Scaling of Bitmaps via Continuous-Space Convolution

Renate Schaaf
https://github.com/rmesch

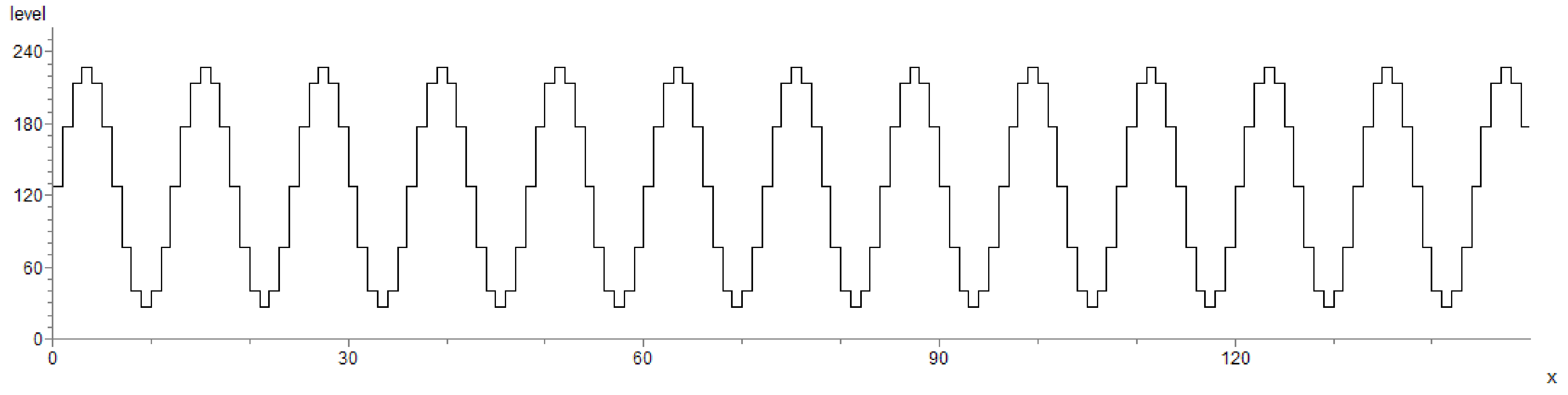
1. History

When Delphi-users ask about advanced scaling routines for bitmaps, they mostly are referred to Anders Melander. His algorithm has been improved implementation-wise by Mike Lischke and used to be part of his GraphicEx package. Anders has moved his algorithm into the Graphics32 project, where over the years it has become faster and better and is also supporting many more filter functions.

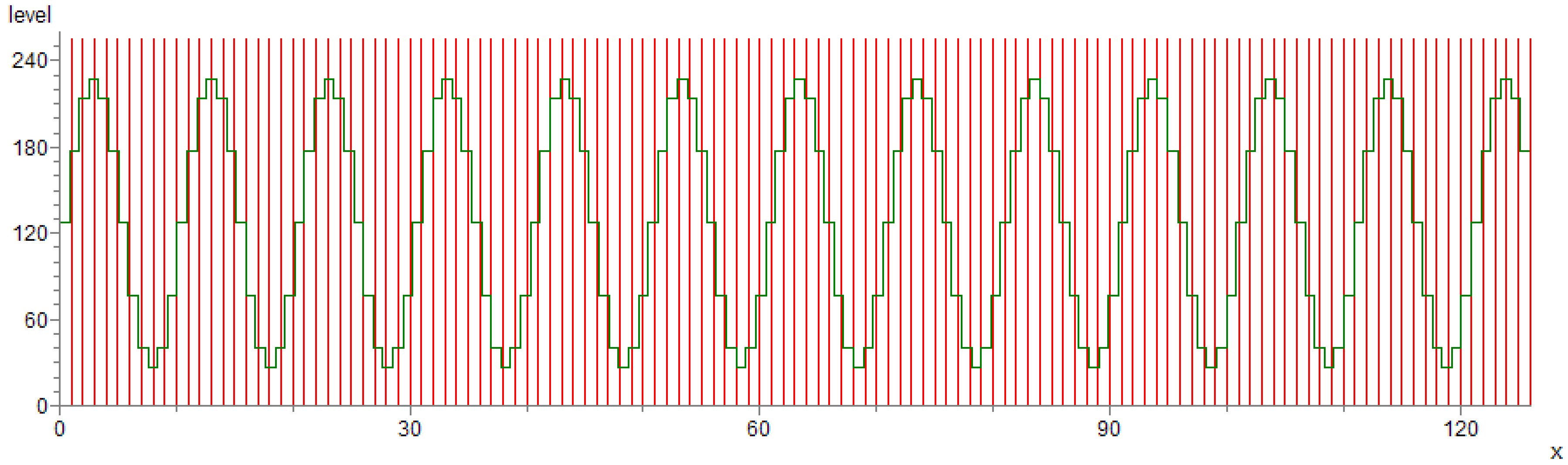
A very beautiful resampling algorithm was proposed by Eric Grange in the Delphi BASM-forum, which I have tried to understand and generalize to filters used by Anders' algorithm. Writing this little paper helped me understand resampling algorithms and writing my own, and so I hope it is useful for others, too.

2. General idea

We usually think of bitmaps being discrete-space objects: the pixels are discrete points [i,j] in space each being assigned its colour-vector (triple or quad). When thinking of resizing bitmaps, it seems better to think of them as step functions in continuous space, having constant values within a square of length one pixel. That makes it easy to visualize them as being first scaled exactly, having constant values in a square of length scale-factor, then being approximated in some optimal way by a step function which is again constant within a square of length one. For simplicity, let's only think of one line of a bitmap and of only one colour value:

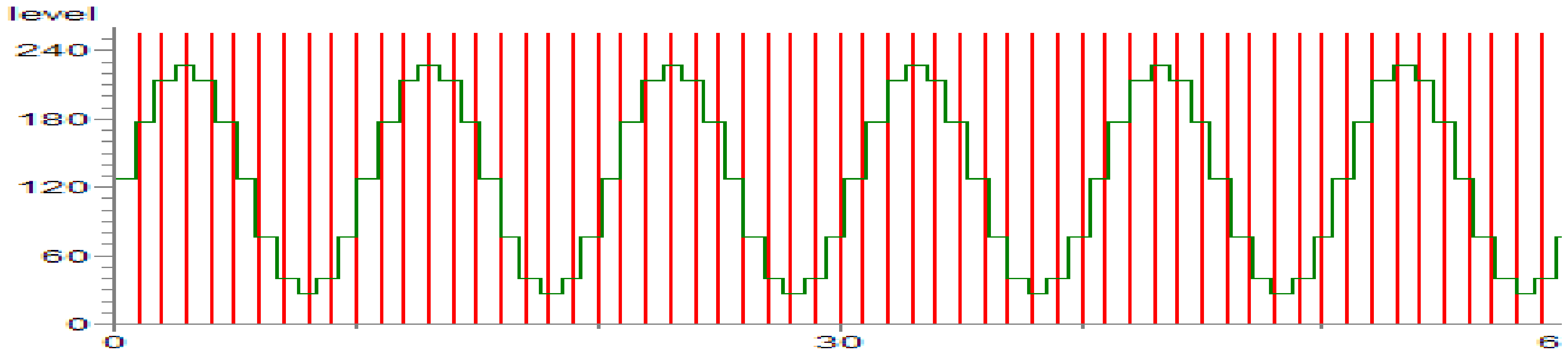


Say, the original bitmap has width 150, so occupies pixel 0 to 149. The step function though has a domain from 0 to 150 since it takes the last pixel's extension into account. Say, the bitmap should be scaled to a width of 126. The exact downscale then looks like this, with the pixel boundaries of the new bitmap being indicated by the red lines:



The downscaled function occupies an interval of length 126, so the correct scaling factor is *scale*=150/126 = OldWidth/NewWidth, and not (OldWidth-1)/(NewWidth-1) as used by some of the existing algorithms, leading to inconsistent scaling of rotated bitmaps. The formula for the scaled step function is
ScaledPixel(x)=OldPixelStep(x*scale).

Now the green levels between the red lines need to be averaged to compute the level of the corresponding target pixel. Here is a zoom into the first part of the downscaled step function for better illustration:



The cheap "stretch" algorithm of TCanvas.Stretchdraw etc. (i.e. Windows.StretchBlt with stretchmode Delete_Scans) would just take the green level at the centre-point of each red interval and assign it to the target pixel. That's nearest-neighbour-resampling. Note how a small shift in the original bitmap will produce a much different result of the smaller bitmap for green levels that jump near the centre of the target pixel interval.

The resampling procedures by Anders Melander use a filter function spread over an interval with a certain radius centred about the target pixel. The colour values of the scaled source pixels that fall into this interval get a weight from the value of the filter function at this location. The weights are normalized to sum up to 1 and the weighted colour values are added to form the colour of the target pixel. The simplest filter is the box filter, a horizontal line over the radius-interval. It gives all pixels that fall into the interval the same weight. The other filter functions used are discussed below. Depending on the filter and the radius one can improve the perceived quality of the resampling a lot compared to nearest-neighbour-resampling.

Eric's algorithm averages the green step function within each new pixel interval:

$$NewPixel(i) = \int_{i}^{i+1} OldPixelStep(x \cdot scale)dx$$

This corresponds to continuous-space convolution with the box filter function, see below. Thus each source pixel that occupies the range of the target pixel also gets weighted by the length of the overlapping step and leads to much smoother results than discrete-space convolution with the box filter. For example a pixel which is completely in the target interval gets a higher weight than one which is already partly in the interval of the next target pixel. In discrete-space convolution with the box filter, both pixels would get the same weight.

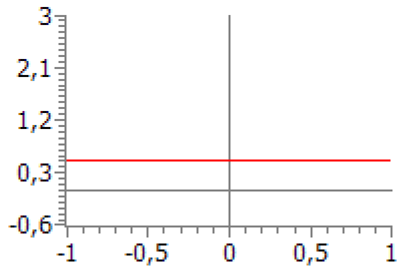
So, the idea is, to implement resampling by continuous-space convolution with other and better filter functions than the box filter, too, and also to allow for variation of the radius of the resampling interval.

3. Filters

Normalized Filters are real functions *filt(x)* which are 0 outside of the interval [-1,1] and which have $\int_{-1}^{1} filt(x)dx = 1$. This way, when multiplied with another function and integrated, they average that function in the interval [-1,1] with weights *filt(x)* at x. You can shift the filter and average the other function in the shifted interval. Scaling the filter to a radius r other than 1 averages the other function over an interval with radius r about the shifted x-value. The scaled filter function is $filt_r(x) = \frac{1}{r} filt(\frac{x}{r})$, it lives on [-r,r] and still has integral 1.

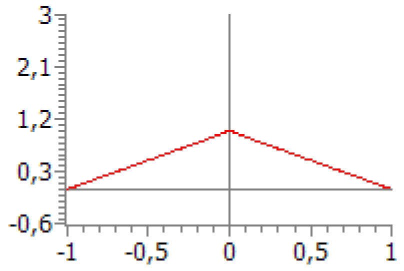
Here are the formulas/graphs for some common normalized filters for x in [-1,1]:

Box



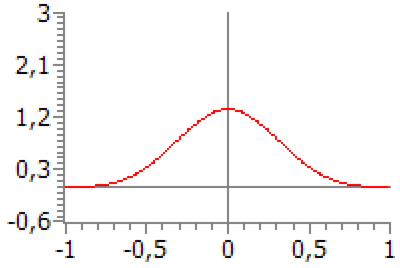
$box(x) = \frac{1}{2}$ gives equal weight to each contributing value, default radius 0.5

Linear



$linear(x) = \begin{cases} x+1 & x \le 0 \\ 1-x & x > 0 \end{cases}$ Gives more weight to centre, continuous weights, default radius 1 or a bit less. Used for bilinear resampling, "bi" because it has to be done both horizontally and vertically.

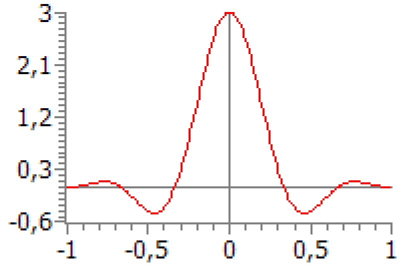
BSpline



$$bspline(x) = \begin{cases} 8x^2(x-1)+\frac{4}{3} & 0 \le x < \frac{1}{2} \\ \frac{8}{3}(1-x)^3 & \frac{1}{2} \le x \le 1 \\ cubic(-x) & x < 0 \end{cases}$$

Has smooth weights, emphasis on centre, default radius 2 or a bit less. Sometimes called cubic filter, but it certainly isn't related to the bicubic photoshop-resampling.

Lanczos 3



$$lanczos(x) = \begin{cases} 3SinC(3x)SinC(x) & 0 \le x \le 1 \\ lanczos(-x) & x < 0 \end{cases}$$

$$SinC(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & x \ne 0 \\ 1 & x = 0 \end{cases}$$

negative weights act as a "sharpener", brilliant results, default radius 3

4. Convolution

The convolution of a real function g with a (filter-)function *filt* is

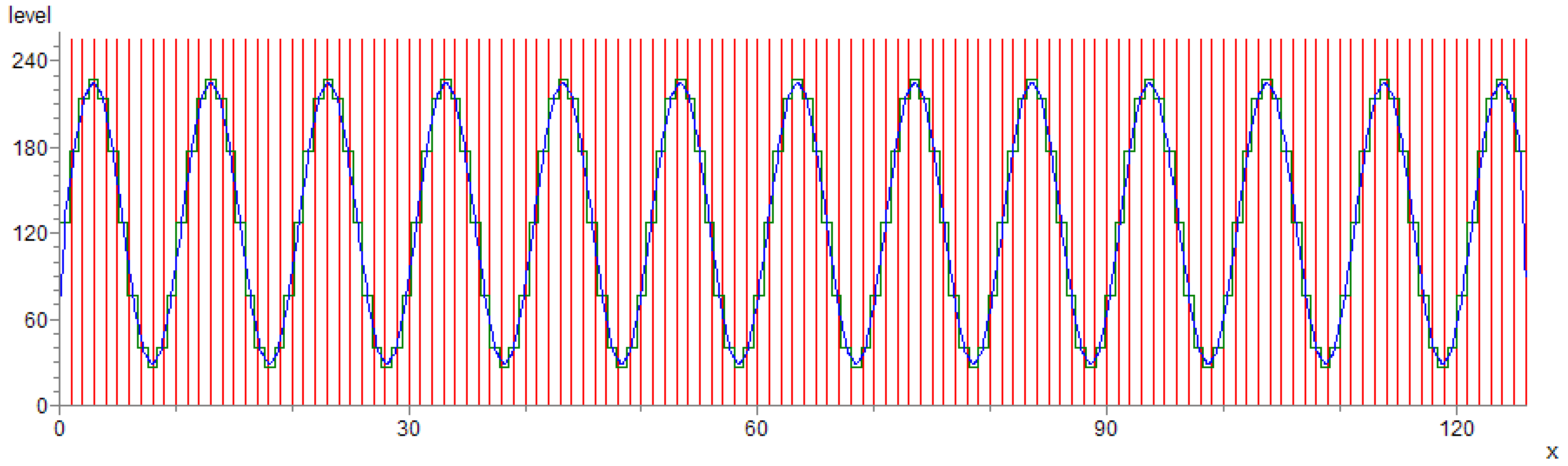
$$conv(x) = \int_{-\infty}^{\infty} filt(y-x)g(y)dy$$

and is the result of averaging the function g by the filter function around each x.

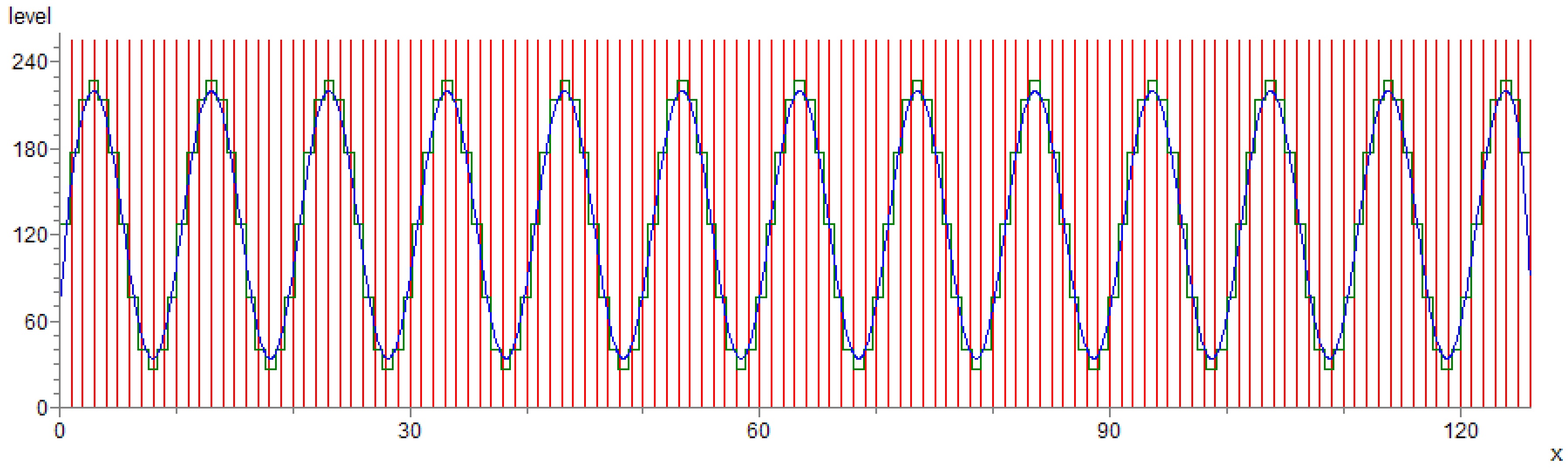
If the filter function has got radius r then the integral really only runs from x-r to x+r and conv(x) is the result of averaging g over the interval [x-r,x+r].
For the bitmap rescaling the input function g is the scaled step function of the colour levels:
$g(y) = OldPixelStep(y \cdot scale)$ where $scale = \dfrac{OldWidth}{NewWidth}$. The averaged function conv(x) is a smoothed-out version of the downscaled step function.

Convolution results for some filters

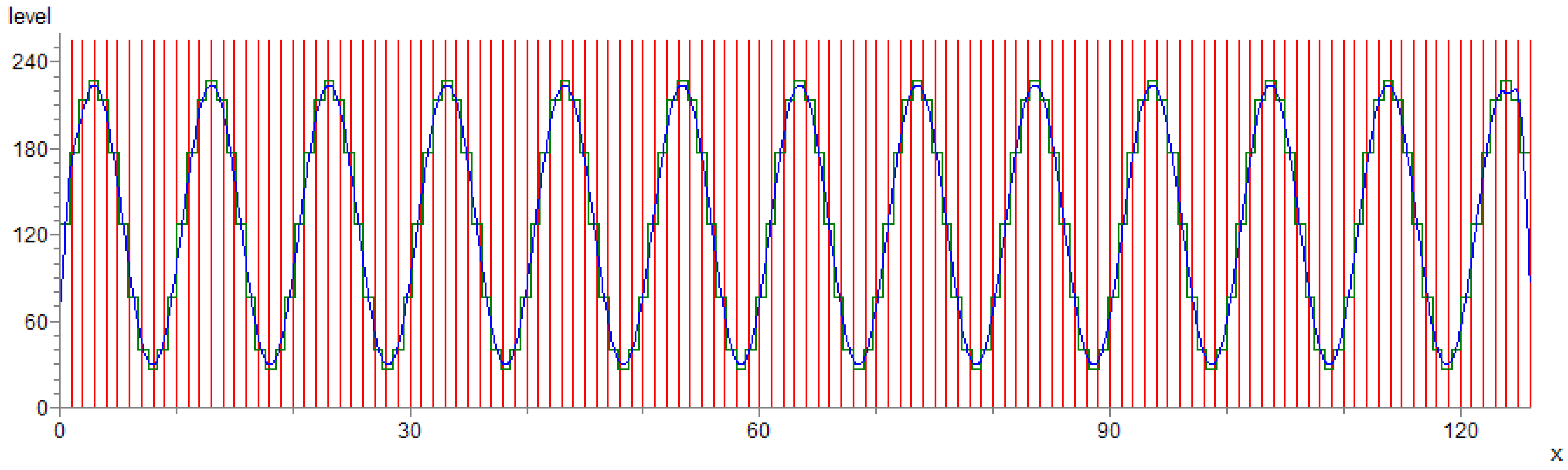
Here is the result, when we average our bitmap line example with the box-filter, radius 0.5:



The blue graph shows the function conv(x). Notice that some of the mountain tops and valley bottoms are rather sharp.
The result for the BSpline-filter with radius 2 is shown below:



The averaged graph is a bit smoother, but the averaging has caused some loss of contrast/sharpness. The result for the Lanczos-filter, or, rather an approximation of it, is shown below:



The filter preserves contrast/sharpness well while being as smooth as the BSpline.

What about the boundary?

Our bitmap step function ends at y=0 and y=126, but the filter-interval for some x-values goes beyond that. It's not a good idea to assume that the input of the bitmap in these stand-over intervals is 0. That's too abrupt a change. There are two good approaches to deal with this:

- We assume the value at the boundary to be repeated throughout the empty intervals (that's what I have done)
- We assume the bitmap to be mirrored in the empty intervals (probably better, that's Anders's and Mike's approach, and I think Eric's, too)

The role of the radius

The larger the radius the more does the convolution approximate a uniform value, the rate depending on the filter. But as the radius tends to 0, the filters which don't change sign approach the "delta-function" and the convolution gets closer and closer to the input function. We need to keep in mind, that our target bitmap consists of length-one pixels, and that we need to sample the conv(x)-function at discrete values x_i to assign a colour to pixel i in the target bitmap. As the radius tends to 0, this results in resampling being closer and closer to nearest-neighbour. So in implementations one must find a good radius in between that gives a most pleasing result. The default radii given for the filter functions are a good starting point. The default radius 0.5 of the box filter seems natural as the filter interval then covers exactly one new pixel.

Colour values for the target pixels

So far the convolution gives an average value for each real x in the interval [0,NewWidth]:

$conv(x) = \int_{x-r}^{x+r} \frac{1}{r} filt(\frac{y-x}{r}) \cdot OldPixelStep(y \cdot scale) dy$ with $scale = \frac{OldWidth}{NewWidth}$ and *filt* being a normalized filter function. The most natural way to assign a colour value to target pixel i would be to take the value of conv(x), where x is in the centre of the pixel-interval [i,i+1]. In fact, that's the only way to make the algorithm symmetric and consistent with rotations and flips of the bitmap. So:

$$NewPixel(i) = conv(i`+0.5) = \int_{i+0.5-r}^{i+0.5+r} \frac{1}{r} filt(\frac{y-i-0.5}{r}) \cdot OldPixelStep(y \cdot scale) dy$$

If we take the box filter with radius 0.5 we end up with

$NewPixel(i) = \int_{i}^{i+1} OldPixelStep(y \cdot scale) dy$, Eric's resampling algorithm.

Now for the implementation of the algorithm it's better to change the variable in the integral to $z = y \cdot scale \quad dz = scale \cdot dy$

$NewPixel(i) = \frac{1}{scale} \int_{(i+0.5-r)\cdot scale}^{(i+0.5+r)\cdot scale} \frac{1}{r} filt(\frac{z/scale - i - 0.5}{r}) \cdot OldPixelStep(z) dz$ (yuck). This can be read a bit better introducing $R = r \cdot scale$ and $x_i = (i+0.5) \cdot scale$:

$$NewPixel(i) = \frac{1}{R}\int_{x_i-R}^{x_i+R} filt(\frac{1}{R}(z-x_i))\cdot OldPixelStep(z)dz$$

This still looks ugly, but the pixel step function is piecewise constant, so the new pixel value can be computed from the old pixel, if we split the integral into pieces of length one.
For each i we need to compute the lowest pixel j = j_{min}(i) and the highest pixel j=j_{max}(i) in the source bitmap whose pixel interval [j,j+1] overlaps [x_i-R, x_i+R]. A safe value for these turns out to be
$j_{\min}(i) = Ceil(x_i - R - 1)$ $j_{\max}(i) = Floor(x_i + R)$ (hope I got that right ☺). And now we get

$$NewPixel(i) = \frac{1}{R}\sum_{j=j_{\min}(i)}^{j=j_{\max}(i)}\int_{j}^{j+1} filt(\frac{1}{R}(z-x_i))dz\cdot OldPixel(j)$$

The integral in this formula defines the weight Weight(i,j) which OldPixel(j) gets in the resampling. Note that up to now it doesn't matter if an interval [j,j+1] is partially outside of [x_i-R, x_i+R], the filter being 0 there takes care of it.
There are several ways to compute the integral in this formula:

Exact integral formula:
Exact value using the antiderivative of the filter, *AntiFilter* being an antiderivative of *filt*:

$$Weight\_Exact(i,j) = \frac{1}{R}\int_{j}^{j+1} filt(\frac{1}{R}(z-x_i))dz = \int_{\frac{j-x_i}{R}}^{\frac{j+1-x_i}{R}} filt(y)dy = AntiFilter(\frac{j-x_i}{R}+\frac{1}{R}) - AntiFilter(\frac{j-x_i}{R})$$

Drawbacks: The antiderivative can't be computed in closed form for the Lanczos-filter. In implementations this evaluation seems to cause loss of detail in the target bitmap.

Using the midpoint rule:
For this we need to intersect the interval for the integral with the support of the filter for a more accurate approximation.

$$Weight\_Mid(i,j) = \int_{\frac{j-x_i}{R}}^{\frac{j+1-x_i}{R}} filt(y)dy = \int_{low(i,j)}^{up(i,j)} filt(y)dy \approx \big(up(i,j) - low(i,j)\big)filt\big(mid(i,j)\big)$$

Where $up(i,j) = \min(\frac{j-x_i}{R}+\frac{1}{R},1)$, $low(i,j) = \max(\frac{j-x_i}{R},-1)$ and $mid(i,j) = \frac{up(i,j)+low(i,j)}{2}$
These weights seem to result in rescalings which preserve details and reduce artefacts almost equally well.

Using the trapezoidal rule:

$$Weight\_Trap(i,j) = \int_{\frac{j-x_i}{R}}^{\frac{j+1-x_i}{R}} filt(y)dy = \int_{low(i,j)}^{up(i,j)} filt(y)dy \approx 0.5\big(up(i,j) - low(i,j)\big)(filt(up(i,j) + filt(low(i,j))$$

with up(i,j) and low(i,j) as before.

In my implementation I use a mixture of the midpoint rule and the trapezoidal rule, resulting in a generalized Simpson's rule.

So we have the contributing pixels of the original bitmap to the pixel value of the new bitmap at i and their weights. When using the midpoint- or trapezoidal rule we also need to modify the weights a tiny bit, so they sum up to 1. This is not necessary when using the exact integral formula. We have left this step out in the summing up below.

Summing up the algorithm for downscaling using the midpoint rule (analogous for the other integral-evaluations):

$scale = \frac{OldWidth}{NewWidth}$ $\quad r = filter\ radius$ $\quad R = r \cdot scale$ $\quad i = new\ pixel\ location$ $\quad x_i = (i + 0.5) \cdot scale$ $\quad j_{\min}(i) = Ceil(x_i - R - 1)$ $\quad j_{\max}(i) = Floor(x_i + R)$

$j_{\min}(i) \le j \le j_{\max}(i) \ = Range\ of\ contributing\ pixel\ locations\ j\ in\ source\ bitmap$

$up(i,j) = \min\left(\frac{j - x_i}{R} + \frac{1}{R}, 1\right)$ $\quad low(i,j) = \max(\frac{j - x_i}{R}, -1)$ $\quad mid(i,j) = \frac{up(i,j) + low(i,j)}{2}$ $\quad weight(i,j) = \left(up(i,j) - low(i,j)\right) filt(mid(i,j))$

$$NewPixel(i) = \sum_{j=j_{\min}(i)}^{j=j_{\max}(i)} weight(i,j) \cdot OldPixel(j)$$

Algorithm for upscaling

The situation for upscaling is very similar to downscaling; only the radius is now measured in pixel width of the source (smaller) bitmap rather than in the units of the target. This means, that the same algorithm as for downscaling applies, except that R has to be replaced by the original filter radius r.

Bitmaps are two-dimensional! – And – about optimization

Sure, up to now we have only resampled one line of a bitmap, and only one colour component. For the other colour components the same weights must be used to get the corresponding colour component of the target pixel.
Since the weights and the contributing pixels of the source bitmap are the same in each line, it is a good idea to compute that only once and use it in each line.
Each scanline of the source bitmap can be resampled as described, but then the result must be resampled in the vertical direction in the same way, using scale=OldHeight/NewHeight.
Again, the weights and the contributors for each column are the same, so should be computed in advance.
One approach is, to first only scale the bitmap horizontally, store the result in a temporary bitmap, and then scale that one vertically. For nitpickers, this isn't good enough, because of the intermediate bitmap using bytes to store the colour channels. This leads to rounding imprecisions and the scaling of a rotated bitmap wouldn't be the same as the rotation of a scaled bitmap. To overcome this, a cache can be used that has integer-valued colour-channels. This approach is used both in Graphics32 and in my implementation. Also, the calculations have to be done in a way that doesn't jump around in bitmaps too much, since that leads to memory being swapped in and out of the processor cache too much.