Generalized Computation of Gaussian Derivatives Using ITK

Release 1.00

Iván Macía[1]

October 5, 2007

[1]VICOMTech, Paseo Mikeletegi 57, San Sebastián (Spain)
imacia@vicomtech.org

Abstract

Computation of local image derivatives is an important operation in many image processing tasks that involve feature detection and extraction, such as edges, corners or more complicated features. However, derivative computation in discrete images is an ill-posed problem and derivative operators without any prior smoothing are known to enhance noise. Here we present a new convolution operator, the `GaussianDerivativeOperator`, that allows to calculate locally Gaussian derivatives of N order. Furthermore, we present some useful classes and examples that make use of this new operator.

Contents

1 Principles of Gaussian Derivative Computation **2**
1.1 Scale-space and Gaussian Derivatives . . . 2
1.2 Gaussian Derivatives for Discrete Signals . . . 3

2 Implementation of Discrete Gaussian Derivative Kernel **3**
2.1 Implementation Details for the Analytical Approach . . . 4

3 Image Functions and Filters Using Gaussian Derivative Operator **5**
3.1 DiscreteGaussianDerivativeImageFunction . . . 5
3.2 DiscreteHessianGaussianImageFunction . . . 6
3.3 DiscreteGaussianDerivativeImageFilter . . . 7

4 Sample Programs **7**
4.1 GaussianDerivativeOperatorCoefficients . . . 7
4.2 GaussianImageDerivatives . . . 7
4.3 GaussianImageDerivatives3D . . . 8
4.4 DiscreteGaussianDerivativeImageFunction . . . 8
4.5 DiscreteHessianGaussianImageFunction . . . 9
4.6 ImageHessianEigenvalues3D . . . 9

1 Principles of Gaussian Derivative Computation

1.1 Scale-space and Gaussian Derivatives

Computation of local image *derivatives* is an important operation in many image processing tasks that involve feature detection and extraction, such as edges, corners or more complicated features.

Usually derivatives are calculated as *convolutions* of the image with some predefined *kernels* or *operators* that try to approximate the derivatives (such as Sobel, Prewitt, Roberts or Laplacian operators). However, derivative computation in discrete images is an ill-posed problem and derivative operators without any prior smoothing are known to enhance noise.

On the other hand, real world entities are only meaningful at a certain range of scales. The *scale-space* representation of an image, first introduced by [1, 2], is a methodology to obtain a meaningful representation of an image at multiples scales.

Given a continuous signal in n-dimensions $f : \mathbb{R}^n \to \mathbb{R}$, its scale-space representation $L : \mathbb{R}^n \mathrm{X} \mathbb{R}_+ \to \mathbb{R}$ is defined by the convolution operation

$$L(\mathbf{x},t) = g(\mathbf{x},t) * f(\mathbf{x}) \tag{1}$$

where t is the *scale-space parameter* and $g : \mathbb{R}^n \mathrm{X} \mathbb{R}_+ \backslash \{0\} \to \mathbb{R}$ is the *continuous Gaussian kernel* given by

$$g(\mathbf{x},t) = \frac{1}{(\sqrt{2\pi t})^n} e^{-\frac{||\mathbf{x}||^2}{2t}} \tag{2}$$

Here the scale-parameter is $t = \sigma^2$ where σ is the standard deviation of the Gaussian function. For simplicity and without loss of generality we will focus only on 2D signals with a continuous Gaussian kernel

$$g(x,y,t) = \frac{1}{2\pi t} e^{-\frac{x^2+y^2}{2t}} \tag{3}$$

Spatial derivatives of this scale-space representation can be defined at different levels of scale as

$$L_{x^i y^j}(x,y,t) = \partial_{x^i y^j} L(x,y,t) \tag{4}$$

Due to the special properties of the Gaussian kernel, the derivative operation commutes with the Gaussian kernel satisfying

$$\partial_{x^i y^j} L = \partial_{x^i y^j}(g * f) = (\partial_{x^i y^j} g) * f = g * (\partial_{x^i y^j} f) \tag{5}$$

The direct implication is that we can pre-compute the convolution of the Gaussian operator with the derivative operator in order to speed-up the computations. This way we will convolve our signal with a single kernel that represent the *Gaussian derivative*.

1.2 Gaussian Derivatives for Discrete Signals

A problem arises when we want to apply these operators to our real images which are *discrete* signals. In the case of the Gaussian kernel, a direct discretization or *sampled Gaussian kernel* could be an initial approximation but, as pointed out by Lindeberg [5], scale-space conditions are not guaranteed to be preserved and may lead to undesired effects, specially when calculating higher order derivatives in noisy data.

Several approaches have been taken in order to address this problem (see for example [6] for a brief description of some of them). For example, Deriche [3] uses recursive filters in order to approximate Gaussian derivatives. This approach is currently implemented in ITK for calculating scale-space derivatives (see for example `itk::RecursiveGaussianImageFilter`). The implementation is fast and accurate enough for some cases, specially when performing operations over the whole input image. However, the current implementation is not suitable to calculate derivatives only at some specific *candidate* points of the input image (`itk::RecursiveGaussianImageFilter` requires all the input image) and is less precise than other approaches, specially at finer scales.

When calculating several local derivatives at the same time (*N-jet*) or when we are interested only in some set of candidate points of the image, a good approach is to use convolution with the *discrete analogue of the Gaussian kernel* and *small-support difference operators*.

Lindeberg [4] showed that the natural way to construct a scale-space representation for discrete signal was by convolution with the *discrete analogue of the Gaussian kernel* given by

$$T(n,t) = e^{-t} I_n(t) \tag{6}$$

where I_n are the modified Bessel functions of integer order.

Lindeberg demonstrates that, with this approach, scale-space properties hold after discretization. For example operators such as finite difference operators commute with the discrete Gaussian kernel and convolution is separable in each direction. This way, a family of kernels that constitute a *discrete analogue of the continuous Gaussian derivatives* are derived. Currently ITK provides an implementation for the discrete Gaussian kernel (see `itk::GaussianOperator`) but not for its derivatives.

2 Implementation of Discrete Gaussian Derivative Kernel

The class `itk::GaussianDerivativeOperator` provides an implementation of a *separable discrete Gaussian derivative kernel*. The interface of the class is very similar to `itk::GaussianOperator` with the main difference that it provides the method `SetOrder()` that allows to specify the order of the Gaussian derivative stored in the new member `m_Order`. Derivatives can be calculated for virtually any order.

The implementation uses two approaches to generate the filter coefficients in `GenerateCoefficients()`. The first approach calculates the coefficients of a polynomial that is multiplied by the Gaussian kernel, thus calculating the final derivative Gaussian kernel the same way we would calculate *analitically* the derivatives of a Gaussian function. This is possible because the discrete Gaussian kernel represents an analogue of the continuous Gaussian kernel and the linear properties are kept.

The second approach calculates the convolution of an `itk::GaussianOperator` with a `itk::DerivativeOperator` to set up the final Gaussian derivative operator. This mode is set when the flag `m_UseDerivativeOperator` is set to `On`.

The analytical approach results in increased accuracy and speed with respect to convolution with the `itk::DerivativeOperator`, which can lead to numerical problems with large values of σ. However, in the analytical approach it takes more time to set up the kernel, for example when the scale changes. In general, it is better to calculate the desired derivatives at all points at a given scale before moving to the next scale.

Like the class `itk::GaussianOperator`, this new class provides the methods `SetMaximumKernelWidth()` and `SetMaximumError()` that allow to specify the desired precision of the Gaussian kernel.

Image spacing can be taken into account when calculating Gaussian derivatives by using `SetSpacing()` to set the spacing in the direction of the kernel (by default 1.0). The operator automatically adjusts the variance to be `m_Variance /= m_Spacing * m_Spacing` and uses also the spacing in the derivative calculation. Thus, at the time of creating image functions that use this operator and have into account image spacing, the variance must not be adjusted. Instead we will simply call `SetSpacing()` for the kernel in each direction.

Figure 1 illustrates plots of the Gaussian derivative kernels of different order obtained using the analytical approach with varying $\sigma = 2.0$ and maximum error. It can be seen that the kernel width augments with σ and decreases with the maximum error. For large values of σ there is a considerable attenuation factor of high order derivatives, which can be compensated by normalization of derivatives across scale-space by calling the method `SetNormalizeAcrossScale()` (see Figure 1 e) and f))

Figure 2 shows a comparison between the analytical and convolution-based kernel computation. It can be seen that for small values of σ (Figure 2 a) and 2 b)) the kernels are very similar but for large σ and high order of derivatives (Figure 2 a) and 2 b)) the convolution implementation gets numerically unstable.

The previous experiments correspond to the provided example `GaussianDerivativeOperatorCoefficients`.

2.1 Implementation Details for the Analytical Approach

The one-dimensional normalized continuous Gaussian kernel is given by

$$g(x,t) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}} \tag{7}$$

and its first order derivative

$$\frac{\partial}{\partial x} g(x,t) = \frac{-x}{t\sqrt{2\pi t}} e^{-\frac{x^2}{2t}} = \frac{-x}{t} g(x,t) \tag{8}$$

We can recursively calculate higher order derivatives

$$\frac{\partial g}{\partial x^2} = \frac{\partial}{\partial x}\left(\frac{-x}{t} g\right) = \frac{\partial}{\partial x}\left(\frac{-x}{t}\right) g - \frac{x}{t}\frac{\partial g}{\partial x} = \frac{-g}{t} + \frac{x^2}{t^2} g = \frac{x^2 - t}{t^2} g \tag{9}$$

$$\frac{\partial g}{\partial x^3} = \frac{\partial}{\partial x}\left(\frac{x^2 - t}{t^2} g\right) = \frac{\partial}{\partial x}\left(\frac{x^2 - t}{t^2}\right) g + \frac{x^2 - t}{t^2}\frac{\partial g}{\partial x} = \frac{2x}{t^2} g + \frac{-x^3 + tx}{t^3} g = \frac{-x^3 + 3xt}{t^3} g \tag{10}$$

$$\frac{\partial g}{\partial x^4} = \frac{\partial}{\partial x}\left(\frac{-x^3+3xt}{t^3}g\right) = \frac{-3x^2+3t}{t^3}g + \frac{x^4-3x^2t}{t^4}g = \frac{x^4-6x^2t+3t^2}{t^4}g \tag{11}$$

$$\frac{\partial g}{\partial x^5} = \frac{4x^3-12xt}{t^4}g + \frac{-x^5+6tx^3-3t^2x}{t^5}g = \frac{-x^5+10tx^3-15t^2x+3t^3}{t^5}g \tag{12}$$

and so on. In the current implementation, the coefficients of the polynomial in the numerator are calculated and stored in the temporal variable `polyCoeffs`, where the polynomial index coincides with the order of the x variable. For example for the third order derivative we have

```
polyCoeffs.push_back(0);  // coefficient for x^0
polyCoeffs.push_back(3);  // coefficient for x^1
polyCoeffs.push_back(0);  // coefficient for x^2
polyCoeffs.push_back(-1); // coefficient for x^3
```

We can also calculate the order of t in the derivative expressions. Let N be the order of the numerator polynomial with respect to x and n the current coefficient. Then

- The order of t in the numerator polynomial is always $(N-n)/2$.
- The order of t in the denominator coincides with the order of the polynomial.

For low order derivatives coefficients are stored directly in the `polyCoeffs` vector but for higher order (> 3) these are computed recursively.

3 Image Functions and Filters Using Gaussian Derivative Operator

Once the `itk::GaussianDerivativeOperator` is implemented, many image functions and filters can be defined to calculate N-dimensional derivatives or other interesting features. Here we present two new *image functions* and a *filter* that make use of this new operator.

3.1 DiscreteGaussianDerivativeImageFunction

The class `itk::DiscreteGaussianDerivativeImageFunction` is used to calculate local Gaussian derivatives of any order in N-dimensional images. It distinguishes from the already existing `itk::GaussianDerivativeImageFunction` in that the measurement is calculated by means of a convolution with a `itk::GaussianDerivativeOperator` instead of making use of a spatial function.

The class provides the methods `SetMaximumKernelWidth()` and `SetMaximumError()` to specify the precision of the operators used internally.

The method `SetOrder()` is used to specify the order of derivatives in each direction. For example, if we want to calculate a second order partial derivative in y for a volumetric image we would type

```
int order[3];
order[0] = 0;
```

```
order[1] = 2;
order[2] = 0;
imageFunction->SetOrder( order );
```

or, since the vector for storing order of derivatives is by default initialized to zero (only Gaussian smoothing for each direction)

```
imageFunction->SetOrder( 1, 2 );
```

Internally the class calculates a N-dimensional kernel that will be convolved with the neighborhood of the current image index. This is done in the method `RecomputeGaussianKernel()`. The 1D operators used to calculate this final kernel are of type `itk::GaussianDerivativeOperator` and are kept in the member `m_OperatorArray`. There is an operator for each direction with the specified order for the derivative.

To calculate the final N-dimensional kernel a small pipeline is configured. First a small image is created which has the same radius as the kernels. The center point is initialized to 1 in order to have an impulse at the center. Then, this small image is convolved N times with each of the 1D Gaussian derivative operators using an `itk::NeighborhoodOperatorImageFilter`. This is done in the following lines of code

```
for( unsigned int direction = 0; direction<itkGetStaticConstMacro(ImageDimension2); ++direction )
{
  convolutionFilter->SetInput( kernelImage );
  convolutionFilter->SetOperator( m_OperatorArray[direction] );
  convolutionFilter->Update();
  kernelImage = convolutionFilter->GetOutput();
  kernelImage->DisconnectPipeline();
}
```

Finally the resulting output image is copied in the final N-dimensional operator stored in `m_DerivativeKernel`. This operator is convolved with the neighborhood of the local index in `EvaluateAtIndex()` whenever we want to calculate the Gaussian derivatives at a point.

3.2 DiscreteHessianGaussianImageFunction

The class `DiscreteHessianGaussianImageFuncion` is used to calculate the local Hessian matrix at points of N-dimensional images. For a volumetric image, the Hessian matrix is given by

$$\begin{bmatrix} \partial I_{xx} & \partial I_{xy} & \partial I_{xz} \\ \partial I_{yx} & \partial I_{yy} & \partial I_{yz} \\ \partial I_{zx} & \partial I_{zy} & \partial I_{zz} \end{bmatrix} \tag{13}$$

which is a symmetric matrix so only 6 components need to be calculated in this case.

The behavior of this class is very similar to `itk::DiscreteGaussianDerivativeImageFunction` but the implementation of the method `RecomputeGaussianKernel()` is a bit more complicated. The reason is we are working with a number of $Nx3$ kernels due to the fact that, for each direction, zero, first and second order derivatives kernels must be stored to calculate the components of the Hessian matrix. As before, these are stored in the member `m_OperatorArray`.

3.3 DiscreteGaussianDerivativeImageFilter

The class `itk::DiscreteGaussianDerivativeImageFilter` calculates Gaussian derivatives for whole images in the form of an `itk::ImageToImageFilter`. It is very similar in conception and implementation to the already existing class `itk::DiscreteGaussianImageFilter`. It uses internally the class `itk::GaussianDerivativeOperator`. The order of the derivatives to be calculated is set using the method `SetOrder()`.

4 Sample Programs

Here we present a set of programs used to demonstrate the capabilities of the newly implemented classes.

4.1 GaussianDerivativeOperatorCoefficients

The test program `GaussianDerivativeOperatorCoefficients` can be used to generate Gaussian derivative kernels and dump them to file. The parameters are the following (parameters in parenthesis are optional):

- `outputTextFile` : output ASCII text file name
- `order` : order of the derivatives
- `sigma` : standard deviation σ of the Gaussian derivative
- `(normalize)` : set to 1 to use normalization across scale-space, otherwise 0
- `(max_error)` : maximum error for Gaussian kernel calculation (typical values 0.02 or less)
- `(use_derivative_operator)` : set to 1 if we want to use the convolution implementation

Example :

```
GaussianDerivativeOperatorCoefficients GaussianOp_O1_s2_0_err0_005.txt 1 2.0 0 0.005 0
```

The results are displayed in Figures 1 and 2 and commented in Section 2.

4.2 GaussianImageDerivatives

The test program `GaussianImageDerivatives` uses the class `itk::NeighborhoodOperatorImageFilter` to perform convolution with two operators of type `itk::GaussianDerivativeOperator`, one for each direction *x* and *y*, over a Gaussian image created with `itk::GaussianImageSource`.

The parameters are the following

- `outputSourceImage` : the generated Gaussian image with $\sigma = 5$
- `outputFilteredImage` : the resulting Derivative of Gaussian image
- `orderX` : order of derivative in *x* direction

- `orderY` : order of derivative in *y* direction
- `operator_sigma` : value of σ (scale) for the derivative operator
- `(maximum_error)` : maximum error for gaussian kernel calculation (typical values 0.02 or less)
- `(maximum_kernel_width)` : if specified it sets a maximum width for the calculated kernel

For example, to calculate a second order derivative in *x* and first order derivative in *y*, with $\sigma = 3.0$ and $max\_error = 0.005$ we would set the following parameters

```
GaussianImageDerivatives sourceImage.png DOG_O21_s3_0_005.png 2 1 3.0 0.005
```

Figure 3 shows the result of using this program to calculate the N-jet (up to order 5) of a 2D Gaussian function.

4.3 GaussianImageDerivatives3D

This is the 3D version of `GaussianImageDerivatives`. There is a new argument that specifies the derivative order for direction *z*.

Example:

```
GaussianImageDerivatives3D sourceImage.png DOG_O222_s3_0_005.png 2
2 2 3.0 0.005
```

Figure 4 illustrates a volume rendering of the partial derivative $\partial_{x^2y^2z^2}G$, where G is a 3D Gaussian function.

4.4 DiscreteGaussianDerivativeImageFunction

The test program `DiscreteGaussianDerivativeImageFunction` uses the class `itk::DiscreteGaussianDerivativeImageFunction` to locally calculate derivatives at non-zero points of a source Gaussian image. This could be easily replaced by any other image, for example using `itk::ImageFileReader`.

The parameters are the following

- `outputFileName` : the resulting output image file name
- `orderX` : order of derivative in *x* direction
- `orderY` : order of derivative in *y* direction
- `sigma` : value of σ (scale) for the derivative operator
- `(maximum_error)` : maximum error for gaussian kernel calculation (typical values 0.02 or less)
- `(maximum_kernel_width)` : if specified it sets a maximum width for the calculated kernel

For example, to calculate a first order derivative in *x* and first order derivative in *y*, with $\sigma = 3.0$ and $max\_error = 0.005$ we would set the following parameters

```
DiscreteGaussianDerivativeImageFunction DOGImageFunc_O11_s3_0_005.png 1 1 3.0 0.005
```

4.5 DiscreteHessianGaussianImageFunction

The test program `DiscreteHessianGaussianImageFunction` uses the class `itk::DiscreteHessianGaussianImageFunction` to locally calculate Hessian matrix at non-zero points of a source Gaussian image. A total of $D*(D+1)/2$ output images are written on output, one for each Hessian matrix component.

Parameters

- `outputFileName` : resulting output image. The program renames this file to `filename_x.ext` where x is the current Hessian components.
- `sigma` : value of σ (scale) for the derivative operator
- `(maximum_error)` : maximum error for gaussian kernel calculation (typical values 0.02 or less)
- `(maximum_kernel_width)` : if specified it sets a maximum width for the calculated kernel

For example, to calculate the Hessian matrix of the Gaussian function with $\sigma = 3.0$ and $max\_error = 0.005$ we would use

```
DiscreteHessianGaussianImageFunction HessianImageFunc_s3_0_005.png 3.0 0.005
```

4.6 ImageHessianEigenvalues3D

The test program `ImageHessianEigenvalues3D` uses the class `itk::DiscreteHessianGaussianImageFunction` to locally calculate Hessian matrix eigenvalues at non-zero points of a 3D input image. The six Hessian component images and three corresponding eigenvalue images are written on output.

This example is for demonstrative purposes but, in general, it is not a good idea to use these image functions to calculate values over the whole input image. For that purpose it is recommended to use faster approaches, such as recursive Gaussian filters.

Parameters

- `inputFileName` : input 3D image
- `sigma` : value of σ (scale) for the derivative operator
- `(maximum_error)` : maximum error for gaussian kernel calculation (typical values 0.02 or less)
- `(maximum_kernel_width)` : if specified sets a maximum width for the calculated kernel

For example, to calculate a first order derivative in x and first order derivative in y, with $\sigma = 3.0$ and $max\_error = 0.005$ we would set the following parameters

```
ImageHessianEigenvalues3D inputImage.mhd 3.0 0.005
```

References

[1] Witkin A.P. Scale-space filtering. In *Proc. 8th Int. Joint Conf. Art. Intell.*, pages 1019–1022, Kalsruhe, Germany, 1983. 1.1

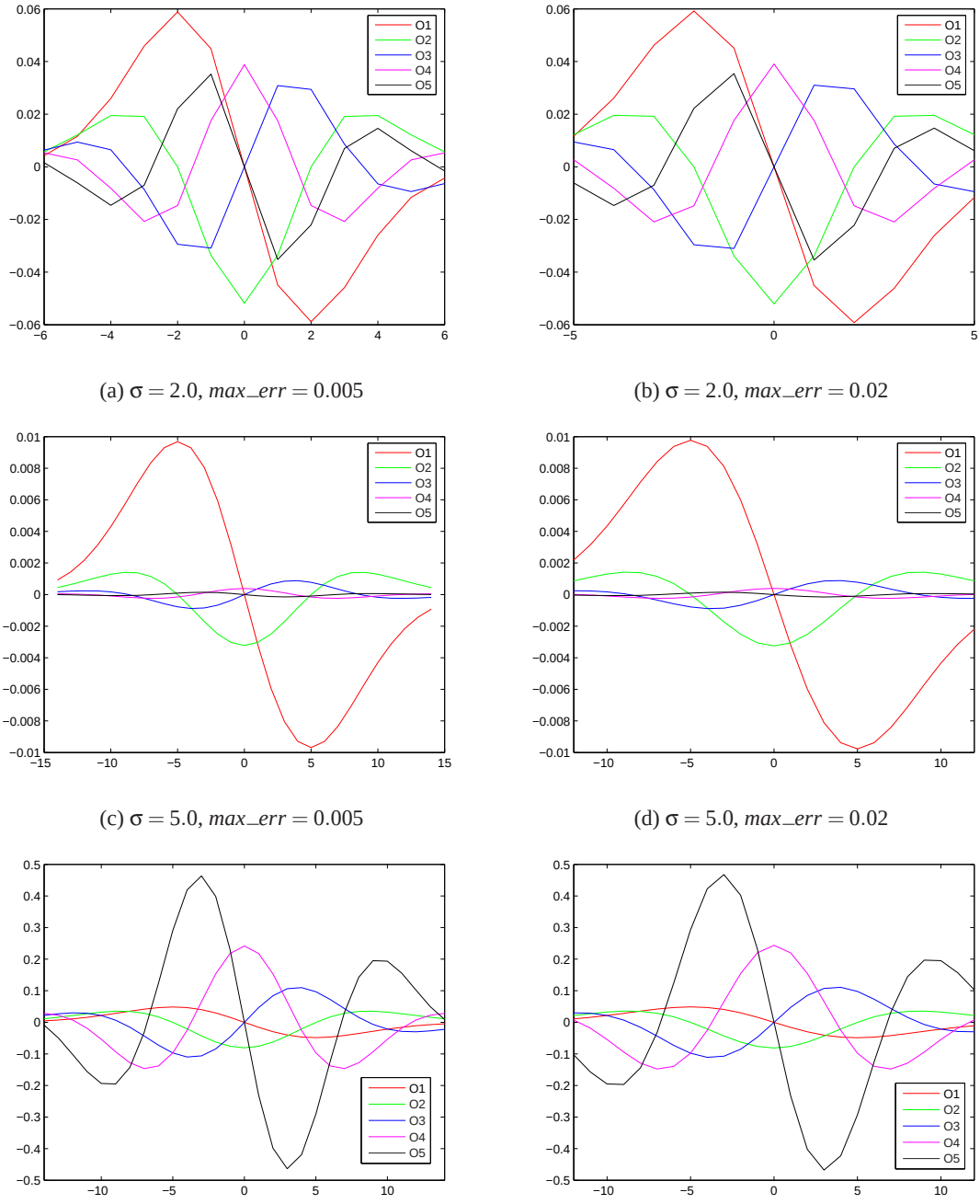
[2] Koenderink J.J. The structure of images. *Biol. Cyb.*, 50:363–370, 1984. 1.1

[3] Deriche R. Recursively implementing the gaussian and its derivatives. Technical Report 1893, Unite de recherche INRIA Sophia-Antipolis, avril 1993. Research Repport. 1.2

[4] Lindeberg T. Scale-space for discrete signals. *IEEE Trans. Pat. Anal. Mach. Intel.*, 12(3):234–254, 1990. 1.2

[5] Lindeberg T. Discrete derivative approximations with scale-space properties : A basis for low-level feature extraction. *J. of Mathematical Imaging and Vision*, 3(4):349–376, 1993. 1.2

[6] Wikipedia. Scale-space implementation. . 1.2



(a) $\sigma = 2.0$, $max\_err = 0.005$

(b) $\sigma = 2.0$, $max\_err = 0.02$

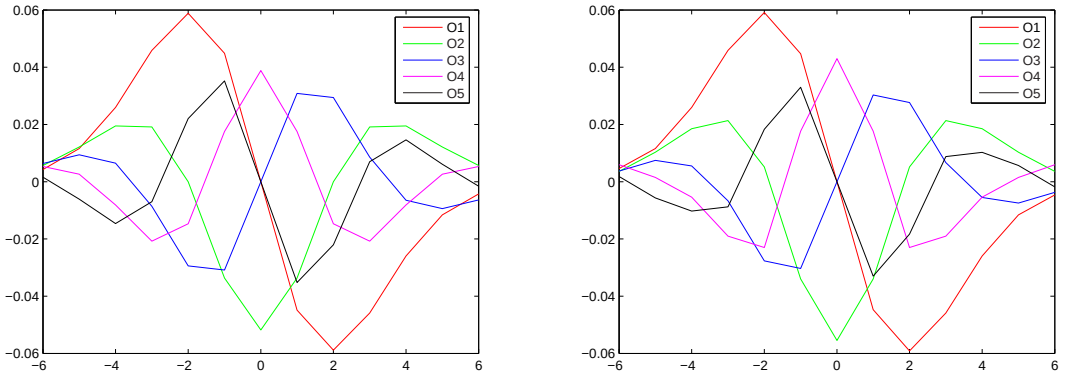
(c) $\sigma = 5.0$, $max\_err = 0.005$

(d) $\sigma = 5.0$, $max\_err = 0.02$

(e) $\sigma = 5.0$, $max\_err = 0.005$, normalized

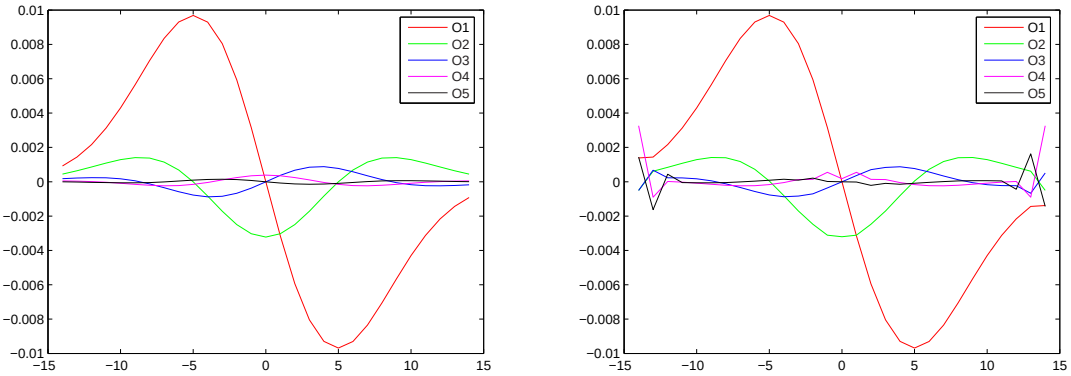
(f) $\sigma = 5.0$, $max\_err = 0.02$, normalized

Figure 1: Resulting Gaussian derivative kernels for different values of σ and maximum error.



(a) $\sigma = 2.0$, $max\_err = 0.005$, analytical

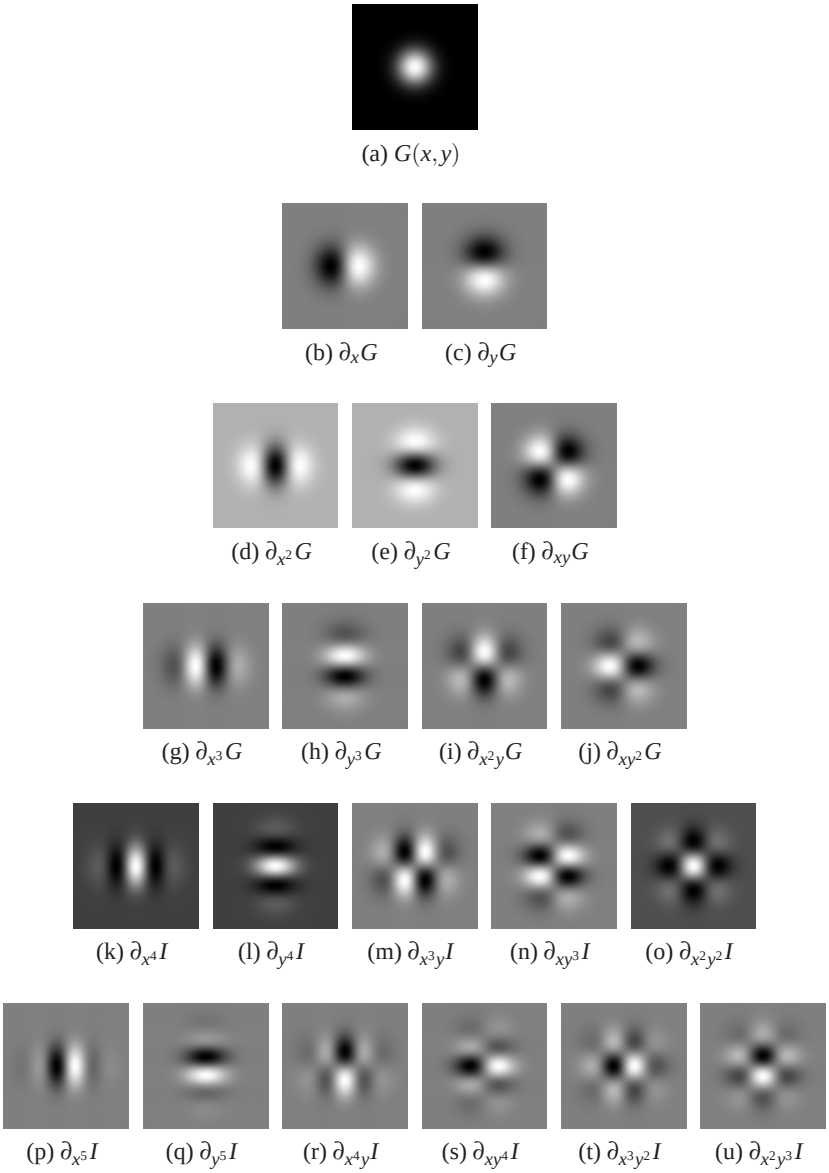
(b) $\sigma = 2.0$, $max\_err = 0.005$, convolution



(c) $\sigma = 5.0$, $max\_err = 0.005$, analytical

(d) $\sigma = 5.0$, $max\_err = 0.005$, convolution

Figure 2: Resulting Gaussian derivative kernels for different values of σ and maximum error with derivatives calculated 2a, 2c analytically and 2b, 2d by convolution.



(a) $G(x,y)$

(b) $\partial_x G$ (c) $\partial_y G$

(d) $\partial_{x^2} G$ (e) $\partial_{y^2} G$ (f) $\partial_{xy} G$

(g) $\partial_{x^3} G$ (h) $\partial_{y^3} G$ (i) $\partial_{x^2y} G$ (j) $\partial_{xy^2} G$

(k) $\partial_{x^4} I$ (l) $\partial_{y^4} I$ (m) $\partial_{x^3y} I$ (n) $\partial_{xy^3} I$ (o) $\partial_{x^2y^2} I$

(p) $\partial_{x^5} I$ (q) $\partial_{y^5} I$ (r) $\partial_{x^4y} I$ (s) $\partial_{xy^4} I$ (t) $\partial_{x^3y^2} I$ (u) $\partial_{x^2y^3} I$

Figure 3: Gaussian derivatives up to order 5 (N-jet) of the 2D Gaussian function depicted in 3a with $\sigma = 5$. Parameters for derivative calculation: $\sigma = 3.0$, $max\_error = 0.001$, normalized.

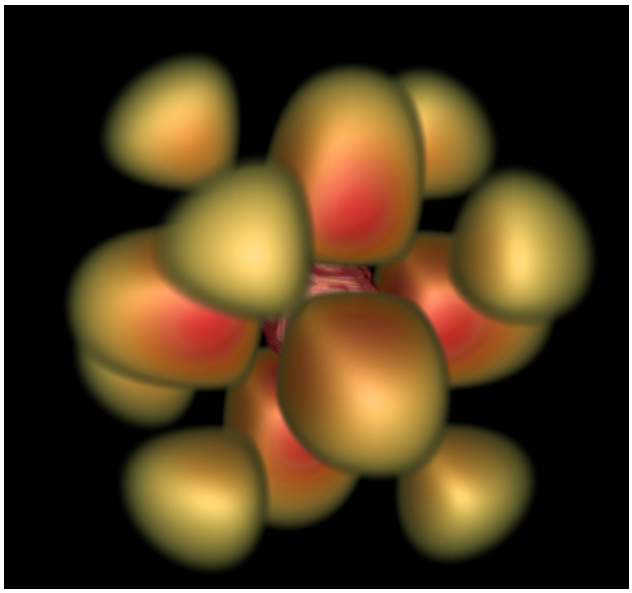


Figure 4: Volume render of derivative of Gaussian $\partial_{x^2y^2z^2}G$, $\sigma(G) = 5.0$, $\sigma_{scale} = 3.0$, $max\_error = 0.001$