CS465 Notes: Sampling and reconstruction

Steve Marschner

September 26, 2004

In graphics we are very often concerned with functions of a continuous variable: an image is the first example you have seen, but you will encounter many more as you continue your exploration of graphics. By their nature continuous functions can't be directly represented in a computer; we have to somehow represent them using a finite number of bits. One of the most useful approaches to representing continuous functions is to use *samples* of the function: just store the values of the function at many different points, and *reconstruct* the values in between when and if they are needed.

You are by now familiar with the idea of representing an image using a two-dimensional grid of pixels – so you have already seen a sampled representation! Think of an image captured by a digital camera: the actual image of the scene that was formed by the camera's lens was a continuous function of the position on the image plane, and the camera converted that function into a two-dimensional grid of samples. Mathematically, the camera converted a function of type $\mathbb{R}^2 \to \mathbf{C}$ to a two-dimensional array of color samples, or a function of type $\mathbb{Z}^2 \to \mathbf{C}$.

Another example is 2D digitizing tablet such as the screen of a pen-based computer or PDA. In this case the original function is the motion of the stylus, which is a time-varying 2D position, or a function of type $\mathbb{R} \to \mathbb{R}^2$. The digitizer measures the position of the stylus at many points in time, resulting in a sequence of 2D coordinates, or a function of type $\mathbb{Z} \to \mathbb{R}^2$. A *motion capture* system does exactly the same thing for a special marker attached to an actor's body: it takes the 3D position of the marker over time ($\mathbb{R} \to \mathbb{R}^3$) and makes it into a series of instantaneous position measurements ($\mathbb{Z} \to \mathbb{R}^3$).

Going up in dimension, a medical CT scanner, used to noninvasively examine the interior of a person's body, measures density as a function of position inside the body. The output of the scanner is a 3D grid of density values: it converts the density of the body ($\mathbb{R}^3 \to \mathbb{R}$) to a 3D array of real numbers ($\mathbb{Z}^3 \to \mathbb{R}$).

All these examples seem very different, but in fact they can all be handled using exactly the same mathematics. In all cases a function is being sampled at the points of a *lattice* in one or more dimensions, and in all cases we need to be able to reconstruct that original continuous function from the array of samples.

From the example of a 2D image, it may seem that the pixels are enough and we never need to think about continuous functions again once the camera has discretized the image. But what if we want to make the image larger or smaller on the screen, particularly by non-

integer scale factors? It turns out that the simplest algorithms to do this perform very badly, introducing obvious visual artifacts known as *aliasing*. Explaining why aliasing happens and understanding how to prevent it requires the mathematics of sampling theory. The resulting algorithms are rather simple, but the reasoning behind them, and the details of making them perform well, can be quite subtle.

Representing contunuous functions in a computer is, of course, not unique to graphics; nor is the idea of sampling and reconstruction. Sampled representations are used in applications from digital audio to computational physics, and graphics is just one (and by no means the first) user of the related algorithms and mathematics. The fundamental facts about how to do sampling and reconstruction have been known in the field of communications since the 1920s and were stated in exactly the form we use them by the 1940s.

This chapter starts by summarizing sampling and reconstruction using the concrete one-dimensional example of digital audio. Then we go on to present the basic mathematics and algorithms that underlie sampling and reconstruction in one and two dimensions. Finally we go into the details of the frequency-domain viewpoint, which provides many insights into the behavior of the algorithms we already presented. (Want to give the idea of seeing the machine first, then the analysis that explains why it does what it does.)

1 Digital audio: sampling and reconstruction in one dimension

Although sampled representations had already been in use for years in telecommunications, the introduction of the compact disc in 1982, following the increased use of digital recording for audio in the previous decade, was the first highly visible consumer application of sampling.

In audio recording, a microphone converts sound, which exists as pressure waves in the air, into a time-varying voltage that amounts to a measurement of the changing air pressure at the point where the microphone is. This electrical signal needs to be stored somehow so that it may be reconstructed, or played back, at a later time and sent (after suitable amplification) to a loudspeaker that converts the voltage back into pressure waves by moving a diaphram in synchronization with the voltage.

The digital approach to recording the audio signal uses sampling: an *analog-to-digital converter* (*A/D converter*, or *ADC*) measures the voltage many thousand times per second, generating a stream of integers that can easily be stored on any number of media, say a disk on a computer in the recording studio, or transmitted to another location, say the disk on a portable audio player. At playback time the data is read from the disk and sent at the appropriate rate to a *digital-to-analog converter* (*D/A converter*, or *DAC*). The DAC produces a voltage according to the numbers it receives, and, provided we took enough samples to fairly represent the variation in voltage, the resulting electrical signal is for all practical purposes identical to the input.

Digital audio: first recordings in 60s; first commercial use in early 70s. CD introduced 1982.

http://history.acusd.edu/gen/recording/digital.html

2 Convolution

Convolution is a simple mathematical concept that underlies the algorithms that are used for sampling, filtering, and reconstruction. It also is the basis of how we will analyze these algorithms later in the chapter.

Convolution is an operation on functions: it takes two functions and combines them to produce a new function. In this book convolution is denoted by a star: the convolution of the functions f and g is $f \star g$. Convolution can be applied both to continuous functions (functions $f(x)$ that are defined for any real argument x) and to discrete sequences (functions $a[i]$ that are defined only for integer arguments i). For convenience in the definitions, we generally assume that the functions' domains go on forever, though of course in practice they will have to stop somewhere, and there will always be boundaries to contend with.

2.1 Discrete convolution

We'll start with the most concrete case of convolution: convolving a discrete sequence $a[i]$ with another discrete sequence $b[i]$. The result is a discrete sequence $(a \star b)[i]$. Here is the definition of $(a \star b)$, expressed as a formula:

$$(a \star b)[i] = \sum_j a[j]b[i-j].$$

By omitting bounds on j we mean that this sum runs over all integers (that is, from $-\infty$ to $+\infty$). In graphics, one of the two functions will usually have *finite support*, which means that it is nonzero only over a finite interval of argument values. If we assume that f has finite support, there is some *radius* r such that $a[i] = 0$ whenever $|i| > r$. In that case we can write the sum above as:

$$(a \star b)[i] = \sum_{j=-r}^{r} a[j]b[i-j].,$$

and we can express the definition in code as

```
FUNCTION convolve( float a[], float b[], int i )
  s = 0
  FOR j = -r TO r
    s = s + a[j]b[i - j]
  RETURN s
```

2.1.1 Convolution filters

Convolution is important because we can use it to perform filtering. For instance, suppose we have a sequence of points from a graphics tablet, and we want to smooth them out

some. One reasonable idea might be to set each point to the average of itself with the previous and next points. This is equivalent to convolving the sequence with the sequence $[\ldots, 0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, \ldots]$. Take a minute to look back at the definitions and convince yourself this is the case.

Example: convolving a box with a step a couple of times.

As in this example, convolution filters are usually designed so that they sum to 1. That way, they don't affect the overall level of the signal.

2.1.2 Properties of convolution

Convolution is a "multiplication-like" operation. Like multiplication or addition of functions, neither the order of the arguments nor the placement of parentheses affects the result. Also, convolution relates to addition in the same way as multiplication. To be precise, convolution is *commutative* and *associative*, and it is *distributive* over addition.

commutative:	$(a \star b)[i] = (b \star a)[i]$
associative:	$(a \star (b \star c))[i] = ((a \star b) \star c)[i]$
distributive:	$(a \star (b + c))[i] = (a \star b + a \star c)[i]$

These properties are very natural if we think of convolution as being like multiplication, and they are very handy to know about because they can let us save work by simplifying convolutions before we actually compute them. For instance, suppose we want to take a sequence b and convolve it with three filters, a_1, a_2, and a_3—that is, we want $a_3 \star (a_2 \star (a_1 \star b))$. If the sequence is long and the filters are short (that is, they have small radii), it is much faster to first convolve the three filters together (computing $a_1 \star a_2 \star a_3$) and finally to convolve the result with the signal, computing $(a_1 \star a_2 \star a_3) \star b$, which we know from commutativity and associativity is the same answer.

Identity for discrete convolution is a discrete impulse.

2.2 Convolution with continuous functions

While it's true that discrete sequences are what we actually work with in a computer program, these sampled sequences are supposed to represent continuous functions, and often we need to reason mathematically about the continuous functions in order to figure out what to do. For this reason it's useful to define convolution between continuous functions, and also between continuous and discrete functions.

The convolution of two continuous functions is the obvious generalization of the definition given above, with an integral replacing the sum:

$$(f \star g)(y) = \int_{-\infty}^{+\infty} f(x)g(y - x)\,dx.$$

One way of reading this definition is that the convolution of f and g, evaluated at the argument y, is the area under the product of the two functions after we shift g to put its zero

point at y. Just like in the discrete case, the convolution is a moving average, with the filter providing the weights for the average.

[Other way of reading the definition—sum of infinitely many copies of filter.]

[Asymmetry in typical usage—one argument is the signal and the other is the filter.]

Just like discrete convolution, convolution of continuous functions is commutative and associative, and it is distributive over addition.

[Identity for continuous convolution is a Dirac impulse.]

[Example. convolving two boxes together.] [Example. convolving two Gaussians together.]

There are two ways to connect the discrete and continuous worlds. One is sampling: we convert a continuous function into a discrete one by writing down the function's value at all integer arguments and forgetting about the rest. Given a continuous function $f_c(x)$ we can sample it to convert to a discrete sequence $f[i]$:

$$f[i] = f(i)$$

Going the other way, from a discrete function, or sequence, to a continuous function, is called *reconstruction*. This is accomplished using yet another form of convolution, the discrete-continuous form. In this case we are filtering a discrete sequence $a[i]$ with a continuous filter $f(x)$.

$$(a \star f)(x) = \sum_i a[i] f(x - i)$$

This can be read to say that the value of the reconstructed function $a \star f$ at x is a weighted sum of the samples $a[i]$ for values of i near x. As with discrete convolution we can express this in code if we have bounds on the support of f. If we know that $f(x) = 0$ for all x outside the interval $(-r, r)$ then we can elimiate all points where the difference between x and i is at least r:

$$(a \star f)(x) = \sum_{j=\lceil x-r \rceil}^{\lfloor x+r \rfloor} a[j] f(x - j)$$

Note that if a point falls exactly at disitance r from x (i.e. if $x - r$ turns out to be an integer) it will be left out of the sum. This is in contrast to the discrete case, where we included the point at $i - r$.

Expressed in code, this comes out to

function reconstruct(sequence a, function f, real x)
$s = 0$
for $j = \lceil x - r \rceil$ to $\lfloor x + r \rfloor$ **do**
  $s = s + a[i] f(x - i)$
return s

[Transposing this to see it as a sum of copies of the filter.]

2.3 A gallery of convolution filters

Now that we have the machinery of convolution, let's examine some of the particular filters commonly used in graphics.

The box. The box filter is a piecewise constant function that integrates to one. As a discrete filter:

$$a_{\text{box},r}[i] = \begin{cases} 1/(2r+1) & |i| \le r \\ 0 & \text{otherwise} \end{cases}$$

Note that for symmetry we include both endpoints. As a continuous filter:

$$f_{\text{box},r}(x) = \begin{cases} 1/(2r) & -r \le x < r \\ 0 & \text{otherwise} \end{cases}$$

In this case we exclude one endpoint. This makes the box of radius 0.5 usable as a reconstruction filter. It is because the box filter is discontinuous that these boundary cases are important, and so for this particular filter we need to pay attention to these boundary cases.

The tent. The tent, or linear, filter is a continuous, piecewise linear function:

$$f_{\text{tent}}(x) = \begin{cases} 1 - |x| & |x| < 1 \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{tent},r}(x) = \frac{f_{\text{tent}}(x/r)}{r}$$

For continuous filters we no longer need to separate the definitions of the discrete and continuous filters. Also note that for simplicity we define $f_{\text{tent},r}$ by scaling the "standard size" tent filter f_{tent}. From now on we'll take this scaling for granted: once we define a filter f, then we can use f_r to mean "the filter f stretched out by r and also scaled down by r." Note that f_r has the same integral as f, and we'll always make sure that that's 1.0.

The Gaussian. The gaussian function, also known as the normal distribution, is an important filter theoretically and practically. We'll see more of its special properties as the chapter goes on.

$$f_g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

The gaussian does not have finite support, although because of the exponential decay its values rapidly become small enough to ignore. When necessary, then, we can trim the tails from the function by setting it to zero outside some radius. The gaussian makes a good sampling filter because it is very smooth; we'll make this statement more precise later in the chapter.

The B-spline cubic. Many filters are defined as piecewise polynomials, and cubic filters with four pieces are often used as reconstruction filters. One such filter is known as the "B-spline" filter because of its origins as a blending function for spline curves (see Chapter 000):

$$f_B(x) = \frac{1}{6}\begin{cases} -3(1-|t|)^3 + 3(1-|t|)^2 + 3(1-|t|) + 1 & -1 \le t \le 1 \\ (2-|t|)^3 & 1 \le |t| \le 2 \\ 0 & \text{otherwise} \end{cases}$$

Among piecewise cubics, the B-spline is special because it has continuous first and second derivatives—that is, it is C^2. A more concise way of defining this filter is $B = b \star b \star b \star b$; proving that the longer form above is equivalent is a nice exercise in convolution (see Problem 000).

The Catmull-Rom cubic. Another piecewise cubic filter named for a spline, the Catmull-Rom filter has the value zero at $x = -2$, -1, 1, and 2, which means it will *interpolate* the samples when used as a reconstruction filter (see below).

$$f_C(x) = \frac{1}{2}\begin{cases} -3(1-|t|)^3 + 4(1-|t|)^2 + (1-|t|) & -1 \le t \le 1 \\ (2-|t|)^3 - (2-|t|)^2 & 1 \le |t| \le 2 \\ 0 & \text{otherwise} \end{cases}$$

The Mitchell-Netravali cubic. For the all-important application of resampling images, Mitchell and Netravali [] made a study of cubic filters and recommended one partway between the previous two filters as the best all-around choice. It is simply a weighted combination of the previous two filters:

$$\begin{aligned} f_M(x) &= \frac{2}{3} f_B(x) + \frac{1}{3} f_C(x) \\ &= \frac{1}{18}\begin{cases} -15(1-|t|)^3 + 18(1-|t|)^2 + 9(1-|t|) + 2 & -1 \le t \le 1 \\ 5(2-|t|)^3 - 3(2-|t|)^2 & 1 \le |t| \le 2 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

2.4 Properties of filters

[Impulse response. Interpolation. Ringing. Continuity (reconstructed function inherits continuity of filter). Vanishing moments?]

[Forward reference to frequency domain as giving more and different insight.]

2.5 Building up 2D filters

So far everything we've said about sampling and reconstruction has been one-dimensional: there has been a single variable x or a single sequence index i. Many of the important

applications of sampling and reconstruction in graphics, though, are to two-dimensional functions—in particular, to 2D images. Fortunately the generalization of sampling algorithms and theory from 1D to 2D, 3D, and beyond is conceptually very simple.

Beginning with the definition of discrete convolution we can generalize it to two dimensions by making the sum into a double sum:

$$(a \star b)[i,j] = \sum_{i'} \sum_{j'} a[i',j']b[i-i',j-j']$$

If f is a finitely supported filter of radius r (that is, it has $(2r+1)^2$ values) then we can write this sum with bounds:

$$(a \star b)[i,j] = \sum_{i'=-r}^{i'=r} \sum_{j'=-r}^{j'=r} a[i',j']b[i-i',j-j']$$

and express it in code:

function convolve2d(float f[][], float g[][], int i, int j)
$a = 0$
for $i' = -r$ to r **do**
  for $j' = -r$ to r **do**
    $a = a + f[i'][j']g[i-i'][j-j']$
return a

This definition can be interpreted in the same ways as in the 1D case: each output sample is a weighted average of an area in the input, using the 2D filter as a "mask" to determine the weight of each sample in the average.

Continuing the generalization we can write continuous–continuous and discrete–continuous convolution in 2D as well:

$$(f \star g)(x,y) = \int \int f(x,y)g(x-x',y-y')\,dx'\,dy' \tag{1}$$

$$(a \star g)(x,y) = \sum_i \sum_j a[i,j]g(x-i,y-j) \tag{2}$$

In each case the result at a particular point is a weighted average of the input near that point. In the first case it's an weighted integral over a region centered at that point, and in the second case it's a weighted average of all the samples that fall near the point.

Once we have gone from 1D to 2D, it should be fairly clear how to generalize further to 3D or even to higher dimensions.

2.6 Separable filters

Now that we have definitions for 2D convolution, what filters should we use? In general they could be any 2D function, and occasionally it's useful to define them this way. But

in most cases we can build suitable 2D (or higher dimensional) filters from the 1D filters we've already seen.

The most useful way of doing this is by using a *separable* filter. The value of a separable filter $f_2(x, y)$ at a particular x and y is simply the product of f_1 evaluated at x and at y:

$$f_2(x, y) = f_1(x)f_1(y).$$

Similarly,

$$a_2[i, j] = a_1[i]a_1[j].$$

Any horizontal or vertical slice through f_2 is a scaled copy of f_1. The integral of f_2 is the square of the integral of f_1, so in particular if f_1 is normalized then so is f_2.

The key advantage of separable filters over other 2D filters has to do with efficiency in implementation. Let's substitute the definition of a_2 into the definition of discrete convolution:

$$(a_2 \star b)[i, j] = \sum_{i'} \sum_{j'} a_1[i']a_1[j']b[i - i', j - j']$$

Note that $a_1[i']$ does not depend on j' and can be factored out of the inner sum:

$$= \sum_{i'} a_1[i'] \sum_{j'} a_1[j']b[i - i', j - j']$$

Let's abbreviate the inner sum as $S[k]$:

$$\mathsf{S}[k] = \sum_{j'} a_1[j']b[k, j - j'] \tag{3}$$

$$(a_2 \star b)[i, j] = \sum_{i'} a_1[i']S[i - i'] \tag{4}$$

With the equation in this form we can first compute and store $S[i - i']$ for each value of i', and then compute the outer sum using these stored values. At first glance this does not seem remarkable, since we still had to do work proportional to $(2r + 1)^2$ to compute all the inner sums. However, it's quite different if we want to compute the value at a whole lot of points $[i, j]$.

Suppose we need to compute $a_2 \star b$ at $[2, 2]$ and $[3, 2]$, and f_1 has a radius of 2. Examining the equation above, we can see that we will need $S[0], \ldots, S[4]$ to compute the result at $[2, 2]$ and we will need $S[1], \ldots, S[5]$ to compute the result at $[3, 2]$. So in the separable formulation we can just compute all six values of S and share $S[1], \ldots, S[4]$.

This savings has great significance for large filters. Filtering an m by n 2D image with a filte of radius r in the general case requires computation of $(2r + 1)^2$ products per pixel, while filtering the image with a separable filter of the same size requires $2(2r + 1)$ products

(at the expense of some intermediate storage). This change in asymptotic complexity from $O(r^2)$ to $O(r)$ enables the use of much larger filters.

Concretely, the algorithm is

function filterImage(image I, filter f)
$r = f.\text{radius}$
$n_x = I.\text{width}$
$n_y = I.\text{height}$
allocate storage array $S[0, \ldots, n_x - 1]$
allocate image $I_{\text{out}}[r, \ldots, n_x - r - 1][r, \ldots, n_y - r - 1]$
initialize S and I_{out} to all zero
for $y = r$ to $n_y - r - 1$ **do**
  for $x = 0$ to $n_x - 1$ **do**
    for $i = -r$ to r **do**
      $S[x] = S[x] + f[i]I[x][y - i]$
  for $x = r$ to $n_x - r - 1$ **do**
    for $i = -r$ to r **do**
      $I_{\text{out}}[x][y] = I_{\text{out}}[x][y] + f[i]S[x - i]$

For simplicity, this function avoids all questions of boundaries by trimming r pixels off all four sides of the output image. In practice there are various ways to handle the boundaries; see Section 000.

3 Image processing using discrete convolution in 2D

Several examples including blur, sharpen, perhaps deconvolve as a special aside.

4 Sampling, reconstruction, and aliasing

The previous sections have developed the basic mathematics and algorithms required to do sampling, filtering, and reconstruction of all kinds of continuous functions. But the details are unclear. SOme of the importnat questions:

- What sample rate is high enough to ensure good results?
- What kinds of filters are appropriate for sampling and reconstruction?
- What are the effects of using the wrong filter?
- How do we manage the cost/quality tradeoffs inherent in choosing filters and sampling rates?

Solid answers to these questions will have to wait until we have developed the theory fully in Section 000, but we can begin to understand the issues now.

What sorts of artifacts should we be watching out for? Let's begin by looking at the sample rate. Suppose we have a complex signal like the one shown in Figure **??**. If we sample the signal at a very high rate, we end up with a set of samples that follows all the details of the signal, and it does not seem difficult to reconstruct a reasonable approximation to the original signal based on those samples.

On the other hand, if we used four times the sample spacing, we would have the set of samples shown in Figure **??**b. Most reasonable ways of connecting these dots to form a continuous function end up looking quite different from the original signal. Intuitively, we have not used enough samples to capture all the detail in the signal.

A more concrete example of the kind of artifacts that can arise from too-low sample frequencies is show in Figure **??**. Here we are sampling a simple sine wave, using a bit less than two samples per cycle. The resulting set of samples is indistinguishable from samples of a low-frequency sine wave. Note that the two frequencies are at equal distance from the sample frequency. Once the sampling has been done, it is impossible to know which of the two signals—the fast or the slow sine wave—was the original, and therefore there's no way we can properly reconstruct the signal in both cases. Because the high frequency signal is "pretending to be" a low-frequency signal, this phenomenon is known as *aliasing*.

Aliasing shows up whenever flaws in sampling and reconstruction lead to artifacts at surprising frequencies. In audio, aliasing takes the form of odd-sounding extra tones—a bell ringing at 10KHz, after being sampled at 8KHz, turns into a 6KHz tone. In images, aliasing often takes the form of *Moiré patterns* that result from the interaction of the sample grid with regular features in an image. For instance, in a photograph of a brick wall the repetetive, high-frequency pattern of mortar lines may alias, causing the wall to turn out with broad bands of red and tan color.

Another example of aliasing in a synthetic image is the familiar stair-stepping on straight lines that are rendered with only black and white pixels. This is another example of small-scale features (the sharp edges of the line) creating artifacts at a different scale (for shallow-slope lines the stair steps are very long). A fuller understanding of what is going on has to wait until the frequency-space analysis later in this chapter.

4.1 Controlling the effects of aliasing

Aliasing can never be completely eliminated, but through suitable use of filters it can be reduced to where it no longer matters. First, a filter is used during sampling to smooth out any small-scale details that would cause aliasing. Second, a reconstruction filter is chosen to avoid introducing fine-scale filtering artifacts.

What the filters are supposed to do, with examples of the effects in the space domain.

The details of choosing a filter are fairly application-specific. There are two tradeoffs to consider: quality vs. cost and sharpness vs. artifacts. The quality/cost tradeoff is straightforward: using larger filters enables the use of higher quality filters but requires more computation. The sharpness/artifacts tradeoff is more troublesome: filters that smooth enough to absolutely quash aliasing artifacts also tend to smooth out some of the small-scale details

that one would like to preserve.

Again the subtleties of the explanation here will be left until the last section, but we now provide a series of practical recommendations for various cases.

Image antialiasing. For sampling an image in a ray tracer or scaling an image to a substantially lower resolution, a primary consideration is smoothing very fine-scale detail effectively. For non-critical applications a box filter of radius 0.5 is quite effective, and for applications where less aliasing is desired, a gaussian with standard deviation between 0.5 and 1.0 is a good choice. Sampling images in renderers is one of the few places in graphics were we have direct control over the sampling filter; in most other cases we are handed the data after it's been sampled by some other device.

Image reconstruction. To scale an image to a sample rate higher than the original, or close to the original, the reconstruction filter is very important. When performance is critical a bilinear filter can be used, but for good quality a bicubic is recommended. The Mitchell-Netravali filter is a good choice for this purpose.

2D curves. Much more will be said on this topic in Chapter 000 where splines are discussed.

Volume data. When resampling volume data a cubic is often used to get the best quality. For point sampling, often the cubic is prohibitively expensive and trilinear is the near-universal choice.

5 Sampling theory

If you are only interested in implementation, you can stop reading here; the algorithms and recommendations in the previous sections will let you implement programs that perform sampling and reconstruction and achieve excellent results. However, there is a deeper mathematical theory of sampling with a history reaching back to the first uses of sampled representations in telecommunications. Sampling theory answers many questions that are difficult to answer with reasoning based strictly on scale arguments.

But most important, sampling theory gives valuable insight into the workings of sampling and reconstruction. It gives the student who learns it an extra set of intellectual tools for reasoning about how to achieve the best results with the most efficient code.

5.1 The Fourier transform

The Fourier transform, along with convolution, is the main mathematical concept that underlies sampling theory. You can read about the Fourier transforms in many math books on analysis, as well as in books on signal processing.

The basic idea behind the Fourier transform is to express any function by adding together sine waves (sinusoids) of all frequencies. By using the appropriate weights for the different frequencies, we can arrange for the sinusoids to add up to any (reasonable) function we want.

As an example, the square wave in Figure **??** can be expressed by a sequence of sine waves:

$$\sum_{n=1,3,5,\ldots}^{\infty} \frac{4}{n\pi} \sin 2\pi nx$$

This *fourier series* starts with a sine wave ($\sin 2\pi x$) that has frequency 1.0—same as the square wave—and the remaining terms add smaller and smaller corrections to reduce the ripples and, in the limit, reproduce the square wave exactly. Note that all the terms in the sum have frequencies that are integer multiples of the frequency of the square wave. This is because other frequencies would produce results that don't have the same period as the square wave.

A surprising fact is that a signal does not have to be periodic in order to be expressed as a sum of sinusoids in this way: it just requires more sinusoids. Rather than summing over a discrete sequence of sinusoids, we will instead integrate over a continuous family of sinusoids. For instance, the box function shown in Figure **??** can be written as the integral of a family of cosine waves [**?**]:

$$\int_{-\infty}^{\infty} \frac{\sin \pi u}{\pi u} \cos 2\pi ux du \tag{5}$$

This integral is adding up infinitely many cosines, weighting the cosine of frequency u by the weight $(\sin \pi u)/\pi u$. The result, as we include higher and higher frequencies, converges to the box function. When a function f is expressed in this way, this weight, which is a function of u, is called the *Fourier transform* of f, denoted $\hat{f}$:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(u) e^{2\pi iux} du \tag{6}$$

This equation is known as the *inverse Fourier transform* because it takes the Fourier transform of f and reconstructs f again.

Note that in Equation 6 the complex exponential $e^{2\pi iux}$ has substituted for the cosine in the previous equation. Also, $\hat{f}$ is a complex-valued function. The machinery of complex numbers is just needed to allow the phase, as well as the frequency, of the sinusoids to be controlled, which is needed to represent any functions that are not symmetric across zero. The magnitude of $\hat{f}$ is known as the *Fourier spectrum*, and for our purposes this is sufficient—we won't need to worry about phase or use any complex numbers directly.

It turns out that computing $\hat{f}$ from f looks very much like computing f from $\hat{f}$:

$$\hat{f}(x) = \int_{-\infty}^{\infty} f(x) e^{-2\pi iux} dx \tag{7}$$

This equation is known as the (forward) *Fourier transform*. The sign in the exponential is the only difference between the forward and inverse Fourier transforms, and it's really just a technical detail. For our purposes we can think of the FT and IFT as the same operation.

Sometimes the f–$\hat{f}$ notation is inconvenient, and then we will denote the Fourier transform of f by $\mathcal{F}\{\{\}$ and the inverse Fourier Transform of $\hat{f}$ by $\mathcal{F}^{-\infty}\{\hat{\{\}}$.

A function and its Fourier transform are related in many useful ways. A few facts (most of them easy to verify) that we'll use later in the chapter are:

- A function and its Fourier transform have the same squared integtral:

  $$\int f(x)^2 dx = \int \hat{f}(u)^2 du$$

  The physical interpretation is that the two have the same energy.

  In particular, scaling a function up by a also scales its Fourier transform by a. That is, $\mathcal{F}\{af\} = a\mathcal{F}\{f\}$.

- Stretching a function along the x axis squashes its Fourier transform along the u axis by the same factor:

  $$\mathcal{F}\{f(x/b)\} = b\hat{f}(bx)$$

  (The renormalization by b is needed to keep the energy the same.)

  This means that if we're interested in a family of functions of different width and height (say all box functions centered at zero) then we only need to know the Fourier transform of one canonical function (say the box function with width and height one) and we can easily know the Fourier transforms of all the scaled and dilated versions of that function. For example, we can instantly generalize Equation **??** to give the Fourier transform of a box of width b and height a:

  $$ab\frac{\sin \pi bu}{\pi bu}$$

- The average value of f is equal to $\hat{f}(0)$. This makes sense since $\hat{f}(0)$ is supposed to be the zero-frequency component of the signal (the DC component if we are thinking of an electrical voltage).

- If f is real (which is always is for us), $\hat{f}$ is an even function—that is, $\hat{f}(u) = \hat{f}(-u)$. Likewise, if f is an even function then $\hat{f}$ will be real (this is not usually the case in our domain, but remember that we really are only going to care about the magnitude of $\hat{f}$.

5.2 Convolution and the Fourier transform

One final property of the Fourier transform that deserves special mention is its relationship to convolution. Briefly,

$$\mathcal{F}\{f \star g\} = \hat{f}\hat{g}.$$

The Fourier transform of the convolution of two functions is the product of the Fourier transforms. Following the by now familiar symmetry,

$$\hat{f} \star \hat{g} = \mathcal{F}\{fg\}.$$

The convolution of two Fourier transforms is the Fourier transform of the product of the two functions. These facts are fairly straightforward to derive from the definitions.

This relationship is the main reason Fourier transforms are useful in studying the effects of sampling and reconstruction. We've seen how sampling, filtering, and reconstruction can be seen in terms of convolution; now the Fourier transform gives us a new domain—the frequency domain—in which these operations are simply products.

5.3 A gallery of Fourier transforms

Now that we have some facts about Fourier transforms, let's look at some examples of individual functions. First, we'll look at some filters from Section 2.3. We have already seen the box function:

$$\mathcal{F}\{h_{\text{box}}\} = \frac{\sin \pi x}{\pi x}$$

The tent function is the convolution of the box with itself, so its fourier transform is just the square of the box's Fourier transform:

$$\mathcal{F}\{h_{\text{tent}}\} = \frac{\sin^2 \pi x}{\pi^2 x^2}$$

By Exercise **??**, we can continue this to the B-spline filter:

$$\mathcal{F}\{h_{\text{B}}\} = \frac{\sin^4 \pi x}{\pi^4 x^4}$$

The Gaussian has a particularly nice Fourier transform:

$$\mathcal{F}\{h_{\text{gauss}}\} = \frac{1}{\sqrt{2\pi}} e^{-u^2/2}$$

It is another Gaussian! The Gaussian with standard deviation 1.0 becomes a Gaussian with standard deviation $1/2\pi$.

5.4 Impulses

We still need one last mathematical idea before we're ready to go on to the main result of sampling theory. That is the idea of an *impulse function*, also called the *Dirac delta function*, denoted $\delta(x)$.

Intuitively, the delta function is a very narrow, very tall spike that infinitesimal width but still has area 1.0. The key defining property of the delta function is that multiplying it by a function selects out the value exactly at zero:

$$\int_{-\infty}^{\infty} \delta(x) f(x) dx = f(0)$$

The delta function does not have a well-defined value at 0 (you can think of its value loosely as $+\infty$), but it does have the value $\delta(x) = 0$ for all $x \neq 0$.

The delta function works like a normal function in that we can scale it and shift it from one place to another:

$$\int_{-\infty}^{\infty} b\delta(x - a) f(x) dx = bf(a)$$

From this property of selecting out single values, it follows that the delta function is the identity for continuous convolution (in the same way that we saw the discrete impulse $[\ldots, 0, 0, 1, 0, 0, \ldots]$ is the identity for discrete convolution). The convolution of δ with a function f is:

$$(\delta \star f)(x) = \int_{-\infty}^{\infty} \delta(t) f(x - t) dt = f(x)$$

So $\delta \star f = f$.

The reason impulses are useful in sampling theory is that we can use them to talk about samples in the context of continuous functions and Fourier transforms. We represent a sample, which has a position and a value, by an impulse translated to that position and scaled by that value. A sample at position a with value b is represented by $b\delta(x - a)$. This way we can express the operation of sampling the function $f(x)$ at a as multiplying f by $\delta(x - a)$. The result is $f(a)\delta(x - a)$.

Sampling a function at a series of equally spaced points is therefore expressed as multiplying the function by the sum of a series of equally spaced impulses, called an *impulse train*:

$$s(x) = \sum_{i=-\infty} \infty\delta(x - i)$$

We can call s an impulse train with period 1. The Fourier transform of s is exactly the same as s: a sequence of impulses at all integer frequencies. You can see why this should be true by thinking about what happens when we multiply the impulse train by a sinusoid and integrate. We wind up adding up the values of the sinusoid at all the integers. This sum will exactly cancel to zero for non-integer frequencies, and it will diverge to $+\infty$ for integer frequencies.

Because of the dilation property of the Fourier transform, we know already that an impulse train with period T (which is a dilation of s by T) is an impulse train with period $1/T$. Making the sampling finer in the space domain makes the impulses farther apart in the frequency domain.

5.5 Sampling and aliasing

Now we have built the mathematical machinery we need to understand the sampling and reconstruction process from the viewpoint of the frequency domian. The key advantage of introducing Fourier transforms is that it makes the effects of convolution filtering on the signal much clearer, and it provides more precise explanations of why we need to filter when sampling and reconstructing.

We start the process with the original, continuous signal. In general its Fourier transform could include components at any frequency, though generally for most kinds of signals (especially images) we expect the content to decrease a the frequency gets higher. Let's see what happens to the Fourier transform if we sample and reconstruct without doing any special filtering.

When we sample the signal, we model the operation as multiplication with an impulse train; the sampled signal is $f \cdot s$. Because of the multiplication–convolution property, the FT of the sampled signal is $\hat{f} \star \hat{s} = \hat{f} \star s$.

Recall that δ is the identity for convolution. This means that

$$\hat{f} \star s = \sum_{i=-\infty} \infty \hat{f}(u - i)$$

That is, convolving with the impulse train makes a whole series of equally spaced copies of the spectrum of f. A good intuitive interpretation of this seemingly odd result is that all those copies just express the fact (as we saw back in Section **??**) that frequencies that differ by an integer multiple of the sampling frequency are indistinguishable once we have sampled—they will produce exactly the same set of samples. The original spectrum is called the *base spectrum* and the copies are known as *alias spectra*.

The trouble begins if these copies of the signal's spectrum overlap, which will happen if the signal contains any significant content beyond half the sample frequency. When this happens, the spectra add, and the information about different frequencies is irreversibly mixed up. This is the first place aliasing can occur, and if it happens here it's due to undersampling—using too low a sample frequency for the signal. The purpose of low-pass filtering when sampling is to limit the frequency range of the signal so that the alias spectra do not overlap the base spectrum.

Suppose we reconstruct the signal using the nearest-neighbor technique. This is equivalent to convolving with a box of width 1. (The discrete–continuous convolution used to do this is the same as a continuous convolution with the series of impulses that represent the samples.) The convolution–multiplication property means that the spectrum of the reconstructed signal will be the product of the spectrum of the sampled signal and the spectrum of the box. The resulting reconstructed Fourier transform contains the base spectrum (though somewhat attenuated at higher frequencies), plus attenuated copies of all the alias spectra. These alias components manifest themselves in the image as the pattern of squares that's characteristic of nearest-neighbor reconstruction.

The leftover alias spectra are the second form of aliasing, due to an inadequate reconstruction filter. From the frequency domain, we can clearly see that a good reconstruction

filter needs to be a good lowpass filter. The purpose of using a reconstruction filter different from the box is to more completely eliminate the alias spectra, reducing the leakage of high-frequency artifacts into the reconstructed signal, while disturbing the base spectrum as little as possible.

5.6 Ideal filters, and useful filters

Following the frequency domain analysis to its logical conclusion, a filter that is exactly a box in the frequency domain would be ideal for both sampling and reconstruction. This would prevent aliasing at both stages without diminishing the frequencies below the Nyquist frequency at all.

Recall that the inverse and forward Fourier transforms are essentially identical, so the spatial domain filter that has a box as its Fourier transform is the function $sin(\pi x)/(\pi x)$. This is known as the *sinc* function.

However, the sinc filter is not generallly used in practice, either for sampling or for reconstruction, because it's impractical and because, even though it's optimal according to the frequency domain criteria, it doesn't produce the best results for many applications.

For sampling, the infinite extent of the sinc filter, and its relatively slow rate of decrease with distance from the center, is a liability. Also, for some kinds of sampling (as we'll see later in antialiasing) the negative lobes are problematic. A gaussian filter makes an excellent sampling filter even for difficult cases where high-frequency patterns must be removed from the input signal, because its Fourier transform falls off exponentially, with no bumps that could let aliases leak through. For less difficult cases, a tent filter generally suffices.

For reconstruction, the size of the sinc function again creates problems, but even more importantly, the many ripples create "ringing" artifacts in reconstructed signals.