I.2

GENERAL FILTERED IMAGE RESCALING

Dale Schumacher
St. Paul, Minnesota

A raster image can be considered a rectangular grid of samples of a continuous 2-D function $f(x, y)$. These samples are assumed to be the exact value of the continuous function at the given sample point. The ideal procedure for rescaling a raster image involves reconstructing the original continuous function and then resampling that function at a different rate (Pratt, 1991; Foley *et al.*, 1990). Sampling at a higher rate (samples closer together) generates more samples, and thus a larger image. Sampling at a lower rate (samples farther apart) generates fewer samples, and thus a smaller image. Fortunately, we don't really have to reconstruct the entire continuous function, but merely determine the value of the reconstructed function at the points that correspond to the new samples, a much easier task (Smith, 1981). With careful choice of filters, this resampling process can be carried out in two passes, stretching or shrinking the image first horizontally and then vertically (or vice versa) with potentially different scale factors. The two-pass approach has a significantly lower run-time cost, **O**(image_width *image_height*(filter_width + filter_height)), than straightforward 2-D filtering, **O**(image_width*image_height*filter_width*filter_height).

The process of making an image larger is known by many names, including magnification, stretching, scaling up, interpolation, and upsampling. I will refer to this process as magnification. The process of making an image smaller is also known by many names, including minification, shrinking, scaling down, decimation, and downsampling. I will refer to this process as minification. The processes will be explained in one

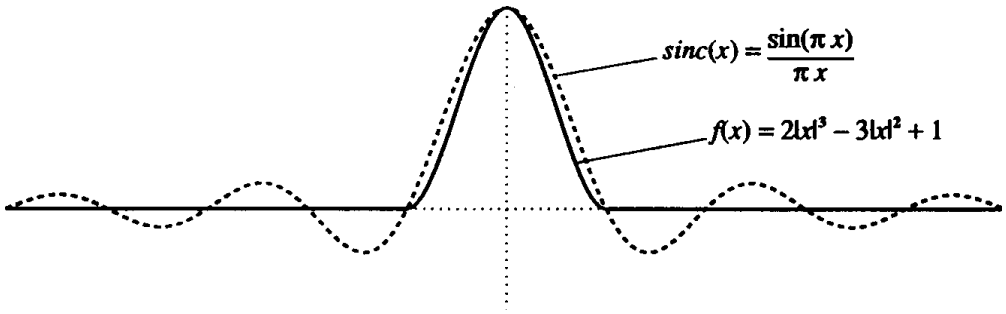
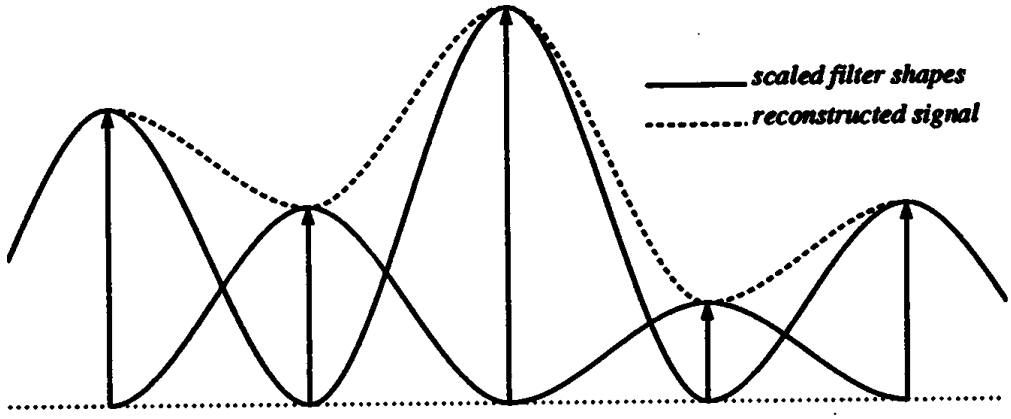


Figure 1.

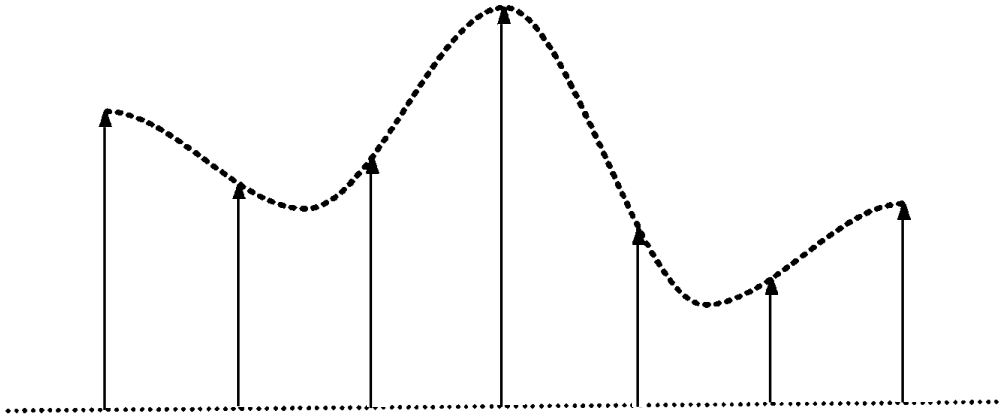
dimension rather than two, since the scaling is carried out in each axis independently.

In magnification, we determine the contribution each source pixel makes to each destination pixel by application of a filter function. Sampling theory states that the *sinc* function, $f(x) = \sin(\pi x)/\pi x$, is ideal for reconstruction; however, we have a finite sample set and need a filter with a finite support (that is, the area over which the filter is nonzero). The filter I use in this example is a cubic function, $f(x) = 2|x|^3 - 3|x|^2 + 1$, from –1 to +1, which covers a unit volume at each sample when applied separably. Figure 1 compares these filter functions. The design of resampling filters is a source of endless debate and is beyond the scope of this gem, but is discussed in many other works (Pratt, 1991; Turkowski, 1990; Mitchell, 1988; Smith, 1982; Oppenheim and Schafer, 1975; Rabiner and Gold, 1975). To apply the filter, we place a copy of our filter function centered around each source pixel, and scaled to the height of that pixel. For each destination pixel, we compute the corresponding location in the source image. We sum the values of the weighted filter functions at this point to determine the value of our destination pixel. Figure 2 illustrates this process.

In minification, the process is similar, but not identical, since we must be concerned with frequency aliasing. Sampling theory defines the Nyquist frequency as the sampling rate that will correctly capture all frequency components in our continuous source signal. The Nyquist frequency is twice the frequency of the highest-frequency component in our source signal. Any frequency component that is higher than half the sampling rate will be sampled incorrectly and will be aliased to a lower frequency.



Input Samples



Output Samples

Figure 2.

Therefore, a reconstructed signal will contain only frequency components of half the sampling rate or less. During magnification, we are stretching our reconstructed signal, lowering its component frequencies. However, during minification, we are shrinking our reconstructed signal, raising its component frequencies, and possibly exceeding the Nyquist frequency of our new sampling rate. To create proper samples, we must eliminate all frequency components above the resampling Nyquist frequency. This can be accomplished by stretching the filter function by the image reduction factor. Also, since the filters at each source pixel are wider, the sums will

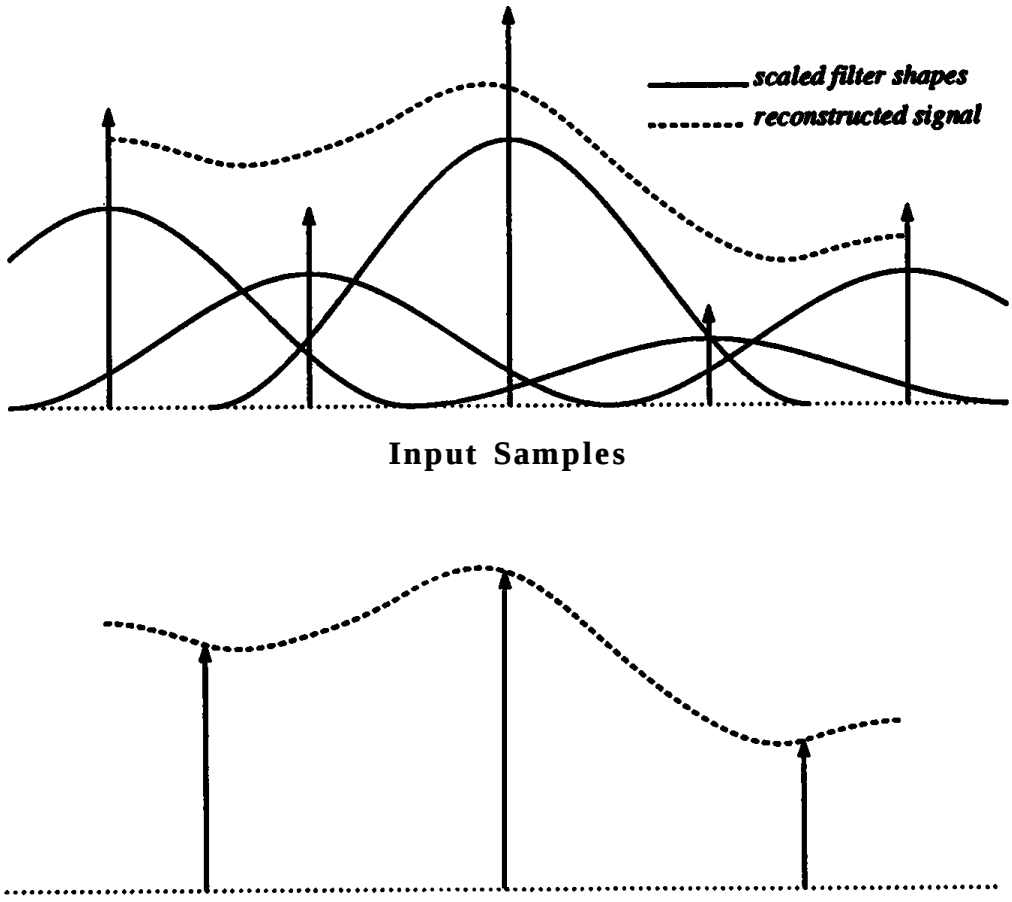


Figure 3.

be proportionally greater and should be divided by the same factor to compensate. Figure 3 illustrates this process.

So far we have only considered the one-dimensional case. We extend this to the two-dimensional case of a typical raster image by scaling first horizontally and then vertically. The further optimization of scaling the smallest destination axis first will not be illustrated here. The filtering operation can lead to a significant number of computations, so we precalculate as much as possible. The scaling process for each row (or column) is identical. The placement and area of the filters is fixed; thus,

we can precalculate the contributors to each destination pixel and the corresponding filter weight. The pseudo-code for calculating the contributors to a destination pixel is as follows:

```
calculate_contributions(destination);
begin
    scale ← dst_size/src_size;
    center ← destination/scale;
    if (scale < 1.0) then begin
        width ← filter_width/scale;
        fscale ← 1.0/scale;
    else
        width ← filter_width;
        fscale ← 1.0;
    end;
    left ← floor(center – width);
    right ← ceiling(center + width);
    for source ← left, source = source + 1, source ≤ right do
        weight ← filter((center – source)/fscale)/fscale;
        add_contributor(destination, source, weight);
        endloop;
    end;
```

After the contributions have been calculated, all the rows (or columns) of the destination image can be processed using the same precalculated filter values. The following pseudo-code shows the application of these values to scale a single destination row.

```
scale_row(destination_row, source_row)
begin
    for i ← 0, i ← i + 1, i < dst_size do
        v ← 0;
        for j ← 0, j ← j + 1, j < contributors[i] do
            s ← contributor[i][j];
            w ← weight_value[i][j];
            v ← v  + (source_row[s]*w);
        endloop;
        destination_row[i] ← v;
    end;
```

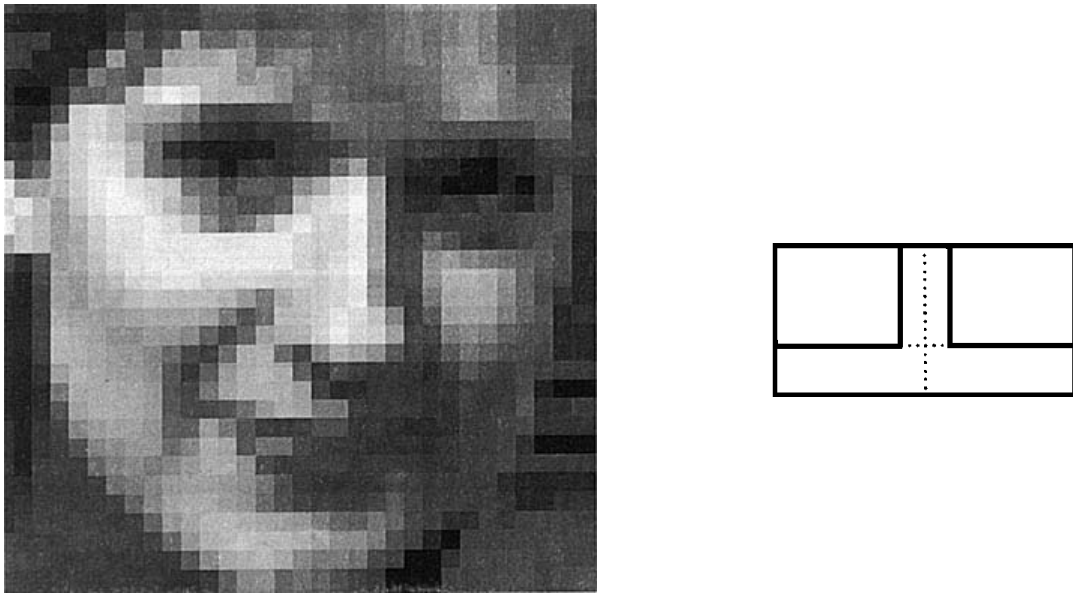


Figure 4.

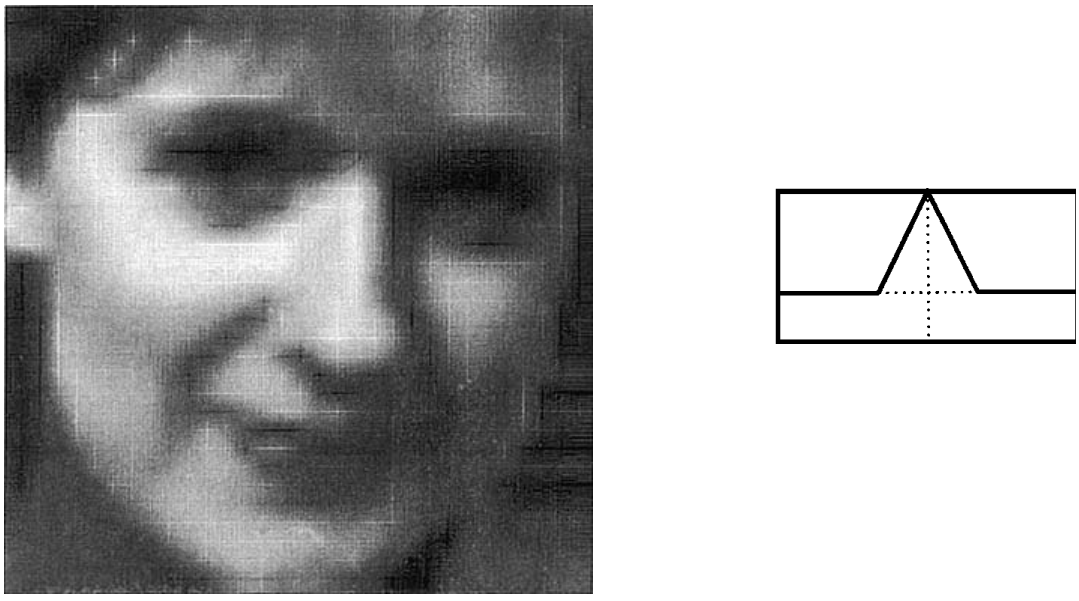


Figure 5.

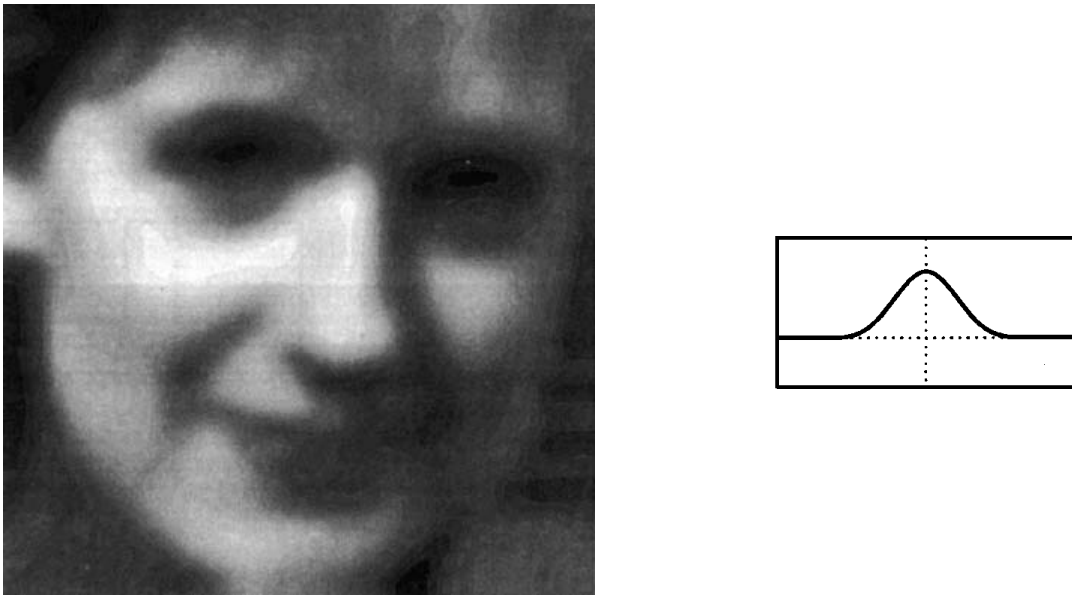


Figure 6.

Figure 7.

Figure 8.

The same process is then applied to the columns of the image—first precalculating the filter contributions according to the vertical scale factor (which may be different than the horizontal), and then processing columns from the intermediate (horizontally scaled) image to the final destination image.

In the source code provided in the Appendix, a number of filter functions are given, and new functions may be easily added. The zoom() function takes the name of the desired filter and the filter support as parameters. Figures 4 through 8 show the effect of various filters on a sample image, along with the impulse response graphs for each filter function.

The sample images have been scaled up by a factor of 12 in both directions. Figure 4 shows a box filter, which is equivalent to direct replication of pixel values, as it shows considerable tiling or "jaggies." Figure 5 shows the triangle or Bartlett filter, a considerable improvement over the box, still computationally simple, but there are still sharp transition lines. Figure 6 shows a cubic B-spline, which creates no sharp transitions, but its width causes excessive blurring. The triangle and B-spline functions are computed by convolving the box filter with itself

one and three times, respectively. Figure 7 shows the Lanczos_3 filter, a *sinc* function damped to zero outside the –3 to +3 range, which shows the excessive "ringing" effect that occurs with the full *sinc* function. Figure 8 shows the Mitchell filter ($B = \frac{1}{3}$, $C = \frac{1}{3}$), a cubic function with no sharp transition, and a good compromise between "ringing" and "blurring" effects.

See also G1, 147; G1, 166; G3, A.1.