CHAPTER 5

Linear Image Operations

1. Simple Smoothing Filter

Consider the following problem. We are given images that are corrupted by random noise and we are asked to clean them. The content of the images can be described as pictures with rather smooth variation in gray level. This is one of the easiest “restoration” problems and has one of the simplest solutions. Although things are not so rosy in practice, it is a good example.

Since the noise is random we will take advantage of averaging. It is well known that if we average say 9 random numbers the result is also a random number but has a standard deviation that is one third the original. So if we replace every pixel of the image with the average of its 3×3 neighborhood we will average out the random noise to one third of its original strength but we will not have a great effect on the original image that had smooth variation anyway.

Congratulations, we designed a really simple filter that works well on images corrupted in heavens. But is it the best we can do? Will it work as well on images corrupted in a less favorable way? Let’s write down this simple method as a formula and try to get some insight

$$I_r[k_0, l_0] = \sum_{-1 \le k \le 1} \sum_{-1 \le l \le 1} \frac{1}{9} I[k_0 - k, l_0 - l] \tag{1.1}$$

where I is the original image and I_r is the “restored” image. It is easy to show that this expression is linear because it is a simple weighted sum of values of image I. Although there are filters that are non linear there is no general methodology to study them, so most of our discussion will concentrate on linear filters. Which means that the basic form of Eq. (1.1) remains the same and the only things that we can vary are the weight of the averaging and the limits of the summation. The weights need not be constants, in fact they can be variables of i and j or of k_0 and l_0 or all four and the limits can be anything we want. Thanks to a French mathematician/engineer/politician that served as officer in Napoleon’s army, we know very well how to analyze filters with weights that vary with k and l. We could do it for weights that vary with k_0 and l_0 or all four but with some difficulty so will avoid it. This engineer, by the way, was named Jean Batiste Joseph Fourier.

It seems that we have restricted ourselves quite a bit here by working only on one particular kind of linear filter. This is partially true because these filters have very predictable behavior and as a result we can tailor them to our exact specifications. Other filters are much more powerful but we can not predict their exact behavior. They are very powerful weapons but we do not know which way they shoot.

We can write Eq. (1.1) in the most general form we are going to use.

$$I_r[k_0, l_0] = \sum_k \sum_l w[k, l] I[k_0 - k, l_0 - l] \tag{1.2}$$

where $w[k, l]$ is the set of weights for this weighted average. This linear operation is called *convolution*. The two dimensional data structure $w[k, l]$ is called convolution template or convolution kernel. The similarity between Eq. (1.2) and the expression for dilation is quite striking.

Apart from this nice coincidence, the formula is not easy to play with. Double summations with four indices are known to cause headache to laboratory animals and since we are going to do quite a bit of algebraic manipulation with Eq. (1.2) we can consider for a while the one dimensional version to demonstrate the most important concepts. After that, we can switch to the two dimensional version and discuss things like separability and circular symmetry which are exclusive to this dimensionality. The expression for the one dimensional case is

$$I_r[k_0] = \sum_k w[k] I[k_0 - k]. \tag{1.3}$$

1.1. Fourier Analysis

We could try to apply this expression on different images I and templates w and see what happens. With a little luck we will notice some pattern, if we choose the proper images and templates to play with. One very promising such “image” is the exponential image

$$I[k] = \alpha^k$$

where α is any number; real, imaginary or complex. Applying Eq. (1.3) we get

$$I_r[k_0] = \sum_k w[k] \alpha^{k_0 - k} = \sum_k w[k] \alpha^{-k} \alpha^{k_0} = \alpha^{k_0} \sum_k w[k] \alpha^{-k}.$$

We can see that

$$W(\alpha) = \sum_k w[k] \alpha^{-k} \tag{1.4}$$

is a constant because it does not depend on k_0 the image index. It depends only on which exponential we used. So

$$I_r[k_0] = W(\alpha) \alpha^{k_0}$$

in other words, the exponential image is scaled! Aren’t we lucky. Unfortunately the exponential is the only such function. Mathematicians call functions that can go through an operation, without any change other than scaling, eigenfunctions (like eigenvectors that when multiplied by a matrix get scaled).

The analysis of convolution can become much simpler if we first decompose every image into a sum of exponentials and that’s where Fourier gets into the picture! If $\alpha = e^{j\omega}$ where j is the imaginary unit we can do the decomposition easily because

$$e^{j\omega} = \cos\omega + j\sin\omega \tag{1.5}$$

and the Fourier analysis decomposes any function into a sum of sines and cosines. If we use this complex version of sines and cosines an image can be written as

$$I[k] = \sum_m c_m e^{jm\omega_0 k} \tag{1.6}$$

where $\omega_0 = 2\pi/N$ and N is the size of the image. This equation does not say much before we know how to compute the c_m's. So

$$c_m = \frac{1}{N}\sum_k I[k]e^{-jm\omega_0 k} \tag{1.7}$$

which is remarkably similar to Eq. (1.6) and even more remarkably similar to Eq. (1.4).

Now we have a way to view the convolution operation. We decompose an image into a sum of exponentials, convolve each exponential, which is easy, and then reconstruct the result.

While everybody is familiar with with the sines and cosines, the complex exponential can look kind of scary. Well, it is not. To see this consider the following expression

$$(\cos\omega + j\sin\omega)^k.$$

It might be a little complicated to attempt to simplify this. But if we replace the sine and cosine expression with the complex exponential the simplification is obvious:

$$(\cos\omega + j\sin\omega)^k = \left(e^{j\omega}\right)^k = e^{jk\omega} = \cos k\omega + j\sin k\omega.$$

The scary thing would be to do this without using the definition of complex exponentials. So, whenever we suspect that exponentials are easier than plain sines and cosines, we can replace every sine with

$$\sin\omega = j\,\frac{e^{-j\omega} - e^{j\omega}}{2}$$

and every cosine with

$$\cos\omega = \frac{e^{j\omega} + e^{-j\omega}}{2}.$$

Notice that both the sine and the cosine have “negative” frequencies $-\omega$. There are no such things as negative frequencies in the real world. The “negative” frequencies are just mathematical artifacts that got life of their own.

Now that we are convinced that not all exponentials are equally useful and we can use the ones defined in Eq. (1.5) to our advantage, we can rewrite Eq. (1.4) to reflect this:

$$W(e^{j\omega}) = \sum_k w[k]e^{j\omega^{-k}} = \sum_k w[k]e^{-j\omega k}. \tag{1.8}$$

Admittedly, the differences between (1.7) and (1.8) are minor. Other that the $1/N$ factor the difference is that ω is replaced by $m\omega_0$. Most textbooks call both formulas Fourier.

2. Pictorial View of Fourier Analysis

There are many details in the Fourier analysis that we cannot cover in this simple first approach to this very deep and fascinating theory. Without expecting to do any great justice to this subject let's see the pictures.

We still assume one dimensional images which we can thing of as single rows taken from regular images. First question is how does a sine or a cosine look like. They cannot even be displayed because they alternate between positive and negative and displays cannot show negative intensities.

Not all images can be displayed directly as a set of intensities because they do not represent such an intensity. The value of a pixel could represent velocities, reflectivity etc. Or we are in the middle of a series of image transformations and these intermediate results contain pixels with negative intensities. But no matter what is the reason that we got negative values we might need to display these images. One of the simplest things to do is to rescale the image to fit between 0 and 255 which is the usual intensity range on a display. To view an image every row of which is a sine using MediaMath, we simply do

```
gshow(gsin(omega * x_img(128,128)), :rescale=t);
```

and the image is shown in Fig. 2.1.

We can also create an image of all the sines, one for each row. There is not much use for such an image of course other than visualization of the basis images (Fig. 2.2). We will see later the effect of different filters on each strip of this image.

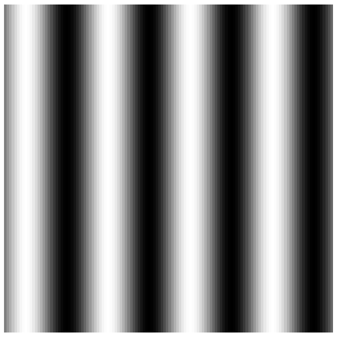

Fig. 2.1. This is how a sine looks like on a screen. The darkest parts of the image correspond to intensity equal to -1, the brightest to intensity equal to 1 and the 50% grey to intensity equal to 0.

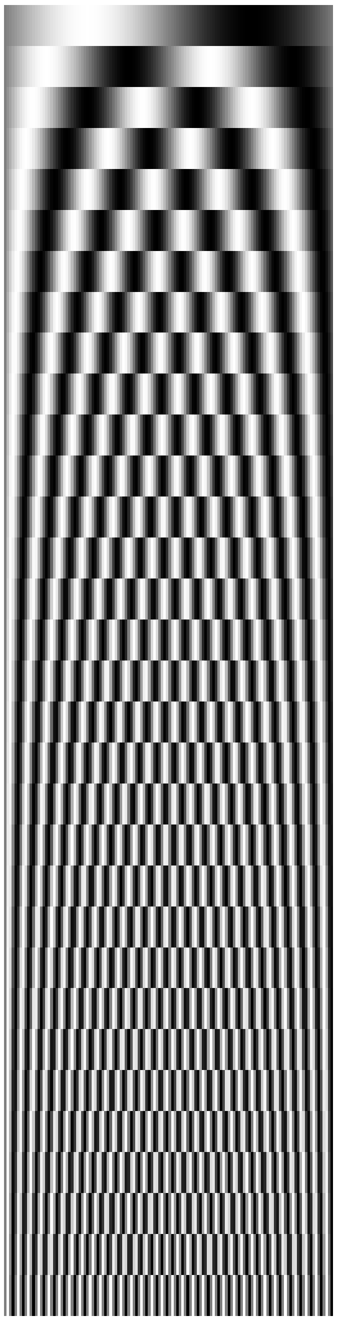

Fig. 2.2 Every 16-row strip in this image contains the $\sin \omega k$, where k is the index along the horizontal axis. The first row contains the fundamental frequency ω_0 and the m^{th} row the frequency $\omega = m\omega_0$. The range of m is between 0 and 64. This image shows only 1 to 32.

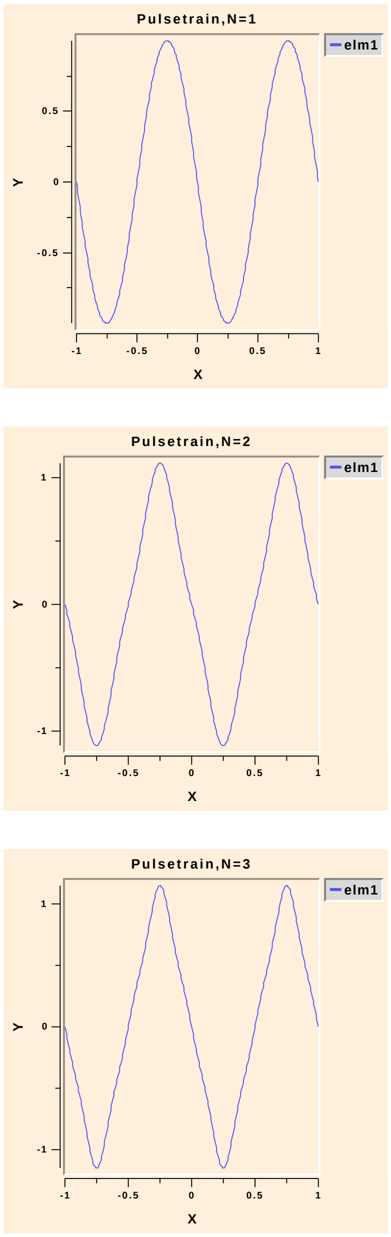

Fig. 2.3 These three plots show the reconstruction of the sawtooth pattern using one two and three Fourier components.

Now we know how sines (and cosines) look like and we cannot blame Laplace for rejecting Fourier's paper. They do not look like they can be used to compose other images. So let's take a few simple images that are likely to give Mr. Fourier a hard time and plot them. The first "image", actually a row of an image, is a sawtooth pattern (or a series of ridges). We can be sure that these nice round sines and cosines cannot reproduce the sharp corners. But Fig. 2.3 proves us wrong. Even with only the first three components the plots look very much like sawtooths. In fact the convergence is quite fast.

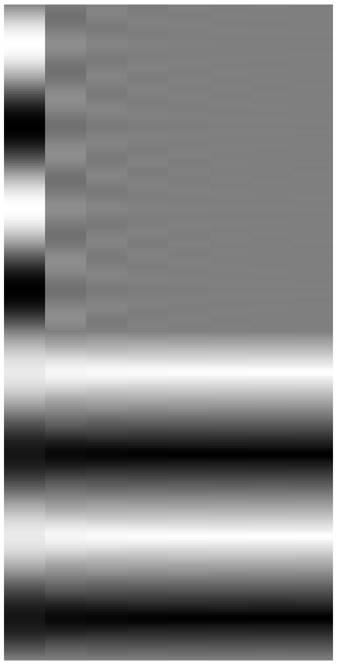

Fig. 2.4 The image on the left shows the a series of strips whose intensity approximates the sawtooth pattern. The first strip contains one component, so it is just a sine, the second two etc. The ridges become successively sharper. The image on the right shows the components.

The speed of the convergence can be verified experimentally by displaying a few images as we did but it can be predicted by looking at the Fourier components. Without getting into details how this is done let's look at the MediaMath code that performed the reconstruction:

```
for (i=1; i<=num; i++)
  {
    j = 2*i-1.0;
    res += ((-1.0)ˆi/jˆ2) * gsin(omega*j*vec_im);
  };
```

We can see that the weight of every component is proportional to 1/`j`^{2} which converges to zero reasonably quickly.

Convergence is not that fast for the next example though. We will show the reconstruction of the square pulse train which is a well known problematic case. No matter what we do there will always be a slight ringing; the flat white and flat black regions will not be exactly flat but slightly rippled. This is known as the Gibbs phenomenon.

In Fig. 2.5 we can see that the convergence is not that fast. After 8 components the reconstruction is not perfect.

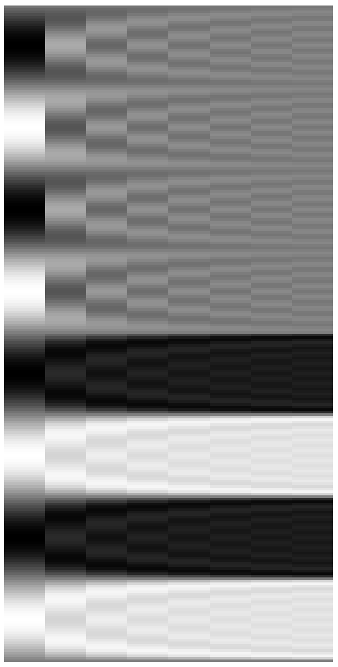

Fig. 2.5 Successive approximations of the square pulse train and its components. The slightly wavy pattern that is especially noticeable on the bright regions will not go away completely no matter how many components we use.

```
for (i=1; i<=num; i++)
  {
    j = 2*i-1.0;
    res += (1.0/j) * gsin(omega*j*vec_im);
  };
```

and we know what to suspect. The 1/`j` is known to converge slowly and it is not summable (the sum of 1/`j` does not converge).

3. Pictorial View of Filtering

What is the effect of filtering on an image? We could take a real image and filter it and see what the effects are. Unfortunately this will give us only the one side of the story. It will be hard to see much difference in an image filtered by two different lowpass filters for instance. But the difference might become apparent at a later stage of processing. So let’s see what is the effect if we apply the filter on an image like the one in Fig. 2.2 which is a series of sines.

Our first filter is the lowpass filter. This filter idealy has no effect on the low frequencies but completely eliminates the high frequencies. So if we want to get rid of all the high frequency Fourier components of an image while retaining the low frequency ones we use this filter (Fig. 3.1). The most common application of this filter is to reduce

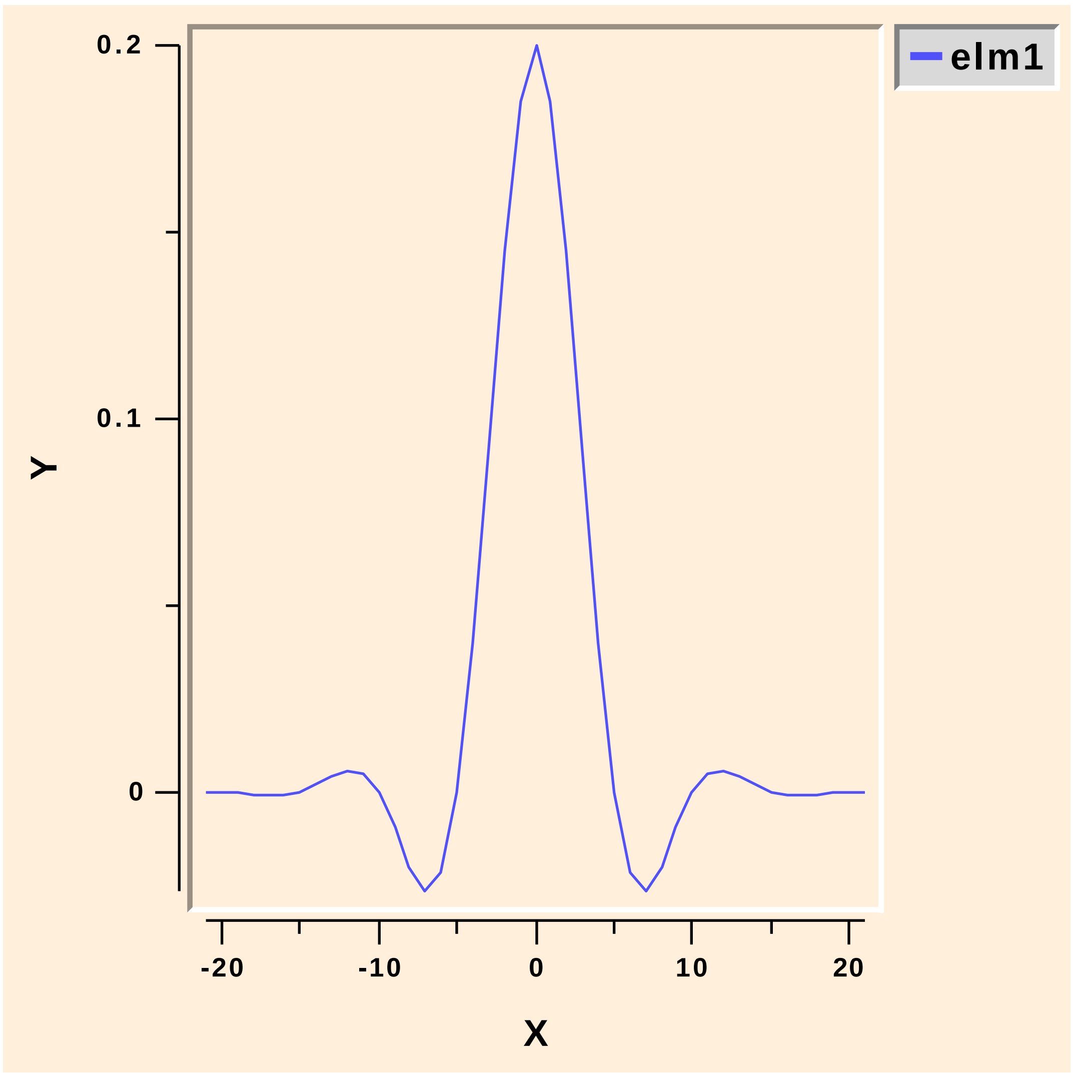

Fig. 3.1 A practical lowpass filter looks like this. The ideal low pass has infinite long tails and it is not practical.

the resolution of an image, when for example we want to create thumbnail images. Since the higher frequency components cannot be represented well in a lower resolution image it is better to eliminate them.

Ideally this filter should scale all the low frequency components by one (e.g. do nothing on them) and scale the high frequency ones by zero (e.g. eliminate them). And the transition at the cutoff frequency should be sharp. Idealy this is nice. In practice we ask for way too much. At best we can expect a gradual transition and some small deviation from one and zero at the passband and the stopband respectively. The sharper the transition and the smaller the deviation the bigger the template and the more costly the convolution. Try

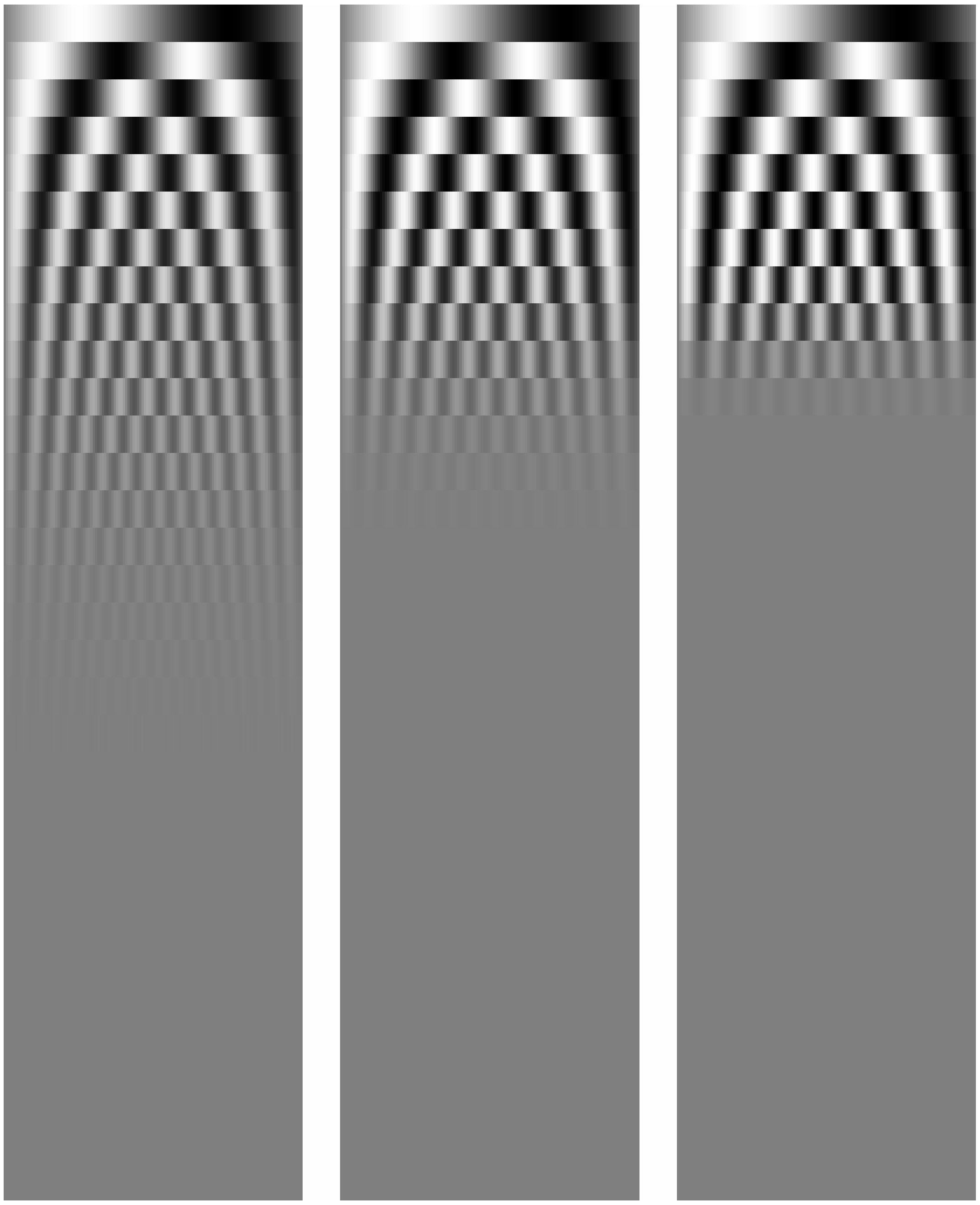

Fig. 3.2 The effect of three different lowpass filters whose transition zones are 1.2, .6 and .3 rad/pixel.

```
for (i=1; i<=5; i++)
  printf("%d0,mk_LP_tmpl(:reduce=3,:sigma=2*i)->vsize);
```

to see the sizes of various lowpass filters. The most expensive of these filters has a transition zone about 0.6 rad/pixel and requires 61 multiplications per pixel (twice that if the image is two dimensional). Given that ω has value between zero and 3.14 rad/pixel, this is not that sharp a transition. In Fig. 3.2 we see the effects of three such filters.

Fig. 3.3 shows the effect of a lowpass, a highpass and a bandpass filter on our favorite set of stripes. The highpass filter lets the high frequency components untouched

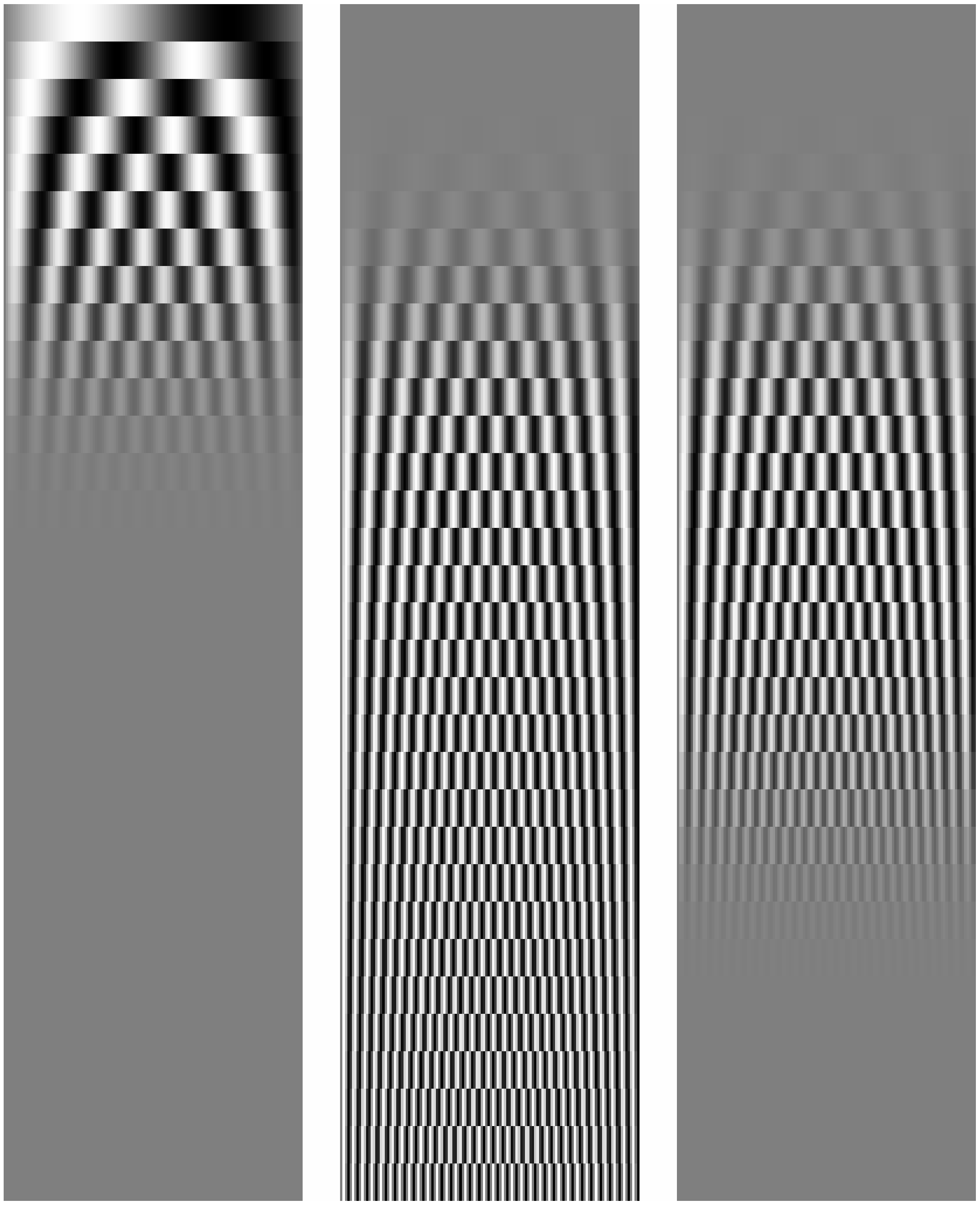

Fig. 3.3 The effect of a lowpass, a highpass and a bandpass filter.

and eliminates the low frequency ones. The bandpass lets the middle frequency filters untouched and eliminates the rest. Again the problems regarding the width of the transition zone are the same as the lowpass filters. In fact the filters are very similar in many ways.

3.1.1. 2-D in Separable Directions

4. 2-D Fourier Transform

The Fourier transform can be applied to more than two dimensions although we will hardly need anything beyond two. The basic intuition is the same. Every signal can be decomposed into a sum of sines and cosines which we represent for convenience with the exponential function

$$e^{j(u_x x + u_y y)}. \tag{4.1}$$

Although it is tempting to use the wavelength of the sinewave as a representation, it is mathematicaly awkward. Eq. (4.1), for example would become

$$e^{j(\frac{2\pi}{\lambda_x}x + \frac{2\pi}{\lambda_y}y)}$$

and it is impossible to represent the overall wavelength as a vector. We will use exclusively the frequencies u_x and u_y along the corresponding axes which in physics are called wavenumbers. Among their advantages is that they can be thought of as a vector

$$\mathbf{u} = \begin{bmatrix} u_x \\ u_y \end{bmatrix}$$

and if we use a similar notation for x and y

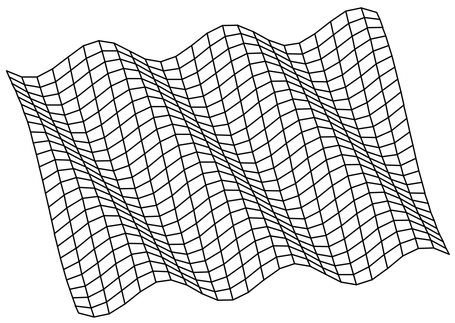

Fig. 4.1. Any image can be decomposed into a sum of sinusoidals like this of various wavelengths and orientation.

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

then Eq. (4.1) can be written as

$$e^{i\mathbf{x}\cdot\mathbf{u}}$$

where the dot $\cdot$ means dot product.

We can extend the one dimensional version of Fourier transform using the above definition of exponential and get the transform of a function $f(x)$

$$F(u_x, u_y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)e^{-j(u_x x + u_y y)}dx\,dy$$

which can be rewritten as

$$F(u_x, u_y) = \int_{-\infty}^{\infty} e^{-ju_x x}\left[\int_{-\infty}^{\infty} f(x, y)e^{-ju_y y}dy\right]dx$$

and we immediately notice that the quantity in the square brackets is the one dimensional Fourier transform of $f(x, y)$ as if x was a constant. This implies that we can take the Fourier transform of an image by replacing every column of it with the Fourier transform of the column and then repeating the same for every row. Due to the duality of the Fourier transform we can do the same for the inverse transform, e.g. apply it first on the columns and the on the rows

$$f(x, y) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{ju_x x}\left[\frac{1}{2\pi}\int_{-\infty}^{\infty} F(u_x, u_y)e^{ju_y y}du_y\right]du_x$$

$$f(x, y) = \frac{1}{4\pi^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u_x, u_y)e^{j(u_x x + u_y y)}du_x\,du_y$$

We can also write the same two equations using the vector notation

$$\begin{aligned} F(\mathbf{u}) &= \int f(\mathbf{x})e^{-j\mathbf{u}\cdot\mathbf{x}}d\mathbf{x} \\ f(\mathbf{x}) &= \frac{1}{4\pi^2}\int F(\mathbf{u})e^{j\mathbf{u}\cdot\mathbf{x}}d\mathbf{u} \end{aligned} \tag{4.2}$$

4.1. Properties of 2-D Fourier Transform

Practicaly everything that is true for the one dimensional version, will be true for the two dimensions as well. If a function $f(x, y)$, for instance, is periodic with periods T_x and T_y it will have a discrete Fourier transform

$$F(u_x, u_y) = \sum_k\sum_l F_{kl}\delta(u_x - kT_x)\delta(u_y - lT_y)$$

The same is true for the most important property, the convolution property. If function r is the convolution of f and g, $r = f(*)g$, then the Fourier transform of r is $R = F \cdot G$. The definition of convolution is a straightforward extension of the formula for one dimension

$$[f(*)g](x_0, y_0) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y)g(x_0 - x, y_0 - y)dx\,dy \tag{4.3}$$

which can also be written in vector form

$$[f(*)g](\mathbf{x_0}) = \int_{-\infty}^{\infty} f(\mathbf{x})g(\mathbf{x_0} - \mathbf{x})d\mathbf{x}$$

4.2. Convolution Templates

A large number of image operations are convolutions like smoothing, edge enhancement, derivatives of any kind, interpolation etc. But convolutions are also very costly operations, so in the end a very large percentage of the time spent processing images is due to convolutions and any small improvement in the performance of convolutions would have significant effects on the overall performance of a system.

It is almost always more convenient to handle continuous convolutions when we do a mathematical analysis of a Vision algorithm but when we implement convolutions we have no other option than discrete convolutions. The definition is straightforward translation of the continuous version. In practice we never have to compute the convolution of an image with another image, but only between an image and a template to get another image and sometimes between two templates in which case we get another template.

An obvious question is why we have different names for image and templates since both are two dimensional data structures. There are many reasons for that. First the $[0, 0]$ of an image is in the upper left corner whereas the $[0, 0]$ of a template can be anywhere. Second images are normaly things recorded by cameras, scanners etc and can be viewed and understood by humans but templates are just collections of numbers and the closest intuitive parallel is that of a painting brush. But the differences do not stop here. The two are treated differently when we apply various operators on them. Images are considered periodic in both directions, so an image represents a single tile of an infinite wall. A template is considered non periodic and it is zero outside its region of definition. And usualy a template is smaller than an image.

Let's look at the case where we convolve an image with a template. To convolve an image I_{ij}, $0 \le i \le i_{\max}$, $0 \le j \le j_{\max}$, with a template t_{kl}, $k_{\min} \le k \le k_{\max}$, $l_{\min} \le l \le l_{\max}$ we apply the formula

$$[I(*)t]_{mn} = \sum_{ij} I_{ij} t_{(m-i),(n-j)} = \sum_{kl} I_{(m-k),(n-l)} t_{kl} \tag{4.4}$$

While we normaly distinguish between images and templates, the mathematical formulas do not. Also the formula does not specify what happens at the borders.

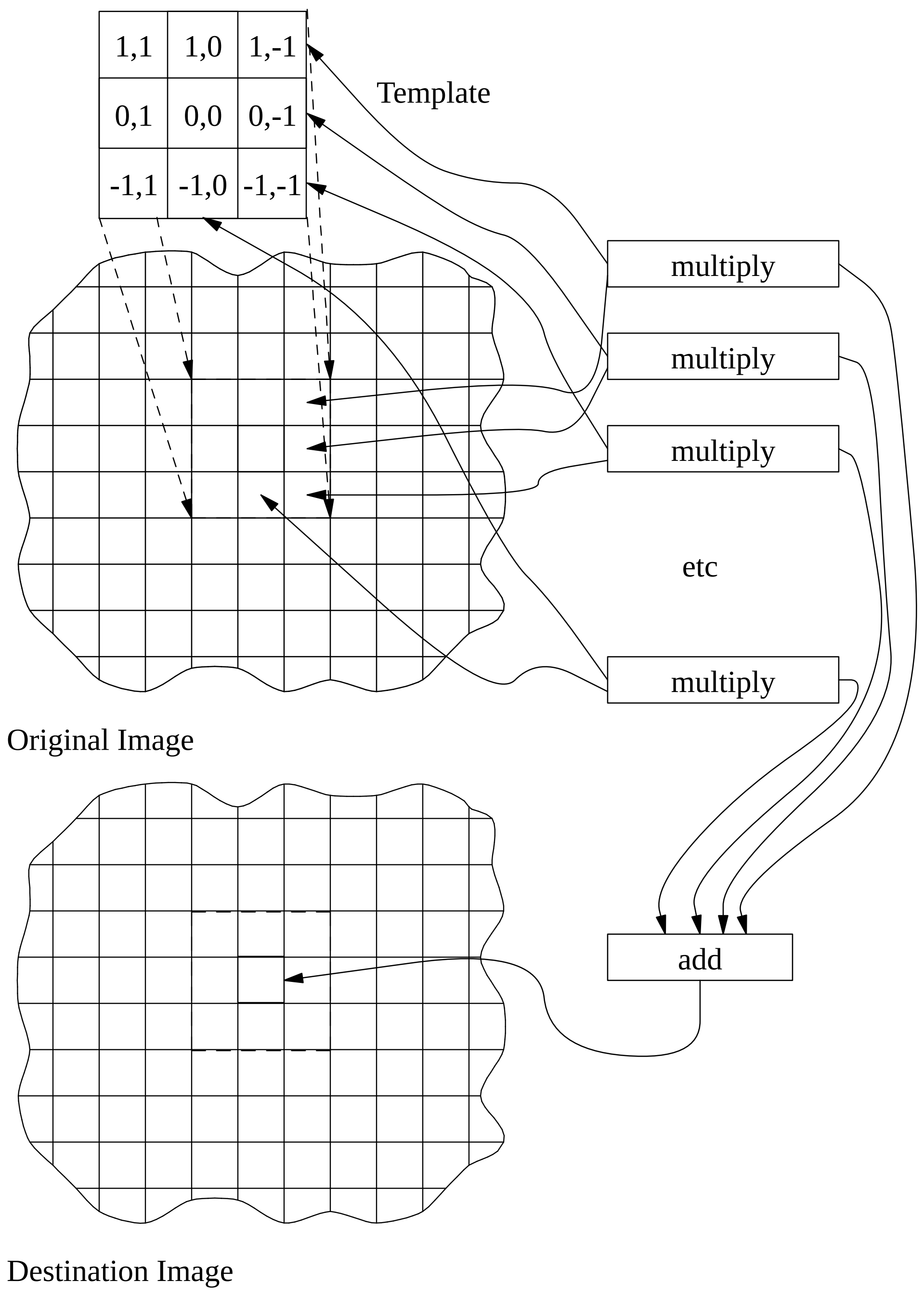

Fig. 4.2. To compute the value of a pixel of a convolution, we take the template, flip it, move it to the corresponding pixel on the source image, multiply the corresponding pixels of the image and the template and then compute the sum of the results.

We can visualize the convolution process between an image I_{ij} and a template t_{kl} easily. Usually the template is much smaller (Fig. 4.2). If we want to compute the value of pixel i, j of the convolution, we take the template, flip it (notice that the indices grow to the left and up in the template in Fig. 4.2) and put it on pixel i, j on the original image (marked with bold lines in Fig. 4.2) so that the $0, 0$ pixel of the template is on top of pixel i, j of the image. The next step is to multiply every pixel of the template with the corresponding pixel on the image and assuming our template is 3×3 we get 9 such products. We sum up these nine numbers and the result is the value stored in pixel i, j of the resulting image.

It is obvious that the cost of such a convolution is nine multiplications and 8 additions per pixel of the image and this can be easily generalized for larger templates. And exactly the same things happen when we convolve a template with another template.

But we have said nothing so far about the borders. What happens when one or more rows of columns of the template are outside the image as when we calculate the border pixels of a convolution? In various image processing libraries there are the following approaches, none of which is the best for everything.

Trim it

This would result in a smaller image as target pixels that require source pixels outside the original image are not computed. This is fine as long as we do not mind an image that shrinks after every convolution. It is at best mathematicaly inconvenient to add or multiply images of different sizes. So this is used mainly for one shot processing with small templates.

Zero padding

It assumes that the source image is surrounded by zeros. This looks much better but it has problems with mathematical inconsistency. We know from mathematics that we can apply convolutions in any order and get the same result, e.g. we can convolve an image with template a first and then with template b and the result is the same as convolving with template b first and then with a. But unless we keep a few extra pixels outside the orginal image (which means let the image grow) the order of convolutions is significant at the borders. Zero padding is used very often for image processing packages that care more about the cosmetics of the image rather than the mathematical consistency. This strategy is also used in MediaMath when convolving two templates, but the extra pixels generated are kept. So templates tend to grow with convolutions.

Pixel replication

The last pixel on the same row or column is replicated. It is just a better version of zero padding but suffers from the same mathematical inconsistency.

Image tiling

The image is assumed that it is periodic and that the image at hand is just a single period. It is perfectly consistent mathematicaly. But can create some unwelcome visual artifacts on pictures. If we smooth the picture of a person with a red hat and the hat reaches up to the top border of the image, we might get a red glow at the bottom of the picture along with some complaints about color coordination by some mathematicaly challenged fashion experts. This strategy is used in convolution between images and templates in MediaMath.

So the rule for MediaMath is to have periodic images and non-periodic templates. Note however that MediaMath will generate an error if we explicitely address a pixel outside its area of definition.

4.3. Separable templates

It is obvious that convolutions are expensive and the cost increases with the size of the template. But it turns out that most templates that are actualy used belong to a special class of templates that are called *separable*. These are templates that have the following property

$$f(x,y) = f^{(1)}(x) f^{(2)}(y) \tag{4.5}$$

which can be written as

$$f(x,y) = \left(f^{(1)}(x)\delta(y) \right) (*) \left(f^{(2)}(y)\delta(x) \right)$$

because if we start from the right hand side and apply Eq. (4.3)

$$\left[\left(f^{(1)}(x)\delta(y) \right) (*) \left(f^{(2)}(y)\delta(x) \right) \right] (x_0, y_0) = \int \int f^{(1)}(x)\delta(y) f^{(2)}(y_0 - y)\delta(x_0 - x) dx\, dy =$$

$$\int f^{(1)}(x)\delta(x_0 - x) \left[\int f^{(2)}(y_0 - y)\delta(y) dy \right] dx =$$

$$\int f^{(1)}(x)\delta(x_0 - x) f^{(2)}(y_0) dx =$$

$$f^{(1)}(x_0) f^{(2)}(y_0) = f(x_0, y_0)$$

Using this identity we can break any convolution of template $f(x,y)$ with an image I in two

$$I(x,y)(*) f(x,y) = \left(I(x,y)(*) \left(f^{(1)}(x)\delta(y) \right) \right) (*) \left(f^{(2)}(y)\delta(x) \right)$$

The advantage of such a decomposition becomes obvious when we consider the discrete version of the template $f^{(1)}(x)\delta(y)$ which is $f^{(1)}{}_i \delta_j$. Recall that the discrete δ_j function is zero everywhere except for $j = 0$ where $\delta_0 = 1$. So if $f^{(1)}{}_i$ is nonzero for $i_{vmin} \le i \le i_{vmax}$ then the template $f^{(1)}{}_i \delta_j$ is only $(i_{vmax} - i_{vmax} + 1) \times 1$ and the resulting convolution costs a lot less. If for example we want a convolution with a separable 7×7 template, we could do it with 14 multiplications and 13 additions instead of $7^2 = 49$ multiplications and 48 additions.

A good example of a separable template is the gaussian

$$f(x,y,) = e^{-\frac{x_2+y_2}{2\sigma^2}} = e^{-\frac{x_2}{2\sigma^2}} \; e^{-\frac{y_2}{2\sigma^2}}$$

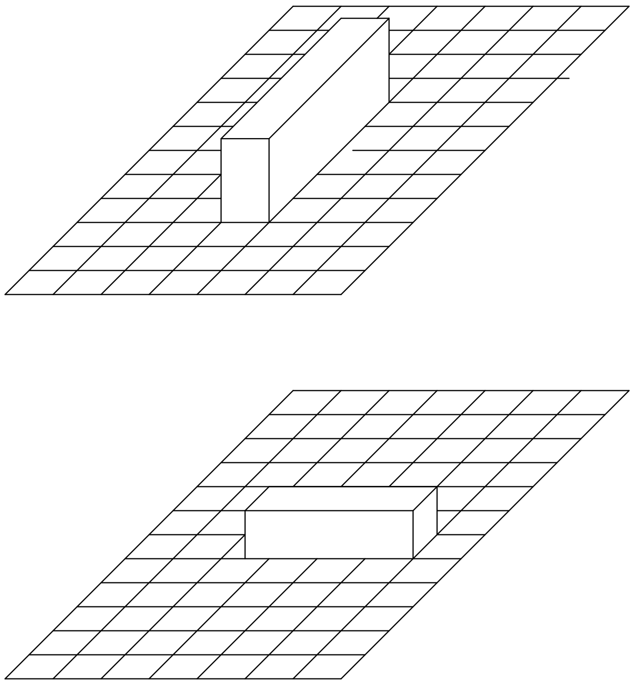

Fig. 4.3. A template that has the form $f^{(2)}(y)\delta(x)$, has a discrete equivalent that is one pixel wide and several pixels long (top) and $f^{(1)}(x)\delta(y)$ is one pixel long and several pixels wide (bottom).

which can be show, by the way, that it is the only separable function that at the same time is circularly symmetric.