Pamscale User Manual

pamscale

Updated: 18 February 2005
Table Of Contents

NAME

pamscale - scale a Netpbm image

SYNOPSIS

pamscale
[
scale_factor
|
{**-xyfit** | **-xyfill** | **-xysize**} *cols rows*
|
-reduce *reduction_factor*
|
[**-xsize=***cols* | **-width=***cols* | **-xscale=***factor*]
[**-ysize=***rows* | **-height=***rows* | **-yscale=***factor*]
|
-pixels *n*
]
[
[**-verbose**]
[
-nomix
|
-filter=*functionName* [**-window=**functionName]
]
[**-linear**]
}
[*pnmfile*]

<?makeman .SH OPTION USAGE ?>

Minimum unique abbreviation of option is acceptable. You may use double hyphens instead of single hyphen to denote options. You may use white space in place of the equals sign to separate an option name from its value.

DESCRIPTION

This program is part of Netpbm.

pamscale scales a Netpbm image by a specified factor, or scales individually horizontally and vertically by specified factors.

You can either enlarge (scale factor > 1) or reduce (scale factor < 1).

The Scale Factors

When you specify an absolute size or scale factor for both dimensions, **pamscale** scales each dimension independently without consideration of the aspect ratio.

If you specify one dimension as a pixel size and don't specify the other dimension, **pamscale** scales the unspecified dimension to preserve the aspect ratio.

If you specify one dimension as a scale factor and don't specify the other dimension, **pamscale** leaves the unspecified dimension unchanged from the input.

If you specify the *scale_factor* parameter instead of dimension options, that is the scale factor for both dimensions. It is equivalent to **-xscale=***scale_factor* **-yscale=***scale_factor*.

Specifying the **-reduce** *reduction_factor* option is equivalent to specifying the *scale_factor* parameter, where *scale_factor* is the reciprocal of *reduction_factor*.

-xyfit specifies a bounding box. **pamscale** scales the input image to the largest size that fits within the box, while preserving its aspect ratio. **-xysize** is a synonym for this. Before Netpbm 10.20 (January 2004), **-xyfit** did not exist, but **-xysize** did.

-xyfill is similar, but **pamscale** scales the input image to the smallest size that completely fills the box, while preserving its aspect ratio. This option has existed since Netpbm 10.20 (January 2004).

-pixels specifies a maximum total number of output pixels. **pamscale** scales the image down to that number of pixels. If the input image is already no more than that many pixels, **pamscale** just copies it as output; **pamscale** does not scale up with **-pixels**.

If you enlarge by a factor of 3 or more, you should probably add a *pnmsmooth* step; otherwise, you can see the original pixels in the resulting image.

Usage Notes

A useful application of **pamscale** is to blur an image. Scale it down (without **-nomix**) to discard some information, then scale it back up using **pamstretch**.

Or scale it back up with **pamscale** and create a "pixelized" image, which is sort of a computer-age version of blurring.

Transparency

pamscale understands transparency and properly mixes pixels considering the pixels' transparency.

Proper mixing does not mean just mixing the transparency value and the color component values separately. In a PAM image, a pixel which is not opaque represents a color that contains light of the foreground color indicated explicitly in the PAM and light of a background color to be named later. But the numerical scale of a color component sample in a PAM is as if the pixel is opaque. So a pixel that is supposed to contain half-strength red light for the foreground plus some light from the background has a red color sample that says full red and a transparency sample that says 50% opaque. In order to mix pixels, you have to first convert the color sample values to numbers that represent amount of light directly (i.e. multiply by the opaqueness) and after mixing, convert back (divide by the opaqueness).

Input And Output Image Types

pamscale produces output of the same type (and tuple type if the type is PAM) as the input, except if the input is PBM. In that case, the output is PGM with maxval 255. The purpose of this is to allow meaningful pixel mixing. Note that there is no equivalent exception when the input is PAM. If the PAM input tuple type is BLACKANDWHITE, the PAM output tuple type is also BLACKANDWHITE, and you get no meaningful pixel mixing.

If you want PBM output with PBM input, use **pamditherbw** to convert **pamscale**'s output to PBM. Also consider **pbmreduce**.

pamscale's function is essentially undefined for PAM input images that are not of tuple type RGB, GRAYSCALE, BLACKANDWHITE, or the _ALPHA variations of those. (By standard Netpbm backward compatibility, this includes PBM, PGM, and PPM images).

You might think it would have an obvious effect on other tuple types, but remember that the aforementioned tuple types have gamma-adjusted sample values, and **pamscale** uses that fact in its calculations. And it treats a transparency plane different from any other plane.

pamscale does not simply reject unrecognized tuple types because there's a possibility that just by coincidence you can get useful function out of it with some other tuple type and the right combination of options (consider **-linear** in particular).

Methods Of Scaling

There are numerous ways to scale an image. **pamscale** implements a bunch of them; you select among them with invocation options.

Pixel Mixing

Pamscale's default method is pixel mixing. To understand this, imagine the source image as composed of square tiles. Each tile is a pixel and has uniform color. The tiles are all the same size. Now lay over that a transparent sheet of the same size, marked off in a square grid. Each cell in the grid stands for a pixel of the target image. For example, if you are scaling a 100x200 image up by 1.5, the source image is 100 x 200 tiles, and the transparent sheet is marked off in 150 x 300 cells.

Each cell covers parts of multiple tiles. To make the target image, just color in each cell with the color which is the average of the colors the cell covers -- weighted by the amount of that color it covers. A cell in our example might cover 4/9 of a blue tile, 2/9 of a red tile, 2/9 of a green tile, and 1/9 of a white tile. So the target pixel would be somewhat unsaturated blue.

When you are scaling up or down by an integer, the results are simple. When scaling up, pixels get duplicated. When scaling down, pixels get thrown away. In either case, the colors in the target image are a subset of those in the source image.

When the scale factor is weirder than that, the target image can have colors that didn't exist in the original. For example, a red pixel next to a white pixel in the source might become a red pixel, a pink pixel, and a white pixel in the target.

This method tends to replicate what the human eye does as it moves closer to or further away from an image. It also tends to replicate what the human eye sees, when far enough away to make the pixelization disappear, if an image is not made of pixels and simply stretches or shrinks.

Discrete Sampling

Discrete sampling is basically the same thing as pixel mixing except that, in the model described above, instead of averaging the colors of the tiles the cell covers, you pick the one color that covers the most area.

The result you see is that when you enlarge an image, pixels get duplicated and when you reduce an image, some pixels get discarded.

The advantage of this is that you end up with an image made from the same color palette as the original. Sometimes that's important.

The disadvantage is that it distorts the picture. If you scale up by 1.5 horizontally, for example, the even numbered input pixels are doubled in the output and the odd numbered ones are copied singly. If you have a bunch of one pixel wide lines in the source, you may find that some of them stretch to 2 pixels, others remain 1 pixel when you enlarge. When you reduce, you may find that some of the lines disappear completely.

You select discrete sampling with **pamscale**'s **-nomix** option.

Actually, **-nomix** doesn't do exactly what I described above. It does the scaling in two passes - first horizontal, then vertical. This can produce slightly different results.

There is one common case in which one often finds it burdensome to have **pamscale** make up colors that weren't there originally: Where one is working with an image format such as GIF that has a limited number of possible colors per image. If you take a GIF with 256 colors, convert it to PPM, scale by .625, and convert back to GIF, you will probably find that the reduced image has way more than 256 colors, and therefore cannot be converted to GIF. One way to solve this problem is to do the reduction with discrete sampling instead of pixel mixing. Probably a better way is to do the pixel mixing, but then color quantize the result with **pnmquant** before converting to GIF.

When the scale factor is an integer (which means you're scaling up), discrete sampling and pixel mixing are identical -- output pixels are always just N copies of the input pixels. In this case, though, consider using **pamstretch** instead of **pamscale** to get the added pixels interpolated instead of just copied and thereby get a smoother enlargement.

pamscale's discrete sampling is faster than pixel mixing, but **pamenlarge** is faster still. **pamenlarge** works only on integer enlargements.

discrete sampling (**-nomix**) was new in Netpbm 9.24 (January 2002).

Resampling

Resampling assumes that the source image is a discrete sampling of some original continuous image. That is, it assumes there is some non-pixelized original image and each pixel of the source image is simply the color of that image at a particular point. Those points, naturally, are the intersections of a square grid.

The idea of resampling is just to compute that original image, then sample it at a different frequency (a grid of a different scale).

The problem, of course, is that sampling necessarily throws away the information you need to rebuild the original image. So we have to make a bunch of assumptions about the makeup of the original image.

You tell **pamscale** to use the resampling method by specifying the **-filter** option. The value of this option is the name of a function, from the set listed below.

To explain resampling, we are going to talk about a simple one dimensional scaling -- scaling a single row of grayscale pixels horizontally. If you can understand that, you can easily understand how to do a whole image: Scale each of the rows of the image, then scale each of the resulting columns. And scale each of the color component planes separately.

As a first step in resampling, **pamscale** converts the source image, which is a set of discrete pixel values, into a continuous step function. A step function is a function whose graph is a staircase-y thing.

Now, we convolve the step function with a proper scaling of the filter function that you identified with **-filter**. If you don't know what the mathematical concept of convolution (convolving) is, you are officially lost. You cannot understand this explanation. The result of this convolution is the imaginary original continuous image we've been talking about.

Finally, we make target pixels by picking values from that function.

To understand what is going on, we use Fourier analysis:

The idea is that the only difference between our step function and the original continuous function (remember that we constructed the step function from the source image, which is itself a sampling of the original continuous function) is that the step function has a bunch of high frequency Fourier components added. If we could chop out all the higher frequency components of the step function, and know that they're all higher than any frequency in the original function, we'd have the original function back.

The resampling method assumes that the original function was sampled at a high enough frequency to form a perfect sampling. A perfect sampling is one from which you can recover exactly the original continuous function. The Nyquist theorem says that as long as your sample rate is at least twice the highest frequency in your original function, the sampling is perfect. So we assume that the image is a sampling of something whose highest frequency is half the sample rate (pixel resolution) or less. Given that, our filtering does in fact recover the original continuous image from the samples (pixels).

To chop out all the components above a certain frequency, we just multiply the Fourier transform of the step function by a rectangle function.

We could find the Fourier transform of the step function, multiply it by a rectangle function, and then Fourier transform the result back, but there's an easier way. Mathematicians tell us that multiplying in the frequency domain is equivalent to convolving in the time domain. That means multiplying the Fourier transform of F by a rectangle function R is the same as convolving F with the Fourier transform of R. It's a lot better to take the Fourier transform of R, and build it into **pamscale** than to have **pamscale** take the Fourier transform of the input image dynamically.

That leaves only one question: What is the Fourier transform of a rectangle function? Answer: sinc. Recall from math that sinc is defined as sinc(x) = sin(PI*x)/PI*x.

Hence, when you specify **-filter=sinc**, you are effectively passing the step function of the source image through a low pass frequency filter and recovering a good approximation of the original continuous image.

Refiltering

There's another twist: If you simply sample the reconstructed original continuous image at the new sample rate, and that new sample rate isn't at least twice the highest frequency in the original continuous image, you won't get a perfect sampling. In fact, you'll get something with ugly aliasing in it. Note that this can't be a problem when you're scaling up (increasing the sample rate), because the fact that the old sample rate was above the Nyquist level means so is the new one. But when scaling down, it's a problem. Obviously, you have to give up image quality when scaling down, but aliasing is not the best way to do it. It's better just to remove high frequency components from the original continuous image before sampling, and then get a perfect sampling of that.

Therefore, **pamscale** filters out frequencies above half the new sample rate before picking the new samples.

Approximations

Unfortunately, **pamscale** doesn't do the convolution precisely. Instead of evaluating the filter function at every point, it samples it -- assumes that it doesn't change any more often than the step function does. **pamscale** could actually do the true integration fairly easily. Since the filter functions are built into the program, the integrals of them could be too. Maybe someday it will.

There is one more complication with the Fourier analysis. sinc has nonzero values on out to infinity and minus infinity. That makes it hard to compute a convolution with it. So instead, there are filter functions that approximate sinc but are nonzero only within a manageable range. To get those, you multiply the sinc function by a *window function*, which you select with the **-window** option. The same holds for other filter functions that go on forever like sinc. By default, for a filter that needs a window function, the window function is the Blackman function.

Filter Functions Besides Sinc

The math described above works only with sinc as the filter function. **pamscale** offers many other filter functions, though. Some of these approximate sinc and are faster to compute. For most of them, I have no idea of the mathematical explanation for them, but people do find they give pleasing results. They may not be based on resampling at all, but just exploit the fact the convolution that is coincidentally part of a resampling calculation.

For some filter functions, you can tell just by looking at the convolution how they vary the resampling process from the perfect one based on sinc:

The impulse filter assumes that the original continuous image is in fact a step function -- the very one we computed as the first step in the resampling. This is mathematically equivalent to the discrete sampling method.

The box (rectangle) filter assumes the original image is a piecewise linear function. Its graph just looks like straight lines connecting the pixel values. This is mathematically equivalent to the pixel mixing method when scaling down, and interpolation (ala **pamstretch**) when scaling up.

Gamma

pamscale assumes the underlying continuous function is a function of brightness (as opposed to light intensity), and therefore does all this math using the gamma-adjusted numbers found in a PNM or PAM image. The **-linear** option is not available with resampling (it causes **pamscale** to fail), because it wouldn't be useful enough to justify the implementation effort.

Resampling (**-filter**) was new in Netpbm 10.20 (January 2004).

The filter functions

Here is a list of the function names you can specify for the **-filter** option. For most of them, you're on your own to figure out just what the function is and what kind of scaling it does. These are common functions from mathematics.

point The graph of this is a single point at X=0, Y=1.

box The graph of this is a rectangle sitting on the X axis and centered on the Y axis with height 1 and base 1.

triangle The graph of this is an isosceles triangle sitting on the X axis and centered on the Y axis with height 1 and base 2.

quadratic

cubic

catrom

mitchell

gauss

sinc

bessel

hanning

hamming

blackman

kaiser

normal

hermite

lanczos

Linear vs Gamma-adjusted

The pixel mixing scaling method described above involves intensities of pixels (more precisely, it involves individual intensities of primary color components of pixels). But the PNM and PNM-equivalent PAM image formats represent intensities with gamma-adjusted numbers that are not linearly proportional to intensity. So **pamscale**, by default, performs a calculation on each sample read from its input and each sample written to its output to convert between these gamma-adjusted numbers and internal intensity-proportional numbers.

Sometimes you are not working with true PNM or PAM images, but rather a variation in which the sample values are in fact directly proportional to intensity. If so, use the **-linear** option to tell **pamscale** this. **pamscale** then will skip the conversions.

The conversion takes time. In one experiment, it increased the time required to reduce an image by a factor of 10. And the difference between intensity-proportional values and gamma-adjusted values may be small enough that you would barely see a difference in the result if you just pretended that the gamma-adjusted values were in fact intensity-proportional. So just to save time, at the expense of some image quality, you can specify **-linear** even when you have true PPM input and expect true PPM output.

For the first 13 years of Netpbm's life, until Netpbm 10.20 (January 2004), **pamscale**'s predecessor **pnmscale** always treated the PPM samples as intensity-proportional even though they were not, and drew few complaints. So using **-linear** as a lie is a reasonable thing to do if speed is important to you. But if speed is important, you also should consider the **-nomix** option and **pnmscalefixed**.

Another technique to consider is to convert your PNM image to the linear variation with **pnmgamma**, run **pamscale** on it and other transformations that like linear PNM, and then convert it back to true PNM with **pnmgamma -ungamma**. **pnmgamma** is often faster than **pamscale** in doing the conversion.

With **-nomix**, **-linear** has no effect. That's because **pamscale** does not concern itself with the meaning of the sample values in this method; **pamscale** just copies numbers from its input to its output.

Precision

pamscale uses floating point arithmetic internally. There is a speed cost associated with this. For some images, you can get the acceptable results (in fact, sometimes identical results) faster with **pnmscalefixed**, which uses fixed point arithmetic. **pnmscalefixed** may, however, distort your image a little. See the **pnmscalefixed** user manual for a complete discussion of the difference.

SEE ALSO

pnmscalefixed, **pamstretch**, **pamditherbw**, **pbmreduce**, **pbmpscale**, **pamenlarge**, **pnmsmooth**, **pamcut**, **pnmgamma**, **pnmscale**, **pnm**, **pam**

HISTORY

pamscale was new in Netpbm 10.20 (January 2004). It was adapted from, and obsoleted, **pnmscale**. **pamscale**'s primary difference from **pnmscale** is that it handles the PAM format and uses the "pam" facilities of the Netpbm programming library. But it also added the resampling class of scaling method. Furthermore, it properly does its pixel mixing arithmetic (by default) using intensity-proportional values instead of the gamma-adjusted values the **pnmscale** uses. To get the old **pnmscale** arithmetic, you can specify the **-linear** option.

The intensity proportional stuff came out of suggestions by Adam M Costello in January 2004.

The resampling algorithms are mostly taken from code contributed by Michael Reinelt in December 2003.

The version of **pnmscale** from which **pamscale** was derived, itself evolved out of the original Pbmplus version of **pnmscale** by Jef Poskanzer (1989, 1991). But none of that original code remains.

Table Of Contents

HISTORY