Algorithms for Linear Scale-Space

Rein van den Boomgaard
University of Amsterdam

January 2003

Linear Scale-Space

Linear scale-space is constructed by embedding an image L_0 into a one parameter family of images $L(\cdot, s)$ that satisfy the *diffusion equation*:

$$\partial_t L = \nabla^2 L.$$

with initial condition $L(\cdot, 0) = L_0$. The solution at 'time' t is given by the Gaussian convolution:

$$L(\cdot, t) = L * G^{\sqrt{2t}}$$

where

$$G^s(\mathbf{x}) = \frac{1}{(s\sqrt{2\pi})^D} \exp(-\frac{\|\mathbf{x}\|^2}{2s^2}).$$

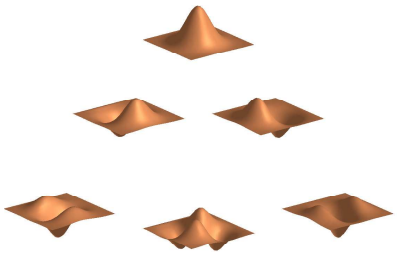
Note that the scale s is not equal to the 'diffusion time' t, instead we have $s = \sqrt{2t}$.

Gaussian Image Derivatives

In *computational* vision we replace the mathematical derivatives by *Gaussian derivatives*:

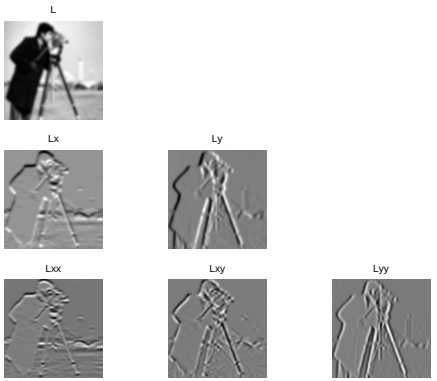
$$\partial L \longrightarrow \partial^s L = \partial(L * G^s) = L * \partial G^s$$

I.e. taking a derivative is replaced by a convolution with the derivative of the Gaussian kernel.



The Gaussian Derivative Kernels. From left to right, top to bottom: G^s, $\partial_x G^s$, $\partial_y G^s$, $\partial_{xx} G^s$, $\partial_{xy} G^s$, $\partial_{yy} G^s$.

Gaussian Image Derivatives



The x-coordinate runs from top to bottom, and the y-coordinate runs from left to right.

The image 'two-jet', i.e. all the derivatives up to order 2, captures (a lot of) the local image structure.

Convolution Integral

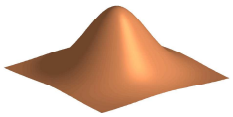
What we would like to calculate is:

$$(L * \partial G^s)(\mathbf{x}) = \int_\Omega L(\mathbf{x}-\mathbf{y})\partial G^s(\mathbf{y})d\mathbf{y}$$

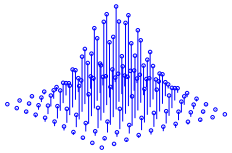
Observe that

- the function L is not known, all we have are 'some' samples,
- the kernel ∂G^s is of infinite spatial extend.
- the samples are only within a finite subset of the infinite domain of the Gaussian function: at the border of the image the kernel is bound to be not entirely within the image domain and

Convolution Sum



The Gaussian Function



The Sampled Gaussian Function

The convolution integral:

$$(L * W)(\mathbf{x}) = \int_\Omega L(\mathbf{x}-\mathbf{y})W(\mathbf{y})d\mathbf{y}$$

is most often approximated with a convolution sum:

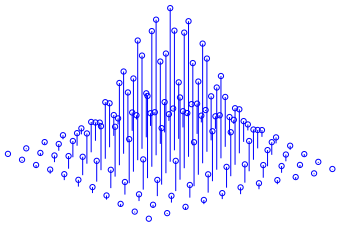
$$(\mathsf{L} \star \mathsf{W})(\mathbf{k}) = \sum_{\mathbf{l}\in\Omega} \mathsf{L}(\mathbf{k}-\mathbf{l})\mathsf{W}(\mathbf{l})$$

where L is the sampled version of L and W is the sampled version of W.

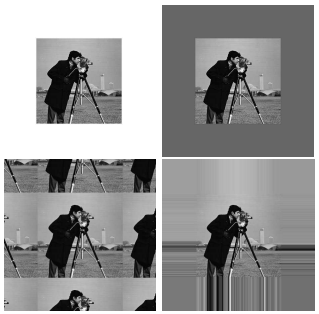
For a Gaussian kernel $W = G^s$, this sampling strategy works fine for scales $s \geq 1$ but not for scales $s < 1$.

Finite Impulse Response

- The discrete (sampled) Gaussian kernel is spatially truncated and only the values $G(k)$ for $|k| \leq N$ are used in the convolution sum.
- The value N is most often taken to be proportional to the scale s. Common choices are $N = 2s$ or $N = 3s$.
- The value α in $N = \alpha s$ depends on the order of differentiation. Higher order differentiating kernels require larger values of α (and thus N).



The Border Problem



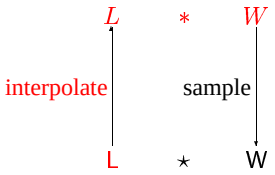
Consider the convolution sum:

$$(\mathsf{L} \star \mathsf{W})(k) = \sum_{l=-N}^{N} \mathsf{L}(k-l)\mathsf{W}(l)$$

In case $k - l \notin [1, S]$ (where S is the size of the 1D image) the following conventions can be found in practice:

- **Zero padding.** We set $\mathsf{L}(m) = 0$ for $m \notin [1, S]$.
- **Periodic Continuation.** We set $\mathsf{L}(m + S) = \mathsf{L}(m)$.
- **Replication.** We set $\mathsf{L}(m)$ for $m \notin [1, S]$ to $\mathsf{L}(1)$ for $m < 1$ and $\mathsf{L}(S)$ for $m > S$.

Convolution and Convolution Sums



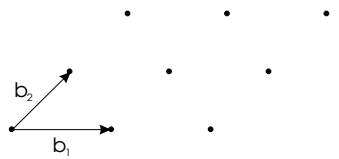
In this section we consider the more general problem of approximating the convolution integral:

$$(L * W)(\mathbf{x}) = \int_{\Omega} L(\mathbf{x}-\mathbf{y})W(\mathbf{y})d\mathbf{y}$$

where we have only been given samples of L on a regular grid.

Instead of *sampling the kernel* W to arrive at a discrete convolution sum, we will *interpolate the image function* given its samples in order to calculate the convolution integral.

Sampling



The sampling grid.

The *sampling grid* Λ is generated by the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_D$ collected in the matrix $\mathbf{B} = \begin{pmatrix}\mathbf{b}_1 & \cdots & \mathbf{b}_D\end{pmatrix}$:

$$\Lambda = \{\mathbf{Bk} \mid \mathbf{k} \in \mathbb{Z}^D\}$$

Sampling the function L results in the discrete function L:

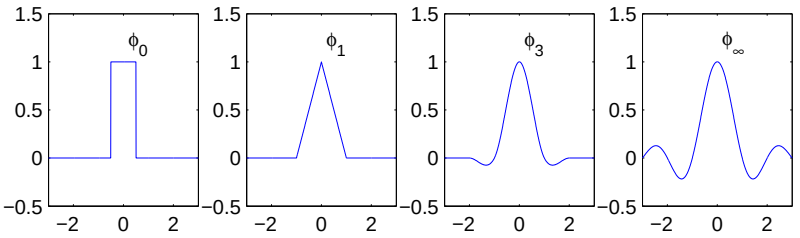
$$\mathsf{L}(\mathbf{k}) = \mathcal{S}_B(L)(\mathbf{k}) = L(\mathbf{Bk})$$

Interpolation

Given a sampled image L we can approximate L using an *interpolation* technique. We consider only interpolation methods of the form:

$$\mathcal{I}_{B,\phi}(\mathsf{L})(x) = \sum_k \phi(x - Bk)\mathsf{L}(k)$$

where ϕ is the *interpolating kernel*.



Approximating the Convolution Integral

The convolution $L * W$ given only the samples L of L is approximated using the interpolation approximation of L:

$$(L * W)(x) \approx (\mathcal{I}_{B,\phi}\mathsf{L} * W)(x)$$

Substituting the interpolation we obtain:

$$\begin{aligned}(\mathcal{I}_{B,\phi}\mathsf{L} * W)(x) &= \int_{\mathbb{R}^D} \left(\sum_k \phi(x - y - Bk)\mathsf{L}(k)\right) W(y)dy \\ &= \sum_k \mathsf{L}(k) \int_{\mathbb{R}^D} \phi(x - y - Bk)W(y)dy \\ &= \sum_k \mathsf{L}(k)\,(\phi * W)\,(x - Bk)\end{aligned}$$

Note that this expression allows us to approximate the convolution in any position in the image domain, even at ‘subpixel’ positions.

Approximating the Convolution Integral

Often we want to store the convolution as a sampled image as well. Sampling the convolution in the same points as the original image we obtain:

$$\begin{aligned}(\mathcal{I}_{B,\phi}\mathsf{L} * W)(Bl) &= \sum_k \mathsf{L}(k)\,(\phi * W)\,(Bl - Bk) \\ &= \sum_k \mathsf{L}(k)\mathcal{S}_B(\phi * W)(l-k) \\ &= (\mathsf{L} \star \mathcal{S}_B(\phi * W))\,(l)\end{aligned}$$

Because $\mathcal{I}_{B,\phi}\mathsf{L} * W$ is the approximation of $L * W$ we can write:

$$\mathcal{S}_B(L * W) \approx \mathcal{S}_B(L) \star \mathcal{S}_B(\phi * W)$$

Approximating the Convolution Integral

The convolution integral is approximated with a convolution sum:

$$\mathcal{S}_B(L * W) \approx \mathcal{S}_B(L) \star \mathcal{S}_B(\phi * W)$$

We thus find that:

- the convolution integral is approximated by a convolution sum of a sampled image and a sampled kernel.
- the kernel to be sampled is *not* the convolution kernel W but the convolution $\phi * W$.
- the accuracy of the approximation is determined by the accuracy of the interpolation.

'Large-scale' Gaussian Convolutions

For $s > 1$ we have that:

$$\mathcal{S}_B(\phi * \partial G^s) \approx \mathcal{S}_B(\partial G^s)$$

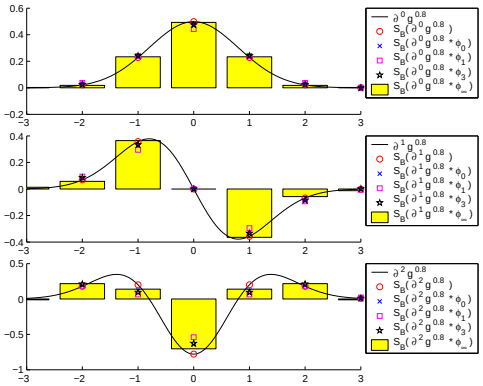
and thus

$$\mathcal{S}_B(L * \partial G^s) \approx \mathcal{S}_B(L) \star \mathcal{S}_B(\partial G^s)$$

showing that *for larger scales the convolution integral is approximated by a convolution sum using the sampled Gaussian (derivative) kernel.*

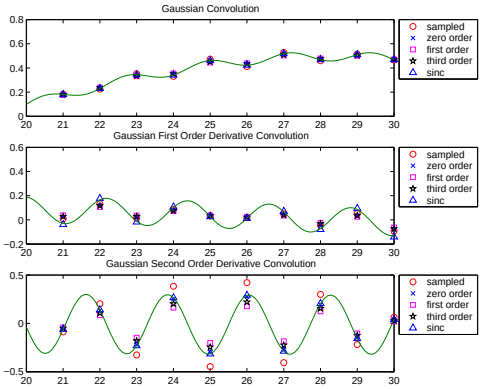
This is true for all the interpolation schemes shown before (ϕ_0, ϕ_1, ϕ_3 and ϕ_∞).

'Small-scale' Gaussian Convolutions



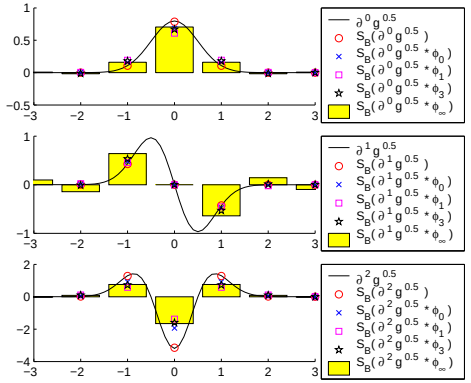
At scale $s = 0.8$ the sampling of the 'interpolated kernel' already differs from the non-interpolated kernel.

‘Small-scale’ Gaussian Convolutions



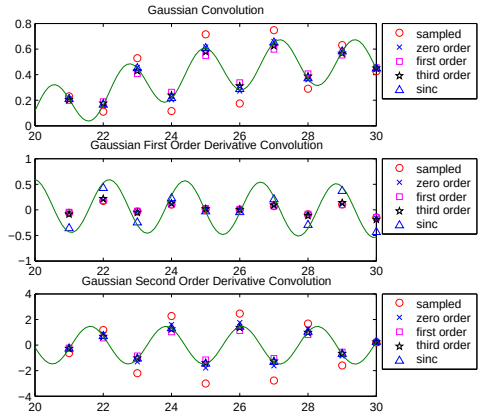
At scale $s = 0.8$ the convolution results using an ‘interpolated kernel’ are slightly better then the results obtained with the non-interpolated kernel.
Especially the Gaussian derivative convolutions are more accurate.

‘Small-scale’ Gaussian Convolutions



At scale $s = 0.5$ the sampling of the ‘interpolated kernel’ differs from the non-interpolated kernel.

‘Small-scale’ Gaussian Convolutions



At scale $s = 0.5$ the convolution results using an ‘interpolated kernel’ are better then the results obtained with the non-interpolated kernel.
Especially the Gaussian derivative convolutions are more accurate.

Algorithms for Gaussian convolutions

We distinguish three important algorithms for image convolutions:

1. **FIR spatial algorithms** (Finite Impulse Response)

   The classical straightforward implementation of the convolution summation.

2. **IIR spatial algorithms** (Infinite Impulse Response)

   A ‘recursive’ filter based on signal processing theory. The resulting filter is an approximation of the Gaussian derivative convolution (and thus there are several variants). The advantage is that the time complexity of the resulting filter is not dependent of the scale.

3. **Frequency domain algorithms** (via the Fourier transform)

   Convolution in the spatial domain is pointwise multiplication in the frequency domain.

Finite Impulse Response Algorithms

- Any Gaussian (derivative) convolution is separable.
- Where should we truncate the Gaussian function (to obtain a FIR filter)?
- Should we normalize for missing density? (and then how?)
- How to calculate the Gaussian derivative kernels (for any order)?
- What is a fast image convolution algorithm ?

Separability

The Gaussian derivative kernel:

$$\begin{aligned}\partial_{\mathbf{k}} G^s(\mathbf{x}) &= \partial_{x_1^{k_1}} \cdots \partial_{x_D^{k_D}} G^s(x_1) \cdots G^s(x_D) \\ &= \partial_{x_1^{k_1}} G^s(x_1) \cdots \partial_{x_D^{k_D}} G^s(x_D)\end{aligned}$$

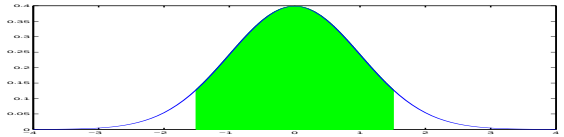
is of the form $W(\mathbf{x}) = W_1(x_1) \cdots W_D(x_D)$ and thus is separable, i.e.

$$L * \partial_{\mathbf{k}} G^s = L * \partial_{x_1^{k_1}} G_1^s * \cdots * \partial_{x_D^{k_D}} G_D^s$$

From now on we thus may focus on the Gaussian derivative convolution of one-dimensional functions.

Truncate the Gaussian Kernel

- By truncation we 'loose' density

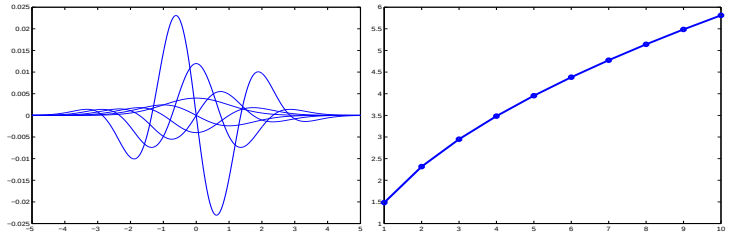


- For the (zero order derivative) Gaussian kernel truncating the Gaussian kernel at $N = \alpha s$ and straightforward sampling leads to the following densities:

α	$N = \lceil \alpha \rceil$	$\int_{-N-\frac{1}{2}}^{N+\frac{1}{2}} G^1(x)dx$
1	1	0.87
2	2	0.98
3	3	0.99
5	5	1.00

Truncate the Gaussian Derivative Kernel

- The Gaussian derivative kernel should be truncated at larger values of x.



- At least a value of s from the last zero crossing in the Gaussian derivative is a nice rule of the thumb (unfortunately there is only an approximative equation for the last zerocrossing):

$$N \geq 2\,s\,\sqrt{n+1-1.15\sqrt[3]{n+1}}$$

Normalization of the Truncated Kernels

Because the Gaussian kernel is truncated the kernel weights do not sum up to one, i.e. the mean value of an image after convolution is lower then the mean value of the original image.

This can be corrected for with a multiplicative normalization of the kernel

$$\overline{G^s}(i) = \frac{G^s(i)}{\sum_{j=-N}^{N} G^s(j)}$$

as can be often found in practical implementations.

The advantage is that the Gaussian convolution preserves the density, a disadvantage is that the accuracy of the Gaussian convolution can be less then for the not normalized kernel.

Normalization of the Truncated Kernels

- Normalization of the Gaussian kernel to preserve the total density is equivalent to the normalization to correctly filter a constant function.
- For Gaussian derivative kernels an equivalent normalization can be carried out by

$$\overline{\partial_k G^s}(i) = \frac{\partial_k G^s(i)}{\sum_{j=-N}^{N} \frac{j^k}{k!} \partial_k G^s(j)}$$

  Note that $h(i) = i^k/k!$ is the canonical function that has k-th order derivative equal to one.
- The first order derivative kernel is thus normalized in such a way that a linear ramp function results in the correct first order derivative.
- A disadvantage is that the normalization only leads to correct results for the canonical functions, not necessarily for other functions.

Gaussian derivative kernels of any order

- The Gaussian derivative of order n is given by:

$$\partial_n G^s(x) = \left(\frac{-1}{s\sqrt{2}}\right)^n H_n\left(\frac{x}{s\sqrt{2}}\right) G^s(x)$$

  where H_n is the Hermite function.
- We thus have to implement two functions: G^s and H_n.
  - The Gaussian function is trivial.
  - For the Hermite function we can use the following recursive definition:

$$\begin{aligned} H_0(x) &= 1 \\ H_1(x) &= 2\,x \\ H_{n+1}(x) &= 2\,x\,H_n(x) - 2\,(n-1)\,H_{n-1}(x) \end{aligned}$$

A bit of Matlab

```
function r = gD( image, scale, ox, oy )
% Gaussian Derivative Convolutions
K = ceil( 4 * scale );
x = -K:K;
Gs = exp( - x.^2 / (2*scale^2) );
Gsx = gDerivative( x, scale, ox );
Gsy = gDerivative( x, scale, oy );

r = imfilter( imfilter( image, Gsx', 'conv', 'replicate' ), ...
                        Gsy, 'conv', 'replicate' );
```

A bit of Matlab

```
function r = gDerivative( x, s, order )
Gs = 1/(s*sqrt(2*pi))*exp( - x.^2 / (2*s^2) );
r = (-1/(s*sqrt(2))).^order*HermiteH(order, x/(s*sqrt(2)) ) .* Gs;


function h = HermiteH( order, x )
switch order
case 0
  h = 0*x + 1;
case 1
  h = 2 * x;
otherwise
  h = 2 * x .* HermiteH( order-1, x ) - ...
      2 * (order-1) * HermiteH( order-2, x );
end
```


Fast Convolution Algorithms

- Accessing slow memory is the bottleneck on a general purpose CPU. Cache behaviour is important.
- Keeping the CPU pipelines filled with data and calculating is important.
- If you need to, you can program with the MMX instruction set of your (Intel) CPU.

- or, you can use approximations of Gaussian derivative convolutions utilizing recursive algorithms, or you can use a Fourier transform based method.

Frequency Domain Algorithms

Let $\hat{\mathsf{L}}$ be the DFT (discrete Fourier transform) of a sampled image L and also let $\hat{\mathsf{W}}$ be the DFT of W, then

$$\begin{array}{ccccc} \mathsf{R} & = & \mathsf{L} & * & \mathsf{W} \\ \Big\uparrow DFT & & \Big\uparrow DFT & & \Big\uparrow DFT \\ \hat{\mathsf{R}} & = & \hat{\mathsf{L}} & \times & \hat{\mathsf{W}} \end{array}$$

i.e. convolution in the spatial domain becomes multiplication in the frequency domain.

- In the change of representation (from the spatial to the frequency domain) we have defined the way the 'border' is treated: *periodic continuation*. This way of dealing with the border gives rise to very peculiar border effects.
- For Gaussian (derivative) kernels we may sample its Fourier transform.

Complexity of frequency domain algorithms

Consider a 1D convolution $\mathsf{L} \star \mathsf{W}$:

- The calculation of the DFT $\hat{\mathsf{L}}$ has computational complexity of order $N \log N$ where N is the number of samples in L
- Let W be a sampled kernel (with M values different from zero) then the spatial implementation of the convolution is of the order NM.
- Thus in case $M > \log N$ the frequency domain implementation has lower computational complexity.
- For a Gaussian derivative kernel M is proportional to the scale. So for large scale the frequency domain algorithms may be advantageous.

Frequency vs Spatial Algorithms

- Small scales:
  - Spatial algorithm is efficient, but not accurate.
  - FFT algorithms is accurate for not too small scales, but not too efficient (compared with spatial algorithm).
- Large scales:
  - Spatial algorithm is inefficient, but accurate.
  - FFT algorithms is not so accurate (sampling a very small scale Gauss in the Fourier domain), but very efficient (compared with spatial algorithm).