Extended Binomial Filter for Fast Gaussian Blur

Abstract

The extended binomial filter algorithm is a very simple and efficient Gaussian blur algorithm where the run time per pixel is independent of the blur radius.

The Gaussian blur is a widely used filter for many effects, especially for image processing. It is important to have a fast and easy algorithm for computation. The runtime of most algorithms for calculating the Gaussian blur like the binomial sequence is proportional to the blur radius *r*. This disadvantage of having a complexity of O(*r*) per pixel makes such algorithm very slow for larger blur radii.

The extended binomial filter is an approximation of the normal binomial filter with constant complexity O(1), making the runtime per pixel independent of the blur radius.

The accuracy of the approximation is chosen by the approximation degree.

Key Features

- Constant complexity O(1) per pixel, independent of the blur radius.
- Minimum pixel readouts, also independent of the blur radius.
- Computation as simple (and fast) as box blur.
- Adequate approximation selectable to the desired accuracy.

Summery

The algorithm uses the advantage that the integral of a constant function is very easy to compute: it’s only one multiplication. The double integral of a constant function is equal easy to calculate: only the differences at the start and end have to be added. This method could be extended to any number of successive integrals of a constant function.

A function is first derived *n* times and then simplified to a sequence of step functions. The algorithm now takes this easy step functions and again integrates it *n* times to get to the approximation of the original function.

The algorithm also makes use of the benefit that consecutive pixels are calculated. So only the differences are added instead of summing up the values for every pixel again. These additions are done for all discrete derivatives. This way only a few additions per pixel have to be done. In its simplest form of the first degree the algorithm does only do a box blur. But the third degree approximation is accurate enough for most 8-bit images.

This principle of approximation is not limited to the Gaussian blur but may used for other filter functions too.

This algorithm is especially useful for large blur radii.

Keywords: Gaussian blur, efficient algorithm, binomial filter, approximation, convolution, gradient.

Content

Acronyms

CAS	Computer Algebra System
DFT	Discrete Fourier Transform
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
Pixel	Picture Element
RMS	Root Mean Square
SKIPSM	Separated-Kernel Image Processing using finite State Machines
USM	Unsharp Masking

1. Mathematical background

This chapter deals with the mathematical stuff behind the algorithm. If you are only interested in the algorithm you can skip this chapter but it helps to better understand the theory of operation.

For the mathematical description the image is thought as a continuous analog function. An image consists of a spatial domain *S*, which is the range of possible positions *x* and *y* of the image and a range domain *P*, which is the range of possible pixel values: $p = \mathrm{f}(x,y)$.

1.1. Unusual functions

A step function is needed for the piecewise function approximation. The Heaviside step function is defined as:

$$\Theta(x) = \frac{x}{2|x|} + \frac{1}{2} = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases} \qquad (1)$$

The Heaviside step function is the integral of the Dirac delta function: $\mathrm{d}\Theta = \delta$.

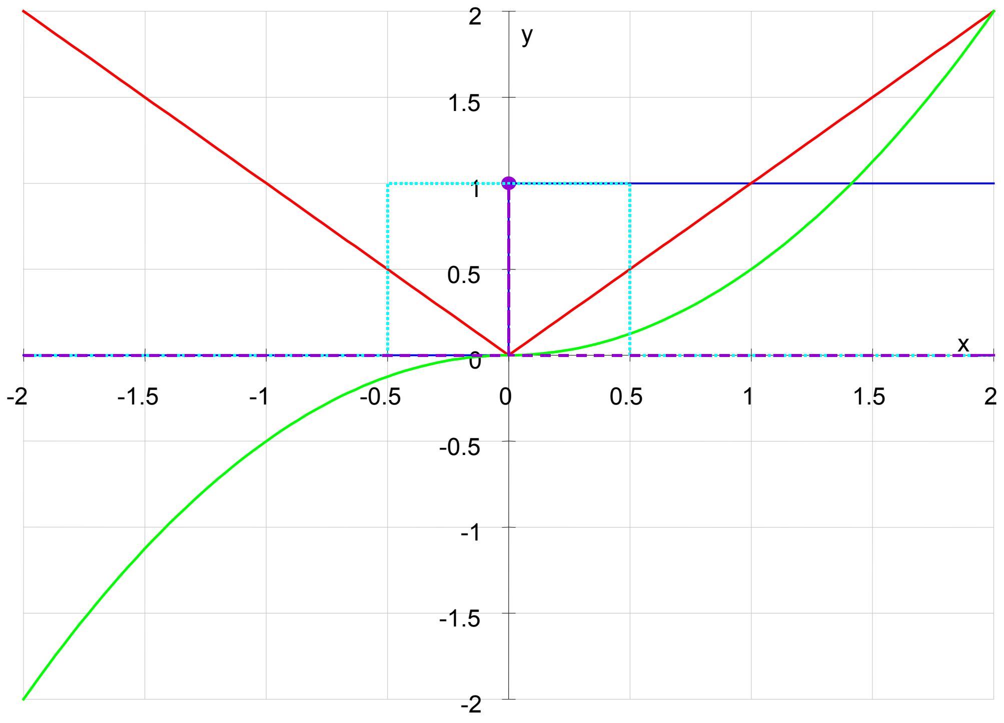
The rectangular function is defined as $\mathrm{rect}(x) = \Theta\left(x + \frac{1}{2}\right) - \Theta\left(x - \frac{1}{2}\right) = \sum_{i=0}^{1} \frac{(-1)^i}{2} \frac{2x+1-2i}{|2x+1-2i|}$.

Another not so common function needed is the absolute value with its derivative and integral:

$$\frac{\mathrm{d}}{\mathrm{d}x}|ax+c| = a\,\mathrm{sign}(ax+c) = a\frac{|ax+c|}{ax+c} = 2a\,\Theta(ax+c) - a\,,$$

$$\int \Theta(ax+c)\mathrm{d}x = \frac{|ax+c| + ax + c}{2a} + C \text{ and } \int |ax+c|\mathrm{d}x = \frac{(ax+c)|ax+c|}{2} + C\,.$$

The value of the Heaviside step function at point $x = 0$ is irrelevant for this application.



Picture 1 Unusual functions

a. Dirac delta function $\delta(x)$
b. Rectangular function
c. Heaviside step function: $\Theta(x)$
d. Absolute value: $|x|$
e. Integration of the absolute value: $\frac{|x|\,x}{2}$

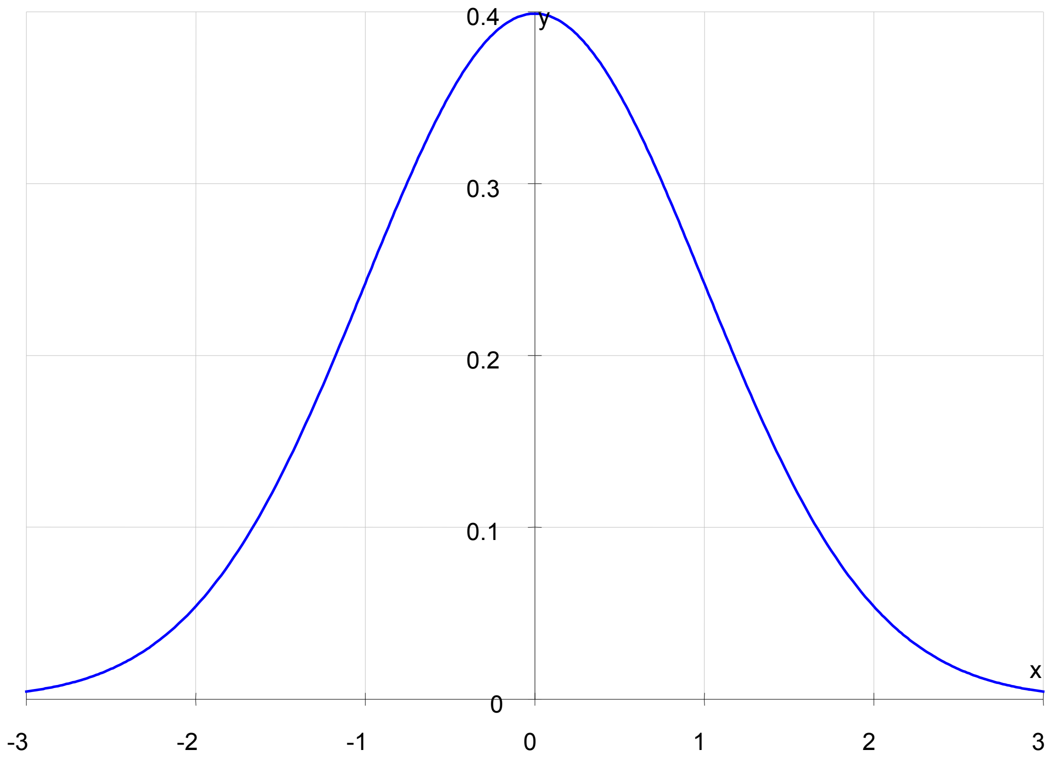
The floor function $\lfloor x \rfloor$ is further needed for discrete sequences.

1.2. Gaussian blur

The Gaussian blur uses the Gaussian function for calculating the transformation. This leads to very harmonic and smooth results. Each new pixel is set to a weighted average of that pixel's neighborhood.

In mathematic this could be expressed as *convolution* (german: *Faltung*) of the image function f(x) by the blur function g(x): $(\mathrm{f} * \mathrm{g})(x) = \int \mathrm{f}(s)\mathrm{g}(x-s)\,\mathrm{d}s$.

The matrix convolution kernel coefficients are selected to approximate the Gaussian distribution. The blur radius defines the size of the kernel.



Picture 2 Gaussian function for σ = 1

$$y = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

The equation of the Gaussian blur function is $\mathrm{g}_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

The parameter μ is the expected value and σ is the standard deviation, which is the blur radius in case of the Gaussian blur.

To ease the following calculations the standard normal deviation is used where σ is set to one and μ set to zero. This simple function is shown in picture 2:

$$\mathrm{g}(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \qquad (2)$$

The standard deviation can be calculated from the standard normal deviation: $\mathrm{g}_{\mu,\sigma}(x) = \frac{1}{\sigma}\mathrm{g}\left(\frac{x-\mu}{\sigma}\right)$.

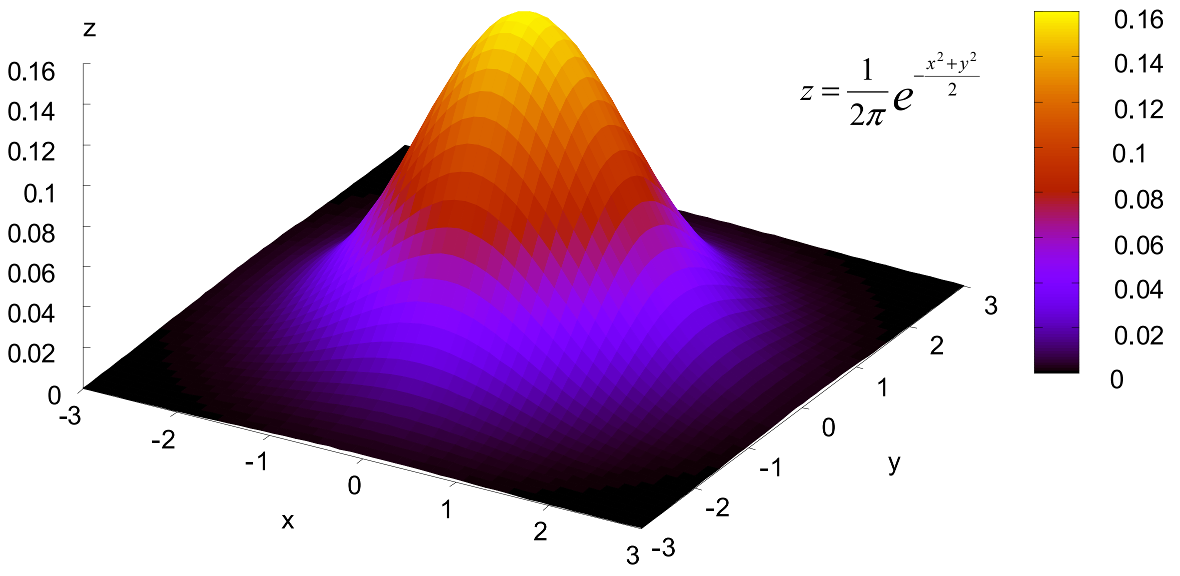
The simplification is taken into account if it influences the calculation.

Filter functions are not limited to one dimension. A two dimensional function is needed for images. If the filter function is linear separable the n dimensional filter function can be calculated by n one dimensional functions: $\mathrm{g}_n(\vec{\mathbf{x}}) = g_n(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} \mathrm{g}(x_i)$.

The Gaussian filter function of two dimensions in picture 3 is $\mathrm{g}_2(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} = \mathrm{g}(x)\mathrm{g}(y)$.

It is possible to select a different blur for x and y direction: $\mathrm{g}_2(x,y)=\frac{1}{2\pi\sigma_x\sigma_y}e^{-\frac{x^2}{2\sigma_x^2}-\frac{y^2}{2\sigma_y^2}}$

or even to perform the blur for any direction θ: $\mathrm{g}_\theta(x,y)=\frac{1}{2\pi\sigma_x\sigma_y}e^{-\frac{x^2\sigma_y^2+y^2\sigma_x^2+(\sigma_x^2-\sigma_y^2)((x^2-y^2)\sin^2\theta+xy\sin 2\theta)}{2\sigma_x^2\sigma_y^2}}$.

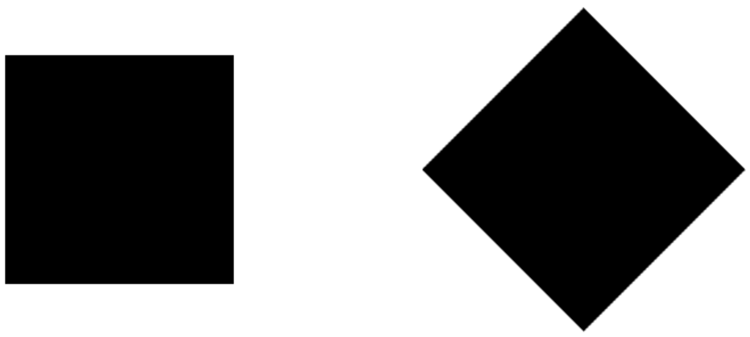


Picture 3 Gaussian bell shape in 3D

The area under function is $\int_{-\infty}^{\infty}\mathrm{g}(x)\,\mathrm{d}x=\int_{-\infty}^{\infty}\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{x^2}{2\sigma^2}}\,\mathrm{d}x=1$.

For two dimensions the volume under the function is $\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{g}_2(x,y)\mathrm{d}x\mathrm{d}y=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}\,\mathrm{d}x\mathrm{d}y=1$.

The approximation error could be measured by the root mean square (RMS) value between exact g(x) and approximation b(x) function: $e=\sqrt{\int(\mathrm{g}(x)-\mathrm{b}(x))^2dx}$. The error is at its maximum e = 1 for no approximation b(x) = 0. Since images are two dimensional and the Gaussian function is linear separable the two dimensional error could be measured by $e=\sqrt{\iint(\mathrm{g}(x)\mathrm{g}(y)-\mathrm{b}(x)\mathrm{b}(y))^2dxdy}$.



Picture 4 Unblurred sample image

Picture 5 shows the Gaussian blur of picture 4 with two squares, one of it rotated 45 degrees. Both look the same and very smooth, just as expected for blurring.

Picture 5 Gaussian blurred image

The standard deviation is defined as

$$\sigma = \sqrt{\int (x-\mu)^2 \, \mathrm{b}(x) \mathrm{d}\,x}\,, \tag{3}$$

where b(x) denotes the density function.

The expected value is defined as

$$\mu = \int x \, \mathrm{b}(x) \mathrm{d}\,x\,. \tag{4}$$

Standard deviation and expected value are defined by equation (3) and (4) for density function of the Gaussian function, since

$$\int_{-\infty}^{\infty} (x-\mu)^2 \, \mathrm{g}_{\mu,\sigma}(x) \mathrm{d}\,x = \int_{-\infty}^{\infty} \frac{(x-\mu)^2}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \mathrm{d}\,x = \sigma^2 \text{ and } \int_{-\infty}^{\infty} x \, \mathrm{g}_{\mu,\sigma}(x) \mathrm{d}\,x = \int_{-\infty}^{\infty} \frac{x}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \mathrm{d}\,x = \mu\,.$$

1.3. Box blur

Calculation of the Gaussian function is costly. Polynomial approximations promise better performance. The question is: which polynomial is the best compromise between accuracy and performance?

The box blur is a first simplification. For the 1st degree of approximation the Gaussian function is replaced by a constant.

The box blur simply sums up the values around a square. That is easy to perform but the result is not as smooth as the Gaussian blur.

The filter function of the box blur (rectangular function) is $\mathrm{b}_1(x) = a\left(\Theta\left(x+\frac{c}{2}\right) - \Theta\left(x-\frac{c}{2}\right)\right)$,

where Θ denotes the Heaviside step function, c is the width and a the high of the box.

One condition to define the constants a and c is that the area beneath the function should be the same for approximation and Gaussian function:

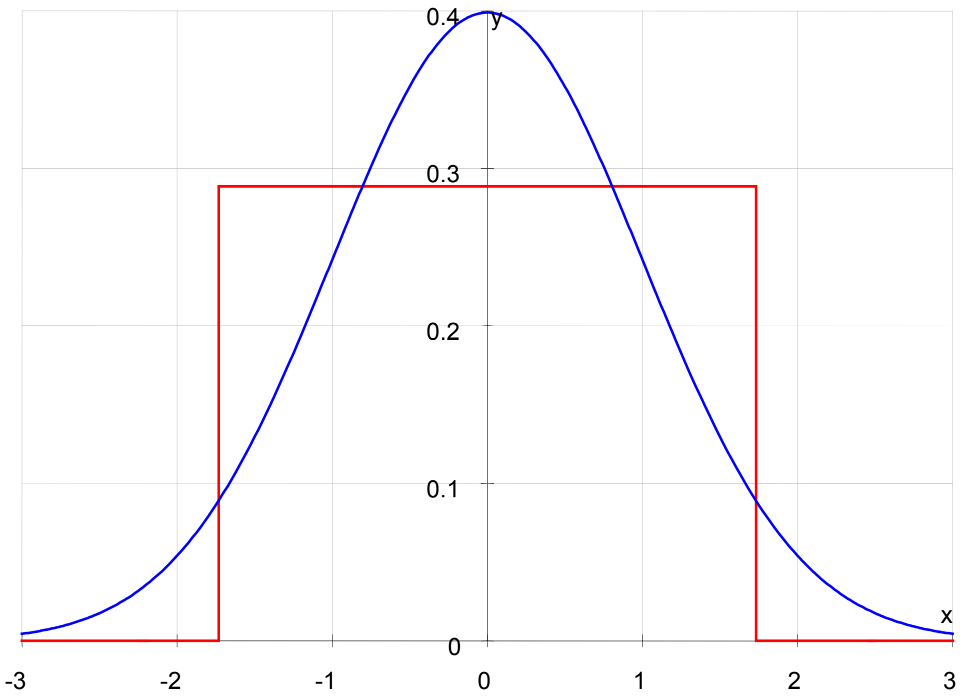
$$\int_{-\infty}^{\infty} \mathrm{b}_1(x) \mathrm{d}\,x = 2\int_0^{c/2} a \, \mathrm{d}\,x = ac = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \mathrm{d}\,x = 1, \text{ so } ac = 1.$$

The second condition is that the standard normal deviation for the approximation and Gaussian function according equation (2) is equal:

$$\sigma^2 = \int x^2 \, \mathrm{b}_1(x) \mathrm{d}\,x = \int_{-c/2}^{c/2} \frac{x^2}{c} \mathrm{d}\,x = \frac{c^2}{12} = 1, \text{ so } c = 2\sqrt{3} \text{ and } a = \frac{1}{2\sqrt{3}}.$$

The box blur function with these definitions is:

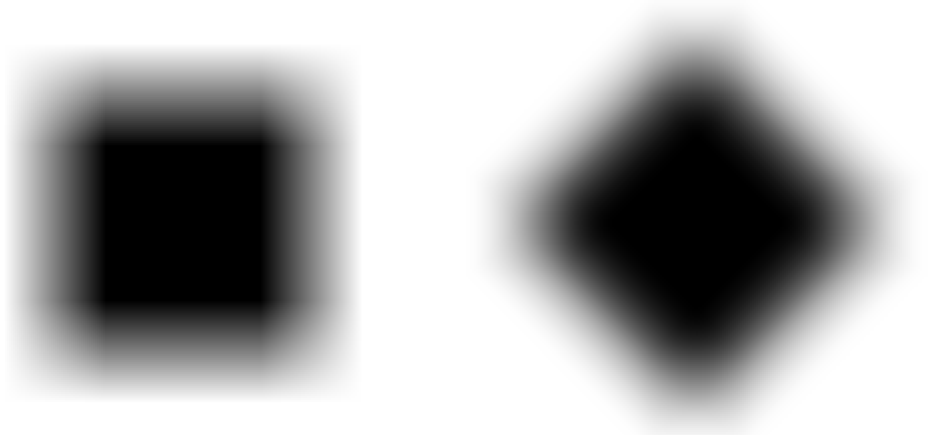
$$\mathrm{b}_1(x) = \frac{\Theta\left(x+\sqrt{3}\right) - \Theta\left(x-\sqrt{3}\right)}{2\sqrt{3}} \tag{5}$$



Picture 6: Box blur function

a. Box blur
b. Gaussian blur

When the area of the Gaussian function is defined as the maximum error, the error of the 1st degree of approximation is $e_1 = \sqrt{\int_{-\infty}^{\infty} (\mathrm{g}(x) - \mathrm{b}_1(x))^2 \, dx} = 0.2036969520$. The two dimensional error is 0.1511405983.



Picture 7: Box blur image

The squares of the box blur can still be seen in the picture 7. Also the two blurred squares don't look the same.

The benefit of the box blur is its speed. Since the blur is applied to every value two sums of neighboring values are nearly identical. Instead of doing individual sums only the differences to the previous sum are added. This way the runtime per value is constant and does not depending on the blur radius.

Can box blur and Gaussian blur somehow be combined to benefit from both algorithms?

1.4. Derivatives of Gauss

The box blur is a kind of simplification of the Gaussian bell shape. Constant values are added instead of a function. The derivative of the box function is two Dirac delta functions with spikes at the beginning and at the end. The integration is simply done during the iteration over all values. This can be used to adapt the approximation better to the bell shape. The step function is integrated again to build a slope function. Two integrations are nearly as simple as one since values must only be added at changes.

The highest degree of derivative consists only of constants. It is calculated by adding the corresponding pixel values.

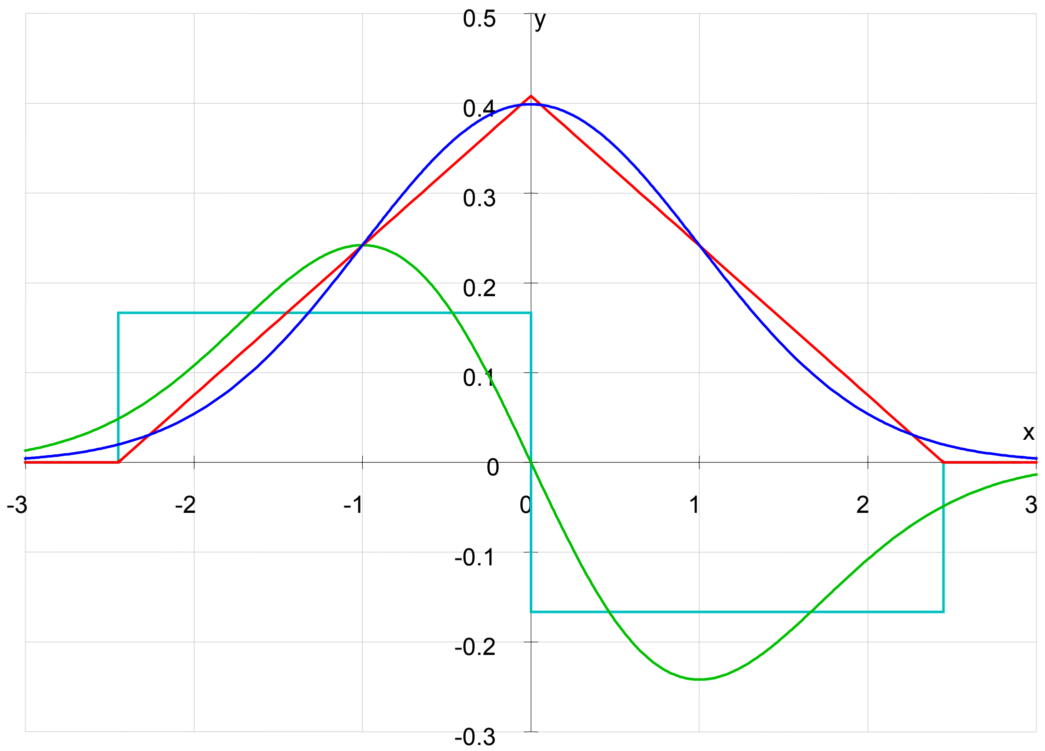
The filter function of the degree *n* is approximated by *n*+1 adjacent polynomial functions of the degree *n*. All derivatives at adjacent points of two polynomials are equal (similar to splines) except of the last derivative which consists of step functions. Height and width of the step functions define the filter function.

The benefit of this approximation is that the *n*-th derivation consists only of constant values, the filter constants. Integration is simply done by addition of new values on adjacent points of two polynomials for every derivative. This makes the calculation time independent from the filter radius.

1.5. Linear approximation

The Gaussian bell shape is adapted by a triangular function.

The second approximation degree of the Gaussian function is $\mathrm{b}_2(x) = a\left(|x+c| - 2|x| + |x-c|\right)$.



Picture 8: Linear approximation

a. $y = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

b. $y = \mathrm{b}_2(x)$

c. $y = -\frac{x}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

d. $y = \mathrm{s}_2(x)$

The first conditions for the constants *a* and *c* is the same than before:

$$\int_{-\infty}^{\infty} \mathrm{b}_2(x)\,\mathrm{d}x = 4a\int_0^c (c-x)\,\mathrm{d}x = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\,\mathrm{d}x = 2ac^2 = 1\,.$$

The second condition is again the same deviation value:

$$\sigma^2 = \int x^2\,\mathrm{b}_2(x)\,\mathrm{d}x = \int_{-c}^{c} \frac{x^2}{2c^2}\left(|x+c| - 2|x| + |x-c|\right)\mathrm{d}x = \frac{c^2}{6} = 1\,.$$ So $a = 1/12$ and $c = \sqrt{6}$.

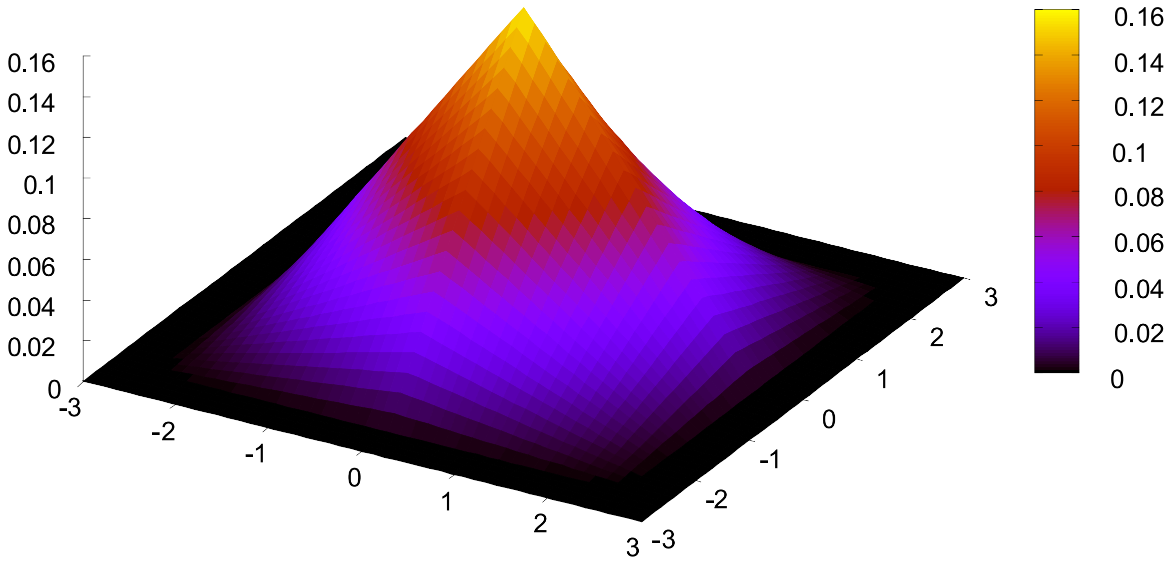
The second approximation degree of the Gaussian function is therefore:

$$b_2(x) = \frac{\left|x+\sqrt{6}\right| - 2|x| + \left|x-\sqrt{6}\right|}{12} \tag{6}$$

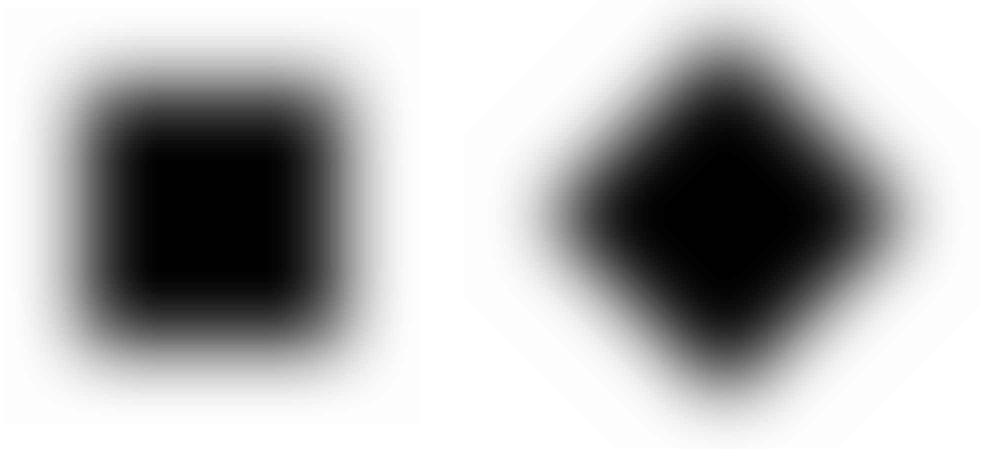
The step function of the second approximation is the derivation of $s_2(x) = \frac{\mathrm{d}\,b_2(x)}{\mathrm{d}\,x}$:

$$s_2(x) = \frac{\Theta\left(x+\sqrt{6}\right) - 2\Theta(x) + \Theta\left(x-\sqrt{6}\right)}{6} \tag{7}$$

The error of the 2nd degree of approximation is $e_2 = \sqrt{\int_{-\infty}^{\infty} (g(x) - b_2(x))^2 dx} = 0.04652434331$. The two dimensional error equals 0.03530803161.



Picture 9: Triangle blur

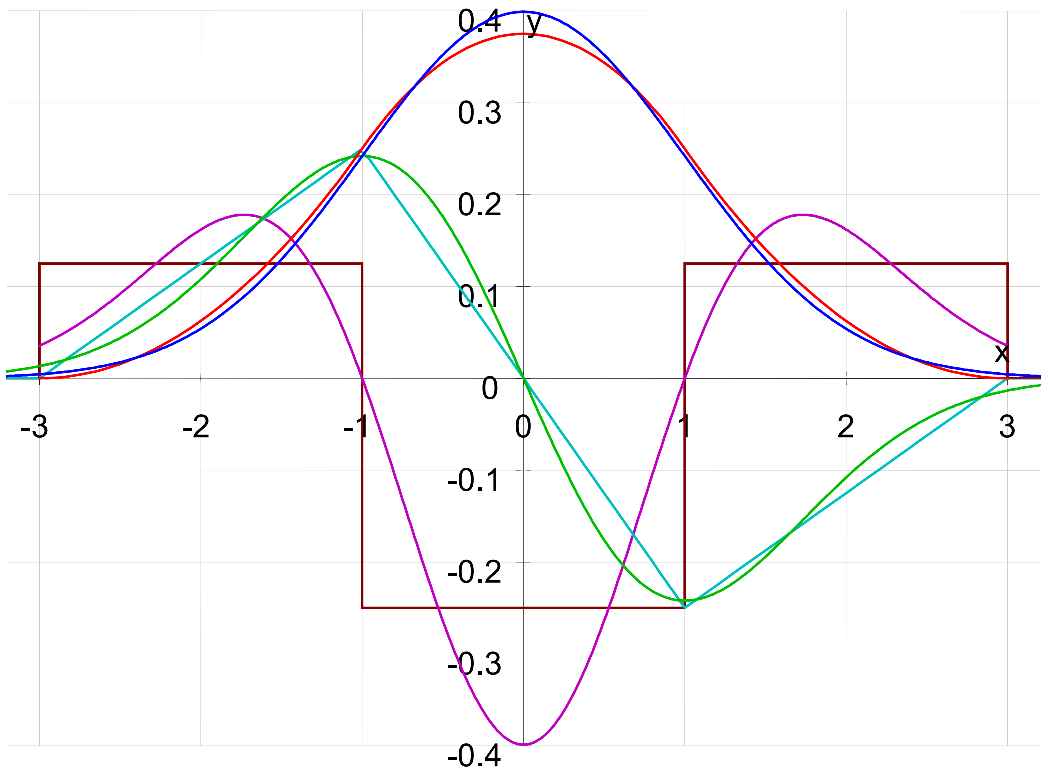


Picture 10: Triangle blur image

The image of picture 10 suggests a good approximation to the Gaussian blur confirmed by the calculation. Applying the approximation to the test image shows no noticeable difference to the Gaussian blur.
A maximum error of a few percent is visible in a picture editor with high pixel amplification. This is in the range of normal 8-bit image editing.

1.6. Square approximation

The process of approximation can be continued if the second degree of is not accurate enough. The bell shaped Gaussian function is differentiated and approximated by triangles. The triangles are then integrated as done before.



Picture 11: Third degree approximation

a. $y=\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$

b. $y = \mathrm{b}_3(x)$

c. 1st derivative

d. $y = \mathrm{b}_3'(x)$

e. 2nd derivative

f. $y = \mathrm{s}_3(x)$

For the third degree approximation the second derivative of the Gaussian function in picture 11e (magenta graph) is simplified to step functions. For simple approximation the width of the three steps (brown graph) is equal. The high of the negative step is twice the high of the positive step because the first derivative of the approximation at point $x = 0$ must be zero (the area of the brown graph must be zero over all). This leads to the following step function:

$$\mathrm{s}_3(x) = a(\Theta(2x+3c) - \Theta(2x-3c)) - 3a(\Theta(2x+c) - \Theta(2x-c))$$

c is the width of the steps and a the amplitude. Two times antiderivative of $\mathrm{s}_3(x)$ makes

$$\mathrm{b}_3(x) = \frac{a}{16}\left((2x+3c)|2x+3c| - (2x-3c)|2x-3c| - 3(2x+c)|2x+c| + 3(2x-c)|2x-c|\right)$$

The conditions for the constants are the same as for the previous approximations. The area beneath the function must be the same for the approximation and the Gaussian function:

$$\int_{-\infty}^{\infty}\mathrm{b}_3(x)\,\mathrm{d}x = \int_{-3c/2}^{3c/2}\mathrm{b}_3(x)\,\mathrm{d}x = \int_{-\infty}^{\infty}\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}\,\mathrm{d}x = 1.$$

So first condition leads to $ac^3 = 1$. For the second condition the standard deviation should be one:

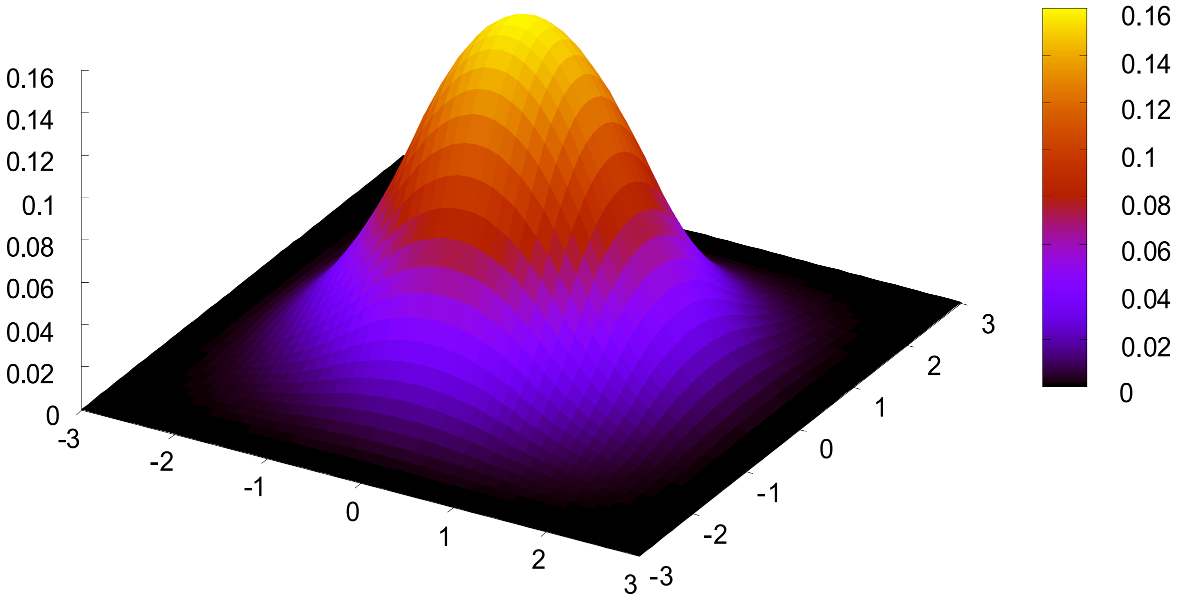
$\sigma^2 = \int x^2\,\mathrm{b}_3(x)\,\mathrm{d}x = \frac{c^4}{16} = 1$, so $a = 1/8$ and $c = 2$.

The step function of the third approximation is therefore

$$\mathrm{s}_3(x) = \frac{1}{8}(\Theta(x+3) - \Theta(x-3)) - \frac{3}{8}(\Theta(x+1) - \Theta(x-1)) \qquad (8)$$

and $\mathrm{b}_3(x) = \frac{1}{32}\left((x+3)|x+3| - (x-3)|x-3| - 3(x+1)|x+1| + 3(x-1)|x-1|\right)$.

The error of the 3rd degree of approximation is $e_3 = \sqrt{\int_{-\infty}^{\infty} (\mathrm{g}(x) - \mathrm{b}_3(x))^2 \, dx} = 0.02531428515$. The two dimensional error equals 0.01954370265.



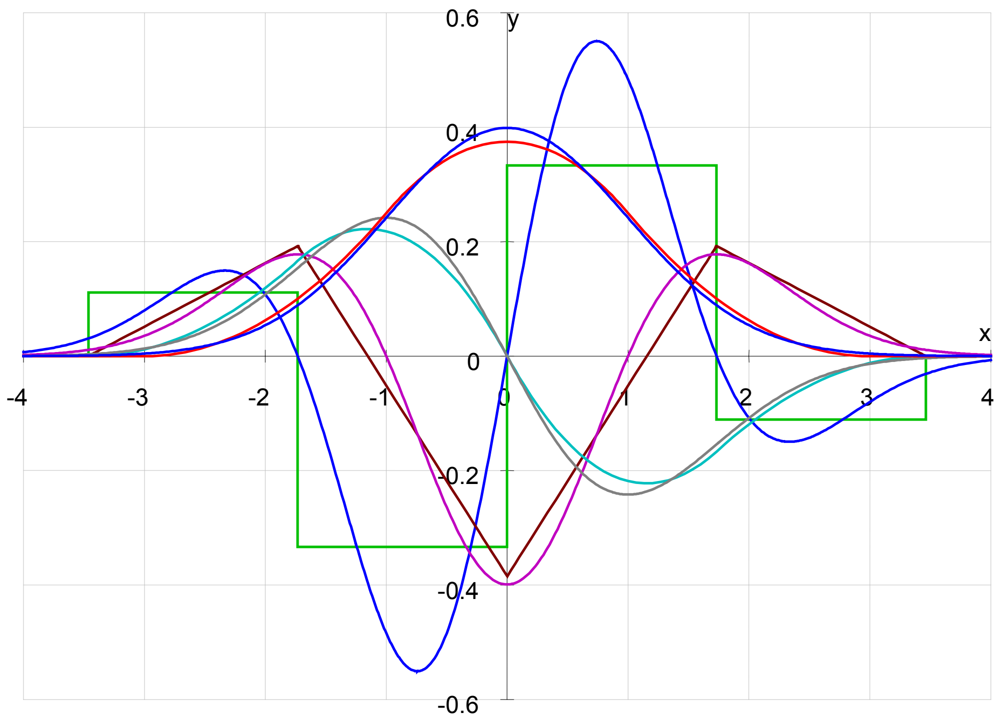
Picture 12: Square approximation

1.7. Higher approximation degree

The same approach as before can be used to calculate the fourth degree of approximation. The formula for the 4th degree step function is derived under the same condition than before: equal step width.

$$\mathrm{s}_4(x) = a(\Theta(x+2\mathrm{c}) + \Theta(x-2\mathrm{c}) - 4\Theta(x+\mathrm{c}) - 4\Theta(x-\mathrm{c}) + 6\Theta(x))$$

$$\mathrm{b}_4(x) = \frac{a}{12}\left((x+2\mathrm{c})^2|x+2\mathrm{c}| + (x-2\mathrm{c})^2|x-2\mathrm{c}| - 4(x+\mathrm{c})^2|x+\mathrm{c}| - 4(x-\mathrm{c})^2|x-\mathrm{c}| + 6x^2|x|\right)$$



Picture 13: Fourth degree approximation functions

a. $y = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

b. $y = \mathrm{b}_4(x)$

c. 1st degree derivative

d. 1st degree approx.

e. 2nd degree derivative

f. 2nd degree approx.

g. 3rd degree derivative

h. step function $\mathrm{s}_4(x)$

The constants are derived from the same conditions than previously:

$$\int_{-\infty}^{\infty} \mathrm{b}_4(x)\,\mathrm{d}x = \int_{-2c}^{2c} \mathrm{b}_4(x)\,\mathrm{d}x = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\,\mathrm{d}x = ac^4 = 1 \text{ and } \sigma^2 = \int x^2\, \mathrm{b}_4(x)\,\mathrm{d}x = \frac{c^4}{9} = 1,$$

which makes $a = 1/9$ and $c = \sqrt{3}$.

The step function of the 4th approximation degree is

$$\mathrm{s}_4(x) = \frac{1}{9}\left(\Theta\left(x+2\sqrt{3}\right) + \Theta\left(x-2\sqrt{3}\right) - 4\Theta\left(x+\sqrt{3}\right) - 4\Theta\left(x-\sqrt{3}\right) + 6\Theta(x)\right) \text{ and}$$

$$\mathrm{b}_4(x) = \frac{1}{108}\left(\left|x+2\sqrt{3}\right|^3 + \left|x-2\sqrt{3}\right|^3 - 4\left|x+\sqrt{3}\right|^3 - 4\left|x-\sqrt{3}\right|^3 + 6|x|^3\right).$$

The error is $e_4 = \sqrt{\int_{-\infty}^{\infty} (\mathrm{g}(x) - \mathrm{b}_4(x))^2\, dx} = 0.01812064909$, the two dimensional error equals 0.01405961129.

It is possible to further increase the approximation degree. The procedure is always the same and it should be possible to define an approach for any degree.

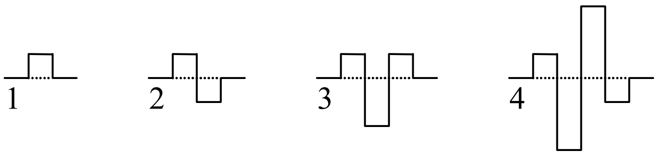
1.8. Arbitrary approximation degree

The previous calculation help to define an arbitrary degree function. To approximate the Gaussian bell shape to the *n*-th degree the function has *n* steps of equal width. Each step is $\binom{n-1}{i}$ high, but with alternating sign and the width is $1/c$. Thus the steps for $n = 1$ are {+1}, for $n = 2$ are {+1, –1}, for $n = 3$ are {+1, –2, +1}, for $n = 4$ are {+1, –3, +3, –1}, and so on. The formula for such a binomial step function is:

$$\mathrm{s}_n(x) = a\sum_{i=0}^{n-1}(-1)^i\binom{n-1}{i}\left(\Theta\left(cx+\frac{n}{2}-i\right) - \Theta\left(cx+\frac{n}{2}-1-i\right)\right) = a\sum_{i=0}^{n}(-1)^i\binom{n}{i}\Theta\left(cx+\frac{n}{2}-i\right) \tag{9}$$

where Θ denotes the Heaviside step function.

The functions of $\mathrm{s}_n(x)$ look like:



When $\mathrm{s}_n(x)$ is integrated n–1 times it gives the *n*-th degree of the Gaussian approximation function $\mathrm{b}_n(x)$.

The constants *a* and *c* have to be defined so that the area of the approximated bell function and the Gaussian function are the same.

This conditions is: $\int_{-\infty}^{\infty} \mathrm{b}_n(x)\,\mathrm{d}x = 2\int_{0}^{n/2c} \mathrm{b}_n(x)\,\mathrm{d}x = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\,\mathrm{d}x = 1.$

The *m*-th antiderivation of $\mathrm{s}_n(x)$ is ($m \ge 0$): $\mathrm{b}_{m,n}(x) = \frac{a}{2m!c^m}\sum_{i=0}^{n}(-1)^i\binom{n}{i}\frac{(cx+n/2-i)^{m+1}}{|cx+n/2-i|}$.

The integration constant is always zero to get the approximation of the Gaussian function for

$$|x| > \frac{n}{2c} \to \mathrm{b}_{m,n}(x) = 0.$$

For the first condition of *a* and *c* function values of $|x| > \frac{n}{2c}$ are zero, so the integral is

$$2\int_0^{n/2c} \mathrm{b}_{n-1,n}(x)\,\mathrm{d}x = 2\,\mathrm{b}_{n,n}\left(\frac{n}{2c}\right) = \frac{a}{c^n} = 1, \text{ because } \sum_{i=0}^{n}(-1)^i\binom{n}{i}(n-i)^{n-1}|n-i| = \sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}i^n = n!$$

The last rightmost equality is described in [2] chapter 1.2.6 page 64 Eq. (34):

$$n! = \sum_{k=0}^{n} k!\binom{0}{k-n} = \sum_{k=0}^{n} k!\sum_{i=0}^{n}\binom{i}{k}\binom{n}{i}(-1)^{n-i} = \sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}\sum_{k=0}^{n}k!\binom{i}{k} = \sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}\sum_{k=0}^{n} i^k = \sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i} i^n \tag{10}$$

with the help of $\mathrm{P}_s(n,k) = \sum_i \binom{s+i}{k}\binom{n}{i}(-1)^{n-i} = \binom{s}{k-n}$. In this case of $s = 0$, $\mathrm{P}_0(n,k)$ equals unity if $n = k$ and zero otherwise. The two rightmost equalities of equation (10) are only true because this cancels out all terms except the term $n = k$.

$\mathrm{P}_s(n,k)$ is proven by induction[1]: for $n = 0$ the term $\mathrm{P}_s(0,k)$ is $\sum_i \binom{s+i}{k}\binom{0}{i}(-1)^{-i} = \binom{s}{k}$ and if $\mathrm{P}_s(n,k)$ is true,

$$\mathrm{P}_s(n+1,k) = \sum_i \binom{s+i}{k}\binom{n+1}{i}(-1)^{n+1-i} = \sum_i \binom{s+i}{k}\left(\binom{n}{i-1}+\binom{n}{i}\right)(-1)^{n+1-i} =$$
$$= \sum_i \binom{s+i}{k}\binom{n}{i-1}(-1)^{n-(i-1)} - \sum_i \binom{s+i}{k}\binom{n}{i}(-1)^{n-i} = \binom{s+1}{k-n} - \binom{s}{k-n} = \binom{s}{k-n-1}.$$

The approximation now got the form: $\mathrm{b}_{m,n}(x) = \frac{c^{n-m}}{2m!}\sum_{i=0}^{n}(-1)^i\binom{n}{i}\frac{(cx+n/2+i)^{m+1}}{|cx+n/2+i|}$.

$\mathrm{b}_{m,n}(x)$ has to be integrated $m = n - 1$ times to get: $\mathrm{b}_n(x) = \frac{c}{2(n-1)!}\sum_{i=0}^{n}(-1)^i\binom{n}{i}\frac{(cx+n+2i)^n}{|cx+n+2i|}$.

For the second condition again the standard deviation of the approximation should be one. This could be solved with integration by parts:

$$\sigma^2 = 2\int_0^{n/2c} x^2\,\mathrm{b}(x)\,\mathrm{d}x = 2\left(\frac{n}{2c}\right)^2 \mathrm{b}_{n,n}\left(\frac{n}{2c}\right) - 4\frac{n}{2c}\mathrm{b}_{n+1,n}\left(\frac{n}{2c}\right) + 4\,\mathrm{b}_{n+2,n}\left(\frac{n}{2c}\right) = 2\left(\frac{n}{2c}\right)^2\frac{1}{2} - \frac{2n}{c}\frac{n}{4c} + 4\frac{n(3n+1)}{48c^2} = 1$$

since $\sum_{i=0}^{n}(-1)^i\binom{n}{i}(n-i)^{n+1} = \sum_{i=0}^{n}(-1)^{n-i}\binom{n}{i}i^{n+1} = \frac{n(n+1)!}{2}$ and $\sum_{i=0}^{n}(-1)^i\binom{n}{i}(n-i)^{n+2} = \frac{n(3n+1)(n+2)!}{24}$.

The constant c is therefore

$$c = \sqrt{n/12}. \tag{11}$$

The n-th degree approximation of the Gaussian function of picture 14 for $n > 0$ makes:

$$\mathrm{b}_n(x) = \frac{\sqrt{3n}}{12(n-1)!}\sum_{i=0}^{n}(-1)^i\binom{n}{i}\frac{\left(x\sqrt{n/12}+n/2-i\right)^n}{\left|x\sqrt{n/12}+n/2-i\right|} \tag{12}$$

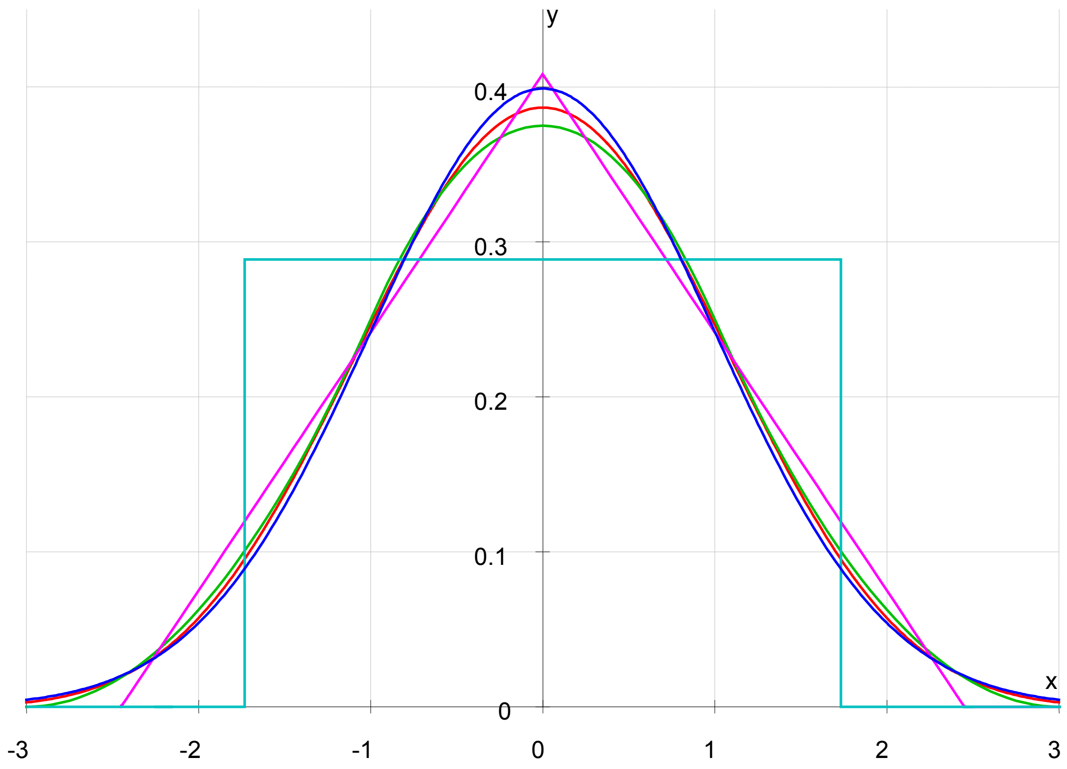
Chapter 1.10 gets to the same equation by a different approach.

The step function for the n-th degree approximation makes:

$$\mathrm{s}_n(x) = \left(\sqrt{\frac{n}{12}}\right)^n \sum_{i=0}^{n}(-1)^i\binom{n}{i}\Theta\left(x\sqrt{\frac{n}{12}}+\frac{n}{2}-i\right) \tag{13}$$

[1] This is exercise 19 of chapter 1.2.6 in [2].

Equations (12) and (13) are also confirmed for $n \le 4$ by the calculations of the previous chapters. The functions equal zero for values outside of $|x| > 3c = \sqrt{3n}$. Picture 14 shows the graphs of equation (12) for different approximation degrees in comparison to the Gaussian function.



Picture 14: nth degree approximation

a. Gaussian blur

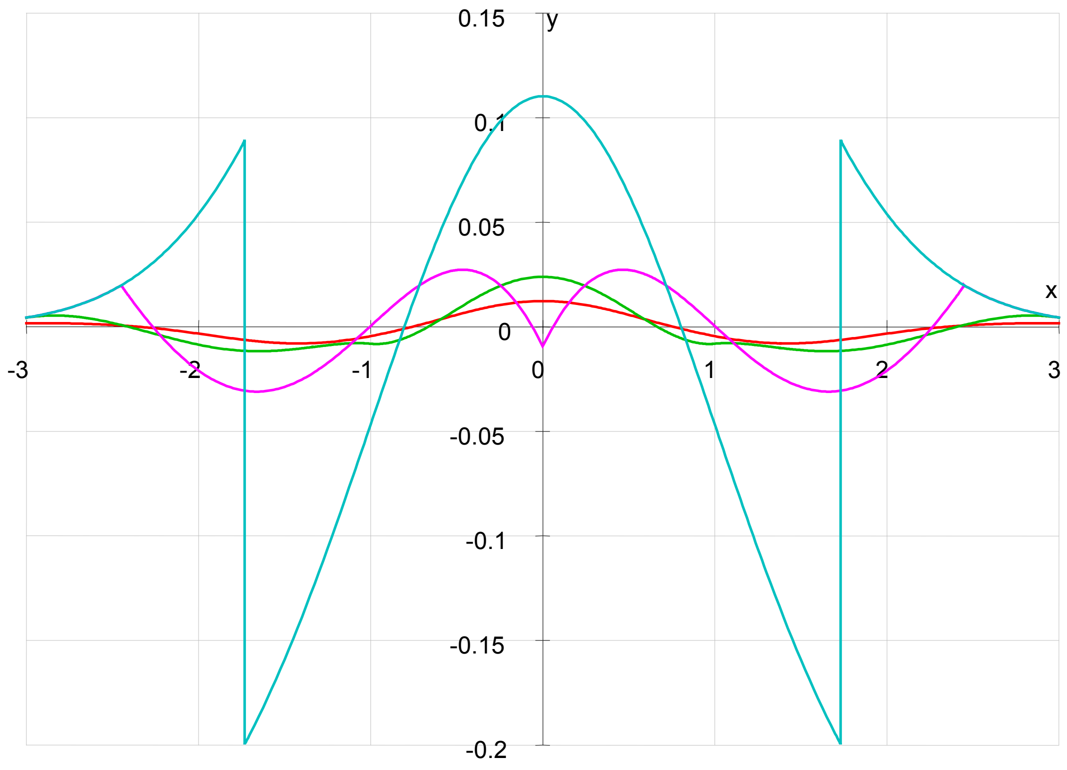
b. 1st degree, box blur

c. 2nd degree, linear blur

d. 3rd degree, square blur

e. 5th degree, quadruple blur

1.9. Accuracy



Picture 15: approximation error

a. 1st degree, box blur

b. 2nd degree, triangle

c. 3rd degree, square blur

d. 5th degree, quadruple blur

The accuracy of the one dimensional approximation of degree n makes

$$e=\sqrt{\int\left(\mathrm{g}(x)-\mathrm{b}(x)\right)^2dx}=\sqrt{2\int_0^{\sqrt{3n}}\left(\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}-\mathrm{b}_n(x)\right)^2\mathrm{d}x+\frac{1-\mathrm{erf}\sqrt{3n}}{2\sqrt{\pi}}}$$ which is about: $e_n\approx\frac{1}{15n-8}$.

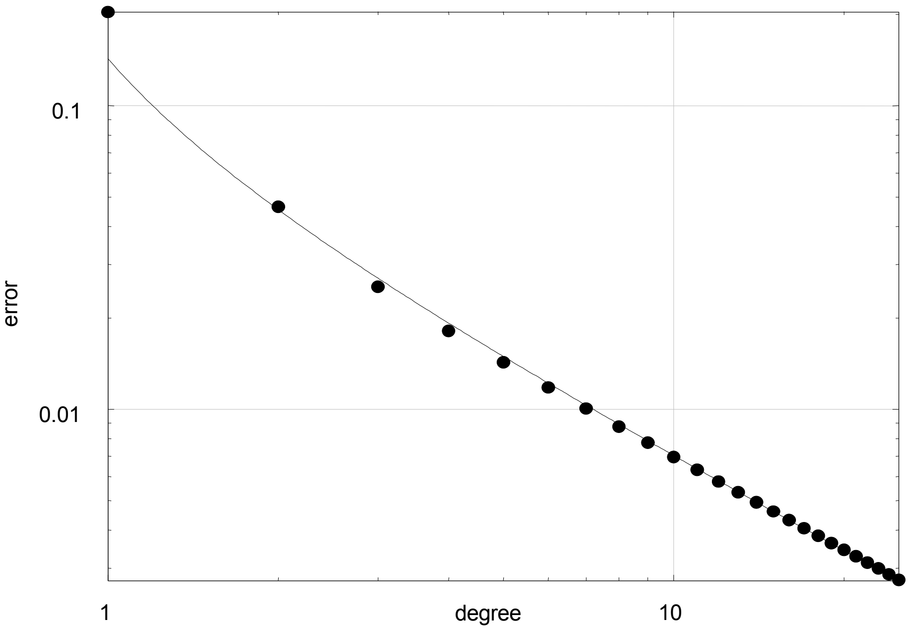
The approximation error of the entire function does not count so much. The error in the frequency spectrum of the next chapter is more important and has a lower and exponential dependency on *n*.

The two dimensional error is

$$e=\sqrt{\iint\left(\mathrm{g}(x)\mathrm{g}(y)-\mathrm{b}(x)\mathrm{b}(y)\right)^2dxdy}=\sqrt{4\int_0^{\sqrt{3n}}\int_0^{\sqrt{3n}}\left(\frac{1}{2\pi}e^{-\frac{x^2+y^2}{2}}-\mathrm{b}_n(x)\mathrm{b}_n(y)\right)^2\mathrm{d}x\,\mathrm{d}y+\frac{1-\mathrm{erf}^2\sqrt{3n}}{4\pi}},$$

where erf(x) denotes the error function: $\mathrm{erf}(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\mathrm{d}t$.

The chosen approximation is not ideal since it converges slowly to the Gaussian function, there are better solutions. But this one has the advantage of easy computation for discrete data.



Picture 16: one dimensional approximation accuracy

1.10. Frequency spectrum

The space domain of the blur function can be transformed into the frequency domain with the help of the Fourier transform. This transform makes it possible to compare the Gaussian blur to the approximations in respect of the affected frequency spectrum.

Fourier's Integral Theorem of the function f(x) is defined as:

$$\mathrm{f}(s)=\frac{1}{\pi}\int_0^\infty\mathrm{d}\,\omega\int_{-\infty}^\infty\mathrm{f}(x)\cos\omega(x-s)\,\mathrm{d}\,x=\int_0^\infty\left(\mathrm{a}(\omega)\cos\omega s+\mathrm{c}(\omega)\sin\omega s\right)\mathrm{d}\,\omega \tag{14}$$

with the spectrum functions: $\mathrm{a}(\omega)=\frac{1}{\pi}\int_{-\infty}^\infty\mathrm{f}(x)\cos\omega x\,\mathrm{d}\,x$ and $\mathrm{c}(\omega)=\frac{1}{\pi}\int_{-\infty}^\infty\mathrm{f}(x)\sin\omega x\,\mathrm{d}\,x$. (15)

The complex part c(ω) of the spectrum equals zero for even functions of f(x) which are used here and the real part a(ω) equals zero for odd.

The frequency spectrum of the Gaussian blur of equation (2) is:

$$\pi\,\mathrm{a}_g(\omega)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^\infty e^{-\frac{x^2}{2}}\cos\omega x\,\mathrm{d}\,x=e^{-\frac{\omega^2}{2}}. \tag{16}$$

The Fourier transform of $\mathrm{b}_n(x)$ seems to be a bit more difficult to calculate. But according to the convolution theorem the Fourier transform in space domain is the point wise product in the frequency domain.
The Fourier transform of the rectangular (box) function is the sinus cardinalis:

$$\int_{-\infty}^{\infty}\mathrm{rect}(x)\cos\omega x\,\mathrm{d}x=\int_{-1}^{1}\sum_{i=0}^{1}\frac{(-1)^i}{2}\frac{2x+1-2i}{|2x+1-2i|}\cos\omega x\,\mathrm{d}x=\frac{\sin\omega/2}{\omega/2}.$$

The convolution of two identical box functions is the triangular (linear) function:

$$\mathrm{tri}(x)=\mathrm{rect}(x)*\mathrm{rect}(x)=\int_{-\infty}^{\infty}\mathrm{rect}(s)\,\mathrm{rect}(s-x)\,\mathrm{d}\,s=\sum_{i=0}^{2}\frac{(-1)^i}{2}\binom{2}{i}\frac{(x+1-i)^2}{|x+1-i|}.$$

The Fourier transform according to the theorem is therefore the square of the sinus cardinalis:

$$\int_{-\infty}^{\infty}\mathrm{tri}(x)\cos\omega x\,\mathrm{d}x=\int_{-2}^{2}\sum_{i=0}^{2}\frac{(-1)^i}{2}\binom{2}{i}\frac{(x+1-i)^2}{|x+1-i|}\cos\omega x\,\mathrm{d}\,x=\left(\frac{\sin\omega/2}{\omega/2}\right)^2.$$

And another convolution by the rectangular function makes:

$$\int_{-\infty}^{\infty}(\mathrm{tri}(x)*\mathrm{rect}(x))\cos\omega x\,\mathrm{d}x=\int_{-3}^{3}\sum_{i=0}^{3}\frac{(-1)^i}{16}\binom{3}{i}\frac{(2x+3-2i)^3}{|2x+3-2i|}\cos\omega x\,\mathrm{d}\,x=\left(\frac{\sin\omega/2}{\omega/2}\right)^3.$$

By further multiplication of both domains the equation of the space domain becomes a well known pattern.

When the operation x^{*n} defines the n-fold iteration of the convolution with itself (convolution power) then the n-th power of the sinus cardinalis is the Fourier transform of the n-th convolution power of the rectangular function:

$$\int_{-\infty}^{\infty}\mathrm{rect}^{*n}(x)\cos\omega x\,\mathrm{d}x=\int_{0}^{n}\sum_{i=0}^{n}\frac{(-1)^i}{(n-1)!}\binom{n}{i}\frac{(x+n/2-i)^n}{|x+n/2-i|}\cos\omega x\,\mathrm{d}x=\left(\frac{\sin\omega/2}{\omega/2}\right)^n. \quad (17)$$

Equation (17) even holds for $n = 0$ since $\int_{-\infty}^{\infty}\mathrm{rect}^{*0}(x)\cos\omega x\,\mathrm{d}x=\int_{-\infty}^{\infty}\delta(x)\cos\omega x\,\mathrm{d}x=1$.

By the inverse Fourier transform we get:

$$\mathrm{rect}^{*n}(x)=\frac{1}{\pi}\int_0^{\infty}\left(\frac{\sin\omega/2}{\omega/2}\right)^n\cos\omega x\,\mathrm{d}\omega \quad (18)$$

and for $n = 0$ another definition of the Dirac delta function as: $\delta(x)=\frac{1}{\pi}\int_0^{\infty}\cos\omega x\,\mathrm{d}\omega=\int_0^{\infty}\cos\pi s x\,\mathrm{d}s$.

A similar relation exists for the step function due to the complex part of the Fourier transform:

$$\int_{-\infty}^{\infty}\Theta(x)\sin\omega x\,\mathrm{d}x=\int_0^{\infty}\sin\omega x\,\mathrm{d}x=\frac{1}{\omega},\text{ and of the inverse transform: }\Theta(x)=\frac{1}{2}+\frac{1}{\pi}\int_0^{\infty}\frac{\sin\omega x}{\omega}\mathrm{d}\omega. \quad (19)$$

In some way the previous pages just derived the powered convolution of the rectangular function:
$\mathrm{b}_n(x)=\sqrt{n/12}\ \mathrm{rect}^{*n}(x\sqrt{n/12})$.
The frequency spectrum $\pi\,\mathrm{a}_n(\omega)$ of the approximation $\mathrm{b}_n(x)$ of equation (12) is therefore simply the power of the sinus cardinalis:

$$\underline{\pi\,\mathrm{a}_n(\omega)=\int_{-\sqrt{3n}}^{\sqrt{3n}}\mathrm{b}_n(x)\cos\omega x\,\mathrm{d}\,x=\left(\frac{\sin\omega\sqrt{3/n}}{\omega\sqrt{3/n}}\right)^n} \quad (20)$$

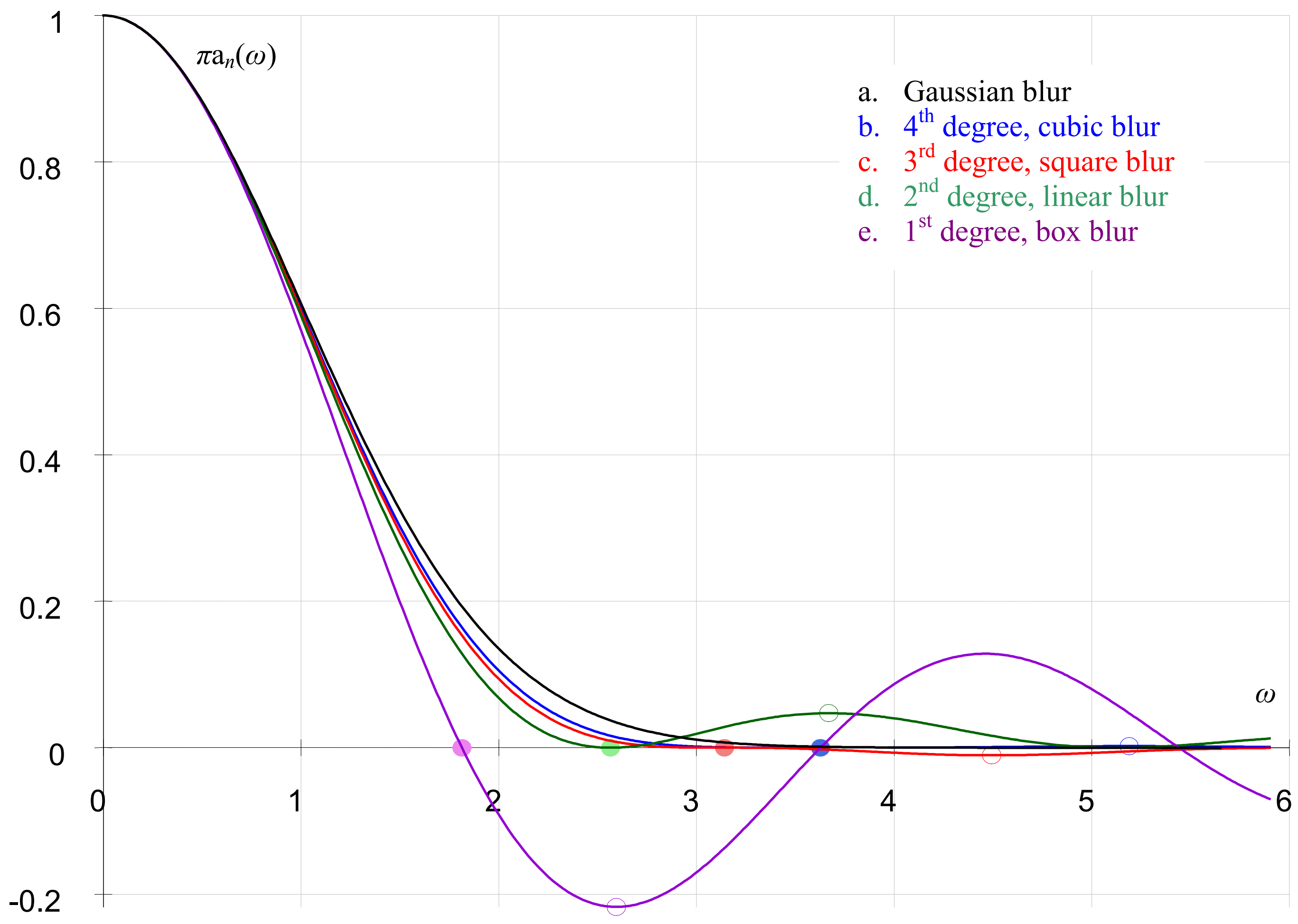
Equations (16) and (20) also suggest the remarkable limit of:

$$\lim_{n\to\infty}\left(\frac{n}{x}\sin\frac{x}{n}\right)^{n^2}=e^{-\frac{x^2}{6}}, \quad (21)$$

since $\lim_{n\to\infty}n^2\ln\left(\frac{n}{x}\sin\frac{x}{n}\right)=\lim_{m\to0}\frac{1}{m^2}\ln\frac{\sin mx}{mx}=\lim_{m\to0}\frac{mx\cos mx-\sin mx}{2m^2\sin mx}=\lim_{m\to0}\frac{x^2\cos mx}{2mx\sin mx-6\cos mx}=-\frac{x^2}{6}$.

The following table contains the frequency spectrum of the first approximation degrees:

Spectrum of degree *n*	Space domain $b_n(x)$	Frequency domain $\pi\, a_n(\omega)$
0th degree (identity):	$\delta(x)$	1
1st degree (box blur):	$\frac{\Theta(x+\sqrt{3})-\Theta(x-\sqrt{3})}{2\sqrt{3}}$	$\frac{\sin \omega\sqrt{3}}{\omega\sqrt{3}}$
2nd degree (linear):	$\frac{\lvert x+\sqrt{6}\rvert-2x+\lvert x-\sqrt{6}\rvert}{12}$	$\frac{1-\cos \omega\sqrt{6}}{3\omega^2}$
3rd degree (square):	$\frac{(x+3)\lvert x+3\rvert-(x-3)\lvert x-3\rvert-3(x+1)\lvert x+1\rvert+3(x-1)\lvert x-1\rvert}{32}$	$\frac{3\sin\omega-\sin 3\omega}{4\omega^3}$
4th degree (cubic):	$\frac{\lvert x+2\sqrt{3}\rvert^3+\lvert x-2\sqrt{3}\rvert^3-4\lvert x+\sqrt{3}\rvert^3-4\lvert x-\sqrt{3}\rvert^3+6\lvert x\rvert^3}{108}$	$\frac{6+2\cos 2\omega\sqrt{3}-8\cos\omega\sqrt{3}}{9\omega^4}$



Picture 17: Frequency spectrum of the analog approximation

Higher frequencies in the spectrum of lower degree approximations are not suppressed as much as they are by the Gaussian convolution.

For any approximation degree *n* there is a minimum frequency ω_0 which is entirely suppressed: $a_n(\omega_0) = 0$. This frequency is calculated by $\omega_0 = \pi\sqrt{n/3}$ and is indicated by a point in picture 17.

The first maxima in the frequency spectra of the approximations is the greatest error of this degree for $\omega > \omega_0$ and is indicated by circle. The frequency ω_p of this peak could be found by setting the derivation of the function equal to zero: $a'_n(\omega_p) = 0$.

The equation to solve this derivation will be: $\tan \omega_p\sqrt{3/n} = \omega_p\sqrt{3/n}$ for the range of $\pi < \omega_p\sqrt{3/n} < 2\pi$ and has no symbolic solution. The numerical calculation of degree n makes: $\omega_p = 2.594271160\sqrt{n}$.

The value of the peak error makes: $\pi\, a_n(\omega_p) = (-4.603338849)^{-n}$. (22)

2. Algorithm

A discrete image comes from an analog world. But for processing images by computers it is necessary to digitize it. This is obtained from an analogue image by sampling and quantization. The analog image is superimposed by a regular grid, consisting of picture elements (pixel), and each pixel is assigned an integer number, the value representing the luminance or brightness.

Such an image has two main characteristics:

- The space domain S, representing a generally rectangular grid of pixels usally in two dimensions of rows and columns. The possible values of the space domain define the resolution of the image. The space of 2048x1536 pixel is for example a 3 mega pixel image.
- The range domain R, representing the pixel value p. The range domain could be expressed by a single value for gray images or by a triple of numbers for color images. Usual ranges are 3x8 or 3x16 bit.

Binomials

The central-limit theorem states that the binomial distribution can be approximated by the Gaussian distribution if n is large enough (de Moivre–Laplace theorem). Reversing this principle, a reasonable approximation to the Gaussian can be obtained by the binomial distribution function.

The approximation of the binomial distribution by the central-limit theorem for large n is:

$$\binom{n}{i} p^i (1-p)^{n-i} \cong \frac{1}{\sqrt{2\pi np(1-p)}} e^{-\frac{(i-np)^2}{2np(1-p)}}. \quad (23)$$

The Gaussian approximation of the probability of $p = 1/2$ for large n is: $\binom{n}{i}\frac{1}{2^n} \cong \sqrt{\frac{2}{\pi n}}\, e^{-\frac{2}{n}\left(i-\frac{n}{2}\right)^2}$.

In this way simple binomial integers can be used for fast calculation of the Gaussian function which would be much slower in direct computation.

2.1. Approximation degree

The polynomial functions are n-times derived till only constants remain. These constants together with the x-position of the polynomial pieces define the filter function. The derivates have to be integrated again to get the filter function. The numerical integration of a constant is very easy. What comes here in handy is that consecutive pixels are calculated. So the integration of the previous pixel is already calculated. Only the difference has to be added. This way the algorithm is very simple.

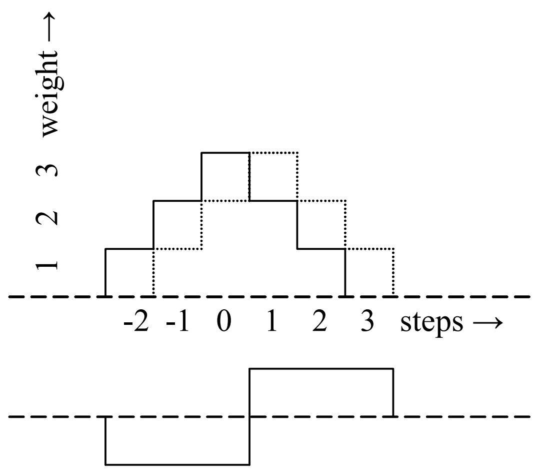
The Gaussian blur of a two dimensional image are applied as two independent one-dimensional calculations.

The degree of approximation to the Gaussian bell shape defines the accuracy. When the degree is one the approximation is a box blur. A degree of $n = 2$ is a linear (triangle) approximation. A degree of $n = 3$ is a square approximation and so on.

At the same time the degree defines the number of pixels to sum up per iteration.

Although the algorithm is only an approximation the result is accurate enough for most applications. For the maximum possible approximation degree the extended binomial filter is equal to the normal binomial filter (having then the same runtime of course).

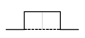
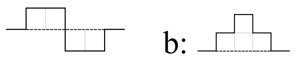
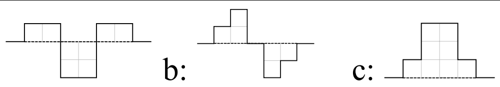
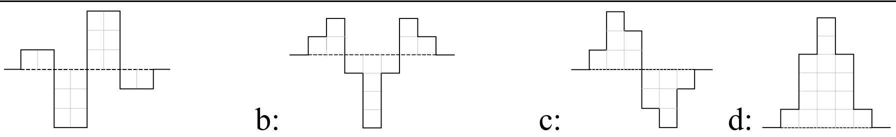
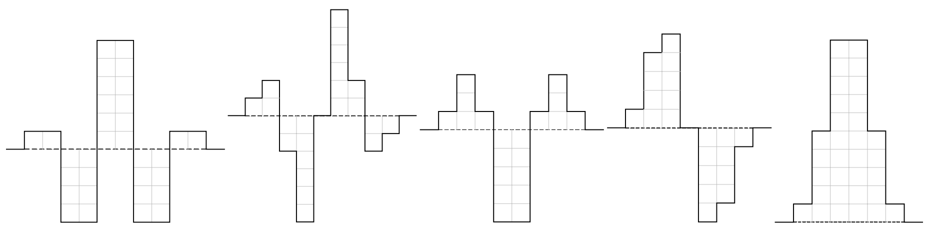
2.2. Calculation



Picture 18: sliding window

The blur is simplified to the form of a triangle. The higher the values the greater is the weight of the pixel. The blur is now done by a sliding window algorithm. Picture 18 shows the approximated triangle blur of the actual pixel as solid line and the wanted blur of the next pixel as doted line. The difference between the two functions is drawn below. It only changes at front of the blur (step 3) by adding the pixel value, at the peak (step 1) by subtracting twice of the value and at the end (step -2) adding again the pixel value.

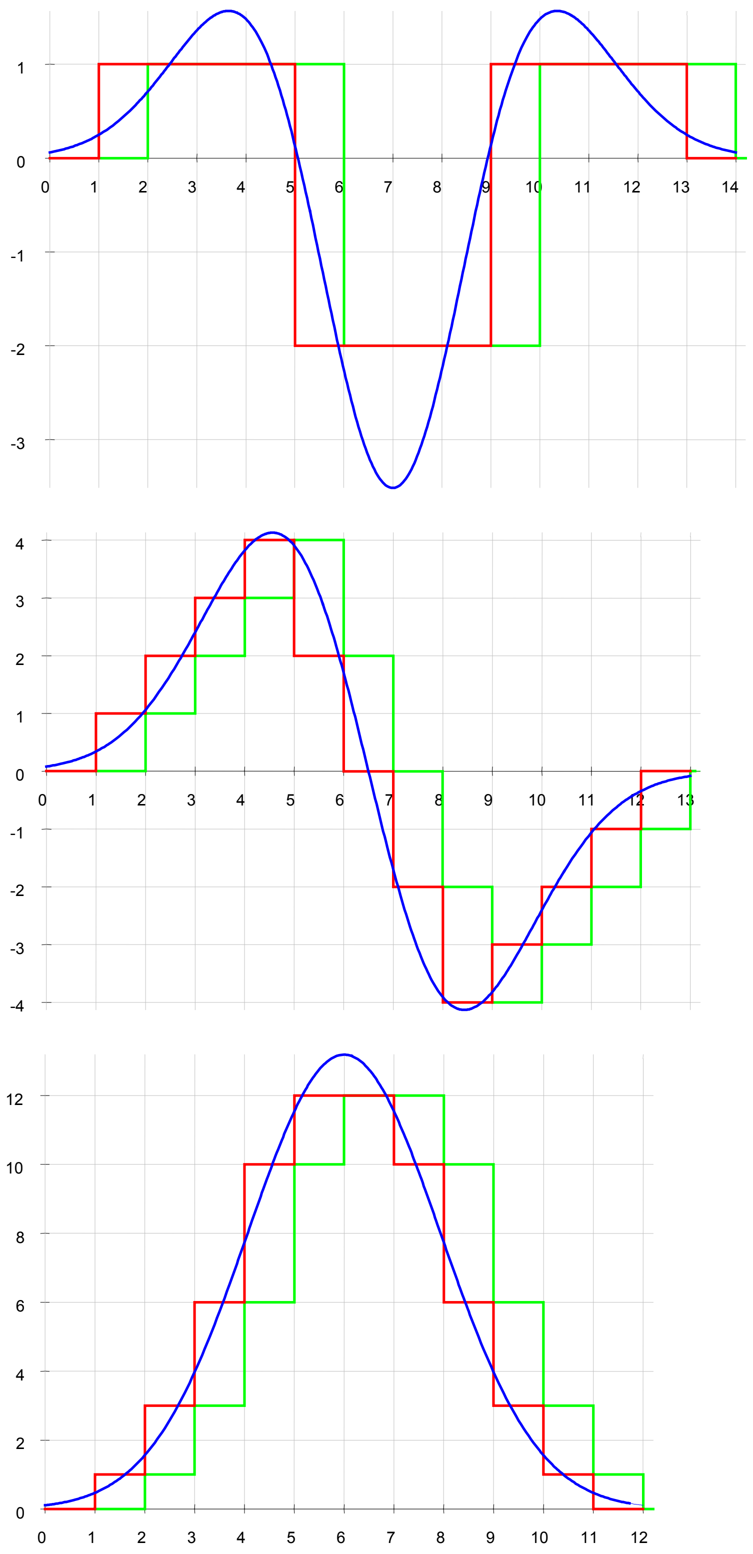
For most cases this approximation will be accurate enough, but the used algorithm can be further developed for higher accuracy. At the beginning we always start with step a function. The levels are the numbers of the Pascal's triangle with alternating sign. The difference of two functions, one shift a step, makes the previous function.

n	Approximation steps for blur radius 2	Weights $w_{n,2}(k)$
1	a: 	a: 1 1
2	a: b: 	a: 1 1 -1 -1 b: 1 2 1
3	a: b: c: 	a: 1 1 -2 -2 1 1 b: 1 2 0 -2 -1 c: 1 3 3 1
4	a: b: c: d: 	a: 1 1 -3 -3 3 3 -1 -1 b: 1 2 -1 -4 -1 2 1 c: 1 3 2 -2 -3 -1 d: 1 4 6 4 1
5	 a: b: c: d: e:	a: 1 1 -4 -4 6 6 -4 -4 1 1 b: 1 2 -2 -6 0 6 2 -2 -1 c: 1 3 1 -5 -5 1 3 1 d: 1 4 5 0 -5 -4 -1 e: 1 5 10 10 5 1

Picture 19: calculation weights

Picture 19 shows the calculation weights of different degrees *n*. The weight numbers of the right column are simple additions of the numbers directly above and to the left like the Pascal's triangle.

If the approximation degree is the blur radius the algorithm calculates exactly the integer representation of the Gaussian blur, the numbers of the Pascal’s triangle. Of course the effort to acquire it is the same as direct calculation. The advantage is that the degree doesn’t have to be as large as the blur radius which is especially useful for large blur radii.



Picture 20: calculation grid, degree 3, radius 4

a) 2nd derivative

red: current pixel

green: next pixel

blue: Gaussian derivative

b) 1st derivative

c) discrete approximation

Let’s assume the red function in picture 20c, which looks very like the Gaussian bell shape, is already calculated. Now the green function of the next pixel to the right should be calculated. The difference between the red and the green function of picture 20c is drawn as red function in picture 20b. Let’s further assume this red function is already calculated too and the next difference should be calculated, drawn as green function in picture 20b. Now the difference between the red and the green function is drawn as red function in picture 20a. Again this function is already calculated and for the difference to the green function of the next pixel only four values have to be added.

How does the algorithm work? Let's assume the three red functions of picture 20a, b and c are already set up by the initialization. The blur of the next pixel in picture 20c (green) should be calculated. The red function of picture 20b is added to the red function of picture 20c to get the green function of picture 20c. Now the red function of picture 20a is added to the red function of picture 20b to get the green function of picture 20b for the next pixel. To get the green function of picture 20a the value of pixel 1 is added, the value of pixel 5 is subtracted two times, the value of pixel 9 is added two times and the value of pixel 13 is subtracted. That's all! With 6 additions (plus one multiplication) and 4 pixel accesses the blur of the next pixel is calculated for a total blur width of 10.

If the 3rd degree of approximation is still not accurate enough higher degrees can be applied. They work the same as previously explained. This way it is possible get any needed accuracy. Of course higher accuracies are more complex but still have a run time independent of the blur radius.

2.3. Extended binomial sequence

Definitions:

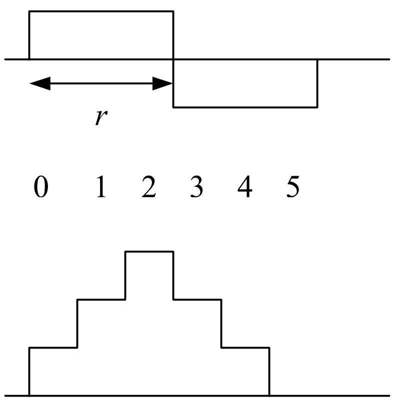
r	discrete blur radius (step width)
n	approximation degree
m	m-th accumulation of the step function
k	steps in horizontal direction

example:

$n = 2$, $r = 3$

$m = 0$

$m = 1$



For $m > 0$ the weight is the sum of the numbers directly above and to the left as the right column of picture 19 shows. This weight function table of the parameter m and k is:

	0	1	2	3	k
0	a_0	a_1	a_2	a_3	a_k
1	a_0	a_0+a_1	$a_0+a_1+a_2$	$a_0+a_1+a_2+a_3$	$\sum_{i=0}^{k} a_i$
2	a_0	$2a_0+a_1$	$3a_0+2a_1+a_2$	$4a_0+3a_1+2a_2+a_3$	$\sum_{i=0}^{k}(k-i+1)a_i$
3	a_0	$3a_0+a_1$	$6a_0+3a_1+a_2$	$10a_0+6a_1+3a_2+a_2$	$\sum_{i=0}^{k}\binom{k-i+2}{2}a_i$
m	a_0	ma_0+a_1	$\binom{m+1}{2}a_0+ma_1+a_2$	$\binom{m+2}{3}a_0+\binom{m+1}{2}a_1+ma_2+a_3$	$\sum_{i=0}^{k}\binom{k-i+m-1}{k-i}a_i$

This table represents the blur calculation of the pixel k in horizontal direction and the accumulation step m vertically. Every entry is the sum of the element above plus to the left.

The discrete weight of step *k* for the first row *m* = 0 makes $a_k = (-1)^{\lfloor k/r \rfloor}\binom{n-1}{\lfloor k/r \rfloor} = \binom{\lfloor k/r \rfloor - n}{\lfloor k/r \rfloor}$, where $\lfloor\ \rfloor$ denotes the floor function. The weights $w_{n,r}(k)$ of the extended binomial sequence of step *k* (for *m* = *n* − 1 accumulations) are therefore:

$$w_{n,r}(k) = \sum_{i=0}^{k} (-1)^{\lfloor i/r \rfloor} \binom{n-1}{\lfloor i/r \rfloor}\binom{n+k-i-2}{k-i} = \binom{n,\ r-1}{k} \qquad [r>0] \qquad (24)$$

The coefficients of the sequence $w_{n,r}(k)$ are polynomial coefficients of the powered univariate polynomial $(1+x+x^2+\ldots+x^r)^n = \sum_{k=0}^{nr}\binom{n,r}{k}x^k$, which could also be expressed as a recursion $\binom{n,r}{k} = \sum_{i=0}^{r}\binom{n-1,r}{k-i}$ or the sum of multinomial coefficients:

$$\binom{n,r}{k} = \sum_{\substack{0\le k_1 \le k_2 \ldots k_r \le n \\ k_1+2k_2+\cdots+rk_r=k}} \binom{n}{n-k_1-\cdots-k_r, k_1, \ldots, k_r} = \sum_{\substack{0\le k_1 \le k_2 \ldots k_r \le n \\ k_1+2k_2+\cdots+rk_r=k}} \frac{n!}{(n-k_1-\cdots-k_r)!k_1!\ldots k_r!}.$$

The parameter *n* and *r* select the approximation degree and the discrete radius.

For *r* = 1 or *n* = 0 the blur calculation of the extended binomial collapses to the identity function $w_{n,1}(k) = w_{0,r}(k) = \binom{n,0}{k} = \binom{0,r}{k} = \binom{0}{k}$ with no blur at all.

For *n* = 1 the weights make $w_{1,r}(k) = \binom{0}{\lfloor k/r \rfloor}$. For *r* = 2 the function is the binomial coefficient $w_{n,2}(k) = \binom{n}{k}$ and for *r* = 3 it is the a kind of trinomial coefficient $w_{n,3}(k) = \binom{n,2}{k} = \sum_{i=0}^{\lfloor k/2 \rfloor}\binom{n}{k-i}\binom{k-i}{i}$.

The size of the sequence (*k*-range) is $s = n(r-1)$. Outside this range of $0 \le k \le s$ the function equals zero.

The sum over all coefficients (total weight) is:

$$w_{n,r} = \sum_{k=0}^{s} w_{n,r}(k) = r^n \qquad (25)$$

This value also defines the needed range of the data type an algorithm must be able to handle in variables for computation. Otherwise an overflow occurs in the calculation. This value grows fast for higher approximation degrees.

The sequence of the coefficients is symmetric: $w_{n,r}(k) = w_{n,r}(s-k)$.

The variance for discrete functions makes:

$$\sigma^2 = \frac{1}{w_{n,r}}\sum_{k=0}^{s}(k-\bar{k})^2\, w_{n,r}(k) = \frac{1}{r^n}\sum_{k=0}^{s}k^2\binom{n,\ r-1}{k} - \bar{k}^2 = \frac{n(r^2-1)}{12},$$

where $\bar{k}$ denotes the mean value, half of the size: $\bar{k} = \frac{1}{w_{n,r}}\sum_{k=0}^{s} k\, w_{n,r}(k) = \frac{s}{2} = \frac{n(r-1)}{2}$.

The standard deviation is therefore:

$$\sigma = \sqrt{\frac{n(r^2-1)}{12}} \qquad (26)$$

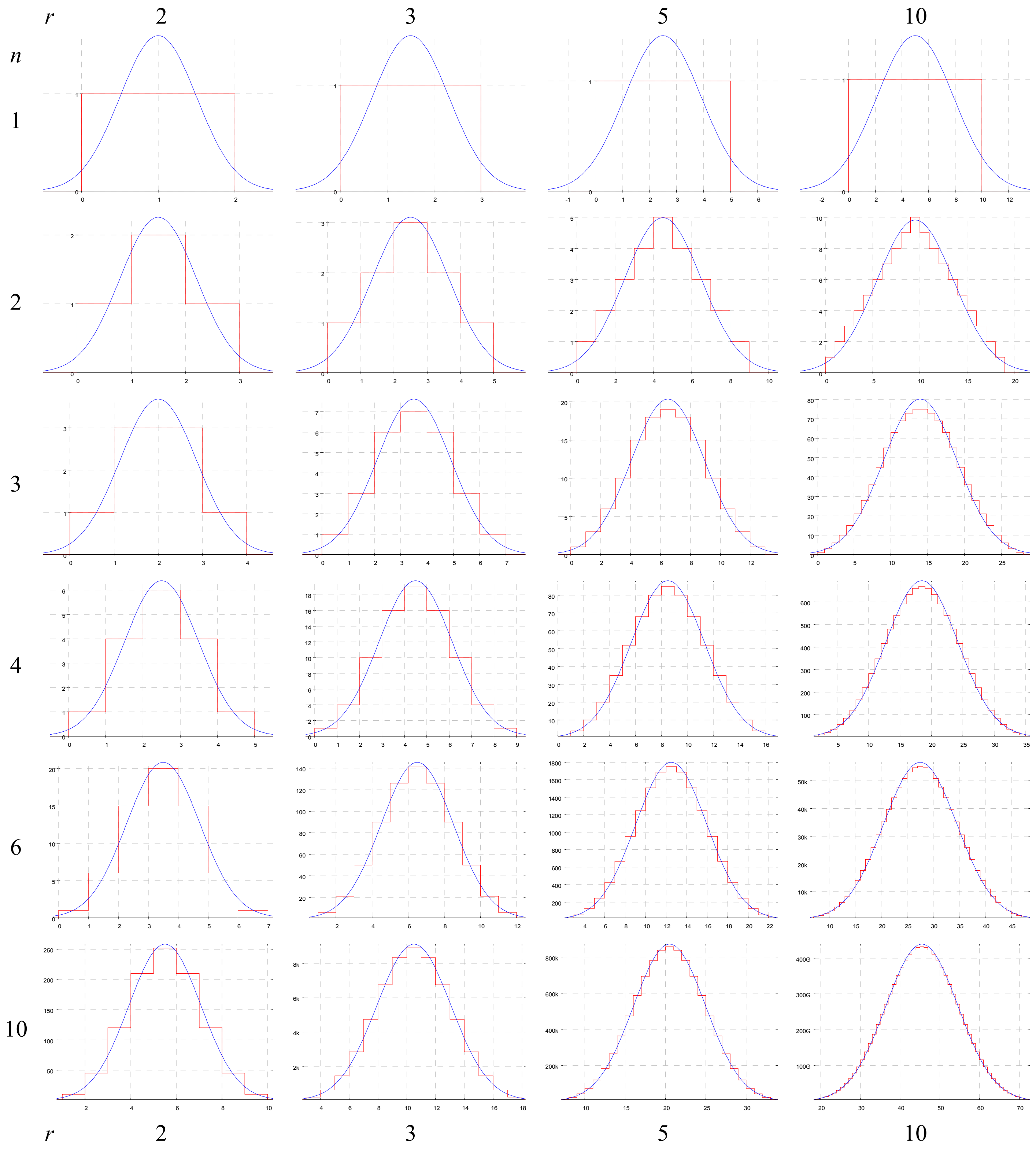
This is the correction factor between the discrete radius *r* and the blur radius σ for the approximation degree *n*. The factor $\sqrt{r^2-1}$ is like a quantization value since there is no blur applied for *r* = 1 (identity function).

For *r* = 2 the radius σ is equal the standard deviation of the binomials: $\sigma = \frac{\sqrt{n}}{2}$.

2.3.1. Extended binomial graphs

Picture 21 shows the discrete extended binomial sequence $w_{n,r}(k)$ (red) for different radius r and approximations degrees n in comparison to the Gaussian function $g(x)$ (blue).

Both axis of the Gaussian function are scaled to the extended binomial to make the graphs comparable.



Picture 21: Extended binomial function graphs

Please note that the extended binomial coefficients are a discrete sequence of integers and not a continuous function.

Strictly spoken the comparison of discrete and continues function is not completely correct since an analog blur function of radius $r = 1$ still performs a blurring on an analog image whereas the discrete blur function of the binomials of discrete radius $r = 1$ does not.

2.4. Discrete frequency spectrum

Discrete values like the extended binomial function could also be transformed from the space domain into the frequency domain by the discrete Fourier transform (DFT).

The sequence of n complex numbers $a_k,\ldots, a_{n-1}$ is transformed into the sequence of n complex numbers $\hat{a}_k,\ldots, \hat{a}_{n-1}$ by the DFT according to the formula: $\hat{a}_k = \sum_{j=0}^{n-1} a_j e^{-2\pi i k \frac{j}{n}}$ for $[k = 0, \ldots, n-1]$. (27)

If the sequence is an even function of real numbers it is possible to get rid of the imaginary part of the transformation. This symmetry could be gained by the condition $a_j = a_{n-j}$.

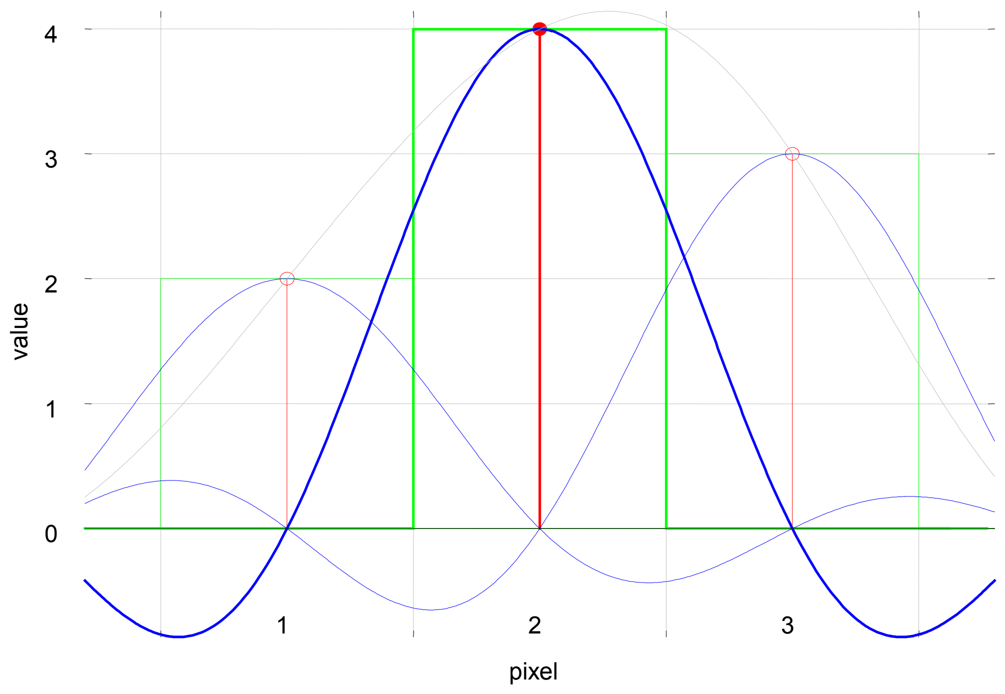
The DFT for the sequence of $n/2$ real numbers $a_j = a_{n-j}$ makes $\hat{a}_k = \sum_{j=0}^{n-1} a_j e^{-2\pi i k \frac{j}{n}} = \sum_{j=0}^{n-1} a_j \cos 2\pi k \frac{j}{n}$. (28)

The transformed sequence of $\hat{a}_k$ also becomes the even series $\hat{a}_k = \hat{a}_{n-k}$ due to the symmetry of the trigonometric function. The same applies for the inverse DFT.

This equation of the DFT is a bit irritating since it has no relation to the spectrum frequency ω. For better understanding the Fourier transform of the extended binomial sequence is developed from equation (15). To make the spectra comparable to previous graphs both axis of the extended binomial blur function are scaled to the Gaussian distribution. Such 'normalized' discrete blur function of the extended binomial sequence is calculated by:

$$\mathrm{f}_{n,r}(x) = \frac{\sigma}{r^n} \sum_{k=0}^{s} \mathrm{w}_{n,r}(k)\, \mathrm{s}(k - \bar{k} + x\sigma)\,. \tag{29}$$

The function s(x) defines the sampling function of a pixel in an image (conversion). Picture 22 shows different possibilities to build an analog image from the three discrete pixel values 2, 4, 3 and vice versa.



Picture 22: pixel sampling functions s(x)

a) red: discrete value
b) green: average value
c) blue: frequency limited value
(gray: sum of blue)

The function of the frequency spectrum makes then:

$$\pi\,\mathrm{a}_{n,r}(\omega) = \int_{-\infty}^{\infty} \mathrm{f}_{n,r}(x) \cos\omega x \,\mathrm{d}x = \frac{\sigma}{r^n} \sum_{k=0}^{s} \mathrm{w}_{n,r}(k) \int_{-\infty}^{\infty} \mathrm{s}(k - \bar{k} + x\sigma) \cos\omega x \,\mathrm{d}x = \frac{1}{r^n}\hat{\mathrm{s}}\left(\frac{\omega}{\sigma}\right) \sum_{k=0}^{s} \mathrm{w}_{n,r}(k) \cos\omega\frac{k-\bar{k}}{\sigma}\,. \tag{30}$$

Since the function $f_{n,r}(x)$ is even the complex part equals zero.

The following table contains different possibilities of sampling functions $s(x)$ and its Fourier transform $\hat{s}(\omega)$.

Pixel sampled as	Function $s(x)$	Fourier transform $\hat{s}(\omega)$	Frequency spectrum $\pi a_{n,r}(\omega)$
a) discrete value at the pixel center	$\delta(x)$	1	$\frac{1}{r^n}\sum_{k=0}^{s} w_{n,r}(k)\cos\omega\frac{k-\bar{k}}{\sigma}$
b) average value (integral) of the pixel area	$\mathrm{rect}(x)$	$\frac{\sin\omega/2}{\omega/2}$	$\frac{2\sigma}{r^n\omega}\sin\frac{\omega}{2\sigma}\sum_{k=0}^{s} w_{n,r}(k)\cos\omega\frac{k-\bar{k}}{\sigma}$
c) frequency limited value	$\frac{\sin\pi x}{\pi x}$	$\mathrm{rect}\left(\frac{\omega}{2\pi}\right)$	$\frac{1}{r^n}\mathrm{rect}\left(\frac{\omega}{2\pi\sigma}\right)\sum_{k=0}^{s} w_{n,r}(k)\cos\omega\frac{k-\bar{k}}{\sigma}$

Other sampling functions would also be possible.

Now the discrete spectrum of the pixel sequence could be calculated.

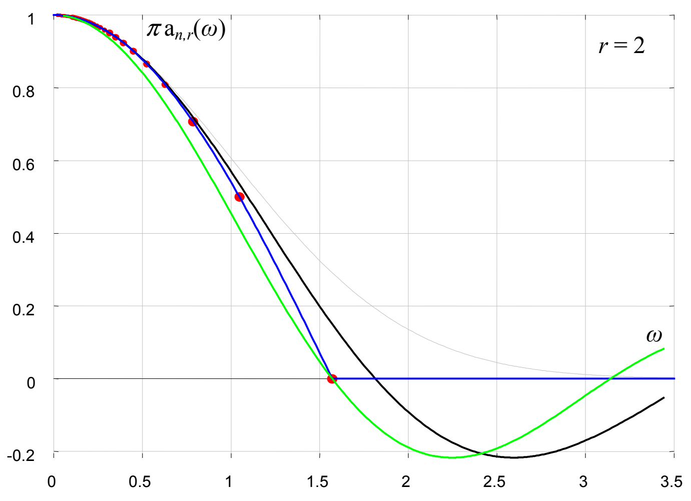
The parameter of the circular frequency ω becomes the discrete pixel distance p and has to be scaled too:

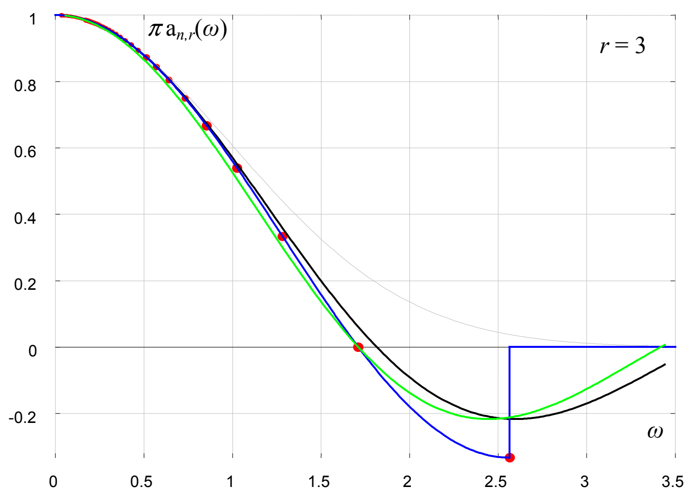
$$p=\frac{2\pi\sigma}{\omega}=\frac{\pi}{\omega}\sqrt{\frac{n(r^2-1)}{3}}\,. \tag{31}$$

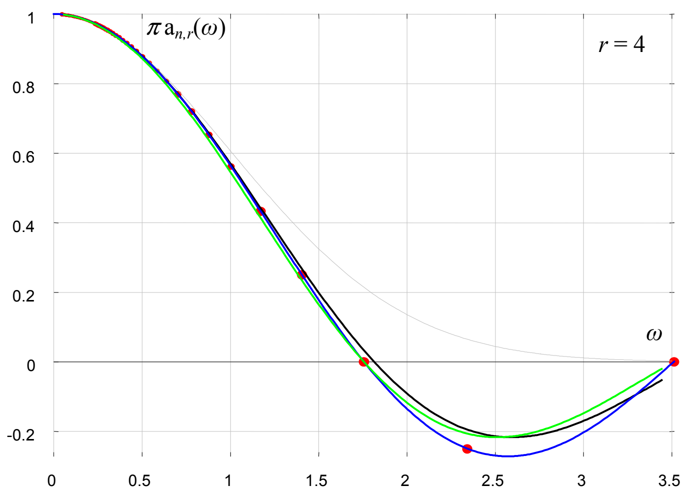
Since an image consists of discrete pixel the spectrum also contains only discrete values of the pixel distance (space period) p. In case of sample function a) the frequency spectrum becomes a DFT and looks now much more like equation (28):

$$\underline{\pi\, a_{n,r}\left(\frac{2\pi\sigma}{p}\right)=\frac{1}{r^n}\sum_{k=0}^{s} w_{n,r}(k)\cos 2\pi\frac{k-\bar{k}}{p}}\,. \qquad [\text{integer } p,\ p>0] \tag{32}$$

The remaining deviation is due to the required different scaling of the axis.







Picture 23: discrete spectra of 1st degree (box blur)

for sampling functions of

a) red: discrete pixel value

b) green: average pixel value

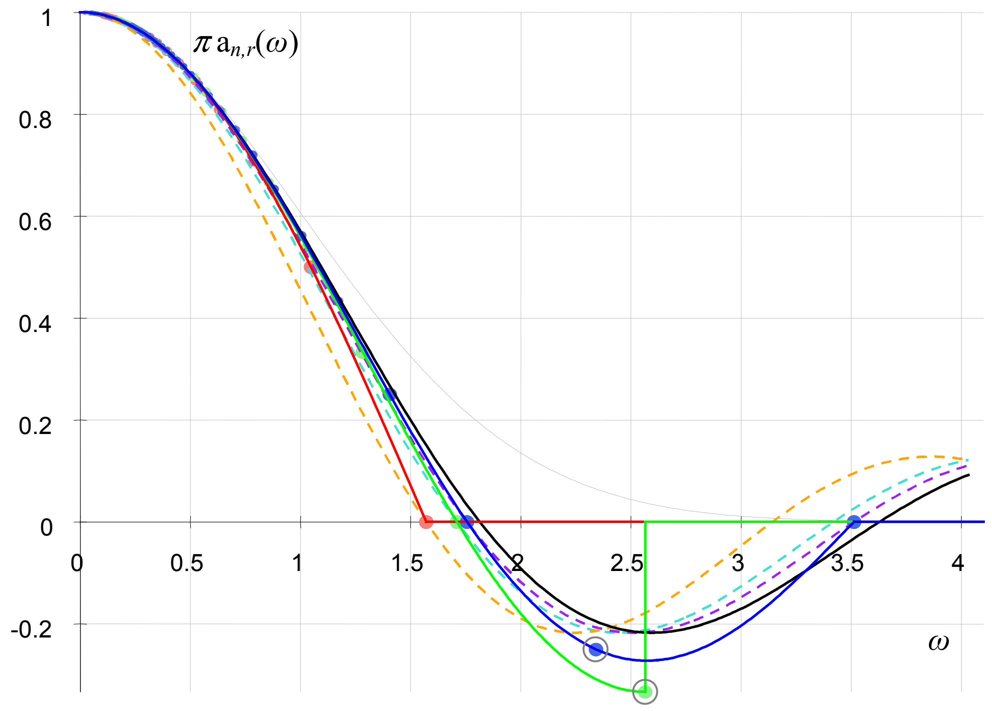
c) blue: frequency limited value

black: analog approximation spectrum $a_1(\omega)$

gray: Gaussian spectrum $a_g(\omega)$

The graphs of picture 23 show the frequency spectrum of the first approximation degree for different radii r. The points in the graphs are at discrete values of possible discrete time periods. The discrete points start at a value of $p = 2$ because of the alias effect.

The maximum visible frequency in an image is only half of the maximum pixel frequency (of space period $p = 1$) because of the alias effect. Due to the Nyquist-Shannon sampling theorem the affected frequency of an image (Nyquist rate) is half of the pixel frequency.



Picture 24: discrete spectrum of 1st degree (box blur)

black: analog, $a_1(\omega)$

red: $r = 2$, $a_{1,2}(\omega)$

green: $r = 3$, $a_{1,3}(\omega)$

blue: $r = 4$, $a_{1,4}(\omega)$

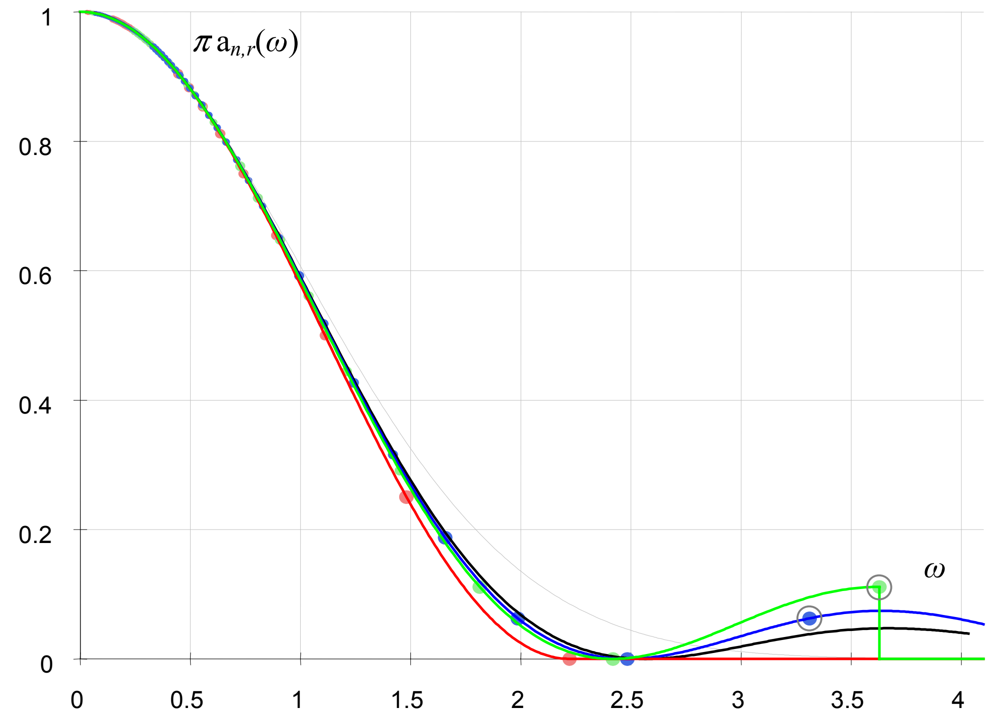
Spectrum of pixel sample:

a) discrete: points

b) average: dashed line

c) frequency limited: line

The function graphs of picture 24 look a bit crowded but the picture just contains all three previous graphs in one for comparison.



Picture 25: discrete spectrum of 2nd degree (linear blur)

black: analog, $a_2(\omega)$

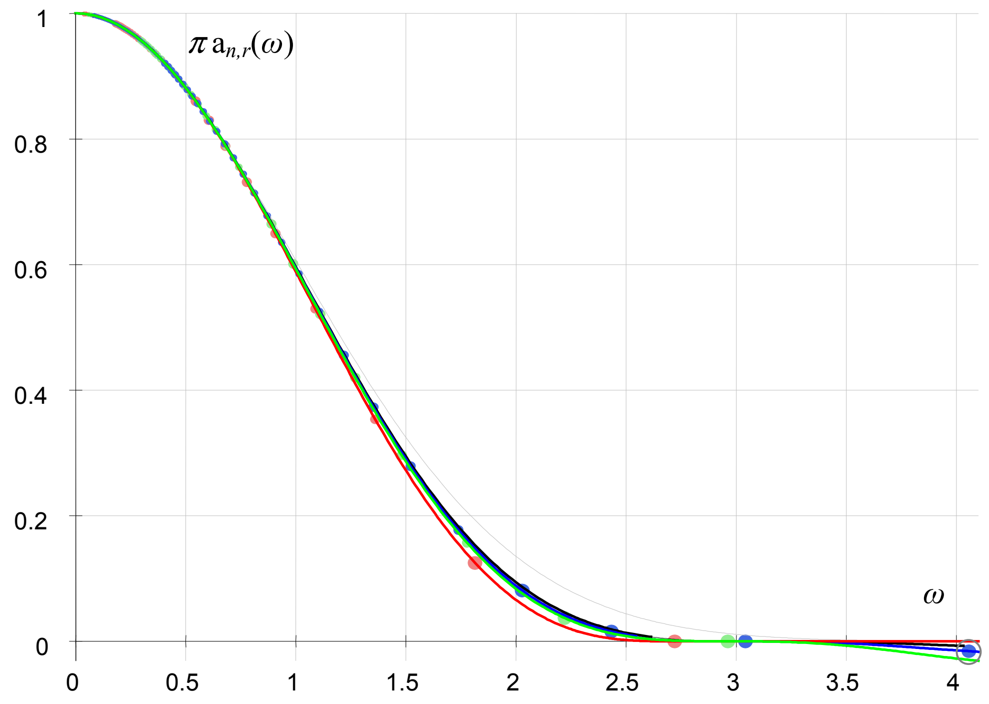
red: $r = 2$, $a_{2,2}(\omega)$

green: $r = 3$, $a_{2,3}(\omega)$

blue: $r = 4$, $a_{2,4}(\omega)$

The graphs of picture 25 to 27 show the frequency spectrum $a_{n,r}(\omega)$ of different approximation degrees and radii. Only the discrete spectrum (points) and frequency limited spectrum (line) are drawn.

The black function shows the analog spectrum function $a_n(\omega)$ of equation (20) of the corresponding approximation degree for comparison. The grey function is the spectrum of the Gaussian blur $a_g(\omega)$.



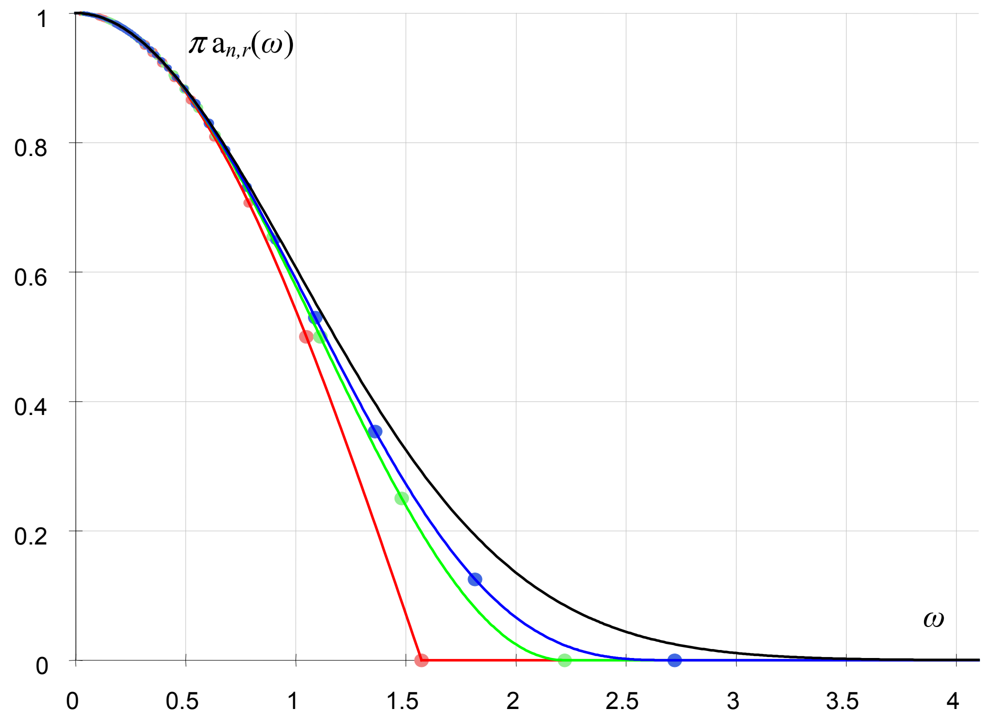
Picture 26: discrete spectrum of 3rd degree (square blur)

black: analog, $a_3(\omega)$

red: $r = 2$, $a_{3,2}(\omega)$

green: $r = 3$, $a_{3,3}(\omega)$

blue: $r = 4$, $a_{3,4}(\omega)$



Picture 27: discrete spectrum of radius $r = 2$ (binomial blur)

black: Gaussian, $a_g(\omega)$

red: 1st degree (box)

green: 2nd degree (linear)

blue: 3rd degree (square)

The graph of picture 27 also makes clear why the binomial sequence is suitable for blur calculation. The discrete spectrum of the image contains no frequency overshoot.

Discrete approximation error:

For any approximation degree n and radius r there is a minimum frequency $\omega_0 = 2\pi\sigma/r = \pi\sqrt{n(1-1/r^2)/3}$ which is entirely suppressed: $a_{n,r}(\omega_0) = 0$ for the discrete period of $p = r$.

What most matters for the approximation is that for a certain frequency ω_0 which is already fully suppressed in the blurred image there is a maximum error of a higher frequencies $\omega_p > \omega_0$ which is not fully suppressed (as already calculated for the analog approximation in chapter 1.10). This peak error in the first overshoot of the frequency is marked by a circle in the graphs. It does not occur for the binomial blur of $r = 2$.

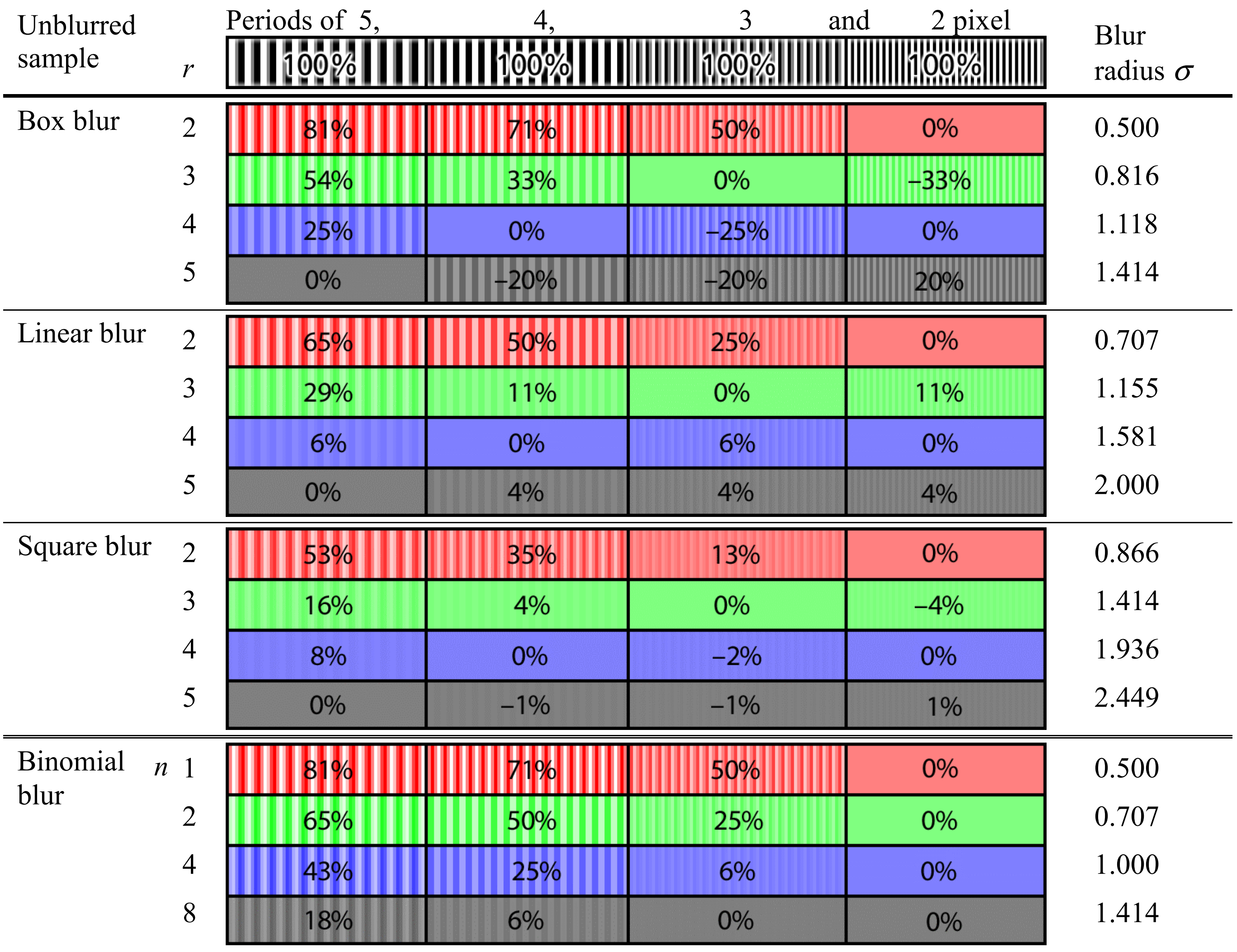
For large $r >> 2$ this peak error is the same than the error already calculated by equation (22) but unfortunately the worst case of the discrete approximation occurs for the radius of $r = 3$ and makes: $\pi\ a_{n,3}(\omega_p) = (-3)^{-n}$. The second largest error occurs for $r = 4$ and makes: $\pi\ a_{n,4}(\omega_p) = (-4)^{-n}$ which is not much more than the error of the analog approximation. It may be an option to possibly avoid the worst case and always work with radii $r > 3$.

2.5. Artificial sample images

Could these findings of the analysis be confirmed by sample images? Not all higher image frequencies are suppressed according the spectrum graphs. What happens if the approximation algorithm is applied on specially prepared artificial images? Such results are visible in picture 28. The colors correspond to the colors of the graphs for certain values of n and r.

As the spectrum graphs indicate higher frequencies are suppressed. But for lower approximation degrees, especially for the box blur, not all frequencies are suppressed well enough. This deficiency is also visible in the samples. Although frequencies of small blur radii in images are suppressed they reappear on larger blur radii. This effect vanishes on higher approximation degree.

Please note that the spectrum in the graphs is sometimes negative. This is a 180 degree phase shift of the image frequency which is also visible in the sample images of picture 28.



	r	Periods of 5,	4,	3 and	2 pixel	Blur radius σ
Unblurred sample		100%	100%	100%	100%	
Box blur	2	81%	71%	50%	0%	0.500
	3	54%	33%	0%	–33%	0.816
	4	25%	0%	–25%	0%	1.118
	5	0%	–20%	–20%	20%	1.414
Linear blur	2	65%	50%	25%	0%	0.707
	3	29%	11%	0%	11%	1.155
	4	6%	0%	6%	0%	1.581
	5	0%	4%	4%	4%	2.000
Square blur	2	53%	35%	13%	0%	0.866
	3	16%	4%	0%	–4%	1.414
	4	8%	0%	–2%	0%	1.936
	5	0%	–1%	–1%	1%	2.449
Binomial blur	n 1	81%	71%	50%	0%	0.500
	2	65%	50%	25%	0%	0.707
	4	43%	25%	6%	0%	1.000
	8	18%	6%	0%	0%	1.414

Picture 28: Remaining contrast of blurred sample images containing certain image frequencies.

The numerical pixel values of the blurred images exactly confirm the calculation of the discrete spectrum analysis. The percentages in the picture represent the remaining contrast after applying the extended binomial blur filter (no blur = 100% contrast).

These examples show specially designed test cases of uniform frequency and maximum contrast which will rarely appear in normal images. But especially lower degrees as box blur for fast approximations should only be used carefully in applications.

Conclusion:

Mathematical conclusion of the spectrum analysis: If r approaches infinity the discrete spectrum $a_{n,r}(\omega)$ of the binomial sequence $w_{n,r}(k)$ approaches to the spectrum $a_n(\omega)$ of the analog function $b_n(x)$. If the degree n approaches infinity the spectrum $a_n(\omega)$ of the analog function $b_n(x)$ approaches to the spectrum $a_g(\omega)$ of the Gaussian function $g(x)$.

Practical conclusion of the spectrum analysis: The maximum frequency error is calculated by equation (22) which could be used as a rule of thumb for the needed approximation degree. For 8-bit images the necessary degree is for example $n = \ln 2^8 / \ln 4.6 = 3.63$, so degree three is accurate enough to avoid the 'reappearing higher frequency effect'. (If absolute accuracy is needed the worst case for the discrete approximation of radius $r = 3$ must be assumed: $n = \ln 2^8 / \ln 3 = 5$.) This error completely vanishes for the binomial blur ($r = 2$).

Advantages of the algorithm: The extended binomial calculation is fast, easy to compute and significantly reduce the complexity of programs.

Disadvantages of the algorithm: Lower degrees of the approximation can still contain high frequency parts of the image. The integer values to handle for computation get very large for greater blur radii (may be 64-bit arithmetic or floating point is needed). The minimum radius is limited (which is more relevant for higher approximation degrees).

2.6. Implementation

The implementation in code is quite easy despite the complex equations used to explain the theory of the algorithm. It is simple an extension of the box blur.

```
Extended Binomial blur:
  For every column
    Do row convolution
  For every row
    Do column convolution
```

```
Convolution:
  Initialization of pixel differences
  For every pixel
    Add differences for all degrees i: Dif[i+1] += Dif[i]
    Write next pixel: Dif[degree]/weight
    Add next Binomial differences (for all combinations i):
      Dif[0] += (-1)^i * combination(degree,i) * pixel(radius*i)
```

The initialization is done the same way than the convolution itself with the range of the first pixel but pixels outside the convolution are ignored.
The complexity of this implementation as well as the number of pixel reads is $O(n)$ and depends only on the approximation degree n.

Examples of different approximation degrees are in the source code chapter in C-code.
Using an approximation instead of the exact Gaussian blur is not really a disadvantage. First of all image data is quantized, so even exact calculation can only be an approximation to the analog Gaussian function. Second the Gaussian function is of infinite range whereas the extended binomial algorithm is of limited range making the calculation easier.
The algorithm presented in this paper is not limited to the Gaussian blur. Its universal concept may be applied to other functions too.

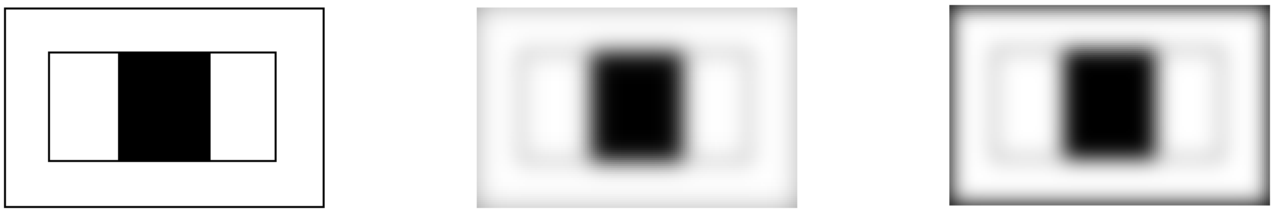
2.6.1. Boarders

Many filter functions like Gaussian blur need surrounding pixels for the calculation. But no pixels exist beyond the boarder.

The easiest way of boarder handling is simply to repeat the pixel at the boarder beyond the boarder. This is the method most image processing programs (like Photoshop) handle this case. So access to pixels beyond the image boarder is saturated to boarder pixel.

Most example programs at the annex do it likewise to make the algorithm not too complex. They access pixels outside the image and assume that the host program can handle the case.

But strictly speaking such handling is not correct for blurring. Assuming a complete white image with a one pixel black boarder, the blurring does not only see a one pixel black boarder because every pixel outside the image is black. This results in a blurred black boarder of the image.


Picture 29a: sample image **b: correct boarder blur** **c: wrong boarder blur**

The correct handling would be to do separate weighting of each pixel near the boarder and therefore ignoring pixels outside the image in the calculated blur. But this approach has the disadvantage of being more complex and slowing down the execution speed.

Fortunately correct boarder handling is easy to implement in the extended binomial algorithm of the first degree. The accumulated pixel blur in the first degree is a triangle. This triangle has simply to be cut of to avoid influences of hypothetic pixel beyond the boarder. Chapter 3.2.5 shows an example program for correct boarder handling.

2.6.2. Floating point blur radius

A disadvantage of using (extended) binomials for blur calculation is that only integer numbers of blur radii are possible. That is not so bad since the effective bur radius is a fraction of the integer radius calculated by equation (26). But it would be convenience not to be limited to integers.

Such floating point computation of the blur radius is possible by calculation two consecutive radii r and r+1 and doing a proper interpolation between the results.

Chapter 3.2.5 shows an example program of calculating a floating point blur radius by interpolation of the second approximation degree.

2.6.3. Pixel buffer

Every blur algorithm needs access to the surrounding pixels. Because the two dimensional blur is done by two successive one dimensional blurs some image buffer is needed to store the temporary results of the first run. Since access to pixels may be costly the problem could be solved by a small pixel buffer.

The pixel buffer saves the source pixels in one dimension of the current surrounding blur. That way the already calculated blur can be stored immediately without overwriting source pixel and the access to the source pixel for the calculation is speed up by the pixel buffer.

2.7. Other efficient blur algorithm

Different other methods are used to speed up the computation for larger blur radius. For example it is possible to do the blur by a FFT transformation. In this way the runtime per value is O(log(r)). But such an algorithm is very complicated and has a huge calculation overhead. In most cases it is not worth the effort. Another opportunity is to use a finite-state-machine (SKIPSM), but this needs a large temporary buffer [7].

Young and van Vliet [4] use a recursive implementation of the Gaussian filter which has also a runtime complexity independent of the blur radius. The disadvantage of this approach is that the length of the temporary output stream is infinite. For practical implementation the stream has to be cut off somewhere introducing an error to the calculation. An implementation of the recursive Gaussian filter by Tom Fiddaman (slightly improved) is at the last chapter.

Down-sampling is another algorithm which uses the effect that the Gaussian blur filter reduces the image's high-frequency components. Because of this low pass effect the image can be down sampled before applying the Gaussian filter. The algorithm for such a down-sampling Gaussian blur filter is simple:

```
Down-sample image by a box blur.
Perform a normal Gaussian blur with also reduced blur radius.
Up-sample the image again by bilinear interpolation.
```

An example implementation of a down-sampled Gaussian blur is at the source code appendix.

It is difficult to compare the runtime of different algorithm. The execution speed depends very much from the implementation. To give a rough estimate the following table shows the runtime for different blur radius:

Run time	Blur radius			O()
Algorithm	1	10	100	r
Conventional	195%	389%	2280%	O(r)
Down sampling	592%	196%	100%	O($1/r^2$)
Extended binomial	168%	170%	190%	O(1)
Recursive	203%	205%	217%	O(1)

The table confirms the consideration that the conventional algorithm has a linear dependency of the blur radius and extended binomial and recursive algorithm have no dependency. Down sampling itself has no dependency either, but the radius for blurring is divided by the down sampling factor and therefore the algorithm gets faster for larger radius.

Of course all algorithms could be further optimized.

2.8. Edge detection

A typical application of the Gaussian blur is edge detection. Edges in an image could be detected by the pixel differences of consecutive pixels of the image data. But this approach has the disadvantage that noise is erroneously detected as edges. So it would be convenient to blur the image before edge detection and differentiate the blur afterwards.

Fortunately these two processes, blurring and derivation, could be optimized to due the rules in convolution. Differentiating a convolution is the same than doing the convolution by a differentiated function. The absolute value (gradient magnitude) is taken as edge information:

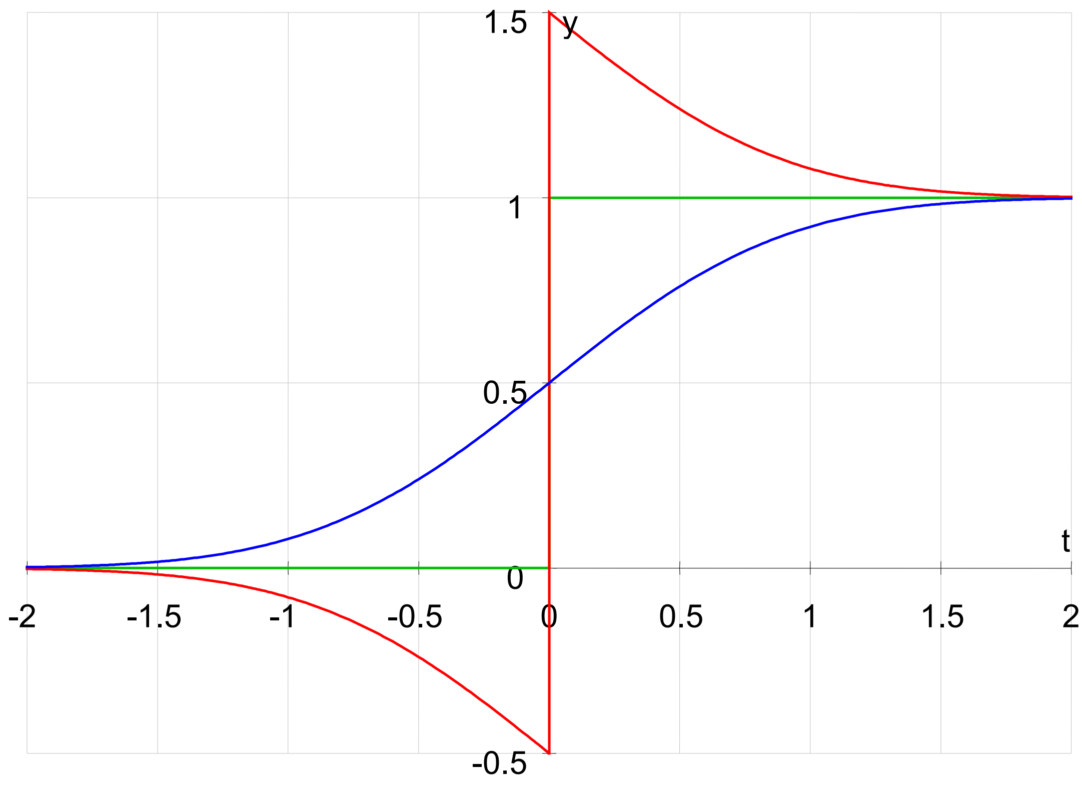
$\left|\mathrm{D}(f * g)\right| = \left|f * \mathrm{D}\,g\right| = \left|f * \left(g_x\,\mathrm{D}\,g_y + g_y\,\mathrm{D}\,g_x\right)\right| = \sqrt{\left(f * g_x\,\mathrm{D}\,g_y\right)^2 + \left(f * g_y\,\mathrm{D}\,g_x\right)^2}$, where D denotes to the differential operator. This is especially useful since the derivation is already part of the extended binomial calculation.

An example implementation of edge detection by convolution by the extended binomial algorithm as added at the source code chapter. The program is short and fast but treads the image as grayscale. Color edges with equal gray value are therefore not recognized.

This method of edge detection is useful in many applications especially for noisy images. It could be used as mask for other operations like blur or sharpen to protect certain parts of the image against modification.

2.9. Sharpening example

The Gaussian blur is needed in many filter algorithms. One application example is sharpening. Sharpening could be seen a negated blur.



Picture 30: sharpen and blur filter

a. original input

b. blur filter

c. sharpen filter

Picture 30 shows blur and sharpen filter response. The sharpening output signal is: *sharpen* = 2×*input* – *blur*.

An example program for a blur-sharpen filter is at the source code annex. Blurring and sharpening have separate sliders and are applied at the same time. If both sliders have the same value of course nothing happens. The difference makes the threshold slider. If the local contrast (of the selected radius) is below the threshold only blurring is applied. For local contrast above the threshold both blur and sharpening is applied. In this way for low contrast like noise the blur filter is applied whereas higher contrast is sharpened.

Without blur this filter works like an USM-filter. In addition a blur could be applied which affects only low contrast (noise).

2.10. References

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery: Numerical Recipes, Cambridge University Press, 3rd Edition, 2007, http://sdu.ictp.it/nr/index.html.

[2] Donald E. Knuth: The Art of Computer Programming, Vol. 1, 3rd Edition, Addison Wesly, 1997.

[3] Hans J. Dirschmid: Mathematische Grundlagen der Elektrotechnik, Vieweg, 1992.

[4] Ian T. Young, Lucas J. van Vliet: Recursive implementation of the Gaussian filter, Delft 1995, http://www.ph.tn.tudelft.nl/~lucas/publications/1995/SP95TYLV/SP95TYLV.pdf

[5] Matthew Aubury, Wayne Luk: Binomial Filters, Journal of VLSI Signal Processing, 1995, http://www.doc.ic.ac.uk/~wl/papers/bf95.pdf

[6] Ian T. Young, Lucas J. van Vliet, Piet W. Verbeek: Recursive Gaussian Derivative Filter, ICPR 1998, http://www.ph.tn.tudelft.nl/~lucas/publications/1998/ICPR98LVTYPV/ICPR98LVTYPV.pdf

[7] Frederick M. Waltza, John W. V. Millerb: An efficient algorithm for Gaussian blur using finite-state machines, Boston 1998, www.engin.umd.umich.edu/~jwvm/ece581/21_GBlur.pdf

[8] Ian T. Young, Jan J. Gerbrands, Lucas J. van Vliet: Fundamentals of Image Processing, Delft University of Technology, V2.2, http://www.diplib.org/home222

[9] Dave Hale: Recursive Gaussian filters, CWP Report 546, 2006, http://ww.cwp.mines.edu/Meetings/Project06/cwp546.pdf

[10] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand: A Gentle Introduction to Bilateral Filtering and its Applications, 2008, http://people.csail.mit.edu/sparis/bf_course/

2.11. Tools

A few tools were necessary to develop this algorithm.

- The computer algebra system Maxima was used for symbolic mathematic and graphs: http://maxima.sourceforge.net/.
- The algorithms were tested with Filtermeister, the graphics plug-in creator: http://www.filtermeister.com/.

3. Source code

3.1. Maxima commands

```
/* The packages draw and functs are needed for function graphs */
load(functs); load(draw);

/* extended binomial function */
b(n,x):=sqrt(3*n)/(12*(n-1)!)*sum((-1)^i*combination(n,i)*(sqrt(n/12)*x+n/2-
i)^n/abs(sqrt(n/12)*x+n/2-i),i,0,n);

/* analog spectrum */
a(n,x):=(sin(x*sqrt(3/n))/(x*sqrt(3/n)))^n;

/* extended binomial sequence */
w(n,r,k):=sum((-1)^floor(i/r)*combination(n-1,floor(i/r))*combination(n+k-i-
2,k-i),i,0,k);

/* discrete spectrum */
u(n,r,x):=sum(w(n,r,k)*cos(x*(k-n*(r-1)/2)/sqrt(n*(r^2-1)/12)),k,0,n*(r-1))
/r^n;

/* rectangular convolution */
rect(n,x):=sum((-1)^i/(2*(n-1)!)*combination(n,i)*(x+n/2-i)^n/abs(x+n/2-
i),i,0,n);

/* extended binomial and Gaussian plot */
Plot_bi(n,r):=( k:(n*r-n)/2, s:sqrt(n*(r^2-1)/12), /* mean,  deviation  */
 draw2d(axis_top=false, axis_right=false, grid=true, xaxis=true, yaxis=false,
  axis_left=false, xtics_axis=true, ytics_axis=true,
  color=red, explicit(w(n,r,floor(x+1/2)),x,k-3*s,k+3*s),
  color=blue, explicit(r^n*exp(-((x-k)/s)^2/2)/sqrt(2*%pi)/s,x,k-3*s,k+3*s)))$
```

3.2. Filtermeister Programs

Programs for Filtermeister were written to test the different algorithms. Filtermeister is a plugin for Photoshop with a C-like programming language and easy access to image information.

Filtermeister is available for testing at http://www.filtermeister.com/.

If you are not familiar with Filtermeister a few words to the C programs:

Filtermeister (in its issue beta 7) does not know C-directives (like #define) and user functions. So the code may look a bit strange some times. No normal C-arrays exist either. A one dimensional integer array of the size 1024 (`N_CELLS`) is available through put/get functions. Three dimensional arrays are available by special functions.

The pixel buffer uses the integer array of put/get and counts on the fact that the access is limited to a ring buffer of the size of `N_CELLS`.

All example programs work with 8 and 16 bit images in any color mode.

The box blur (first degree approximation) is too simple and the blur unsatisfactorily for a program.

- The 2nd, 3rd and 4th approximation degree programs are example implementations of the algorithm and could be easily extended to any degree.
- The arbitrary approximation degree program is for comparing the effect of different degrees on images.
- The improved second degree example handles large and small radii and boarders correctly.
- The edge detection and blur-sharpen example show some application of the algorithm.
- Recursive and down-sampled Gaussian blur examples are two other algorithm for blurring.

3.2.1. Second degree approximation

The program for the second degree approximation is a great improvement to the box blur. It is as simple and has only a slight approximation error.

```
%ffp

Category: "easy.Filter"
Title:    "Gauss2"
Version:  "1.1"
Filename: "gauss2.8bf"
Author:   "Alois Zingl"
Copyright:"© 2010"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Second degree of the extended binomial Gaussian blur filter algorithm."

ctl(0): "Radius (pixel)", divisor = (i0=3), range=(0, (int)(N_CELLS*i0/4.9))

OnFilterStart:
{                                          // set padding to blur radius
  int radius = doingProxy?(ctl(0)+zoomFactor/2)/zoomFactor:ctl(0);
  needPadding = sqrt(6.)*radius/i0+1;      // sqrt(6) = degree factor
  bandWidth = 100+4*needPadding;
  isTileable = !doingProxy;                // split large images
  return false;
}

ForEveryTile:
{
  int radius = sqrt(6.*ctl(0)*ctl(0)/(scaleFactor*scaleFactor*i0*i0)+1);

  for (z = 0; z < Z; ++z)                  // for all color planes
  {
    for (x = x_start; x < x_end; ++x)      // vertical blur...
    {
      int dif = 0, sum = (radius*radius)>>1;
      for (y = y_start-2*radius; y < y_end; ++y)
      {                                    // inner vertical blur loop
        if (y >= y_start)
        {
          pset(x, y, z, sum/(radius*radius));  // set blurred pixel
          dif += get(y-radius)-2*get(y);       // sum up differences: +1, -2, +1
        } else if (y+radius >= y_start) dif -= 2*get(y);
        sum += dif += p = src(x, y+radius, z); // accumulate pixel blur
        put(p, y+radius);                      // buffer next pixel
      } // y
    } // x
    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    for (y = y_start; y < y_end; ++y)      // horizontal blur...
    {
      int dif = 0, sum = (radius*radius)>>1;
      for (x = x_start-2*radius; x < x_end; ++x)
      {                                    // inner vertical blur loop
        if (x >= x_start)
        {
          pset(x, y, z, sum/(radius*radius));   // set blurred pixel
          dif += get(x-radius)-2*get(x);        // sum up differences: +1, -2, +1
        } else if (x+radius >= x_start) dif -= 2*get(x);
        sum += dif += p = pget(x+radius, y, z); // accumulate pixel blur
        put(p, x+radius);                      // buffer next pixel
      } // x
    } // y
  } // color plane

  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```

3.2.2. Third degree approximation

The program for the third degree approximation is accurate enough for most applications. The only disadvantage of this 3rd degree approximation is the half pixel image offset for every half pixel blur radius.

```
%ffp

Category: "easy.Filter"
Title:    "Gauss3"
Version:  "1.1"
Filename: "gauss3.8bf"
Author:   "Alois Zingl"
Copyright:"© 2010"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Third degree of the extended binomial Gaussian blur filter algorithm."

ctl(0): "Radius (pixel)", divisor = 2, range=(0,N_CELLS/3)

OnFilterStart:
{
  int radius = doingProxy?(ctl(0)+zoomFactor/2)/zoomFactor:ctl(0);
  needPadding = 3*radius/2+2;
  bandwidth = 100+4*needPadding;
  isTileable = !doingProxy; // split large images
  return false;
}

ForEveryTile:
{
  int radius = sqrt(1.0+ctl(0)*ctl(0)/(scaleFactor*scaleFactor));

  for (z=0; z < Z; ++z)                        // for all color planes
  {
    for (x = x_start; x < x_end; ++x)          // vertical blur for every row
    {
      int dif = 0, der = 0; float sum = 0;
      for (y = y_start-3*radius; y < y_end; ++y)
      {                                          // inner vertical blur loop
        if (y >= y_start)
        {
          dif += 3*(get(y)-get(y+radius))-get(y-radius);    // {+1,-3,+3,-1}
          pset(x, y, z, (int)(sum/(radius*radius*radius))); // set blurred pixel
        } else if (y+radius >= y_start) dif += 3*(get(y)-get(y+radius));
          else if (y+2*radius >= y_start) dif -= 3*get(y+radius);
        sum += der += dif += p = src(x, y+3*radius/2, z);   // accumulate pixel blur
        put(p, y+2*radius);                                 // buffer next pixel
      } // y
    } // x
    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    for (y = y_start; y < y_end; ++y)      // horizontal blur for every column
    {
      int dif = 0, der = 0; float sum = 0;
      for (x = x_start-3*radius; x < x_end; ++x)  // inner horizontal blur loop
      {
        if (x >= x_start)
        {
          dif += 3*(get(x)-get(x+radius))-get(x-radius);    // {+1,-3,+3,-1}
          pset(x, y, z, (int)(sum/(radius*radius*radius))); // set blurred pixel
        } else if (x+radius >= x_start) dif += 3*(get(x)-get(x+radius));
          else if (x+2*radius >= x_start) dif -= 3*get(x+radius);
        sum += der += dif += p = pget(x+3*radius/2, y, z);  // accumulate pixel blur
        put(p, x+2*radius);                                 // buffer next pixel
      } // x
    } // y
  } // color plane
  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```

3.2.3. Fourth degree of approximation

The fourth degree approximation has nearly no difference to the true Gaussian blur and could even be used for 16-bit images.

```
%ffp

Category: "easy.Filter"
Title:    "Gauss4"
Version:  "1.1"
Filename: "gauss4.8bf"
Author:   "Alois Zingl"
Copyright:"© 2010"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Fourth degree of the extended binomial Gaussian blur filter algorithm."

ctl(0): "Radius (pixel)", range=(0,(int)(N_CELLS/4/1.73)) // limit to buffer

OnFilterStart:
{
  int radius = doingProxy?(ctl(0)+scaleFactor/2)/scaleFactor:ctl(0);
  needPadding = 2*1.73*radius+3; // 1.73 = sqr(12/degree) = degree factor
  isTileable = !doingProxy; // split large images
  bandwidth = 100+4*needPadding;
  return false;
}

ForEveryTile:
{
  int radius = sqrt(3.0*ctl(0)*ctl(0)/(scaleFactor*scaleFactor)+1);
  int x, y, z, p;
  float weight = 1.0/((double)radius*radius*radius*radius);

  for (z=0; z < Z; ++z) // for all color planes
  {
    for (x = x_start; x < x_end; ++x)     // vertical blur...
    {
      float dif = 0., der1 = 0., der2 = 0., sum = 0.;
      for (y = y_start-4*radius; y < y_end; ++y)
      {                                   // set up init values for the first blur
        if (y >= y_start)
        {            //{+1,-4,+6,-4,+1}
          dif += -4*(get(y-radius)+get(y+radius))+6*get(y)+get(y-2*radius);
          pset(x, y, z, (int)(sum*weight)); // set blurred pixel
        } else
        {
          if (y+3*radius >= y_start) dif -= 4*get(y+radius);// -4,
          if (y+2*radius >= y_start) dif += 6*get(y);        //    +6,
          if (y+  radius >= y_start) dif -= 4*get(y-radius);//       -4,(+1)}
        }
        sum += der1 += der2 += dif += p = src(x, y+2*radius-1, z);// accumulate blur
        put(p, y+2*radius);  // buffer pixel, min buffer size: 4*radius
      } // y
    } // x
    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();
```

```
      for (y = y_start; y < y_end; ++y)     // horizontal blur...
      {
        float dif = 0., der1 = 0., der2 = 0., sum = 0.;
        for (x = x_start-4*radius; x < x_end; ++x)
        {                                  // set up init values for the first pixel
          if (x >= x_start)
          {          //{+1,-4,+6,-4,+1}
            dif += -4*(get(x-radius)+get(x+radius))+6*get(x)+get(x-2*radius);
            pset(x, y, z, (int)(sum*weight)); // set blurred pixel
          } else
          {
            if (x+3*radius >= x_start) dif -= 4*get(x+radius);// -4,
            if (x+2*radius >= x_start) dif += 6*get(x);       //    +6,
            if (x+  radius >= x_start) dif -= 4*get(x-radius);//       -4,(+1)}
          }
          sum += der1 += der2 += dif += p = pget(x+2*radius-1, y, z);// accumulate blur
          put(p, x+2*radius);  // buffer pixel, min buffer size: 4*radius
        } // x
      } // y
    } // color plane
    return true;  //Done!
  }

  OnFilterEnd:
  {
    updateProgress(0, 1);
    return false;
  }
```

3.2.4. Arbitrary approximation degree

The approximation degree can be freely chosen. This example program only tests the algorithm. It can be used to compare different approximation degrees on the blur effect. An implementation of fixed degree is simpler and faster in Filtermeister. It needs at least Filtermeister version 1.0 beta 9.

```
%ffp

Category: "easy.Filter"
Title:    "Gauss"
Version:  "1.1"
Filename: "gauss.8bf"
Author:   "Alois Zingl"
Copyright:"© 2010"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Arbitrary degree of the extended binomial Gaussian blur filter algorithm."

ctl(0): "Radius (pixels)", divisor = (i0=5), range = (0,255*i0)
ctl(1): "Approximation degree", range = (1, 8), page = 2
ctl(2): StaticText, "n = 1, r = 1, Radius = 0.00", size = (100,*)

OnFilterStart:
{                                                       // set padding to blur radius
  int radius = doingProxy?(ctl(0)+zoomFactor/2)/zoomFactor:ctl(0);
  needPadding = ctl(1)*radius*sqrt(3.0/ctl(1))/i0+3; // degree factor
  bandwidth = 100+4*needPadding;
  isTileable = !doingProxy;                          // split large images
  allocArray(0, ctl(1), 0, 0, 8);                    // double array for blur derivatives
  return false;
}

OnCtl(n):
{      // the get/put pixel buffer has only N_CELLS entries, so limit slider values
  int f = N_CELLS*i0*sqrt(ctl(1)/12.0)-i0;          // degree factor
  if (e == FME_VALUECHANGED && n < 2 && ctl(0)*ctl(1) > f)
    setCtlVal(1-n, f/ctl(n));            // limit needed pixel buffer to N_CELLS
  return false;
}

ForEveryTile:
{
  int radius = (ctl(0)+scaleFactor/2)/scaleFactor;  // Blur radius
  int x, y, z, p, i, degree = ctl(1);
  radius = sqrt(12.0*radius*radius/degree/i0/i0+1); // degree factor
  setCtlTextv(2,"n = %d, r = %d, Radius = %.2f",    // view degree, discrete &
    ctl(1), radius, sqrt(ctl(1)*(radius*radius-1)/12.0)*scaleFactor);//effective radius

  for (z=0; z < Z; ++z)                             // for all color planes
  {
    // Update progress bar and cancel if ESC key was pressed
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    for (x = x_start; x < x_end; ++x)               // vertical Blur...
    {
      double dif = 0, sum;
      ffillArray(0, 0.0);                           // set derivatives to 0

      for (y = y_start-degree*radius; y < y_end; ++y) // inner vertical Blur loop
      {                                             // max buffer size: radius*degree
        put(src(x, y+degree*(radius-1)/2, z), y);   // buffer pixel

        for (p = 1, i = 0; i <= degree; ++i)        // accumulate differences
        {
          if (y+(degree-i)*radius < y_start) break; // still in init stage, break
          dif += p*get(y-i*radius);                 // pixel
          p = p*(i-degree)/(i+1);                   // next binomial
        }

        for (sum = dif/radius, p = degree-1; p--; ) // accumulate pixel Blur
          fputArray(0, p, 0, 0, sum = fgetArray(0, p, 0, 0)+sum/radius); // slow in FM

        if (y >= y_start) pset(x, y, z, (int)sum);  // set Blurred pixel
      } // y
    } // x
    for (y = y_start; y < y_end; ++y)               // horizontal Blur...
    {
      double dif = 0, sum;
      ffillArray(0, 0.0);                           // set derivatives to 0

      for (x = x_start-degree*radius; x < x_end; ++x)
      {                                             // inner horizontal Blur loop
        put(pget(x+degree*(radius-1)/2, y, z), x);  // buffer pixel

        for (p = 1, i = 0; i <= degree; ++i)        // accumulate differences
        {
          if (x+(degree-i)*radius < x_start) break; // still in init stage, break
          dif += p*get(x-i*radius);                 // pixel
          p = p*(i-degree)/(i+1);                   // next binomial
        }

        for (sum = dif/radius, p = degree-1; p--; ) // accumulate pixel Blur
          fputArray(0, p, 0, 0, sum = fgetArray(0, p, 0, 0)+sum/radius); // slow in FM

        if (x >= x_start) pset(x, y, z, (int)sum);  // set Blurred pixel
      } // x
    } // y
  } // color plane
  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  freeArray(0);
  return false;
}
```

3.2.5. Improved second degree approximation

This example program uses the second approximation degree as previously but with two improvements. First instead of limiting the blur radius to integer it can handle floating points for small blur radii. And second the blur of boarder pixel is calculated correctly (better than PhotoShop).

```
%ffp

/*
   Fast Gaussian blur by extended binomial algorithm of first degree
```

```
    * simple: very easy blur calculation
    * fast: no run time dependency on blur radius
    * accurate: correct blur calculation of pixel near the boarder
    * large blur range of floating point radius
    * works for all color modes including 16 bit and image tiling
*/

Category: "easy.Filter"
Title:    "Gaussian Blur"
Version:  "1.0"
Filename: "gauss.8bf"
Author:   "Alois Zingl"
Copyright:"© 2010 GPL"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Fast Gaussian blur filter algorithm."

// i0 = CtlDivisor(0); needed number of cells for get/put: 2.45*2*radius+1
ctl(0): "Radius (pixel)", divisor = (i0=10), range = (0, (int)(N_CELLS*i0/(2.45*2)-1))

OnFilterStart:
{                                         // set padding to blur radius
  int Radius = ctl(0)/(doingProxy ? zoomFactor*i0 : i0) + 1;
  needPadding = 2.45*Radius+1; // 2.45 = sqr(12/degree) = degree factor
  bandWidth = 100+3*needPadding;
  isTileable = !doingProxy;               // split large images
  return false;
}

ForEveryTile:
{
  float Weight = 2.45*ctl(0)/(i0*scaleFactor), fRadius = sqrt(Weight*Weight+1);
  int x, y, z, p, Radius = floor(fRadius); // integer blur radius
  fRadius -= Radius;                      // fraction of radius
  Weight = Radius*Radius + fRadius*(2*Radius+1);

  for (z = 0; z < Z; ++z)                 // for all color planes
  {
    for (x = x_start; x < x_end; ++x)    // vertical blur...
    {
      float sum = src(x, y_start-Radius-1, z), dif = -sum;

      for (y = y_start-2*Radius-1; y < y_end; ++y)
      {                                  // inner vertical blur loop
        p = src(x, y+Radius, z);         // next pixel
        put(p, y+Radius);                // buffer pixel
        sum += dif + fRadius*p; dif += p; // accumulate pixel Radius

        if (y >= y_start)
        {
          int s = 0, w = 0;              // boarder blur correction
          sum -= get(y-Radius-1)*fRadius; // addition for fraction blur
          dif += get(y-Radius)-2*get(y);  // sum up differences: +1, -2, +1

          // cut off accumulated blur area of pixel beyond the boarder
          // assume: added pixel values beyond boarder = value at boarder
          p = Radius-y;                  // top part to cut off
          if (p > 0)
          {
            p = p*(p-1)/2 + fRadius*p;
            s += get(0)*p;
            w += p;
          }
          p = y+Radius-Y+1;              // bottom part to cut off
          if (p > 0)
          {
            p = p*(p-1)/2 + fRadius*p;
            s += get(Y-1)*p;
            w += p;
          }
          pset(x, y, z, (int)((sum-s)/(Weight-w))); // set blurred pixel

        } else if (y+Radius >= y_start) dif -= 2*get(y);
      } // y
    } // x
    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    for (y = y_start; y < y_end; ++y)     // horizontal blur...
    {
      float sum = pget(x_start-Radius-1, y, z), dif = -sum;

      for (x = x_start-2*Radius-1; x < x_end; ++x)
      {                                   // inner vertical blur loop
        p = pget(x+Radius, y, z);         // next pixel
        put(p, x+Radius);                 // buffer pixel
        sum += dif + fRadius*p; dif += p; // accumulate pixel blur

        if (x >= x_start)
        {
          int s = 0, w = 0;               // boarder blur correction
          sum -= get(x-Radius-1)*fRadius; // addition for fraction blur
          dif += get(x-Radius)-2*get(x);  // sum up differences: +1, -2, +1

          // cut off accumulated blur area of pixel beyond the boarder
          p = Radius-x;                   // left part to cut off
          if (p > 0)
          {
            p = p*(p-1)/2 + fRadius*p;
            s += get(0)*p;
            w += p;
          }
          p = x+Radius-X+1;               // right part to cut off
          if (p > 0)
          {
            p = p*(p-1)/2 + fRadius*p;
            s += get(X-1)*p;
            w += p;
          }
          pset(x, y, z, (int)((sum-s)/(Weight-w))); // set blurred pixel

        } else if (x+Radius >= x_start) dif -= 2*get(x);
      } // x
    } // y
  } // color plane
```

```
  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```

3.2.6. Edge detection example

This example program detects edges in images by a derivation of the blur. It uses only the gray values of the pixel and could be improved by considering all color planes separately before summing up the edge information. It works only for 8-bit RGB images and uses the second degree blur approximation algorithm.

```
%ffp

Category: "easy.Filter"
Title:    "Edge detection"
Version:  "1.0"
Filename: "edgedetection.8bf"
Author:   "Alois Zingl"
Copyright:"© 2008"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Detection monochrome edges by derivative convolution."
// fast and easy:  works only in RGB 8bit mode and considers only grey pixel values

SupportedModes: RGBMode

ctl(0): "Blur (pixels)", divisor = 2, range=(1,N_CELLS/3)

OnFilterStart:
{
  int Blur = doingProxy?(ctl(0)+scaleFactor/2)/scaleFactor:ctl(0); // blur radius
  needPadding = 3*Blur/2+2;
  bandWidth = 100+4*needPadding;
  isTileable = !doingProxy; // split large images
  return false;
}

ForEveryTile:
{
  int Blur = max(1, (ctl(0)+scaleFactor/2)/scaleFactor);  // blur radius
  int x, y, z, p, Blur2 = (3*Blur+1)/2;
  double weight = 1.0/(Blur*Blur*Blur);

  for (x = x_start; x<x_end; ++x)      // vertical blur for every row
  {
    int dif = 0, der = 0, sum = 0;
    for (y = y_start-3*Blur; y < y_end; ++y)  // initialize row
    {
      p = srcp(x, y+Blur2); // pixel value
      p = 5*Rval(p)+9*Gval(p)+2*Bval(p); // convert to gray level
      put(p, y+3*Blur);         // buffer pixel, min buffer size: 3*Blur
      dif += p;            // accumulate differences {+1,
      if (y+2*Blur >= y_start) dif -= 3*get(y+2*Blur);// -3,
      if (y+Blur >= y_start) dif += 3*get(y+Blur);    //    +3,
      if (y >= y_start)           //                          -1}
      {
        dif -= get(y);
        p = (der*Blur*Weight/2)+2048; // derivation and ..
        psetp(x, y, (int)(sum*Weight)+(p<<12)); // .. blur stored in 12 bit accuracy
      }
      sum += der; der += dif;    // accumulate pixel blur
    } // y
  } // x

  //Update progress bar and cancel if ESC key was pressed
  if (updateProgress(doingProxy*y_start+y_end, doingProxy?y_end-y_start:Y))
    abort();

  for (y = y_start; y < y_end; ++y)     // horizontal blur for every column
  {
    int dif = 0, der = 0, sum = 0; // blur
    int dif2 = 0, sum2 = 0, p2;    // derivative blur
    for (x = x_start-3*Blur; x < x_end; ++x)  // initialize column
    {
      p = pgetp(x+Blur2, y)&0xffffff; // pixel value
      put(p, x+3*Blur);       // buffer pixel, min buffer size: 3*Blur
      dif += p>>12; dif2 += p&0xfff;// accumulate differences {+1,
      if (x+2*Blur >= x_start)        //                      -3,
      {
        p = get(x+2*Blur); dif -= 3*(p>>12); dif2 -= 3*(p&0xfff);
      }
      if (x+Blur >= x_start)          //                      +3,
      {
        p = get(x+Blur); dif += 3*(p>>12); dif2 += 3*(p&0xfff);
      }
      if (x >= x_start)               //                      -1}
      {
        p = get(x); dif -= p>>12; dif2 -= p&0xfff;

        p = 2*sum*Weight-4096;    // horizontal edges
        p2 = sum2*Blur*Weight;  // vertical edges
        p = min(255, sqr(p*p+p2*p2)/12);   // sum up edges
        psetp(x, y, p*0x10101); // set RGB pixel
      }
      sum += der; der += dif; sum2 += dif2; // accumulate pixel blur
    } // x
  } // y

  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```

3.2.7. Blur-Sharpen example

This example shows the application of a blur filter to sharpen images. See chapter 2.8 for slider explanation.

```
%ffp
Category: "easy.Filter"
Title:    "Blur - Sharpen"
```

```
Version:  "1.0"
Filename: "blursharpen.8bf"
Author:   "Alois Zingl"
Copyright:"© 2008"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Improve contrast by blur and sharpen."

ctl(0): "Radius (pixels)", val = 4, divisor = 2, range=(0,340)
ctl(1): "Blur (%)", range = (0, 100), val = 32
ctl(2): "Sharpen (%)", val = 64
ctl(3): "Threshold (0..255)", val = 16

OnFilterStart:
{
  int Blur = doingProxy?(ctl(0)+scaleFactor/2)/scaleFactor:ctl(0); // blur radius
  needPadding = 3*Blur/2+2;
  isTileable = !doingProxy; // split large images
  bandwidth = 100+4*needPadding;
  return false;
}

ForEveryTile:
{
  int Blur = sqrt(1.0+ctl(0)*ctl(0)/(scaleFactor*scaleFactor));  // blur radius
  int x, y, z, i, Blur2 = (3*Blur+1)/2;
  double Weight = Blur*Blur*Blur;
  int th = (imageMode<GRAY16MODE?1:128)*ctl(3);  // threshold

  for (z=0; z < Z; ++z) // for all color planes
  {
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    for (x = x_start; x < x_end; ++x)     // vertical blur for every row
    {
      double dif = 0., der = 0., sum = 0.;
      for (y = y_start-3*Blur; y < y_start; ++y)  // initialize row
      {
        sum += der; der += dif;    // accumulate pixel blur
        p = src(x, y+Blur2, z); // pixel value
        put(p, y+3*Blur);         // buffer pixel, min buffer size: 3*Blur
        dif += p;             // accumulate differences {+1,
        if (y+2*Blur >= y_start) dif -= 3*get(y+2*Blur);// -3,
        if (y+Blur >= y_start) dif += 3*get(y+Blur);    //     +3, (-1)}
      }
      for (; y < y_end; ++y)
      {                                   // inner vertical blur loop
        sum += der; der += dif;    // accumulate pixel blur
        pset(x, y, z, (int)(sum/Weight)); // set blurred pixel
        p = src(x, y+Blur2, z); // next pixel
        put(p, y+3*Blur);         // buffer pixel
        dif += p+3*(get(y+Blur)-get(y+2*Blur))-get(y);  // {+1,-3,+3,-1}
      } // y
    } // x

    for (y = y_start; y < y_end; ++y)     // horizontal blur for every column
    {
      double dif = 0., der = 0., sum = 0.;
      for (x = x_start-3*Blur; x < x_start; ++x)  // initialize column
      {
        sum += der; der += dif;     // accumulate pixel blur
        p = pget(x+Blur2, y, z); // pixel value
        put(p, x+3*Blur);        // buffer pixel, min buffer size: 3*Blur
        dif += p;            // accumulate differences {+1,
        if (x+2*Blur >= x_start) dif -= 3*get(x+2*Blur); // -3,
        if (x+Blur >= x_start) dif += 3*get(x+Blur);     //     +3, (-1)}
      }
      for (; x < x_end; ++x)
      {                                 // inner horizontal blur loop
        sum += der; der += dif;     // accumulate pixel blur
        p = (int)(sum/Weight)-src(x, y, z);  // local contrast
        p = (2*ctl(1)*p - ctl(2)*(abs(p-th)-abs(p+th)+2*p))/200; // blur - sharpen
        pset(x, y, z, src(x, y, z)+p);
        p = pget(x+Blur2, y, z); // next pixel
        put(p, x+3*Blur);         // buffer pixel
        dif += p+3*(get(x+Blur)-get(x+2*Blur))-get(x);  // {+1,-3,+3,-1}
      } // x
    } // y
  } // color plane
  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```

3.2.8. Down-sampled Gaussian blur example

Fast Gaussina blur example by down-sampling the image.

```
%ffp

Title      :"downsampleGauss"
Author     :"Alois Zingl"
Version    :"August 2008"
Copyright  :"© 2008  GPL"
Category   :"easy.Filter"
Filename   :"downsampleGauss.8bf"
URL        :"http://free.pages.at/easyfilter/gauss.html"
Description:"Fast Gaussian blur filter by downsampling."

ctl(0): "Blur (pixels)", divisor=10, range=(0,500), page=100, gamma=200

OnFilterStart:
{
  double s = (doingProxy?(ctl(0)+scaleFactor/2)/scaleFactor:ctl(0))/7.4+0.9; // sigma
  isTileable = !doingProxy;
  needPadding = s*sqrt(log(256.0)/log(2.0)); // 8-bit resolution radius

  i0 = max((int)s/4, 1); // downsample scale
  s = -i0*i0/(s*s); // calc Gaussian bell shape
  i1 = -32768;  // weight
  for (i = 0; i <= needPadding/i0; i++)
  {
    put((int)(32768.0*exp(i*i*s)), i);
    i1 += get(i)*2;
  }
  bandWidth = 100+4*needPadding+i0;
  // no allocArray here since X,x_end,.. are not updated yet after changes
  return false;
}

ForEveryTile:
{
  int x, y, z, Blur = needPadding/i0-1;  // blur radius
  int xe = (x_end-x_start)/i0;
  int ye = (y_end-y_start)/i0;

  if (getArrayDim(0,0)==0)
    allocArrayPad(0, xe, ye, 2, 4, Blur+1); // downsampled image buffer

  for (z=0; z < planesWithoutAlpha; ++z)
  {
    if (updateProgress(y_start+(y_end-y_start)*z/Z, Y)) abort();

    // box downsampling
    for (y = -Blur; y < ye+Blur; ++y)
      for (x = -Blur; x < xe+Blur; ++x)
      {
        int xi, yi, s = (i0*i0)>>1;
        for (yi = 0; yi < i0; ++yi)
          for (xi = 0; xi < i0; ++xi)
            s += src(x*i0+xi+x_start, y*i0+y_start+yi, z);
        putArray(0, x, y, 0, s/(i0*i0));
      }

    // vertical blur...
    for (x = -Blur; x <= xe+Blur; ++x)
      for (y = -1; y <= ye; ++y)
      {
        double sum = 0;
        for (i = y-Blur; i <= y+Blur; ++i)
          sum += getArray(0,x,i,0)*get(abs(i-y)); // pixel * weight
        putArray(0, x, y, 1, (int)(sum/i1));
      }

    // horizontal blur...
    for (y = -1; y <= ye; ++y)
      for (x = -1; x <= xe; ++x)
      {
        double sum = 0;
        for (i = x-Blur; i <= x+Blur; ++i)
          sum += getArray(0,i,y,1)*get(abs(i-x)); // pixel * weight
        putArray(0, x, y, 0, (int)(sum/i1));
      }

    // linear upsampling
    for (y = -1; y <= ye; ++y)
      for (x = -1; x <= xe; ++x)
      {
        int xi, yi, i = i0>>1;
        int a = getArray(0, x, y, 0),   b = getArray(0, x+1, y, 0)-a;
        int c = getArray(0, x, y+1, 0), d = getArray(0, x+1, y+1, 0)-c;
        for (yi = 0; yi < i0; ++yi)
          for (xi = 0; xi < i0; ++xi)
            pset(x*i0+xi+x_start+i, y*i0+y_start+yi+i, z,
                 ((b*xi+a*i0)*(i0-yi)+(d*xi+c*i0)*yi+2*i*i)/(i0*i0));
      }
  }
  return true;  //Done!
}

OnFilterEnd:
{
  updateProgress(0, 1);
  freeArray(0);
  return false;
}
```

3.2.9. Recursive Gaussian blur example

An implementation by Tom Fiddaman (slightly improved) of the recursive Gaussian blur according to Young et al. [4].

```
%ffp

// Recursive gaussian demo code
// implemented by Tom Fiddaman, www.sd3.info/pf828
// from the article:
//    Title: Recursive implementation of the Gaussian filter
//    Authors: I.T. Young, L.J. van Vliet
//    in: Signal Processing, vol. 44, no. 2, 1995, 139-151.
// http://www.ph.tn.tudelft.nl/~lucas/publications/1995/SP95TYLV/SP95TYLV.html

// changes by Alois Zingl:
// continues boundary condition added instead of reflected boundaries
//    (the boarder pixel is copied beyond the boarder)
// the image is processed in tiles

Category: "easy.filter"
Title:    "Recursive Gaussian"
Version:  "1.0"
Filename: "recursivegauss.8bf"
Author:   "Tom Fiddaman, Alois Zingl"
URL:      "http://free.pages.at/easyfilter/gauss.html"
Description:"Recursive implementation of the Gaussian filter."

ctl(0): "Blur (pixel)"

OnFilterstart:
{
  isTileable = !doingProxy; // split large images
  needPadding = 4*(doingProxy?(ctl(0)+scaleFactor/2)/scaleFactor:ctl(0));
  bandWidth = 100+4*needPadding;
  return false;
}

ForEveryTile:
{
  int radius = needPadding;
  double b3 = max(radius-4, radius*2/3)/4.0, wn, wn1, wn2, wn3;
  double b0 = 1.57825 + b3*(2.44413 + b3*(1.4281 + 0.422205*b3));
  double b1 = b3*(2.44413 + b3*(2.85619 + 1.26661*b3))/b0;
  double b2 = (-1.4281 - 1.26661*b3)*b3*b3/b0;
  b3 = 0.422205*b3*b3*b3/b0;
  b0 = 1.0 - (b1+b2+b3);

  for (z = 0; z < Z; z++)  // loop through all channels
  {
    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress((1-doingProxy)*y_start*Z*2+(y_end-y_start)*(2*z+1),
                       (doingProxy?y_end-y_start:Y)*Z*2))
       abort();

    // DO COLUMNS

    for (x = x_start; x < x_end; x++)       // loop through all rows
    {
      y = y_start-radius;
      wn1 = wn2 = wn3 = src(x,y,z); // initialize value

      while (++y < y_start)  // initialize row
      {
        wn = b0*src(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }

      while (++y < y_end)   // real forward pass
      {
        wn = b0*src(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        pset(x,y,z, (int)wn);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }

      while (++y < y_end+radius)  // post process forward
      {
        wn = b0*src(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        put((int)wn, y);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }
      wn1 = wn2 = wn3 = src(x,y-1,z); // initialize values

      while (--y > y_end)   // initialize backward
      {
        wn = b0*get(y) + b1*wn1+b2*wn2+b3*wn3;
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }

      while (y-- > y_start)   // real backward pass
      {
        wn = b0*pget(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        pset(x,y,z, (int)wn);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }
    } // for x

    //Update progress bar and cancel if ESC key was pressed
    if (updateProgress((1-doingProxy)*y_start*Z*2+(y_end-y_start)*(2*z+2),
                       (doingProxy?y_end-y_start:Y)*Z*2))
       abort();

    // DO ROWS (a bit simpler since rows aren't split in tiles)

    for (y = y_start; y < y_end; y++)   // loop through all columns
    {
      int p = pget(x_end-1,y,z);   // remember boarder pixel
      wn1 = wn2 = wn3 = pget(x_start,y,z);

      for (x = x_start; x < x_end; x++)  // real forward pass
      {
        wn = b0*pget(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        pset(x,y,z, (int)wn);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }

      for (; x < x_end+radius; x++)  // post process forward
      {
        wn = b0*p + b1*wn1+b2*wn2+b3*wn3;
        put((int)wn, x);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }
      wn1 = wn2 = wn3 = p; // initialize values

      while (--x > x_end)   // initialize backward
      {
        wn = b0*get(x) + b1*wn1+b2*wn2+b3*wn3;
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }

      while (x-- > x_start)  // real backward pass
      {
        wn = b0*pget(x,y,z) + b1*wn1+b2*wn2+b3*wn3;
        pset(x,y,z, (int)wn);
        wn3 = wn2; wn2 = wn1; wn1 = wn;
      }
    } // for y
  } // for z

  //Stop processing and apply the effect
  return true;
}

OnFilterEnd:
{
  updateProgress(0, 1);
  return false;
}
```