Fast Gaussian Blur

David Eberly
Magic Software, Inc.
http://www.magic-software.com

Created: March 2, 1999
Modified: October 28, 2002

The most popular method we have used here for blurring is to do it as a convolution of a Gaussian kernel with the image by using fast Fourier transforms. However, the implementations of FFTs usually have two problems. The first problem is that a nonnegative image when blurred by an FFT may have negative values as a result of numerical round–off errors. A region of positive measure for which the initial image is identically zero becomes a region for which there are many sign fluctuations on numbers of small magnitude. The fluctuations create many problems in my other applications which require Gaussian blurring. The second problem is that for large scale blurring, the FFTs produced artifacts (near the four corners of 2–dimensional images) which look like bright four–point stars. This also causes problems in my applications.

An alternate approach to Gaussian blurring is to notice that it is equivalent to solving an initial value problem for the partial differential equation $u_t = \nabla^2 u$ over $\mathbb{R}^n$ where the initial data is the image you want blurred. Large t corresponds to large scale blurring. Unfortunately, you must iterate through time. To get a blurred image at large scale using the standard finite difference methods requires a lot of iterations.

I have a different approach. The standard deviation of the Gaussian is related to time by $t = \sigma^2/2$. The partial differential equation in this coordinate system is $\sigma u_\sigma = \sigma^2 \nabla^2 u$. You can set up a finite difference scheme by observing that

$$\sigma u_\sigma = \lim_{b \to 1} \frac{u(x, b\sigma) - u(x, \sigma)}{\ln(b)}$$

and

$$\sigma^2 u_{xx} = \lim_{h \to 0} \frac{u(x + h\sigma, \sigma) - 2u(x, \sigma) + u(x - h\sigma, \sigma)}{h^2}.$$

Similar spatial derivative formulas occur for each spatial variable. If you have samples $u(x_i, \sigma)$ for positions x_i, sample scale as $\sigma_k = \sigma_0 b^k$ for some $\sigma_0 > 0$ and for some $b > 1$, and if you want to sample $u(x_i, \sigma_j)$, then the finite difference scheme to do so is

$$u(x_i, \sigma_{j+1}) = u(x_i, \sigma_j) + \ln(b)[u(x_i + \sigma_j, \sigma_j) - 2u(x_i, \sigma_j) + u(x_i - \sigma_j)].$$

The quantities $x_i \pm \sigma_j$ may not be grid points themselves, but I do a linear interpolation of the nearest image values to estimate u at the nongrid points. If the $x_i \pm \sigma_j$ are outside the grid, I just clip them to the image boundary. The ideas extend naturally to higher dimensions.

You still need to worry about numerical stability of the algorithm. Using a method similar to the one you apply to the regular finite difference problem, you need $\ln(b) < 0.5$ for dimension 1, so $b < \exp(0.5) \doteq 1.649$. For dimension 2 with equal x and y grid spacings, you need $b < \exp(0.25) \doteq 1.284$. For dimension 3 with equal x, y, and z grid spacings, you need $b < \exp(1/6) \doteq 1.181$. However, note that the scale sampling is *geometric*, not linear: b is a multiplier. Moreover, the method is asymptotically stable, so the total error

tends to decrease as scale increases. This means you can blur to a larger scale by much fewer steps if you use the above approach.

A simple test on a fast workstation for a $256 \times 256 \times 128$ data set showed that Gaussian blurring via a general purpose FFT program took about 1 hour (which unfortunately included some swap time), but the finite difference scheme as described took about 5 minutes for a *single scale*. For larger scales, the finite difference scheme must be iterated. As long as the number of iterations keeps the total time less than that for the FFT program, the algorithm is effective. Of course, the trade–off is that the FFT generally has smaller errors than the finite difference. But the real measure needed for processing large data sets is the *cost* which balances error and runtime.

For an implementation, see the image analysis source directory, file `FastBlur.cpp`.