Correlation and Convolution

Class Notes for CMSC 426, Fall 2005

David Jacobs

Introduction

Correlation and Convolution are basic operations that we will perform to extract information from images. They are in some sense the simplest operations that we can perform on an image, but they are extremely useful. Moreover, because they are simple, they can be analyzed and understood very well, and they are also easy to implement and can be computed very efficiently. Our main goal is to understand exactly what correlation and convolution do, and why they are useful. We will also touch on some of their interesting theoretical properties; though developing a full understanding of them would take more time than we have.

These operations have two key features: they are *shift-invariant*, and they are *linear*. Shift-invariant means that we perform the same operation at every point in the image. Linear means that this operation is linear, that is, we replace every pixel with a linear combination of its neighbors. These two properties make these operations very simple; it's simpler if we do the same thing everywhere, and linear operations are always the simplest ones.

We will first consider the easiest versions of these operations, and then generalize. We'll make things easier in a couple of ways. First, convolution and correlation are almost identical operations, but students seem to find convolution more confusing. So we will begin by only speaking of correlation, and then later describe convolution. Second, we will start out by discussing 1D images. We can think of a 1D image as just a single row of pixels. Sometimes things become much more complicated in 2D than 1D, but luckily, correlation and convolution do not change much with the dimension of the image, so understanding things in 1D will help a lot. Also, later we will find that in some cases it is enlightening to think of an image as a continuous function, but we will begin by considering an image as *discrete*, meaning as composed of a collection of pixels.

Notation

We will use uppercase letters such as *I* and *J* to denote an image. An image may be either 2D (as it is in real life) or 1D. We will use lowercase letters, like *i* and *j* to denote indices, or positions, in the image. When we index into an image, we will use the same conventions as Matlab. First, that means that the first element of an image is indicated by 1 (not 0, as in Java, say). So if *I* is a 1D image, *I(1)* is its first element. Second, for 2D images we give first the row, then the column. So *I(3,6)* is the pixel in the third row of the image, and the sixth column.

An Example

One of the simplest operations that we can perform with correlation is local averaging. As we will see, this is also an extremely useful operation. Let's consider a simple averaging operation, in which we replace every pixel in a 1D image by the average of that pixel and its two neighbors. Suppose we have an image *I* equal to:

5	4	2	3	7	4	6	5	3	6

Averaging is an operation that takes an image as input, and produces a new image as output. When we average the fourth pixel, for example, we replace the value 3 with the average of 2, 3, and 7. That is, if we call the new image that we produce *J* we can write: *J(4) = (I(3)+I(4)+I(5))/3 = (2+3+7)/3 = 4.* Or, for example, we also get: *J(3) = (I(2)+I(3)+I(4))/3 = (4+2+3)/3 = 3.* Notice that every pixel in the new image depends on the pixels in the old image. A possible error is to use *J(3)* when calculating *J(4).* Don't do this; *J(4)* should only depend on *I(3)*, *I(4)* and *I(5)*. Averaging like this is shift-invariant, because we perform the same operation at every pixel. Every new pixel is the average of itself and its two neighbors. Averaging is linear because every new pixel is a linear combination of the old pixels. This means that we scale the old pixels (in this case, we multiply all the neighboring pixels by 1/3) and add them up. This example illustrates another property of all correlation and convolution that we will consider. The output image at a pixel is based on only a small neighborhood of pixels around it in the input image. In this case the neighborhood contains only three pixels. Sometimes we will use slightly larger neighborhoods, but generally they will not be too big.

Boundaries: We still haven't fully described correlation, because we haven't said what to do at the boundaries of the image. What is *J(1)*? There is no pixel on its left to include in the average, ie., *I(0)* is not defined. There are four common ways of dealing with this issue.

.	.	.	*0*	*0*	5	4	2	3	7	4	6	5	3	6	*0*	*0*	.	.	.

In the first method of handling boundaries, the original image is padded with zeros (in red italics).

The first way is to imagine that *I* is part of an infinitely long image which is zero everywhere except where we have specified. In that case, we have *I(0) = 0,* and we can say: *J(1) = (I(0) + I(1) + I(2))/3 = (0 + 5 + 4)/3 = 3.* Similarly, we have: *J(10) = (I(9)+I(10)+I(11))/3 = (3 + 6 + 0)/3 = 3.*

.	.	.	*5*	*5*	5	4	2	3	7	4	6	5	3	6	*6*	*6*	.	.	.

In the second method of handling boundaries, the original image is padded with the first and last values (in red italics).

The second way is to also imagine that *I* is part of an infinite image, but to extend it using the first and last pixels in the image. In our example, any pixel to the left of the first pixel in *I* would have the value 5, and any pixel to the right of the last pixel would have the value 6. So we would say: *J(1) = (I(0) + I(1) + I(2))/3 = (5 + 5 + 4)/3 = 4 2/3,* and *J(10) = (I(9)+I(10)+I(11))/3 = (3 + 6 + 6)/3 = 5.*

.	.	.	*3*	*6*	5	4	2	3	7	4	6	5	3	6	*5*	*4*	.	.	.

In the third method of handling boundaries, the original image is repeated cyclically (in red italics).

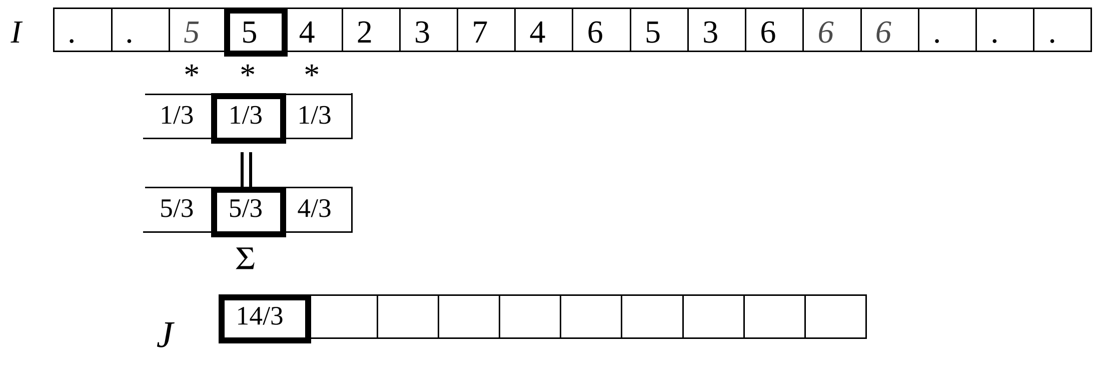
Third, we can imagine the image as being like a circle, so that the pixel values repeat over and over again. The pixel to the left of the first pixel, then, would be the last pixel in the image. That is, in our example, we would define *I(0)* to be *I(10).* Then we would have *J(1) = (I(0) + I(1) + I(2))/3= (I(10) + I(1) + I(2))/3 = (6 + 5 + 4)/3 = 5,* and *J(10) = (I(9)+I(10)+I(11))/3 = (I(9)+I(10)+I(1))/3 = (3 + 6 + 5)/3 = 4 2/3.*

Finally, we can simply say that the image is undefined beyond the values that we have been given. In that case, we cannot compute any average that uses these undefined values, so *J(1)* and *J(10)* will be undefined, and *J* will be smaller than *I.*

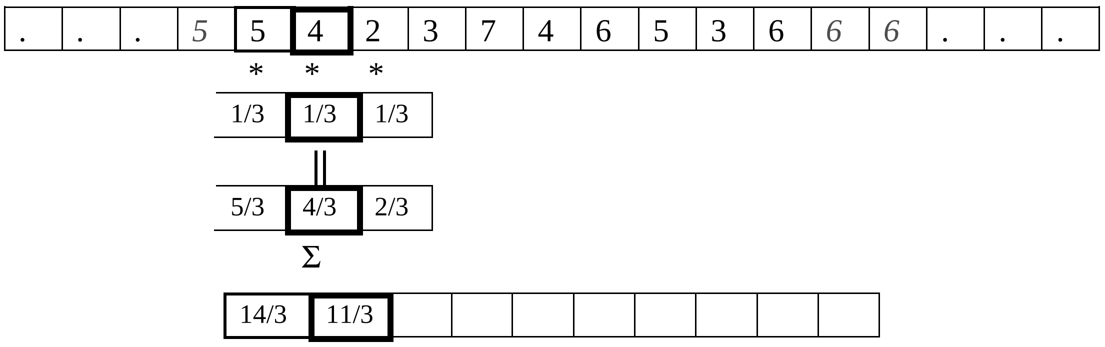
These four methods have different advantages and disadvantages. If we imagine that the image we are using is just a small window on the world, and we want to use values outside the boundary that are most similar to the values that we would have obtained if we'd taken a bigger picture, than the second approach probably makes the most sense. That is, if we had to guess at the value of *I(0),* even though we can't see it, the value we can see in *I(1)* is probably a pretty good guess. In this class, unless we explicitly state otherwise, you should use the second method for handling boundaries.

Correlation as a Sliding, Windowed Operation

We're now going to look at the same averaging operation in a slightly different way which is more graphical, and perhaps more intuitive to generalize. In averaging, for a specific pixel we multiply it and its neighbors by 1/3 each, and then add up the three resulting numbers. The numbers we multiply, (1/3, 1/3, 1/3) form a *filter.* This particular filter is called a *box* filter. We can think of it as a 1x3 structure that we slide along the image. At each position, we multiply each number of the filter by the image number that lies underneath it, and add these all up. The result is a new number corresponding to the pixel that is underneath the center of the filter. The figure below shows us producing *J(1)* in this way.



To produce the next number in the filtered image, we slide the filter over a pixel, and perform the same operation.



We continue doing this until we have produced every pixel in *J*. With this view of correlation, we can define a new averaging procedure by just defining a new filter. For example, suppose instead of averaging a pixel with its immediate neighbors, we want to average each pixel with immediate neighbors and their immediate neighbors. We can define a filter as (1/5, 1/5, 1/5, 1/5, 1/5). Then we perform the same operation as above, but using a filter that is five pixels wide. The first pixel in the resulting image will then be: *J(1) = (I(-1)/5 + I(0)/5 + I(1)/5 + I(2)/5 + I(3)/5) = 1+1+1+4/5 + 2/5 = 4 1/5.*

A Mathematical Definition for Correlation

It's helpful to write this all down more formally. Suppose *F* is a correlation filter. It will be convenient notationally to suppose that *F* has an odd number of elements, so we can suppose that as it shifts, its center is right on top of an element of *I*. So we say that *F* has *2N+1* elements, and that these are indexed from -*N* to *N*, so that the center element of *F* is *F(0)*. Then we can write:

$$F \circ I(x) = \sum_{i=-N}^{N} F(i) I(x+i)$$

where the circle denotes correlation. With this notation, we can define a simple box filter as:

$$F(i) = \begin{array}{lll} \frac{1}{3} & \text{for} & i = -1,0,1 \\ 0 & \text{for} & i \neq -1,0,1 \end{array}$$

Constructing an Filter from a Continuous Function

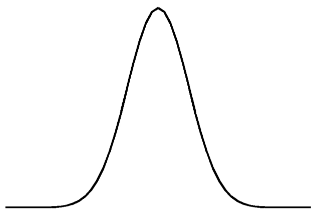
It is pretty intuitive what a reasonable averaging filter should look like. Now we want to start to consider more general strategies for constructing filters. It commonly occurs that we have in mind a continuous function that would make a good filter, and we want to come up with a discrete filter that approximates this continuous function. Some reasons for thinking of filters first as continuous functions will be given when we talk about the

Fourier transform. But in the mean time we'll give an example of an important continuous function used for image smoothing, the Gaussian.

A one-dimensional Gaussian is:

$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-(x-\mu)^2 \Big/ 2\sigma^2 \right)$$

This is also known as a Normal distribution. Here μ is the mean value, and σ is the variance. Here's a plot of a Gaussian:



The mean, μ, gives the location of the peak of the function. The parameter σ controls how wide the peak is. As σ gets smaller the peak becomes narrower.

The Gaussian has a lot of very useful properties, some of which we'll mention later. When we use it as a filter, we can think of it as a nice way to average. The averaging filter that we introduced above replaces each pixel with the average of its neighbors. This means that nearby pixels all play an equal role in the average, and more distant pixels play no role. It is more appealing to use the Gaussian to replace each pixel with a weighted average of its neighbors. In this way, the nearest pixels influence the average more, and more distant pixels play a smaller and smaller role. This is more elegant, because we have a smooth and continuous drop-off in the influence of pixels on the result, instead of a sudden, discontinuous change. We will show this more rigorously when we discuss the Fourier transform. When we use a Gaussian for smoothing, we will set μ *=0,* because we want each pixel to be the one that has the biggest effect on its new, smoothed value. So our Gaussian has the form:

$$G_0(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-x^2 \Big/ 2\sigma^2 \right)$$

σ will serve as a parameter that allows us to control how much we smooth the image (that is, how big a neighborhood we use for averaging). The bigger σ is, the more we smooth the image.

We now want to build a discrete filter that looks like a Gaussian. We can do this by just evaluating the Gaussian at discrete locations. That is, although *G* is defined for any continuous value of *x*, we will just use its values at discrete locations (… -3, -2, -1, 0, 1, 2, 3…). One way to think about this is that we are approximating our original, continuous function with a piecewise constant function, where the pieces are each one pixel wide.

However, we cannot evaluate *G* at every integer location, because this would give us a filter that is infinitely long, which is impractical. Fortunately, as *x* gets away from 0, the value of *G(x)* will approach 0 pretty quickly, so it will be safe to ignore values of *x* with a high absolute value. There is no hard and fast rule about when it is safe to ignore some values of *G(x)*, but a reasonable rule of thumb is to make sure we capture 99% of the function. What we mean by this is that the sum of values in our filter will contain at least 99% of what we would get in an infinitely long filter, that is, we choose *N* so that:

$$\frac{\sum_{-N}^{N} G(x)}{\sum_{-\infty}^{\infty} G(x)} > .99$$

There is one final point. An averaging filter should have all its elements add up to 1. This is what averaging means; if the filter elements don't add up to one, then we are not only smoothing the image, we are making it dimmer or brighter. The continuous Gaussian integrates to one (after all, it's a probability density function, and probabilities must add up to one), but when we sample it and truncate it, there is no guarantee that the values we get will still add up to one. So we must normalize our filter, meaning the elements of our filter will be calculated as:

$$F(i) = \frac{G(i)}{\sum_{-N}^{N} G(x)} \quad for\ i = -N...N$$

Taking Derivatives with Correlation

Averaging, or smoothing, is one of the most important things we can do with a filter. However, another very useful thing is taking a derivative. This allows us to measure how quickly an image is changing. Taking derivatives is something we usually think of doing with a continuous function. Technically, derivatives aren't even defined on a discrete image, because a derivative measures how quickly the image is changing over an interval, in the limit, as the interval becomes infinitesimally small. However, just as we were able to produce a discrete filter that was an approximation to a continuous Gaussian, we can discretely approximate a continuous derivative operator.

Intuitively, a derivative is found by taking the difference as we go from one part of a function to another. That is, it's the change in *y* divided by the change in *x*. So it's natural that a derivative filter will look something like: (-1/2 0 1/2). When we apply this filter, we get $J(i) = (I(i+1)-I(i-1))/2$. This is taking the change in *y*, that is, the image intensity, and dividing it by the change in *x*, the image position.

Let's consider an example. Suppose image intensity grows quadratically with position, so that *I(x)* = x^2. Then if we look at the intensities at positions 1, 2, 3, … they will look like:

1	4	9	16	25	36	.	.	.

If we filter this with a filter of the form (-1/2 0 1/2) we will get:

1 1/2	4	6	8	10	12	.	.	.

We know that the derivative of *I(x), dI/dx = 2x.* And sure enough, we have, for example, that *J(2) = 4, J(3) = 6, …*Notice that *J(1)* is not equal to *2* because of the way we handle the boundary, by setting *I(0) = 1* instead of *I(0) = 0.* So at the boundary, our image doesn't really reflect *I(x)* = x^2.

We could just as easily have used a filter like (0 -1 1), which computes the expression: *J(i) = (I(i+1)-I(i))/1*, or a filter (-1 1 0), which computes *J(i) = (I(i) – I(i-1))/1.* These are all reasonable approximations to a derivative. One advantage of the filter (-1/2 0 1/2) is that it treats the neighbors of a pixel symmetrically.

Matching with Correlation

Part of the power of correlation is that we can use it, and related methods, to find locations in an image that are similar to a template. To do this, think of the filter as a template; we are sliding it around the image looking for a location where the template overlaps the image so that values in the template are aligned with similar values in the image.

First, we need to decide how to measure the similarity between the template and the region of the image with which it is aligned. A simple and natural way to do this is to measure the sum of the square of the differences between values in the template and in the image. This increases as the difference between the two increases. For the difference between the filter and the portion of the image centered at *x,* we can write this as:

$$\begin{aligned}\sum_{i=-N}^{N}(F(i)-I(x+i))^2 &= \sum_{i=-N}^{N}\left(F^2(i)+I^2(x+i)-2F(i)I(x+i)\right)\\ &= \sum_{i=-N}^{N}\left(F^2(i)\right)+\sum_{i=-N}^{N}\left(I^2(x+i)\right)-2\sum_{i=-N}^{N}(F(i)I(x+i))\end{aligned}$$

As shown, we can break the Euclidean distance into three parts. The first part depends only on the filter. This will be the same for every pixel in the image. The second part is the sum of squares of pixel values that overlap the filter. And the third part is twice the negative value of the correlation between *F* and *I.* We can see that, all things being equal, as the correlation between the filter and the image increases, the Euclidean distance between them decreases. This provides an intuition for using correlation to match a template with an image. Places where the correlation between the two is high tend to be locations where the filter and image match well.

This also shows one of the weaknesses of using correlation to compare a template and image. Correlation can also be high in locations where the image intensity is high, even if it doesn't match the template well. Here's an example. Suppose we correlate the filter

3	7	5

with the image:

3	2	4	1	3	8	4	0	3	8	0	7	7	7	1	2

We get the following result:

40	43	39	34	64	85	52	27	61	65	59	84	105	75	38	27

Notice that we get a high result (85) when we center the filter on the sixth pixel, because (3,7,5) matches very well with (3,8,4). But, the highest correlation occurs at the 13th pixel, where the filter is matched to (7,7,7). Even though this is not as similar to (3,7,5), its magnitude is greater.

One way to overcome this is by just using the sum of square differences between the signals, as given above. This produces the result:

25	26	26	41	29	2	59	54	34	26	78	13	20	32	61	38

We can see that using Euclidean distance, (3,7,5) best matches the part of the image with pixel values (3,8,4). The next best match has values (0,7,7), but this is significantly worse.

Another option is to perform *normalized correlation* by computing::

$$\frac{\sum_{i=-N}^{N}\left(F(i)I(x+i)\right)}{\sqrt{\sum_{i=-N}^{N}\left(I(x+i)\right)^2}\sqrt{\sum_{i=-N}^{N}\left(F(i)\right)^2}}.$$

This measure of similarity is similar to correlation, but it is invariant to scaling the template or a corresponding region in the image. If we scale *I(x-N)…I(x+N),* by a single, constant factor, this will scale the numerator and denominator by the same amount. Consequently, for example, (3,7,5) will have the same normalized correlation with (7,7,7) that it would have with (1,1,1). When we perform normalized correlation between (3,7,5) and the above image we get:

.946	.877	.934	.73	.81	.989	.64	.59	.78	.835	.61	.931	.95	.83	.57	.988

Now, the region of the image that best matches the filter is (3,8,4). However, we also get a very good match between (3,7,5) and the end of the image, which, when we replicate the last pixel at the boundary, is (1,2,2). These are actually quite similar, up to a scale factor. Keep in mind that normalized correlation would say that (3,6,6) and (1,2,2) match perfectly. Normalized correlation is particularly useful when some effect, such as changes in lighting or the camera response, might scale the intensities in the image by an unknown factor.

Correlation as an inner product

One way to get a better intuition for correlation and normalized correlation for matching is to consider these comparisons as based on an inner product. To do this, we can consider our filter as a vector, *F = (F(-N), F(-N+1)…F(N))*. We are comparing this to a portion of an image, which we can also think of as a vector: $I_x = (I(x-N), I(x-N+1), \ldots I(x+N))$. Then, when we compute the correlation between *F* and *I* at the position *x*, we are computing $<F, I_x>$. *F* and I_x are vectors in a (2N+1)-dimensional space, but still, everything we've learned about inner products applies. In particular,

$$\langle F, I_x \rangle = \|F\| \|I_x\| \cos\theta$$

where θ is the angle between *F* and I_x. This means that the correlation between the two depends on the magnitude of each vector, and on the angle between them. For *F* and I_x of fixed magnitudes, the correlation between them is maximized when the angle between them is zero, which occurs when *F* and I_x are scaled versions of each other. The correlation decreases as the angle between them increases. Normalized correlation is correlation divided by $||F||$ and $||I_x||$, so it just computes cos θ. So normalized correlation only computes the angle between *F* and I_x. This serves as a good measure of their similarity if we are not interested in whether their magnitudes are similar.

2D Correlation

Images are 2D, so we really want to perform correlation in 2D. The basic idea is the same, except that the image and filter are now 2D. We can suppose that our filter is square and has an odd number of elements, so it is represented by a (2N+1)x(2N+1) matrix. We don't lose anything with these assumptions, because we can take any filter and pad it with zeros to make it square with an odd width. Padding a filter with zeros does not change its behavior.

Given a square filter, we can compute the results of correlation by aligning the center of the filter with a pixel. Then we multiply all overlapping values together, and add up the result. We can write this as:

$$F \circ I(x,y) = \sum_{j=-N}^{N} \sum_{i=-N}^{N} F(i,j) I(x+i, y+j)$$

Example: We can perform averaging of a 2D image using a 2D box filter, which, for a 3x3 filter, will look like:

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

F

Below we show an image, *I,* and the result of applying the box filter above to *I* to produce the resulting image, *J.*

8	3	4	5
7	6	4	5
4	5	7	8
6	5	5	6

I

8	*8*	*3*	*4*	*5*	*5*
8	8	3	4	5	*5*
7	7	6	4	5	*5*
4	4	5	7	8	*8*
6	6	5	5	6	*6*
6	*6*	*5*	*5*	*6*	*6*

I with padded boundaries

6.44	5.22	4.33	4.67
5.78	5.33	5.22	5.67
5.56	5.44	5.67	6
5.22	5.33	5.78	6.33

J = F*o*I

Separable filters: Generally, 2D correlation is more expensive than 1D correlation because the sizes of the filters we use are larger. For example, if our filter is (N+1)x(N+1) in size, and our image contains MxM pixels, then the total number of multiplications we must perform is $(N+1)^2M^2$. However, with an important class of filters, we can save on this computation. These are separable filters, which can be written as a combination of two smaller filters. For example, in the case of the box filter, we have:

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & 0 \end{bmatrix} \circ \begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 \end{bmatrix}$$

That is, the box filter is equal to the correlation of two filters that have a row and a column of equal values, and all other values equal to zero (here we are handling boundaries by using 0 values outside the boundary). When this is the case, correlating an image with the box filter produces the same result as correlating it separately with the two filters on the right side of the equals sign. Intuitively, this makes sense. Correlating with the rightmost filter averages each pixel with its neighbors on the same row. The second correlation then averages these resulting pixels with those neighbors in the same column, which are already averages of neighbors on the same row. This produces an average of all neighboring pixels.

The advantage of separating a filter in this way is that when we perform correlation with the matrices on the right, we don't need to actually perform any multiplications where there are zeros. In fact, it is more convenient to simply write these filters as 3x1 and 1x3, ignoring the zeros. This means that for each pixel in the image, we only need to perform 6 multiplications instead of 9. In general, if we can separate a (N+1)x(N+1) filter into two filters that are (N+1)x1 and 1x(N+1), the total work we must do in correlation is reduced from $(N+1)^2M^2$ to $2(N+1)M^2$.

Gaussian filtering has the same separability. For a 2D Gaussian, we use a function that is rotationally symmetric, where the height of the filter falls off as a Gaussian function of the distance from the center. This has the form:

$$G_0(x,y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\left(x^2+y^2\right) \big/ 2\sigma^2 \right)$$

This can be separated because it can be expressed as:

$$G_0(x,y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\left(x^2+y^2\right) \big/ 2\sigma^2 \right) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-x^2 \big/ 2\sigma^2 \right) \exp\left(-y^2 \big/ 2\sigma^2 \right)$$

It is the product of two components, one of which only depends on *x*, while the other only depends on *y*. The first component can be expressed with a filter that has a single row, and the second can be expressed with a filter that has a single column.

Convolution

Convolution is just like correlation, except that we flip over the filter before correlating. For example, convolution of a 1D image with the filter (3,7,5) is exactly the same as correlation with the filter (5,7,3). We can write the formula for this as:

$$F * I(x) = \sum_{i=-N}^{N} F(i) I(x-i)$$

In the case of 2D convolution we flip the filter both horizontally and vertically. This can be written as:

$$F * I(x,y) = \sum_{j=-N}^{N} \sum_{i=-N}^{N} F(i,j) I(x-i, y-j)$$

Notice that correlation and convolution are identical when the filter is symmetric.

The key difference between the two is that convolution is associative. That is, if *F* and *G* are filters, then $F*(G*I) = (F*G)*I$. If you don't believe this, try a simple example, using $F=G=(-1\ 0\ 1)$, for example. It is very convenient to have convolution be associative. Suppose, for example, we want to smooth an image and then take its derivative. We could do this by convolving the image with a Gaussian filter, and then convolving it with a derivative filter. But we could alternatively convolve the derivative filter with the Gaussian to produce a filter called a *Difference of Gaussian (DOG)*, and then convolve this with our image. The nice thing about this is that the DOG filter can be precomputed, and we only have to convolve one filter with our image.

In general, people use convolution for image processing operations such as smoothing, and they use correlation to match a template to an image. Then, we don't mind that correlation isn't associative, because it doesn't really make sense to combine two templates into one with correlation, whereas we might often want to combine two filter together for convolution. When discussing the Fourier series as a way of understanding filtering, we will use convolution, so that we can introduce the famous *convolution theorem.*

The Fourier Series (this section is a bit more advanced)

The Fourier Series gives us a very important and useful way of representing an image. While it is a very powerful representation for many reasons, we will mainly introduce it as a way of understanding the effect of convolution. First, we will consider what properties a good image representation should have by looking at familiar representations. Then we will describe the Fourier Series, which has these properties. Finally, we will describe the convolution theorem, which helps us better understand the effect of convolution with different filters.

Orthonormal Basis for Vectors

How do we represent points in 2D? Using their *x* and *y* coordinates. What this really means is that we write any point as a linear combination of two vectors (1,0) and (0,1).

$(x,y) = x(1,0) + y(0,1)$.

These vectors form an *orthonormal basis* for the plane. That means they are orthogonal (ie, perpendicular), $\langle(1,0), (0,1)\rangle = 0$, and of unit magnitude. Why is an orthonormal basis a good representation? There are many reasons, but two are:

1) Projection. To find the *x* coordinate, we take $\langle(x,y), (1,0)\rangle$. In general, when we have an orthonormal basis for a space, we can describe any point with coordinates, and find each coordinate of a point by taking an inner product with the basis vector for this coordinate.
2) Pythagorean theorem. $\|(x,y)\|^2 = x^2 + y^2$. When we have an orthonormal basis, the squared length of a vector is the sum of the square of its coordinates in that basis.

As an example of a less attractive basis, supposed we represented points using the two basis vectors (1,0) and (1,1). These are not orthogonal, because <(1,0),(1,1)> = 1. Still, given any point, (x,y), we could always represent it as:

(x,y) = u(1,0) + v(1,1)

for some choice of u and v. We can compute u and v by taking v = y, u = x – y. For example, we would represent the point (7,3) as 4(1,0) + 3(1,1). So rather than represent the point with the Euclidean coordinates (7,3), we can represent this same point in this new basis with the coordinates (4,3). But note that the Pythogorean theorem wouldn’t hold. And, if I give you two basis vectors that are not orthonormal, it is not so easy to come up with a formula for determining the coordinates of a point using these basis vectors (especially as we look at higher dimensions).

The Fourier Series

The Fourier series is a good representation because it provides an orthonormal basis for images. We want to explore it in the simplest possible setting, to get the key intuitions. This is one case in which it is easier to get the idea by working with continuous functions, rather than discrete ones. So, instead of thinking of an image as a list of pixels, we’ll think of it as a continuous function, *I(x)*. To simplify, we’ll also only do things in 1D, but all this extends to 2D images. We will also consider only periodic functions that go from 0 to 2π, and then repeat over and over again. This is equivalent to the third way that we describe above for handling pixels outside of the boundary of images. The main thing I’ll do to make things easier, though, is to be very sloppy mathematically, skipping precise conditions and giving intuitions instead of complete proofs.

The following functions provide an orthonormal basis for functions:

$$\frac{1}{\sqrt{2\pi}}, \frac{\cos(kx)}{\sqrt{\pi}}, \frac{\sin(kx)}{\sqrt{\pi}} \quad for\ k = 1,2,3,...$$

These functions form the *Fourier Series.* The first one is a constant function, and the others are just sines and cosines of increasing frequency.

To define things like whether a set of functions are orthonormal, we need to define an inner product between two functions. We do this with a continuous version of the inner product. With discrete vectors, to compute the inner product we multiply together the values of matching coordinates, and then add up the results. We do the same thing with continuous functions; we multiply them together, and integrate (add) the result. So the inner product between two functions, f(x) and g(x), defined in the domain from 0 to 2π, is:

$$\langle f,g\rangle = \int_0^{2\pi} f(x)g(x)dx$$

When we say that a function, f, has unit magnitude, we mean that <f,f> = 1. When we say that two functions are orthogonal, we mean that <f,g> = 0. One reason that these are sensible definitions is that if we approximate a function by sampling its values, and building them into a vector, then the definitions in the continuous case give us the same result as the limit of what we get in the discrete case, as we sample the function more and more densely.

We can use this definition to show that the functions in the Fourier Series have unit magnitude, by taking the inner product of each function with itself, and showing that this is equal to 1. We can show they are orthogonal by showing that their inner products with each other are zero. Doing this requires solving some integrals of trigonometric functions.

To show that the Fourier Series forms a good coordinate system for all functions we must also show that we can express any function as a linear combination of the elements of the Fourier Series. That is, we must show that for any function, *f*, we can write:

$$f(x) = a_0 \frac{1}{\sqrt{2\pi}} + \sum_{k=1}^{\infty}\left(b_k \frac{\cos(kx)}{\sqrt{\pi}} + a_k \frac{\sin(kx)}{\sqrt{\pi}} \right)$$

Then the values (a_0, b_1, a_1, b_2, a_2, …) are the coordinates of the function in this new coordinate system provided by the Fourier Series. This is a very important fact, but we will skip the proof here. Keep in mind that I'm leaving out some necessary conditions, such as that this only works when *f* is a continuous function.

This result means that any function can be broken down into the sum of sine waves of different amplitudes and phases. We say that the part of the function that is due to longer frequency sine waves is the low frequency part of the function. The part that is due to high frequency sine waves is the high frequency component of the function. If a function is very bumpy, with small rapid changes in its value, these rapid changes will be due to the high frequency component of the function. If we can eliminate these high-frequency components, we can make the function smoother.

Implications of the orthonormality of the Fourier Series.

As with an orthonormal basis for vectors, the orthonormality of the Fourier Series means that we can use projection and (a generalization of) the Pythagorean theorem.

We can solve the for the values (a_0, b_1, a_1, b_2, a_2, …) just by taking inner products. So:

$$a_0 = \left\langle f, \frac{1}{\sqrt{2\pi}} \right\rangle = \int_0^{2\pi} \frac{f(x)}{\sqrt{2\pi}} dx \qquad b_k = \left\langle f, \frac{\cos(kx)}{\sqrt{\pi}} \right\rangle = \int_0^{2\pi} \frac{f(x)\cos(kx)}{\sqrt{\pi}} dx$$

$$a_k = \left\langle f, \frac{\sin(kx)}{\sqrt{\pi}} \right\rangle = \int_0^{2\pi} \frac{f(x)\sin(kx)}{\sqrt{\pi}} dx$$

The analog to the Pythagorean theorem is called Parsevaal's theorem. This is:

$$\int_0^{2\pi} f^2(x)dx = a_0^2 + \sum_1^{\infty} \left(a_i^2 + b_i^2\right)$$

Convolution Theorem

Now we get to the reason why the Fourier Series is important in understanding convolution. This is because of the convolution theorem. Let F, G, H be the fourier series that represents the functions f, g, and h. That is, F,G, and H are infinite vectors. The convolution theorem states that convolution in the spatial domain is equivalent to component-wise multiplication in the transform domain, ie, f*g = h ó FG = H. Here, by FG, we mean that we multiply each element of F by the corresponding element of G and use this as an element of H.

This is a remarkable theorem. As an example of its significance, suppose we convolve a filter, *F*, with an image *I*, and that *F* just consists of a single sine wave, *F=sin(x)*. Then the Fourier Series representation of *F* is:

$$a_1 = \sqrt{\pi}, \quad \text{all other components} = 0.$$

This means that no matter what *I* is, if we define *J=F*I*, then the components of the Fourier series of *J* will all be zero except for a_1, because all other components of *F* are 0, and when we multiply these by the corresponding components of *I* we always get 0. More generally, this means that convolution with a filter attenuates every frequency by different amounts, and we can understand the effects of convolution by finding the Fourier Series of the convolution kernel.

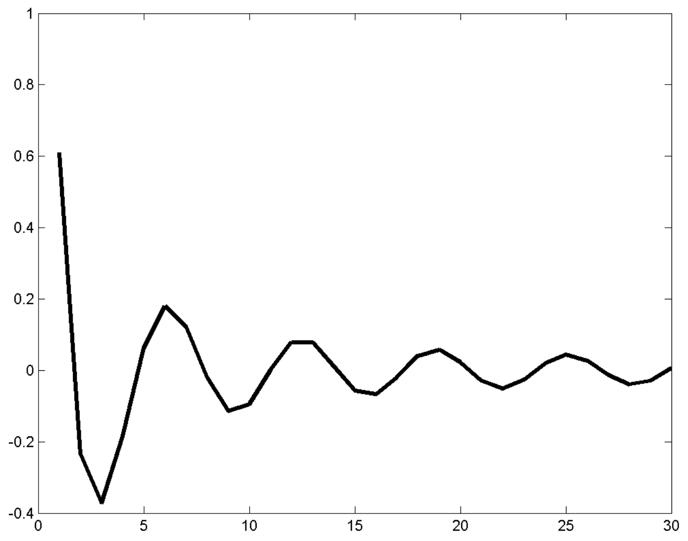
As another important example, we point out that the Fourier series representation of a Gaussian is also a Gaussian. If g(t) is a Gaussian, the broader g is the narrower G is (and vice versa). This means that smoothing with a Gaussian reduces high frequency components of a signal. The more we smooth, the more we reduce the high frequency components. In fact, attenuating high frequency components of a signal can be taken to be the definition of smoothing. To some extent, this is why we learned the Fourier series. It allows us to understand what smoothing really does.

We can also see why simple averaging isn't as good a way of smoothing. Let f(x) be an averaging filter, which is constant in the range –T to T. Then, we can compute its Fourier Series as:

$$a_0 = \int_{-T}^{T} \frac{1}{\sqrt{2\pi}} dx = \frac{2T}{\sqrt{2\pi}}$$
$$a_k = \int_{-T}^{T} \frac{\sin(kx)}{\sqrt{\pi}} dx = \frac{2\cos(kT)}{k\sqrt{\pi}}$$

$$b_k = \int_{-T}^{T} \frac{\cos(kx)}{\sqrt{\pi}} dx = \frac{-2\sin(kT)}{k\sqrt{\pi}}$$

This is called the sinc function. If we graph it, we see that it looks like:



This does not decay as fast as a Gaussian. Also it has oscillations that produce odd effects. As a result of these oscillations, some high frequency components of an image will be wiped out by averaging, while some similar high frequency components remain.

Aliasing and the Sampling Theorem

Sampling means that we know the value of a function, *f*, at discrete intervals, *f(nπ/T)* for *n = 0, +-1, +-T*. That is, we have 2T + 1 uniform samples of *f(x)*, assuming *f* is a periodic function. Suppose now that *f* is *band-limited* to T. That means,

$$f(x) = a_0 \frac{1}{\sqrt{2\pi}} + \sum_{k=1}^{T} \left(b_k \frac{\cos(kx)}{\sqrt{\pi}} + a_k \frac{\sin(kx)}{\sqrt{\pi}} \right)$$

We know the value of *f(x)* at 2T + 1 positions. If we plug these values into the above equation, we get 2T + 1 equations. The a_i and b_i give us 2T + 1 unknowns. These are linear equations. So we get a unique solution. This means that we can reconstruct *f* from its samples. However, if we have fewer samples, reconstruction is not possible. If there is a higher frequency component, then this can play havoc with our reconstruction of all low frequency components, and be interpreted as odd low frequency components.

This means that if we want to sample a signal, the best thing we can do is to low-pass filter it and then sample it. Then, we get a signal that matches our original signal perfectly in the low-frequency components, and this is the best we can do. To shrink images, in fact, we low-pass filter them and then sample them.