspond to the correct sum of the area. Also, if the image is large it will yield large numbers, so the right-bottom regions of the integral image can overflow. One needs to use double or even long double. Finally, precision becomes a problem too. The final values for sums of areas may be rounded up. Despite all these problems, the big advantage is speed.

An IplImage structure can be used, as long as each pixel is large enough. One can compute the precision limit by considering that the image contains pixels of maximum value, as well as the image size. For example:

$$max = width * height * 255 \tag{3.14}$$

Suppose that the $width = 640$ and $height = 480$, then $max = 78336000$. Type `unsigned int` could hold up to 4216631295. Therefore, 4 bytes per pixel would suffice in this case, with room to spare.

When creating the integral image, it is convenient to create a matrix (or IplImage) with one extra column and one extra row, both initialised to zero. This is because there is a recursive way of computing the integral image from a frame:

$$ii(x, y) = ii(x - 1, y) + ii(x, y - 1) - ii(x - 1, y - 1) + i(x - 1, y - 1) \tag{3.15}$$

Where: $i(x, y)$ is the original image and $ii(x, y)$ is the resulting integral image.

One needs to be careful with the indexes of the the integral image and the frame. For example, to compute the sum of the area defined by the index points $(1, 1), (1, 2), (2, 1), (2, 2)$, the indexes of the integral image would actually be:

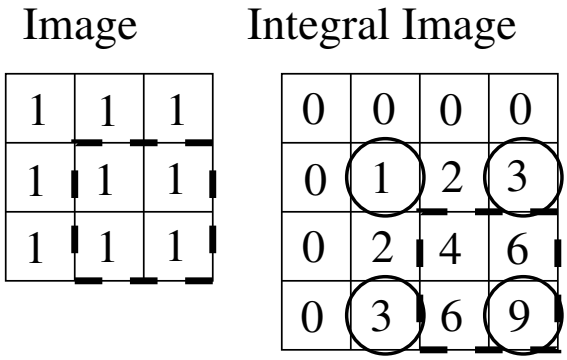

Figure 3.14: Integral image example.

$$(1,1),(3,1),(1,3),(3,3)$$

So, in the example in figure 3.14, the area of the marked four pixels would be:

$$Pt_{(3,3)} - Pt_{(1,3)} - Pt_{(3,1)} + Pt_{(1,1)} = 9 - 3 - 3 + 1 = 4$$

3.5.1 Other integral images

The concept of the integral image can be extended to other kinds of sums of pixels. For example, the sum of the squares of the pixels can help to compute the variance over a certain area of the image. Equation 3.15 would be re-written as:

$$sii(x,y) = sii(x-1,y) + sii(x,y-1) - sii(x-1,y-1) + i(x-1,y-1)^2 \tag{3.16}$$

Where: $sii(x,y)$ is the square integral image.

Computing the variance repeatedly for different sub-windows using equation 3.11 would be computationally expensive. However, one can re-write the equation as a function of the sums of the squares, using a variance estimator and the sample mean:

$$\sigma_a^2 = \frac{1}{N}\sum (i(x,y) - \mu_a)^2 = \frac{1}{N}\left(\sum i(x,y)^2 - \frac{\left(\sum i(x,y)\right)^2}{N}\right) \tag{3.17}$$

One practical use of integral images is related to the implementation of the Kuwahara filter. Equation 3.17 can be simplified to cater for two integral images: one representing the *square of the sum* of the pixels, the other representing the *sum of the squares* of the pixels. The comparison of the variances between the four areas required by the Kuwahara filter can be made without actually using the true values for the variance (ignoring the $1/N$ coefficient does not affect the comparison).

$$\sigma_{comp}^2 = \sum i(x,y)^2 - \frac{\left(\sum i(x,y)\right)^2}{N} \tag{3.18}$$

To estimate the variance for a rectangular area defined by Pt_1 to Pt_4, one needs two values:

$$\sum i(x,y)^2_{Pt_1 \ to \ Pt_4} = sii(Pt_4) - sii(Pt_3) - sii(Pt_2) + sii(Pt_1) \tag{3.19}$$

$$\sum i(x,y)_{Pt_1 \ to \ Pt_4} = ii(Pt_4) - ii(Pt_3) - ii(Pt_2) + ii(Pt_1) \tag{3.20}$$

3.6 Affine Transformations

Sometimes it is convenient to transform an image due to some distortion or to get some special geometric effect. In this case there is no direct intensity change, rather the position of pixels are changed. Affine transformations are a special case of more general transformations can be carried out using the following equations:

$$x' = a_1\, x + a_2\, y + a_3 \tag{3.21}$$

$$y' = b_1\, x + b_2\, y + b_3 \tag{3.22}$$

Where: x' and y' are the new pixel positions for the transformed image.

In this case, the pixel values from point (x, y) are copied to point (x', y'). These equations present a problem to digital images. As the positions of the resulting image are computed, there may be gaps that cannot be filled up due to the rounding up of the results. One way to improve that is inverting the approach, i.e., for every x' and y' find the correspondent source x and y, and then copy the pixel.

There are also problems regarding noise, which can be minimised by using interpolation techniques. These techniques are not covered here, but one can assume that opencv uses bilinear interpolation by default.

There are special cases of the affine transformations, they are: *rotation*, *scaling*, *translation* and *skewing*. The equations 3.25 to 3.29 refer to these special affine transformations:

As an example, follows the derivation of the equations for rotation. Let's assume that a certain pixel on (x, y) is going to be copied to (x', y') in figure 3.15.

If the length between (x, y) and the origin is l, one can write:

$$l = \frac{x}{cos(\theta_1)} = \frac{x'}{cos(\theta_2)} = \frac{y}{sin(\theta_1)} = \frac{y'}{sin(\theta_2)} \tag{3.23}$$

Let the difference between θ_1 and θ_2 be α. The following relation is true:

$$\begin{aligned}
&x'cos(\theta_1) = x\, cos(\theta_2) = x\, cos(\theta_1 - \alpha) \\
&x'cos(\theta_1) = x\, (cos(\theta_1)\, cos(\alpha) + sin(\theta_1)\, sin(\alpha)) \\
&\texttt{Replacing}\ x\, sin(\theta_1)\ \texttt{with}\ y\, cos(\theta_1) : \\
&x'cos(\theta_1) = x\, cos(\theta_1)\, cos(\alpha) + y\, cos(\theta_1)\, sin(\alpha)) \\
&x' = x\, cos(\alpha) + y\, sin(\alpha))
\end{aligned} \tag{3.24}$$

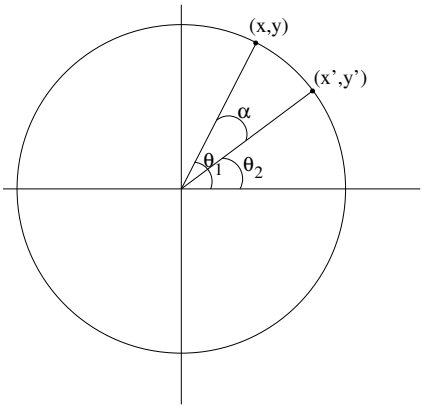

Figure 3.15: Rotation of a point (x, y) with center at $(0, 0)$.

Therefore, **Rotation** by angle α from the origin (or you have to redefine where is the centre of rotation):

$$x' = x\, cos(\alpha) + y\, sin(\alpha) \tag{3.25}$$

And the derivation of the destination y' is left as an exercise:

$$y' = -x\, sin(\alpha) + y\, cos(\alpha) \tag{3.26}$$

Scaling, a scale for x axis and b scale for y axis:

$$x' = a\, x \tag{3.27}$$
$$y' = b\, y \tag{3.28}$$

Skewing by angle α:

$$\begin{aligned} x' &= x + y\, tan(\alpha) \\ y' &= y \end{aligned} \tag{3.29}$$

3.6.1 Affine Transformations using matrices

Any affine transformation can be written as a matrix multiplication followed by a vector addition. The matrix multiplication does linear transformations such as rotations and scalings. Vector addition does translations.

A common way to represent affine transformations is to use a 2x3 matrix to multiply the vector (x, y) or $\begin{vmatrix} x \\ y \end{vmatrix}$.

It is common to use an augmented vector of the form $\begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$ to represent points in 2D.

The advantage of using the augmented vector is that an arbitrary affine transformation can be written as a simple matrix multiplication:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix} \tag{3.30}$$

In OpenCV you will find that there are lots of matrix operations already implemented. There are also ways of automatically computing the transformations matrix given certain parameters.

For example, suppose that you want to rotate an image by 45°, with the centre of rotation in the origin.

An affine matrix can be produced in one line:

```
Mat rotmatrix=getRotationMatrix2D(Point(0,0), angle, scale);
```

Where: angle is in degrees, and scale is a floating point operator (in this case = 1.0).

The computed matrix would look like:

$$\begin{vmatrix} 0.707 & 0.707 & 0.0 \\ -0.707 & 0.707 & 0.0 \end{vmatrix} \tag{3.31}$$

Now one can use the rotmatrix to actually rotate the image:

```
warpAffine(imgsource, imgdest, rotmatrix, imgsource.size());
```

3.7 Reading

Read Chapter 3 of Gonzales and Woods.

3.8 Exercises

3.8.1 Exercise 1

1. Compute an affine matrix for: centre at the middle of the image (image size is 176x144), angle=30°, and scaling=1.
2. Write a program using OpenCV that implements rotation, translation, scaling and skewing for a grey-scale image, based on equations from section 3.6.
3. Is there anything wrong with the resulting image? Why?
4. Fix the problem using an inverted approach.
5. Use OpenCV approach to achieve the same results. (e.g., use `warpAffine()` with a 2x3 matrix)

3.8.2 Exercise 2

1. Write a program that smooths the image using a linear filter. The program should allow the kernel for the linear filter to be created at different sizes (take a parameter for the kernel size).

3.8.3 Exercise 3

1. Write a program for histogram equalisation.
2. Modify the program for histogram *specification*. Your program should take 6 parameters:

   `equalise input.bmp output.bmp x y x' y'`

   This should load the input.bmp image and produce the output.bmp file. The equalisation should be specified by the histogram from the rectangular area in the image defined by the points (x,y) and (x',y').

Bibliography

[1] N. Otsu, "A threshold selection method from gray-level histograms," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 9, pp. 62–66, Jan 1979.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice Hall, 2008.

[3] M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita, *Digital processing of biomedical images*, ch. Digital processing of biomedical images, pp. 187–203. Plenum Press, New York, NY, 1976.