Gaussian Derivative Package

Markus van Almsick
Technische Universiteit Eindhoven

Version 2.5.7

■ Prefix

```
TimeStamp["MathVisionTools`GaussianDerivative`"] = {2006, 6, 1, 17, 11, 0}
```

The preamble of the package containing all kinds of remarks about authorship, contence, and copyright.

```
(*:Name: MathVisionTools`GaussianDerivative` *)

(*:Author: Bart M.ter Haar Romeny Ph.D.and Markus van Almsick*)

(*:Email address: B.terHaarRomeny@tue.nl, M.v.Almsick@tue.nl*)

(*:Context: MathVisionTools`GaussianDerivative`*)

(*:Package Version: 2.5 .7 *)

(*:Mathematica Version: 5.0 *)

(*:Copyright: Copyright 2001-2006, Technische Universiteit Eindhoven*)

(*:Title: Gaussian Kernel, Gaussian Derivative, and Gaussian Filter *)

(*:Summary: This package implements the Gaussian derivative for Scale
    Space theory. "Front-End Vision and Multiscale Image Analysis"
    by Bart M.ter Haar Romeny, Springer, 2003.*)

(*:Keywords: Gaussian, Gaussian derivatives, multiscale,
 scale-space, computer vision, differential operator *)

(*:Requirements: None *)

(*:Source: "Front-End Vision and Multiscale Image Analysis",
 by Bart M.ter Haar Romeny, Springer,2003 *)

(*:History: Version 1.0 by Bart M.ter Haar Romeny and Markus van Almsick,
 October 2001
            Version 1.1 by Bart M.ter Haar Romeny and Markus van Almsick, May 2002
            Version 1.2 by Bart
  M.ter Haar Romeny and Markus van Almsick, Sept 2002
            Version 2.0 by Bart M.ter Haar Romeny and Markus van Almsick, Dec 2002
            Version 2.1 by Bart M.ter Haar Romeny and Markus van Almsick, Jan 2003
            Version 2.2 by Bart M.ter Haar Romeny and Markus van Almsick, Nov 2003
            Version 2.4 by Bart M.ter Haar Romeny and Markus van Almsick, Jan 2004
            Version 2.5 by Bart M.ter
  Haar Romeny and Markus van Almsick, April 2006 *)

(*:Limitations: *)

(*:Discussion: *)

(*:To do:  Recursive Implementation, Directional, Parallel *)
```

■ Begin Package

Declaring the package and unlocking all symbols defined in the code.

```
Utilities`Notation`AutoLoadNotationPalette = False;

BeginPackage[
 "MathVisionTools`GaussianDerivative`",
 "MathVisionTools`Common`MathVisionToolsCommon`",
 "MathVisionTools`Convolve`",
 "MathVisionTools`ImageFrame`",
 "Utilities`Notation`"
]

Unprotect[
 G,
 NGaussianKernel,
 GaussianKernel,
 AngularGaussianKernel,
 GaussianKernelRange,
 NGaussianDKernel,
 GaussianDKernel,
 GaussianD,
 GaussianDerivative,
 GaussianDerivativeAt,
 Method,
 KernelRange,
 ImageBoundary,
 Convolve,
 Fourier,
 Cyclic,
 Truncate,
 Reflective,
 Constant
]
```

■ Online Help and Options

DefaultScaleParameter::usage =

"The default linear scale parameter t = $\frac{\sigma^2}{2}$. Thus the width or standard deviation of the Gaussian kernel is 1."

G::usage = "G_t[x] is a short input notation for GaussianKernel[t, x]. $G_t^{(n)}$[x] is a short input notation for GaussianDKernel[t,n,x]."

GaussianKernel::usage =

"GaussianKernel[t,x] renders the gaussian kernel numerically and symbolically. Note that t is half the variance, which is the square of the Gaussian kernel width σ."

AngularGaussianKernel::usage =

"AngularGaussianKernel[t,α] renders the angular gaussian kernel numerically and symbolically. The angular kernel is the Greens function of the diffusion equation on the interval [-π,π] with periodic boundary conditions. Note, that t is half the variance, which is the square of the Gaussian kernel width σ."

NGaussianKernel::usage =

"NGaussianKernel[t,x] renders the gaussian kernel numerically for Real t and x with compiled code. Note that t is half the variance, which is the square of the Gaussian kernel width σ."

GaussianDKernel::usage =

"GaussianDKernel[t,n,x]. Note that t is half the variance, which is the square of the Gaussian kernel width σ. n is the derivative order."

NGaussianDKernel::usage =

"NGaussianDKernel[n][t,x] renders the n^{th} derivative of the gaussian kernel numerically for non-negative Integer n and Real t and x with compiled code. Note that t is half the variance, which is the square of the Gaussian kernel width σ."

GaussianD::usage =

"GaussianD[img,{x,t,n}] performs the n^{th} Gaussian partial derivation on image img with respect to x. The Gaussian kernel variance is 2t. GaussianD[img,{x, t_x,n_x},{y,t_y,n_y},...] is the n_x^{th} Gaussian partial derivation with respect to x, the n_y^{th} Gaussian partial derivation with respect to y, and so on."

GaussianDerivative::usage =

"GaussianDerivative[{t_x,n_x},{t_y,n_y},...][img] repesents the (n_x,n_y)th Gaussian derivative of image img. The Gaussian kernel variance is 2t_x in x- and 2t_y in y-direction."

GaussianDerivativeAt::usage =

"GaussianDerivativeAt[{x,t_x,n_x},{y,t_y,n_y},...][img] repesents the (n_x,n_y)th Gaussian derivative of image img at the specified location (x,y) in scale space. The Gaussian kernel variance is 2t_x in x- and 2t_y in y-direction. To determine Gaussian derivatives at

```
   more than one location enter GaussianDerivativeAt[{{x1,tx1,nx1},
   {y1,ty1,ny1},...},{{x2,tx2,nx2},{y2,ty2,ny2},...},...][img]."

KernelRange::usage =
 "KernelRange is an option of GaussianD and GaussianDerivative. It
   determines the default range of the kernel used in the convolution."

(* ImageBoundary-Option *)
If[
 Head[ImageBoundary::usage] === MessageName,
 ImageBoundary::usage =
  "ImageBoundary is an option of GaussianD and GaussianDerivative
    working with the Convolve method. ImageBoundary determines the
    handling of an image frame in a convolution.",
 If[
  Not[StringMatchQ[ImageBoundary::usage, "*GaussianDerivative*"]],
  ImageBoundary::usage =
   ImageBoundary::usage <> "\nImageBoundary is also an option of GaussianD and
      GaussianDerivative working with the Convolve method. ImageBoundary
      determines the handling of an image frame in a convolution."
 ]
]

(* Convolve-Option *)
If[
 Head[Convolve::usage] === MessageName,
 Convolve::usage =
  "Is a value for the GaussianDerivative option Method. It calls upon the
    ListConvolve command to perform the convolution of the Gaussian derivation.",
 If[
  Not[StringMatchQ[Convolve::usage, "*GaussianDerivative*"]],
  Convolve::usage =
   Convolve::usage <> "\nIt is also a value for the GaussianDerivative
      option Method. It calls upon the ListConvolve command to
      perform the convolution of the Gaussian derivation."
 ]
]

(* Fourier-Option *)
If[
 Not[StringMatchQ[Fourier::usage, "*GaussianDerivative*"]],
 Fourier::usage = Fourier::usage <> "\nFourier is also an option value
     for the GaussianDerivative Method. The convolution of the Gaussian
     derivative is performed via multiplication in the Fourier domain."
]

(* Cyclic-Option *)
If[
 Head[Cyclic::usage] === MessageName,
 Cyclic::usage = "Is a value for the GaussianDerivative
    option ImageBoundary. It calls upon the ListConvolve command
    to perform a cyclic convolution to maintain the image size.",
 If[
  Not[StringMatchQ[Cyclic::usage, "*GaussianDerivative*"]],
  Cyclic::usage =
```

```
   Cyclic::usage <> "\nIt is also a value for the GaussianDerivative option
      ImageBoundary. It calls upon the ListConvolve command to perform a cyclic
      convolution to maintain the image size."
 ]
]

(* Truncate-Option *)
If[
 Head[Truncate::usage] === MessageName,
 Truncate::usage = "Is a value for the GaussianDerivative option
    ImageBoundary. It calls upon the ListConvolve command to truncate the
    incomplete convolution of the Gaussian kernel at the image boundary.",
 If[
  Not[StringMatchQ[Truncate::usage, "*GaussianDerivative*"]],
  Truncate::usage =
   Truncate::usage <> "\nIt is also a value for the GaussianDerivative option
      ImageBoundary. It calls upon the ListConvolve command to truncate the
      incomplete convolution of the Gaussian kernel at the image boundary."
 ]
]

(* Reflective-Option *)
If[
 Head[Reflective::usage] === MessageName,
 Reflective::usage =
  "Is a value for the GaussianDerivative option ImageBoundary. It calls upon
    the ListConvolve command to ap-/prepend the reflected image on each image
    boundary before performing the convolution of the Gaussian kernel.",
 If[
  Not[StringMatchQ[Reflective::usage, "*GaussianDerivative*"]],
  Reflective::usage =
   Reflective::usage <> "\nIt is also a value for the GaussianDerivative
      option ImageBoundary. It calls upon the ListConvolve command
      to ap-/prepend the reflected image on each image boundary
      before performing the convolution of the Gaussian kernel."
 ]
]

(* Constant-Option *)
If[
 Not[StringMatchQ[Constant::usage, "*GaussianDerivative*"]],
 Constant::usage =
  Constant::usage <> "\nIt is also a value for the GaussianDerivative
     option ImageBoundary. It calls upon the ListConvolve to
     extend the value of each boundary pixel to infinity."
]

Options[GaussianDerivative] = {KernelRange → Automatic, Method → Convolve,
   ImageBoundary → Cyclic, WorkingPrecision → Log[10, 2^16.]};

Options[GaussianDerivativeAt] = {KernelRange → Automatic,
   ImageBoundary → Cyclic, WorkingPrecision → Log[10, 2^16.]};
```

■ Package Code

Starting the private context of the package.

```
Begin["`Private`"]
```

Open the palette for Gaussian Derivatives if started from a local kernel.

```
If[Not[$Remote], NotebookOpen[First[FileNames["GaussianDerivativePalette.nb",
     {ToFileName[{$UserAddOnsDirectory, "Applications", "MathVisionTools",
        "FrontEnd", "Palettes"}], ToFileName[{$AddOnsDirectory,
        "Applications", "MathVisionTools", "FrontEnd", "Palettes"}]}]]]];
```

■ Gaussian Kernel

NOTE: we use the linear scale parameter $t = \frac{2}{\tau}\sigma^2$ to denote the width of the Gaussian bell shape kernel. σ is the standard deviation and σ^2 the variance of the kernel.

```
(* Gaussian Kernel *)

τ = 4; (* global factor for the scaling parameter t, default is 4 (or 1) *)

DefaultScaleParameter = 2 / τ;
```

■ Numeric Implementation

Implementing a compiled version of the Gaussian kernel to speed up evaluation by a factor 3.

```
(* Numeric Implementation *)

NGaussianKernel = Compile[{{t, _Real}, {x, _Real}}, Exp[-x^2 / (τ t)] / Sqrt[Pi (τ t)]];

GaussianKernel[0] = 1 / Sqrt[Pi (τ t)];

GaussianKernel[x_?NumberQ] := NGaussianKernel[DefaultScaleParameter, x]

GaussianKernel[0 | 0., x_?NumberQ] = DiracDelta[x];

GaussianKernel[t_?NumberQ, x_?NumberQ] := NGaussianKernel[t, x]
```

■ Symbolic Implementation

```
(* Symbolic Implementation *)

GaussianKernel[0, {x__}] = DiracDelta[x];
GaussianKernel[0, x_] = DiracDelta[x];

GaussianKernel[t_: DefaultScaleParameter, x_] = Exp[-x^2 / (τ t)] / Sqrt[Pi (τ t)];

GaussianKernel /: Derivative[0, n_][GaussianKernel] := GaussianDKernel[n, #1, #2] &
```

■ Multidimensional Implementation

```
(* Multidimensional Implementation *)

GaussianKernel[x_List] := Apply[Times, Map[GaussianKernel, x]]

GaussianKernel[t_List, x_List] :=
 Inner[GaussianKernel, t, x, Times] /; Dimensions[t] == Dimensions[x]

GaussianKernel[t_, x_List] := Apply[Times, Map[GaussianKernel[t, #] &, x]]
```

■ GaussianKernel Input Notation

```
Notation[G[x__] ⟺ GaussianKernel[{x__}]]

(* GaussianKernel Input Notation *)

MakeExpression[RowBox[{lhs___, "G", "[", x__, "]", rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianKernel", "[",
      RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x], "}"}],
      "]"}], rhs}], StandardForm]

MakeBoxes[GaussianKernel[{x__}], StandardForm] := RowBox[{"G", "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]

Notation[G[x_] ⟺ GaussianKernel[x_]]

(* GaussianKernel Input Notation *)

MakeExpression[RowBox[{lhs___, "G", "[", x_, "]", rhs___}], StandardForm] :=
 MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianKernel", "[", x, "]"}], rhs}], StandardForm]

MakeBoxes[GaussianKernel[x_], StandardForm] :=
 RowBox[{"G", "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G_t_[x__] ⟺ GaussianKernel[t_, {x__}]]

MakeExpression[RowBox[{lhs___, SubscriptBox["G", t_], "[", x_RowBox, "]", rhs___}],
  StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianKernel", "[", RowBox[{t, ",",
       RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x], "}"}]}]}],
      "]"}], rhs}], StandardForm]

MakeBoxes[GaussianKernel[t_, {x__}], StandardForm] :=
 RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]], "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]

Notation[G_t_[x_] ⟺ GaussianKernel[t_, x_]]
```

```
MakeExpression[RowBox[{lhs___, SubscriptBox["G", t_], "[", x_, "]", rhs___}],
  StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianKernel", "[", RowBox[{t, ",", x}], "]"}], rhs}],
  StandardForm]

MakeBoxes[GaussianKernel[t_, x_], StandardForm] :=
 RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]],
   "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G_t_[x__] ⇔ GaussianKernel[{t__}, {x__}]]

MakeExpression[RowBox[{lhs___, SubscriptBox["G", t__], "[", x__, "]", rhs___}],
  StandardForm] := MakeExpression[RowBox[{lhs, RowBox[{"GaussianKernel", "[",
      RowBox[{RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[t], "}"}],
        ",", RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x],
          "}"}]}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianKernel[{t__}, {x__}], StandardForm] := RowBox[{SubscriptBox["G",
    Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{t}, StandardForm, None]], "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]
```

■ Angular Gaussian Kernel

■ Symbolic Implementation

```
(* Symbolic Implementation *)

AngularGaussianKernel[0, φ_] = DiracDelta[φ];

AngularGaussianKernel[t_: DefaultScaleParameter, φ_] = 1/(2 π) EllipticTheta[3, φ / 2, ℯ^-t];
```

■ Gaussian Derivative Kernel

```
(* Gaussian Derivative Kernel *)
```

■ Numeric Implementation

To obtain efficient formulas for the n-th derivative of the Gaussian kernel we apply the Horner scheme to the hermite polynomials that occur when differentiating $\frac{1}{\sqrt{\pi t}}\, e^{\frac{-x^2}{t}}$.

```
<< Algebra`Horner`
```

We also apply the substitution $\frac{x^2}{t} \to$ y2 for even *n*, and $\frac{x}{\sqrt{t}} \to y$ for odd *n*.

```
EfficientGaussianKernel[n_?EvenQ] :=
 Horner[Simplify[D[GaussianKernel[t, x], {x, n}] /. x → Sqrt[y2 t]] * Sqrt[π] t^((n+1)/2)/ℯ^-y2] *
   ℯ^-y2/Sqrt[π * t^(n+1)] /. t → (τ t)
```

```
EfficientGaussianKernel[n_?OddQ] :=
 Horner[Simplify[D[GaussianKernel[t, x], {x, n}] /. x → y Sqrt[t]] * Sqrt[π] t^((n+1)/2) / E^-y^2] *
  E^-y^2 / (Sqrt[π] t^((n+1)/2)) /. t → (τ t)

EfficientGaussianKernel[3] // InputForm
```

```
(y*(12 - 8*y^2))/(E^y^2*Sqrt[Pi]*t^2)
```

The resulting formulas are placed into the body of a compiled function for the first 12 *n*.

```
(* Numeric Implementation *)

NGaussianDKernel[0] =
  Compile[{{t, _Real}, {x, _Real}}, Exp[-x^2 / (τ t)] / Sqrt[Pi (τ t)]];

NGaussianDKernel[1] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x / Sqrt[τ t]}, (-2 * y) / (E^y^2 * Sqrt[Pi] * (τ t))], {{y, _Real}}];

NGaussianDKernel[2] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x^2 / (τ t)}, (-2 + 4 * y) / (E^y * Sqrt[Pi (τ t)^3])], {{y, _Real}}];

NGaussianDKernel[3] = Compile[{{t, _Real}, {x, _Real}}, With[{y = x / Sqrt[τ t]},
    (y * (12 - 8 * y^2)) / (E^y^2 * Sqrt[Pi] * (τ t)^2)], {{y, _Real}}];

NGaussianDKernel[4] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x^2 / (τ t)}, (12 + y * (-48 + 16 * y)) / (E^y * Sqrt[Pi * (τ t)^5])], {{y, _Real}}]

NGaussianDKernel[5] = Compile[{{t, _Real}, {x, _Real}}, With[{y = x / Sqrt[τ t]},
    (y * (-120 + y^2 * (160 - 32 * y^2))) / (E^y^2 * Sqrt[Pi] * (τ t)^3)], {{y, _Real}}];

NGaussianDKernel[6] = Compile[{{t, _Real}, {x, _Real}}, With[{y = x^2 / (τ t)},
    (-120 + y * (720 + y * (-480 + 64 * y))) / (E^y * Sqrt[Pi * (τ t)^7])], {{y, _Real}}];

NGaussianDKernel[7] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x / Sqrt[τ t]}, (y * (1680 + y^2 * (-3360 + y^2 * (1344 - 128 * y^2)))) /
     (E^y^2 * Sqrt[Pi] * (τ t)^4)], {{y, _Real}}];

NGaussianDKernel[8] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x^2 / (τ t)}, (1680 + y * (-13440 + y * (13440 + y * (-3584 + 256 * y)))) /
     (E^y * Sqrt[Pi * (τ t)^9])], {{y, _Real}}];

NGaussianDKernel[9] = Compile[{{t, _Real}, {x, _Real}}, With[{y = x / Sqrt[τ t]},
    (y * (-30240 + y^2 * (80640 + y^2 * (-48384 + y^2 * (9216 - 512 * y^2))))) /
     (E^y^2 * Sqrt[Pi] * (τ t)^5)], {{y, _Real}}];

NGaussianDKernel[10] = Compile[{{t, _Real}, {x, _Real}}, With[{y = x^2 / (τ t)},
    (-30240 + y * (302400 + y * (-403200 + y * (161280 + y * (-23040 + 1024 * y))))) /
     (E^y * Sqrt[Pi * (τ t)^11])], {{y, _Real}}];

NGaussianDKernel[11] =
  Compile[{{t, _Real}, {x, _Real}}, With[{y = x / Sqrt[τ t]}, (y * (665280 +
        y^2 * (-2217600 + y^2 * (1774080 + y^2 * (-506880 + y^2 * (56320 - 2048 * y^2)))))) /
     (E^y^2 * Sqrt[Pi] * (τ t)^6)], {{y, _Real}}];

NGaussianDKernel[12] =
  Compile[{{t, _Real}, {x, _Real}}, With[{y = x^2 / (τ t)}, (665280 + y *
       (-7983360 + y * (13305600 + y * (-7096320 + y * (1520640 + y * (-135168 + 4096 * y)))))) /
     (E^y * Sqrt[Pi] * Sqrt[(τ t)^13])], {{y, _Real}}];

NGaussianDKernel[n_Integer?Positive] = Compile[{{t, _Real}, {x, _Real}},
   With[{y = x / Sqrt[τ t]},
    -HermiteH[n, y] / (Sqrt[Pi] (-Sqrt[τ t]) ^ (n + 1)) * Exp[-y^2]
   ], {{y, _Real}}];
```

Refer to the numeric implementation whenever the Gaussian kernel is called with numerical arguments.

```
GaussianDKernel[0, x_?NumberQ] := NGaussianKernel[DefaultScaleParameter, x]

GaussianDKernel[n_Integer, x_?NumberQ] :=
 NGaussianDKernel[n][DefaultScaleParameter, x] /; 1 ≤ n ≤ 12

GaussianDKernel[0 | 0., n_Integer, x_] := Derivative[n][DiracDelta][x]

GaussianDKernel[t_?NumberQ, 0, x_?NumberQ] := NGaussianKernel[t, x]

GaussianDKernel[t_?NumberQ, n_Integer, x_?NumberQ] :=
 NGaussianDKernel[n][t, x] /; 1 ≤ n ≤ 12
```

■ Symbolic Implementation

For non-numeric arguments one has to render symbolic formulas. We consider 3 different cases:
numeric order of differentiation, symbolic order of differentiation, and kernel of zero width resulting in a Dirac delta distribution.

```
(* Symbolic Implementation *)

GaussianDKernel[t_: DefaultScaleParameter, 0, x_] := GaussianKernel[t, x]

GaussianDKernel[t_: DefaultScaleParameter, n_Integer?NonNegative, x_] :=
 Simplify[-HermiteH[n, x / Sqrt[τ t]] / (Sqrt[Pi] (-Sqrt[τ t])^(n + 1))] * Exp[-x^2 / (τ t)]

GaussianDKernel[t_: DefaultScaleParameter, n_Symbol, x_] :=
 If[
  N[t] === 0.,
  Derivative[n][DiracDelta][x],
  -HermiteH[n, x / Sqrt[τ t]] / (Sqrt[Pi] (-Sqrt[τ t])^(n + 1)) * Exp[-x^2 / (τ t)]
 ]

GaussianDKernel /: Derivative[0, 0, n_][GaussianDKernel] :=
 GaussianDKernel[#1 + n, #2, #3] &
```

■ Multidimensional Implementation

```
(* Multidimensional Implementation *)

GaussianDKernel::dimfail1 =
  "The number of variables and the number of derivative indices do not match.";

GaussianDKernel::dimfail2 = "The number of variables, the number of
    derivative indices, and the number of scale parameters do not match.";

GaussianDKernel[{0 ..}, x_List] := GaussianKernel[x];

GaussianDKernel[n_List, x_List] :=
  Inner[GaussianDKernel, n, x, Times] /; Dimensions[n] == Dimensions[x];

GaussianDKernel[n_List, x_List] :=
  (Message[GaussianDKernel::dimfail1]; Throw[$Failed]) /;
   Dimensions[n] ≠ Dimensions[x];

GaussianDKernel[t_List, {0 ..}, x_List] := GaussianKernel[t, x];

GaussianDKernel[t_List, n_List, x_List] :=
  Apply[Times, Apply[GaussianDKernel, Transpose[{t, n, x}], {1}]] /;
   Dimensions[n] == Dimensions[t] == Dimensions[x];

GaussianDKernel[t_List, n_List, x_List] :=
  (Message[GaussianDKernel : dimfail2]; Throw[$Failed]) /;
   Not[Dimensions[n] == Dimensions[t] == Dimensions[x]];

GaussianDKernel[t_, {0 ..}, x_List] := GaussianKernel[t, x];

GaussianDKernel[t_, n_List, x_List] :=
  Apply[Times, Apply[GaussianDKernel[t, ##] &, Transpose[{n, x}], {1}]] /;
   Dimensions[n] == Dimensions[x];

GaussianDKernel[t_, n_List, x_List] :=
  (Message[GaussianDKernel : dimfail2]; Throw[$Failed]) /;
   Dimensions[n] ≠ Dimensions[x];
```

■ GaussianDKernel Input Notation

```
Notation[G′[x_] ⟺ GaussianDKernel[1, x_]]

(* GaussianDKernel Input Notation *)

MakeExpression[RowBox[{lhs___, SuperscriptBox["G", "′"], "[", x_, "]", rhs___}],
  StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{"1", ",", x}], "]"}], rhs}],
  StandardForm]

MakeBoxes[GaussianDKernel[1, x_], StandardForm] :=
 RowBox[{SuperscriptBox["G", "′", MultilineFunction → None],
   "[", MakeBoxes[x, StandardForm], "]"}]
```

```
Notation[G'_t_[x_] ⟺ GaussianDKernel[t_, 1, x_]]

MakeExpression[
  RowBox[{lhs___, SubsuperscriptBox["G", t_, "′"], "[", x_, "]", rhs___}],
  StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{t, ",", "1", ",", x}], "]"}],
    rhs}], StandardForm]

MakeBoxes[GaussianDKernel[t_, 1, x_], StandardForm] :=
 RowBox[{SubsuperscriptBox["G", MakeBoxes[t, StandardForm], "′",
    MultilineFunction → None], "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G''[x_] ⟺ GaussianDKernel[2, x_]]

MakeExpression[RowBox[{lhs___, SuperscriptBox["G", "′′"], "[", x_, "]", rhs___}],
  StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{"2", ",", x}], "]"}], rhs}],
  StandardForm]

MakeBoxes[GaussianDKernel[2, x_], StandardForm] :=
 RowBox[{SuperscriptBox["G", "′′", MultilineFunction → None],
   "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G''_t_[x_] ⟺ GaussianDKernel[t_, 2, x_]]

MakeExpression[
  RowBox[{lhs___, SubsuperscriptBox["G", t_, "′′"], "[", x_, "]", rhs___}],
  StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{t, ",", "2", ",", x}], "]"}],
    rhs}], StandardForm]

MakeBoxes[GaussianDKernel[t_, 2, x_], StandardForm] :=
 RowBox[{SubsuperscriptBox["G", MakeBoxes[t, StandardForm], "′′",
    MultilineFunction → None], "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G^(n_)[x_] ⟺ GaussianDKernel[n_, x_]]

MakeExpression[
  RowBox[{lhs___, SuperscriptBox["G", TagBox[RowBox[{"(", n_, ")"}], Derivative]],
    "[", x_, "]", rhs___}], StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{n, ",", x}], "]"}], rhs}],
  StandardForm]

MakeBoxes[GaussianDKernel[n_, x_], StandardForm] :=
 RowBox[{SuperscriptBox["G", TagBox[RowBox[{"(", MakeBoxes[n, StandardForm], ")"}],
     Derivative], MultilineFunction → None], "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G_t_^(n_)[x_] ⟺ GaussianDKernel[t_, n_, x_]]
```

```
MakeExpression[RowBox[
   {lhs___, SubsuperscriptBox["G", t_, TagBox[RowBox[{"(", n_, ")"}], Derivative]],
    "[", x_, "]", rhs___}], StandardForm] := MakeExpression[
  RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{t, ",", n, ",", x}], "]"}],
    rhs}], StandardForm]

MakeBoxes[GaussianDKernel[t_, n_, x_], StandardForm] :=
 RowBox[{SubsuperscriptBox["G", MakeBoxes[t, StandardForm],
    TagBox[RowBox[{"(", MakeBoxes[n, StandardForm], ")"}], Derivative],
    MultilineFunction → None], "[", MakeBoxes[x, StandardForm], "]"}]

Notation[G^(n__)[x__] ⟺ GaussianDKernel[{n__}, {x__}]]

MakeExpression[
  RowBox[{lhs___, SuperscriptBox["G", TagBox[RowBox[{"(", n__, ")"}], Derivative]],
    "[", x__, "]", rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianDKernel", "[",
      RowBox[{RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[n], "}"}],
        ",", RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x],
          "}"}]}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianDKernel[{n__}, {x__}], StandardForm] :=
 RowBox[{SuperscriptBox["G",
    TagBox[RowBox[{"(", Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{n},
        StandardForm, None], ")"}], Derivative], MultilineFunction → None], "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]

Notation[G_t^(n__)[x__] ⟺ GaussianDKernel[t_, {n__}, {x__}]]

MakeExpression[RowBox[
   {lhs___, SubsuperscriptBox["G", t_, TagBox[RowBox[{"(", n__, ")"}], Derivative]],
    "[", x__, "]", rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianDKernel", "[", RowBox[{t, ",",
        RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[n], "}"}], ",",
        RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x], "}"}]}]},
      "]"}], rhs}], StandardForm]

MakeBoxes[GaussianDKernel[t_, {n__}, {x__}], StandardForm] :=
 RowBox[{SubsuperscriptBox["G", MakeBoxes[t, StandardForm],
    TagBox[RowBox[{"(", Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{n},
        StandardForm, None], ")"}], Derivative], MultilineFunction → None], "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]

Notation[G_t__^(n__)[x__] ⟺ GaussianDKernel[{t__}, {n__}, {x__}]]
```

```
MakeExpression[RowBox[
   {lhs___, SubsuperscriptBox["G", t__, TagBox[RowBox[{"(", n__, ")"}], Derivative]],
    "[", x__, "]", rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianDKernel", "[",
      RowBox[{RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[t], "}"}],
        ",", RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[n], "}"}],
        ",", RowBox[{"{", Utilities`Notation`Private`stripSpuriousRowBox[x],
          "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianDKernel[{t__}, {n__}, {x__}], StandardForm] :=
 RowBox[{SubsuperscriptBox["G",
    Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{t}, StandardForm, None],
    TagBox[RowBox[{"(", Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{n},
        StandardForm, None], ")"}], Derivative], MultilineFunction → None], "[",
   Utilities`Notation`Private`makeHeldRowBoxOfBoxes[{x}, StandardForm, None], "]"}]
```

■ GaussianD

■ N-dimensional Gaussian derivative

```
(* N-dimensional Gaussian derivative *)
```

Summation rule:

```
GaussianD[img1_ + img2_, stn : {_Symbol | _Slot, _, _ : 0} .., opts___Rule] :=
 GaussianD[img1, stn, opts] + GaussianD[img2, stn, opts]
```

Constant factor:

```
GaussianD[c_ img_, stn : {_Symbol | _Slot, _, _ : 0} .., opts___Rule] :=
 c GaussianD[img, stn, opts] /; Apply[And, Map[FreeQ[c, First[#]] &, {stn}]]
```

Concatenation of Gaussian derivations:

```
GaussianD[
  GaussianD[img_, stn1 : {_Symbol | _Slot, _, _ : 0} .., opts1___Rule],
  stn2 : {_Symbol | _Slot, _, _ : 0} .., opts2___Rule
 ] :=
 GaussianD[
   img,
   Sequence @@ Map[
     (Cases[{stn1}, {#, __}] /. {} → 0 + Cases[{stn2}, {#, __}] /. {} → 0) &,
    Union[Map[First, {stn1}], Map[First, {stn2}]]
    ],
   Sequence @@ Union[{opts1}, {opts2}]
  ] /; Length[{stn1}] === Length[{stn2}] ∧ Map[First, {stn1}] === Map[First, {stn2}]
```

Processing a List of functions:

```
GaussianD[img_List, stn : {_Symbol | _Slot, _, _ : 0} .., opts___Rule] :=
 Map[GaussianD[#, stn, opts] &, img]
```

Gaussian derivation via Fourier transformation:

```
GaussianD[img_, stn : {_, _, _Integer : 0} .., opts___Rule] :=
 Module[
  {result, xs, ts, ns,
   ys = Table[Unique["y"], {Length[{stn}]}],
   ωs = Table[Unique["ω"], {Length[{stn}]}]},
  {xs, ts, ns} = Transpose[{stn}];
  result /;
   If[
    FreeQ[
     result = FourierTransform[img /. Thread[xs → ys], ys, ωs], FourierTransform
    ],
    {ts, ns} = Transpose[Map[Rest, {stn}]];
    FreeQ[
     result = InverseFourierTransform[
        FullSimplify[
         Apply[Times, (-i ωs)^ns] Exp[-ts.(ωs^2)] result,
         Prepend[Thread[ts > 0], Element[ωs, Reals]]
        ],
        ωs, ys
       ] /. Thread[ys → xs],
     InverseFourierTransform
    ],
    False
   ]
 ]

GaussianD[GaussianDerivative[tn__][img_][s__Symbol], stn : {_Symbol, _, _ : 0} ..,
  opts___Rule] := GaussianDerivative[Apply[Sequence, {tn} + Replace[{s},
       Append[Map[(First[#] → Rest[#]) &, {stn}], _ → {0, 0}], {1}]], opts][img][s]

GaussianD[img_Symbol[s : (__Symbol | __Slot)], stn : {_Symbol | _Slot, _, _ : 0} ..,
  opts___Rule] := GaussianDerivative[Apply[Sequence, Replace[{s},
      Append[Map[(First[#] → Rest[#]) &, {stn}], _ → {0, 0}], {1}]], opts][img][s]

GaussianD[img_?NumericQ, stn : {_Symbol, _, _ : 0} .., opts___Rule] = 0;
```

■ GaussianD Input Notation

```
Notation[∂_{G_t_[x_]} f_ ⟺ GaussianD[f_, {x_, t_, 1}]]

(* GaussianD Input Notation *)

MakeExpression[
  RowBox[{lhs___, SubscriptBox["∂", RowBox[{SubscriptBox["G", t_], "[", x_, "]"}]],
    f_, rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[", RowBox[{f, ",", RowBox[
         {"{", RowBox[{x, ",", t, ",", "1"}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x_, t_, 1}], StandardForm] :=
 RowBox[{SubscriptBox["∂", RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]],
      "[", MakeBoxes[x, StandardForm], "]"}]], Parenthesize[f, StandardForm, D]}]

Notation[∂^(n_)_{G_t_[x_]} f_ ⟺ GaussianD[f_, {x_, t_, n_}]]
```

```
MakeExpression[RowBox[
   {lhs___, SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", t_], "[", x_, "]"}],
     RowBox[{"(", n_, ")"}]], f_, rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[", RowBox[{f, ",",
       RowBox[{"{", RowBox[{x, ",", t, ",", n}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x_, t_, n_}], StandardForm] := RowBox[
  {SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]],
      "[", MakeBoxes[x, StandardForm], "]"}], RowBox[
     {"(", MakeBoxes[n, StandardForm], ")"}]], Parenthesize[f, StandardForm, D]}]

Notation[∂_{G_{t1_,t2_}[x1_,x2_]}^{(n1_,n2_)} f_ ⟺ GaussianD[f_, {x1_, t1_, n1_}, {x2_, t2_, n2_}]]

MakeExpression[
  RowBox[{lhs___, SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", RowBox[
         {t1_, ",", t2_}]], "[", RowBox[{x1_, ",", x2_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_}], ")"}]], f_, rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[", RowBox[
       {f, ",", RowBox[{"{", RowBox[{x1, ",", t1, ",", n1}], "}"}], ",", RowBox[
         {"{", RowBox[{x2, ",", t2, ",", n2}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t1_, n1_}, {x2_, t2_, n2_}], StandardForm] :=
 RowBox[{SubsuperscriptBox["∂", RowBox[{SubscriptBox["G",
       RowBox[{MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm]}]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm]}], "]"}],
    RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
       MakeBoxes[n2, StandardForm]}], ")"}]], Parenthesize[f, StandardForm, D]}]

Notation[∂_{G_t_[x1_,x2_]}^{(n1_,n2_)} f_ ⟺ GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_}]]

MakeExpression[RowBox[{lhs___, SubsuperscriptBox["∂",
     RowBox[{SubscriptBox["G", t_], "[", RowBox[{x1_, ",", x2_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_}], ")"}]], f_, rhs___}], StandardForm] :=
 MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[",
      RowBox[{f, ",", RowBox[{"{", RowBox[{x1, ",", t, ",", n1}], "}"}], ",", RowBox[
        {"{", RowBox[{x2, ",", t, ",", n2}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_}], StandardForm] := RowBox[
  {SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm]}], "]"}],
    RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
       MakeBoxes[n2, StandardForm]}], ")"}]], Parenthesize[f, StandardForm, D]}]

Notation[∂_{G_{t1_,t2_,t3_}[x1_,x2_,x3_]}^{(n1_,n2_,n3_)} f_ ⟺
  GaussianD[f_, {x1_, t1_, n1_}, {x2_, t2_, n2_}, {x3_, t3_, n3_}]]
```

```
MakeExpression[
  RowBox[{lhs___, SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", RowBox[
         {t1_, ",", t2_, ",", t3_}]], "[", RowBox[{x1_, ",", x2_, ",", x3_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_}], ")"}]}], f_, rhs___}],
  StandardForm] := MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[",
      RowBox[{f, ",", RowBox[{"{", RowBox[{x1, ",", t1, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{x2, ",", t2, ",", n2}], "}"}], ",", RowBox[
         {"{", RowBox[{x3, ",", t3, ",", n3}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t1_, n1_}, {x2_, t2_, n2_}, {x3_, t3_, n3_}],
  StandardForm] := RowBox[{SubsuperscriptBox["∂",
    RowBox[{SubscriptBox["G", RowBox[{MakeBoxes[t1, StandardForm], ",",
         MakeBoxes[t2, StandardForm], ",", MakeBoxes[t3, StandardForm]}]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm],
        ",", MakeBoxes[x3, StandardForm]}], "]"}], RowBox[
     {"(", RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm],
        ",", MakeBoxes[n3, StandardForm]}], ")"}]}], Parenthesize[f, StandardForm, D]}]

Notation[
 ∂_{G_t_[x1_,x2_,x3_]}^{(n1_,n2_,n3_)} f_ ⟺ GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_}, {x3_, t_, n3_}]]

MakeExpression[RowBox[{lhs___, SubsuperscriptBox["∂",
     RowBox[{SubscriptBox["G", t_], "[", RowBox[{x1_, ",", x2_, ",", x3_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_}], ")"}]}], f_, rhs___}],
  StandardForm] := MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[",
      RowBox[{f, ",", RowBox[{"{", RowBox[{x1, ",", t, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{x2, ",", t, ",", n2}], "}"}], ",", RowBox[
         {"{", RowBox[{x3, ",", t, ",", n3}], "}"}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_}, {x3_, t_, n3_}],
  StandardForm] := RowBox[
  {SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm],
        ",", MakeBoxes[x3, StandardForm]}], "]"}], RowBox[
     {"(", RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm],
        ",", MakeBoxes[n3, StandardForm]}], ")"}]}], Parenthesize[f, StandardForm, D]}]

Notation[∂_{G_{t1_,t2_,t3_,t4_}[x1_,x2_,x3_,x4_]}^{(n1_,n2_,n3_,n4_)} f_ ⟺
  GaussianD[f_, {x1_, t1_, n1_}, {x2_, t2_, n2_}, {x3_, t3_, n3_}, {x4_, t4_, n4_}]]
```

```
MakeExpression[RowBox[{lhs___, SubsuperscriptBox["∂",
     RowBox[{SubscriptBox["G", RowBox[{t1_, ",", t2_, ",", t3_, ",", t4_}]],
       "[", RowBox[{x1_, ",", x2_, ",", x3_, ",", x4_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_, ",", n4_}], ")"}]}], f_, rhs___}],
  StandardForm] := MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[",
      RowBox[{f, ",", RowBox[{"{", RowBox[{x1, ",", t1, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{x2, ",", t2, ",", n2}], "}"}], ",",
        RowBox[{"{", RowBox[{x3, ",", t3, ",", n3}], "}"}], ",", RowBox[
         {"{", RowBox[{x4, ",", t4, ",", n4}], "}"}]}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t1_, n1_},
   {x2_, t2_, n2_}, {x3_, t3_, n3_}, {x4_, t4_, n4_}], StandardForm] :=
 RowBox[{SubsuperscriptBox["∂", RowBox[{SubscriptBox["G",
       RowBox[{MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm], ",",
         MakeBoxes[t3, StandardForm], ",", MakeBoxes[t4, StandardForm]}]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm],
        ",", MakeBoxes[x3, StandardForm], ",", MakeBoxes[x4, StandardForm]}], "]"}],
    RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
       MakeBoxes[n2, StandardForm], ",", MakeBoxes[n3, StandardForm], ",",
       MakeBoxes[n4, StandardForm]}], ")"}]}], Parenthesize[f, StandardForm, D]}]

Notation[∂_{G_[x1_,x2_,x3_,x4_]}^{(n1_,n2_,n3_,n4_)} f_ ⇔
  GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_}, {x3_, t_, n3_}, {x4_, t_, n4_}]]

MakeExpression[
  RowBox[{lhs___, SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", t_], "[",
       RowBox[{x1_, ",", x2_, ",", x3_, ",", x4_}], "]"}],
     RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_, ",", n4_}], ")"}]}], f_, rhs___}],
  StandardForm] := MakeExpression[RowBox[{lhs, RowBox[{"GaussianD", "[",
      RowBox[{f, ",", RowBox[{"{", RowBox[{x1, ",", t, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{x2, ",", t, ",", n2}], "}"}], ",",
        RowBox[{"{", RowBox[{x3, ",", t, ",", n3}], "}"}], ",", RowBox[
         {"{", RowBox[{x4, ",", t, ",", n4}], "}"}]}]}], "]"}], rhs}], StandardForm]

MakeBoxes[GaussianD[f_, {x1_, t_, n1_}, {x2_, t_, n2_},
   {x3_, t_, n3_}, {x4_, t_, n4_}], StandardForm] := RowBox[
  {SubsuperscriptBox["∂", RowBox[{SubscriptBox["G", MakeBoxes[t, StandardForm]], "[",
      RowBox[{MakeBoxes[x1, StandardForm], ",", MakeBoxes[x2, StandardForm], ",",
        MakeBoxes[x3, StandardForm], ",", MakeBoxes[x4, StandardForm]}], "]"}],
    RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
       MakeBoxes[n2, StandardForm], ",", MakeBoxes[n3, StandardForm], ",",
       MakeBoxes[n4, StandardForm]}], ")"}]}], Parenthesize[f, StandardForm, D]}]
```

■ GaussianDerivative

■ Automatic Kernel Range

To ensure a given precision for Gaussian derivatives, one must provide a Gaussian kernel with a minimum range [-r,r]. If ϵ is the admissable error, the constraint for a continuous Gaussian kernel is

$$\int_{-r}^{r} \text{GaussianDKernel}[t, 0, x]\, dx \geq 1 - \epsilon$$

For n-th order derivative kernels this generalizes to

$$\int_{-r}^{r} \frac{(-x)^n}{n!}\, \text{GaussianDKernel}[t, n, x]\, dx \geq 1 - \epsilon$$

Unfortunately, the above inequality cannot be solved for **r** symbolically. Hence, we solve the inequality numerically for
t = 1 and $\epsilon = 2^{-m}$ with **m** = 1,..., 32. Note, that t scales **x** by $\sqrt{t}$ and thus also **r**.

$$\text{RangeError}[n\_] := \text{RangeError}[n] = \int_{-r}^{r} \frac{(-x)^n}{n!}\, \text{GaussianDKernel}[1, n, x]\, dx$$

For the Gaussian kernel we obtain the explicit result

```
r /. Solve[RangeError[0] == 1 - 2^-m, r][[1]]
```

```
2 InverseErf[0, 2^-m (-1 + 2^m)]
```

A list of results for derivatives up the 16th order is generated by

```
RangeErrorList[m_] :=
 Rest[FoldList[(r /. FindRoot[Evaluate[RangeError[#2] == 1 - 2^-m], {r, #1}]) &,
   2 InverseErf[0, 2^-m (-1 + 2^m)], Range[0, 16]]]
```

In accordance with Zernickes formula we fit the results.

```
RangeErrorFunction[m_] :=
 Sqrt[Fit[Transpose[{Range[0, 16], RangeErrorList[m]^2}], {n + 1, (n + 1)^(1/3)}, n]]
```

Thus, for each **m** we can generate a Gaussian kernel range function.

```
GaussianKernelRange[m_] = Compile[{{t, _Real}, {n, _Integer}},
  {-#, #} &[Evaluate[Ceiling[Sqrt[t τ / 4] RangeErrorFunction[m]]]]]
```

We do so explicitly for **m** = 4, 8, 12,... , 52.

```
GaussianKernelRange[4] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(2.3243180666111423 +
       2.3243180666111423 * n - 0.9765965923097873 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[8] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(2.2224289812448683 +
       2.2224289812448683 * n + 1.6011247864558187 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[12] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(2.114376372864952 +
       2.114376372864952 * n + 4.0288081480154725 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[16] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.998216846965191 +
       1.998216846965191 * n + 6.392974885898669 * (1 + n)^(1 / 3)) * t * τ]]]]
```

```
GaussianKernelRange[20] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.874991609534614 +
       1.874991609534614 * n + 8.723038909063387 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[24] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.7458525263268259 +
       1.7458525263268259 * n + 11.032362584249514 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[28] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.6117501067548536 +
       1.6117501067548536 * n + 13.328052128995822 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[32] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.473435732709178 +
       1.473435732709178 * n + 15.61428837974991 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[36] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.3315035827972916 +
       1.3315035827972916 * n + 17.893710961739284 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[40] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.1864219273587364 +
       1.1864219273587364 * n + 20.168092531572174 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[44] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(1.0385758971718777 +
       1.0385758971718777 * n + 22.438510420428443 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[48] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(0.889875356726046 +
       0.889875356726046 * n + 24.692137806054756 * (1 + n)^(1 / 3)) * t * τ]]]]

GaussianKernelRange[52] =
 Compile[{{t, _Real}, {n, _Integer}}, {-#, #} &[Ceiling[Sqrt[(0.7367547195975274 +
       0.7367547195975274 * n + 26.829859007281208 * (1 + n)^(1 / 3)) * t * τ]]]]
```

Earlier versions of this package used the formula of Zernicke for the upper bound of Gaussian width.

```
GaussianKernelRange = Compile[{{t, _Real}, {n, _Integer}},
   Ceiling[4 * Sqrt[τ t] * Sqrt[n + 2 - 1.15 ∛(n + 2)]]];

GaussianDerivative::widthfail =
 "Insufficient kernel width for `1`-order differentiation.";
GaussianDerivative::nowidth = "No kernel width specified.";
```

■ Automatic Minimal Scale

Scale is needed to regularize a discrete dataset. Here we determine the minimal scale needed to render correct derivative values for a given precision.

In Fourier space an n-th order Gaussian derivation is performed via multiplication with

$$\frac{(-i\,\omega)^n\, e^{-t\,\omega^2}}{\sqrt{2\,\pi}}$$

This Gaussian derivative kernel in Fourier space extends from -∞ to ∞. However, due to the discreteness of the data we are bounded by its Nyquist frequency. If we take the lattice constant of the data grid as 1, we cannot sample the data with frequencies higher than π. Hence, we miss the values in Fourier space from π to ∞. We calculate this error and determine the scale, with which this error is kept below a given precision threshold.

```
error[n_] := error[n] = Simplify[(∫_0^π (-i ω)^n e^(-t ω^2) / Sqrt[2 π] dω) / (∫_0^∞ (-i ω)^n e^(-t ω^2) / Sqrt[2 π] dω)]
```

Here are the minimal scales *t* for Gaussian derivatives of order *n* from 0 through 16 to obtain a precision of 2^{-b} with *b* running from 4 to 52 in steps of 4.

```
results = Table[MapIndexed[{First[#2] - 1, t /. FindRoot[#1 == 1 - 2^-b, {t, 1}]} &,
    Table[error[n], {n, 0, 16}]], {b, 4, 52, 4}];
```

For each precision we fit the terms of Zernicke's formula to these minmal scales.

```
Map[Fit[#, {1, n + 1, (n + 1)^(1/3)}, n] &, results]
```

```
MinimalScale[4] = Compile[{{n, _Integer}}, -0.07225602550763996` +
    0.1934994115318552` * (1 + n)^(1/3) + 0.05458391922630867` (1 + n)];
```

```
MinimalScale[8] = Compile[{{n, _Integer}},
   0.04813185222160912` + 0.31489231691130615` (1 + n)^(1/3) + 0.05841271321345174` (1 + n)];
```

```
MinimalScale[12] = Compile[{{n, _Integer}},
   0.23138720823850856` + 0.3870206014942941` (1 + n)^(1/3) + 0.06201023916198861` (1 + n)];
```

```
MinimalScale[16] = Compile[{{n, _Integer}},
   0.4427462125429202` + 0.43729757816428666` (1 + n)^(1/3) + 0.06533784418156777` (1 + n)];
```

```
MinimalScale[20] = Compile[{{n, _Integer}},
   0.6700763376887527` + 0.47527046426284586` (1 + n)^(1/3) + 0.06841700241661773` (1 + n)];
```

```
MinimalScale[24] = Compile[{{n, _Integer}},
   0.9076871517131073` + 0.5053978829196788` (1 + n)^(1/3) + 0.07127851131749753` (1 + n)];
```

```
MinimalScale[28] = Compile[{{n, _Integer}},
   1.1524506471180527` + 0.5301181105158642` (1 + n)^(1/3) + 0.07395088137222257` (1 + n)];
```

```
MinimalScale[32] = Compile[{{n, _Integer}},
   1.4024642991094307` + 0.5509059382924002` (1 + n)^(1/3) + 0.0764583548853869` (1 + n)];
```

```
MinimalScale[36] = Compile[{{n, _Integer}},
   1.656487038699375` + 0.568717809132497` (1 + n)^(1/3) + 0.07882117438223667` (1 + n)];
```

```
MinimalScale[40] = Compile[{{n, _Integer}},
   1.9136820125603298` + 0.5841887192822285` (1 + n)^(1/3) + 0.08105828273659765` (1 + n)];
```

```
MinimalScale[44] = Compile[{{n, _Integer}},
   2.1736973881589376` + 0.5976246623874925` (1 + n)^(1/3) + 0.0831900868437959` (1 + n)];
```

```
MinimalScale[48] = Compile[{{n, _Integer}},
   2.442533291990515` + 0.6028349507086076` (1 + n)^(1/3) + 0.08585656496350498` (1 + n)];
```

```
MinimalScale[52] = Compile[{{n, _Integer}},
   2.627515797245066` + 0.6649239314410514` (1 + n)^(1/3) + 0.08356758359896967` (1 + n)];
```

```
GaussianDerivative::scalefail = "Scale too small for `1`-order differentiation.";
```

The former approach for 12-bit precision to estimate the lower boumd of admissable scale paramterer values was:

```
MinimalScale = Compile[{{n, _Integer}}, Re[Sqrt[n + 1 - 1.15 (n + 1)^(1/3)]]];
```

■ Symbolical Implementation

```
GaussianDerivative[{t_, 0}][UnitStep] := 1/2 (1 + Erf[#/(2 Sqrt[t])]) &

GaussianDerivative[tn : {0, 0} ..][img_] = img;

GaussianDerivative[tn : {_, _Integer?Positive} ..][img_Symbol] :=
 0 /; MemberQ[Attributes[img], Constant]

GaussianDerivative[tn1 : {_, _Integer : 0} .., opts1___Rule][
  GaussianDerivative[tn2 : {_, _Integer : 0} .., opts2___Rule][img_]] :=
 GaussianDerivative[Apply[Sequence, {tn1} + {tn2}],
    Apply[Sequence, Union[{opts1}, {opts2}]]][img] /; Length[{tn1}] === Length[{tn2}]

GaussianDerivative[tn : {_, _} .., opts___Rule][f_] :=
 (Evaluate[preliminary$Result] &) /;
  Not[ArrayQ[f, _, NumeriQ]] ∧ Not[MatchQ[Head[preliminary$Result =
       GaussianD[f @@ Thread[Slot[Range[Length[{tn}]]]], Sequence @@
         MapThread[Prepend, {{tn}, Thread[Slot[Range[Length[{tn}]]]]}]]],
     HoldPattern[GaussianDerivative[tn, opts][f]]
    ]
   ] /; Not[ArrayQ[f]]
```

Old Code:

```
GaussianDerivative[tn : {_, _Integer : 0} .., opts___Rule][img_Symbol] :=
 Module[
  {result, ts, ns,
   xs = Table[Unique["x"], {Length[{tn}]}],
   ωs = Table[Unique["ω"], {Length[{tn}]}],
   ys = Table[Unique["y"], {Length[{tn}]}]},
  (Evaluate[result /. Thread[ys → Table[Slot[i], {i, Length[ys]}]]] &) /;
   If[
    FreeQ[result = FourierTransform[Apply[img, xs], xs, ωs], FourierTransform],
    {ts, ns} = Transpose[{tn}];
    FreeQ[result =  InverseFourierTransform[Apply[Times, (-I ωs)^ns]
        Exp[-ts.(ωs^2)] result, ωs, ys], InverseFourierTransform],
    False
   ]
 ]
```

■ Numerical Implementation

The GaussianDerivative checks if the image and parameters allow numeric differentiation and calls up the apporpriate Method.

```
GaussianDerivative::nomethod = "Derivation Method `1` not implemented!";

GaussianDerivative[tn : {_?NumericQ, _Integer : 0} .., opts___Rule][img_List] :=
 Switch[
   Method /. {opts} /. Options[GaussianDerivative],
   Convolve, ListConvolveGaussianDerivative[tn, opts][img],
   Fourier, FourierGaussianDerivative[Apply[Sequence, Reverse[{tn}]], opts][img],
   _, Message[GaussianDerivative::nomethod, Method]; Throw[$Failed]
  ] /; TensorRank[img] ≥ Length[{tn}]
```

ListConvolve Method

```
GaussianDerivative::noboundopt = "Invalid ImageBoundary option.";

GaussianKernelInterval[{0, 0}, _, rangeopt_: Automatic] = {-1, 1};

GaussianKernelInterval[{t_, n_}, bits_, rangeopt_: Automatic] :=
  (If[t ≤ 0, Message[GaussianDerivative::scalefail, n]; Throw[$Failed]];
   If[t < MinimalScale[bits][n], Message[GaussianDerivative::scalefail, n]];
   Switch[
    rangeopt,
    Automatic, GaussianKernelRange[bits][t, n],
    Infinity, {-1, 1} * Floor[Sqrt[-τ t * Log[$MinMachineNumber]]],
    {-Infinity, Infinity}, {-1, 1} * Floor[Sqrt[-τ t * Log[$MinMachineNumber]]],
    {_Integer, Infinity},
    {rangeopt[[1]], Floor[Sqrt[-τ t * Log[$MinMachineNumber]]]},
    {-Infinity, _Integer}, {-Floor[Sqrt[-τ t * Log[$MinMachineNumber]]],
     rangeopt[[2]]},
    {_Integer, _Integer}, rangeopt,
    _, Message[GaussianDerivative::nowidth]; Throw[$Failed]
   ]);

ListConvolveGaussianDerivative1D[
   {{0, 0}, {kmin_, kmax_}, boundaryopt_}, img_, depth_, opts___] = img;

ListConvolveGaussianDerivative1D[
  {{t_, n_}, {kmin_, kmax_}, boundaryopt_}, img_, depth_, opts___] :=
 Module[
  {kern = Nest[List, Table[NGaussianDKernel[n][t, x], {x, kmin, kmax}], depth - 1]},
  Switch[
   boundaryopt,

   Cyclic,
   ListConvolve[kern, img,
    {PadLeft[{-kmin + 1}, depth, 1], PadLeft[{-kmax - 1}, depth, 1]}] ,

   Truncate,
   Map[AttachFrame1D[#, {-kmin, kmax}] &,
    ListConvolve[kern, img, {PadLeft[{-1}, depth, 1], PadLeft[{1}, depth, 1]}],
    {depth - 1} ],

   _?NumericQ,
   ListConvolve[kern, img,
    {PadLeft[{-kmin + 1}, depth, 1], PadLeft[{-kmax - 1}, depth, 1]}, N[boundaryopt]] ,
```

```
    Constant,
    ListConvolve[kern, ImageFrame1D[img, {{-kmin, kmax}, depth, Constant}]] ,

    Reflective,
    ListConvolve[kern, ImageFrame1D[img, {{-kmin, kmax}, depth, Reflective}]],

    _, Message[GaussianDerivative::nobound]; Throw[$Failed]
   ]
  ]

ListConvolveGaussianDerivative[tn__List, opts___Rule][img_List] :=
 Catch[Module[
   {
    dim = TensorRank[img], nn = Length[{tn}], dimrot,
    bounds = (ImageBoundary /. {opts} /. Options[GaussianDerivative]),
    kernrange = (KernelRange /. {opts} /. Options[GaussianDerivative]),
    kernpbits = Min[52, Max[4, 4 * Ceiling[
         Log[2, 10^(WorkingPrecision /. {opts} /. Options[GaussianDerivative])
          ] / 4]]]
   },
   If[
    dim === nn,
    dimrot = RotateLeft[Range[nn]];
    Fold[
     ListConvolveGaussianDerivative1D[#2, Transpose[#1, dimrot], dim, opts] &,
     img,
     Map[{#, GaussianKernelInterval[#, kernpbits, kernrange], bounds} &,
      RotateLeft[{tn}]]],
    dimrot = Join[Range[1, dim - nn], RotateLeft[Range[dim - nn + 1, dim]]];
    Transpose[
     Fold[
      ListConvolveGaussianDerivative1D[#2, Transpose[#1, dimrot], dim, opts] &,
      Transpose[img, Join[Range[nn + 1, dim], Range[nn]]],
      Map[{#, GaussianKernelInterval[#, kernpbits, kernrange], bounds} &,
       RotateLeft[{tn}]]
     ],
     Join[Range[dim - nn + 1, dim], Range[1, dim - nn]]
    ]
   ]
  ]]
```

Fourier Method

NFourierGaussianDKernelList[] renders the Fourier transformed kernel scaled by factor $(-i)^n \sqrt{r}$. This way the kernel remains real and the multiplication with the Fourier transformed image is performed twice as fast. The complex scaling is applied separately at a later stage. The Fourier transformed Gaussian derivative kernel is capped by **10.^($MachinePrecision-3)** so that numerical instabilities for negative **t** do not occur.

```
NFourierGaussianDKernelList =
  Compile[
   {{t, _Real}, {n, _Integer}, {r, _Integer}},
   Join[{If[n === 0, 1, 0]}, #, (-1)^n Reverse[If[EvenQ[r], Drop[#, -1], #]]] &[
    With[{stepsize = 2. Pi / r},
     Table[(omega)^n * Exp[-omega^2 τ t / 4], {omega, stepsize, Pi, stepsize}]
    ]
   ],
   {{omega, _Real}, {stepsize, _Real}}
  ];

FourierGaussianDerivative[tn__List, opts___Rule][img_List] :=
 Module[
  {dim = TensorRank[img],
   nn = Length[{tn}],
   kern, norm, res},
  kern =
   Apply[
    FastOuterTimes,
    Apply[
     NFourierGaussianDKernelList,
     Transpose[Append[Transpose[{tn}], res = Take[Dimensions[img], nn]]],
     {1}
    ]
   ];
  If[Not[Apply[And, Positive[Transpose[{tn}][[1]]]]] &&
    (Max[kern] > 10.^($MachinePrecision - 3)),
   kern = Map[Min[#, 10.^($MachinePrecision - 3)] &, kern, {-1}]];
  norm = Apply[Times, (-I)^(Transpose[{tn}][[2]])] ;
  If[
   dim === nn,
   Chop[Re[InverseFourier[norm * kern * Fourier[img]]]],
   Transpose[
    Map[
     Chop[Re[InverseFourier[norm * kern * Fourier[#]]]] &,
     Transpose[img, Join[Range[nn + 1, dim], Range[nn]]],
     {dim - nn}
    ],
    Join[Range[dim - nn + 1, dim], Range[1, dim - nn]]
   ]
  ]
 ]
```

■ GaussianDerivative Input Notation

```
Notation[f_G_t_ ⟺ GaussianDerivative[{t_, 0}][f_]]
```

```
(* GaussianDerivative Input Notation *)

MakeExpression[SubscriptBox[f_, SubscriptBox["G", t_]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{"{", RowBox[{t, ",", "0"}], "}"}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 0}][f_], StandardForm] := SubscriptBox[
  MakeBoxes[f, StandardForm], SubscriptBox["G", MakeBoxes[t, StandardForm]]]

Notation[f_G't_ ⟺ GaussianDerivative[{t_, 1}][f_]]

MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_], "′"], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{"{", RowBox[{t, ",", "1"}], "}"}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 1}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]], "′", MultilineFunction → None]

Notation[f_G''t_ ⟺ GaussianDerivative[{t_, 2}][f_]]

MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_], "″"], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{"{", RowBox[{t, ",", "2"}], "}"}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 2}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]], "″", MultilineFunction → None]

Notation[f_G^(n_)t_ ⟺ GaussianDerivative[{t_, n_}][f_]]

MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_], RowBox[{"(", n_, ")"}]],
  StandardForm] := MakeExpression[RowBox[
   {RowBox[{"GaussianDerivative", "[", RowBox[{"{", RowBox[{t, ",", n}], "}"}], "]"}],
    "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, n_}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm], SubscriptBox["G",
   MakeBoxes[t, StandardForm]], RowBox[{"(", MakeBoxes[n, StandardForm], ")"}]]

Notation[f_G_t1_,t2_ ⟺ GaussianDerivative[{t1_, 0}, {t2_, 0}][f_]]

MakeExpression[SubscriptBox[f_, SubscriptBox["G", RowBox[{t1_, ",", t2_}]]],
  StandardForm] := MakeExpression[RowBox[{RowBox[{"GaussianDerivative",
      "[", RowBox[{RowBox[{"{", RowBox[{t1, ",", "0"}], "}"}], ",", RowBox[
        {"{", RowBox[{t2, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, 0}, {t2_, 0}][f_], StandardForm] :=
 SubscriptBox[MakeBoxes[f, StandardForm], SubscriptBox["G",
   RowBox[{MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm]}]]]

Notation[f_G^(n1_,n2_)t1_,t2_ ⟺ GaussianDerivative[{t1_, n1_}, {t2_, n2_}][f_]]
```

```
MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", RowBox[{t1_, ",", t2_}]],
   RowBox[{"(", RowBox[{n1_, ",", n2_}], ")"}]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{RowBox[{"{", RowBox[{t1, ",", n1}], "}"}], ",", RowBox[
         {"{", RowBox[{t2, ",", n2}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, n1_}, {t2_, n2_}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm], SubscriptBox["G", RowBox[
    {MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm]}]], RowBox[{"(",
    RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm]}], ")"}]]

Notation[f_G_t_ ⟺ GaussianDerivative[{t_, 0}, {t_, 0}][f_]]

MakeExpression[SubscriptBox[f_, SubscriptBox["G", t_]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{RowBox[{"{", RowBox[{t, ",", "0"}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 0}, {t_, 0}][f_], StandardForm] := SubscriptBox[
  MakeBoxes[f, StandardForm], SubscriptBox["G", MakeBoxes[t, StandardForm]]]

Notation[f_G_t_^(n1_,n2_) ⟺ GaussianDerivative[{t_, n1_}, {t_, n2_}][f_]]

MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_],
   RowBox[{"(", RowBox[{n1_, ",", n2_}], ")"}]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[{"GaussianDerivative", "[",
      RowBox[{RowBox[{"{", RowBox[{t, ",", n1}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", n2}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, n1_}, {t_, n2_}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]], RowBox[{"(",
    RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm]}], ")"}]]

Notation[f_G_t1_,t2_,t3_ ⟺ GaussianDerivative[{t1_, 0}, {t2_, 0}, {t3_, 0}][f_]]

MakeExpression[
  SubscriptBox[f_, SubscriptBox["G", RowBox[{t1_, ",", t2_, ",", t3_}]]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t1, ",", "0"}], "}"}],
        ",", RowBox[{"{", RowBox[{t2, ",", "0"}], "}"}], ",", RowBox[
         {"{", RowBox[{t3, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, 0}, {t2_, 0}, {t3_, 0}][f_], StandardForm] :=
 SubscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", RowBox[{MakeBoxes[t1, StandardForm], ",",
     MakeBoxes[t2, StandardForm], ",", MakeBoxes[t3, StandardForm]}]]]

Notation[f_G_t1_,t2_,t3_^(n1_,n2_,n3_) ⟺ GaussianDerivative[{t1_, n1_}, {t2_, n2_}, {t3_, n3_}][f_]]
```

```
MakeExpression[
  SubsuperscriptBox[f_, SubscriptBox["G", RowBox[{t1_, ",", t2_, ",", t3_}]],
   RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_}], ")"}]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t1, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{t2, ",", n2}], "}"}], ",", RowBox[
         {"{", RowBox[{t3, ",", n3}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, n1_}, {t2_, n2_}, {t3_, n3_}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", RowBox[{MakeBoxes[t1, StandardForm], ",",
     MakeBoxes[t2, StandardForm], ",", MakeBoxes[t3, StandardForm]}]],
  RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
      MakeBoxes[n2, StandardForm], ",", MakeBoxes[n3, StandardForm]}], ")"}]]

Notation[f_G_t_ ⟺ GaussianDerivative[{t_, 0}, {t_, 0}, {t_, 0}][f_]]

MakeExpression[SubscriptBox[f_, SubscriptBox["G", t_]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t, ",", "0"}], "}"}],
        ",", RowBox[{"{", RowBox[{t, ",", "0"}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 0}, {t_, 0}, {t_, 0}][f_], StandardForm] :=
 SubscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]]]

Notation[f_G_t_^(n1_,n2_,n3_) ⟺ GaussianDerivative[{t_, n1_}, {t_, n2_}, {t_, n3_}][f_]]

MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_],
   RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_}], ")"}]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{t, ",", n2}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", n3}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, n1_}, {t_, n2_}, {t_, n3_}][f_], StandardForm] :=
 SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]],
  RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",",
      MakeBoxes[n2, StandardForm], ",", MakeBoxes[n3, StandardForm]}], ")"}]]

Notation[
 f_G_t1_,t2_,t3_,t4_ ⟺ GaussianDerivative[{t1_, 0}, {t2_, 0}, {t3_, 0}, {t4_, 0}][f_]]
```

```
MakeExpression[
  SubscriptBox[f_, SubscriptBox["G", RowBox[{t1_, ",", t2_, ",", t3_, ",", t4_}]]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t1, ",", "0"}], "}"}],
        ",", RowBox[{"{", RowBox[{t2, ",", "0"}], "}"}], ",",
        RowBox[{"{", RowBox[{t3, ",", "0"}], "}"}], ",", RowBox[
         {"{", RowBox[{t4, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, 0}, {t2_, 0}, {t3_, 0}, {t4_, 0}][f_],
  StandardForm] := SubscriptBox[MakeBoxes[f, StandardForm], SubscriptBox["G",
   RowBox[{MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm],
     ",", MakeBoxes[t3, StandardForm], ",", MakeBoxes[t4, StandardForm]}]]]

Notation[f_(n1_,n2_,n3_,n4_)_(G_(t1_,t2_,t3_,t4_)) ⇔
  GaussianDerivative[{t1_, n1_}, {t2_, n2_}, {t3_, n3_}, {t4_, n4_}][f_]]

MakeExpression[SubsuperscriptBox[f_,
   SubscriptBox["G", RowBox[{t1_, ",", t2_, ",", t3_, ",", t4_}]],
   RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_, ",", n4_}], ")"}]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t1, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{t2, ",", n2}], "}"}], ",",
        RowBox[{"{", RowBox[{t3, ",", n3}], "}"}], ",", RowBox[
         {"{", RowBox[{t4, ",", n4}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t1_, n1_}, {t2_, n2_}, {t3_, n3_}, {t4_, n4_}][f_],
  StandardForm] := SubsuperscriptBox[MakeBoxes[f, StandardForm], SubscriptBox[
   "G", RowBox[{MakeBoxes[t1, StandardForm], ",", MakeBoxes[t2, StandardForm],
     ",", MakeBoxes[t3, StandardForm], ",", MakeBoxes[t4, StandardForm]}]],
  RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm],
      ",", MakeBoxes[n3, StandardForm], ",", MakeBoxes[n4, StandardForm]}], ")"}]]

Notation[f_(G_t_) ⇔ GaussianDerivative[{t_, 0}, {t_, 0}, {t_, 0}, {t_, 0}][f_]]

MakeExpression[SubscriptBox[f_, SubscriptBox["G", t_]], StandardForm] :=
 MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t, ",", "0"}], "}"}],
        ",", RowBox[{"{", RowBox[{t, ",", "0"}], "}"}], ",",
        RowBox[{"{", RowBox[{t, ",", "0"}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", "0"}], "}"}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, 0}, {t_, 0}, {t_, 0}, {t_, 0}][f_], StandardForm] :=
 SubscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]]]

Notation[
 f_(G_t_)^(n1_,n2_,n3_,n4_) ⇔ GaussianDerivative[{t_, n1_}, {t_, n2_}, {t_, n3_}, {t_, n4_}][f_]]
```

```
MakeExpression[SubsuperscriptBox[f_, SubscriptBox["G", t_],
   RowBox[{"(", RowBox[{n1_, ",", n2_, ",", n3_, ",", n4_}], ")"}]],
  StandardForm] := MakeExpression[RowBox[{RowBox[
     {"GaussianDerivative", "[", RowBox[{RowBox[{"{", RowBox[{t, ",", n1}], "}"}],
        ",", RowBox[{"{", RowBox[{t, ",", n2}], "}"}], ",",
        RowBox[{"{", RowBox[{t, ",", n3}], "}"}], ",", RowBox[
         {"{", RowBox[{t, ",", n4}], "}"}]}]}], "]"}], "[", f, "]"}], StandardForm]

MakeBoxes[GaussianDerivative[{t_, n1_}, {t_, n2_}, {t_, n3_}, {t_, n4_}][f_],
  StandardForm] := SubsuperscriptBox[MakeBoxes[f, StandardForm],
  SubscriptBox["G", MakeBoxes[t, StandardForm]],
  RowBox[{"(", RowBox[{MakeBoxes[n1, StandardForm], ",", MakeBoxes[n2, StandardForm],
      ",", MakeBoxes[n3, StandardForm], ",", MakeBoxes[n4, StandardForm]}], ")"}]]
```

■ GaussianDerivativeAt

■ Numerical Implementation

The GaussianDerivative checks if the image and parameters allow numeric differentiation at one or several locations in scale space.

```
GaussianDerivativeAt[
   tnp : {_?NumericQ, _?NumericQ, _Integer : 0} .., opts___Rule][img_List] :=
 Catch[Module[
    {
     bounds = (ImageBoundary /. {opts} /. Options[GaussianDerivativeAt]),
     kernrange = (KernelRange /. {opts} /. Options[GaussianDerivativeAt]),
     kernpbits = Min[52, Max[4, 4 * Ceiling[
          Log[2, 10^(WorkingPrecision /. {opts} /. Options[GaussianDerivativeAt])
           ] / 4]]],
     tnpList = Reverse[Map[PadRight[#, 3] &, {tnp}]],
     positions, intervals, frameWidth, offsets, framedImage, x
    },
    positions = tnpList[[All, 1]];
    intervals = Map[
      GaussianKernelInterval[Rest[#], kernpbits, kernrange] &,
      tnpList, {1}
     ];
    frameWidth = Transpose[Apply[
       {1 - Map[Min[1, #] &, #1],
         MapThread[(Max[#1, #2] - #2) &, {#2, Take[Dimensions[img], Length[#2]]}]} &,
       Transpose[Round[positions] + intervals]
      ]];
    offsets = frameWidth[[All, 1]];
    framedImage = ImageFrame[img, frameWidth, ImageBoundary → bounds];
    Fold[
     Dot[#2, #1] &,
     Apply[Take[framedImage, ##] &, Round[positions] + offsets + intervals],
     MapThread[
      Table[
        NGaussianDKernel[Last[#1]][#1[[-2]], x],
        Evaluate[Join[{x}, Reverse[#2] + First[#1] - Round[First[#1]], {-1}]]
```

```
        ] &,
       {tnpList, intervals}
      ]
     ]
    ]] /; TensorRank[img] ≥ Length[{tnp}]

GaussianDerivativeAt[
   tnps : {{_?NumericQ, _?NumericQ, _Integer : 0} ..} .., opts___Rule][img_List] :=
 Catch[Module[
    {
     bounds = (ImageBoundary /. {opts} /. Options[GaussianDerivativeAt]),
     kernrange = (KernelRange /. {opts} /. Options[GaussianDerivativeAt]),
     kernpbits = Min[52, Max[4, 4 * Ceiling[
          Log[2, 10^(WorkingPrecision /. {opts} /. Options[GaussianDerivativeAt])
           ] / 4]]],
     tnpList = Map[Reverse, Map[PadRight[#, 3] &, {tnps}, {2}]],
     positions, intervals, frameWidth, offsets, framedImage, x
    },
    positions = tnpList[[All, All, 1]];
    intervals =
     Map[GaussianKernelInterval[Rest[#], kernpbits, kernrange] &, tnpList, {2}];
    frameWidth = MapThread[
        {1 - Min[1, First[#1]], Max[Last[#1], #2] - #2} &,
        {#, Take[Dimensions[img], Length[#]]}
       ] &[PseudoTranspose[Round[positions] + intervals, {3, 1, 2}]];
    offsets = frameWidth[[All, 1]];
    framedImage = ImageFrame[img, frameWidth, ImageBoundary → bounds];
    MapThread[
     Fold[Dot[#2, #1] &, Apply[Take[framedImage, ##] &, #1], #2] &,
     {Map[(# + offsets) &, Round[positions]] + intervals,
      MapThread[
       Table[
         NGaussianDKernel[Last[#1]][#1[[-2]], x],
         Evaluate[Join[{x}, Reverse[#2] + First[#1] - Round[First[#1]], {-1}]]
        ] &,
       {tnpList, intervals}, 2
      ]}
    ]
   ]] /; Apply[And, Map[(TensorRank[img] ≥ Length[#]) &, {tnps}]]
```

■ Utility Functions

A utility function to speed up outer multiplications by a factor 15 compared to Outer[].

```
FastOuterTimes[x_List, y__List] :=
  Fold[With[{v = #2}, Map[Times[#, v] &, #1, {TensorRank[#1]}]] &, x, {y}];
FastOuterTimes[x_List] = x;
```

A transpose function that can handle matrices with varying row length.

```
PseudoTranspose[expr : {{__} ..}] :=
 DeleteCases[
  Transpose[PadRight[expr, {Length[expr], Max[Map[Length, expr]]}, Null]], Null, {2}]

PseudoTranspose[expr : {{__} ..}, perm_List] :=
 With[
  {dim = Drop[NestWhileList[
      Max[Map[Length, expr, {i++}]] &, i = 1; Length[expr], (# > 0) &], -1]},
  If[Length[perm] > Length[dim], Message[Transpose::tperm, perm]; Return[]];
  DeleteCases[
   Transpose[PadRight[expr, dim, Null], perm], Null | {Null ..}, Length[perm]]
 ]
```

Attaching a frame of specified value and thickness around an image.

```
AttachFrame1D[data_List, {widthleft_Integer, widthright_Integer}] :=
 If[widthleft == widthright,
  Join[#, data, #] &[
   Table[0, {widthleft}, ##] &@@ Transpose[{Dimensions[First[data]]}]],
  Join[Table[0, {widthleft}, ##], data, Table[0, {widthright}, ##]] &@@
   Transpose[{Dimensions[First[data]]}]
 ]
```

■ Closing Package

Closing the package and locking all the symbols.

```
End[]

Protect[
 G,
 GaussianKernelRange,
 NGaussianKernel,
 GaussianKernel,
 AngularGaussianKernel,
 NGaussianDKernel,
 GaussianDKernel,
 GaussianD,
 GaussianDerivative,
 GaussianDerivativeAt,
 Method,
 KernelRange,
 ImageBoundary,
 Convolve,
 Fourier,
 Cyclic,
 Truncate,
 Reflective
]

EndPackage[]
```

MathVisionTools: GaussianDerivative

- `GaussianDerivative[{ t, n}][f]` represents the *n*-th Gaussian derivative of a one-dimensional function or data list *f* at scale *t*. *t* is equivalent to $\sigma^2/2$ in a Gaussian distribution. A known function *f* is processed with respect to its first slot. A data list *f* is automatically convolved with the respective Gaussian derivative kernel.
- `GaussianDerivative[{ t_x, n_x}, {t_y, n_y}][f]` gives the (n_x, n_y)-th Gaussian derivative of function or data list *f* at scale $t_x$and t_y respectively. A known function *f* is processed with respect to its first slot as variable *x* and with respect to its second slot as variable *y*. A two-dimensional data list *f* is automatically convolved with the respective Gaussian derivative kernel.

- `GaussianDerivative` is also applicable to dimensions larger than 2.
- The following options can be given:

`Method`	`Convolve`	determines the convolutions process. `Method` → `Convolve` causes `GaussianDerivative` to perform the convolution with the built-in `Listvolve` command. With `Method` → `Fourier`, the convolution is done via multiplication in Fourier space.
`KernelRange`	`Automatic`	specifies the size of the Gaussian convolution kernel if the derivation is done via `ListConvolve`. `Infinity` sets the kernel size equal to the data size and any integer or list of integers specifies the dimensions of the kernel.
`ImageBoundary`	`Cyclic`	specifies the boundary condition of the convolution. Convolutions via the `Fourier` method can only handle the cyclic boundary condition. However, if `Method` is set to `Convolve`, several boundary conditions are available: `Cyclic` (default), `Truncate`, `Constant`, `Reflective`, and any numeric value as background.

- Note, that Gaussian derivatives need a minimal scale *t* to render regularized results. Typical lower bounds are 0.8 for first-order derivatives and 1.2 for second-order derivatives.
- $f_{G_t}^{(n)}$ is a short notation for `GaussianDerivative[{ t, n}][f]`.
- See also: and `GaussianDerivativeAt`, `GaussianDKernel`, and `GaussianD`.
- New in Version 1.

Further Examples

This loads the *MathVisionTools* package.

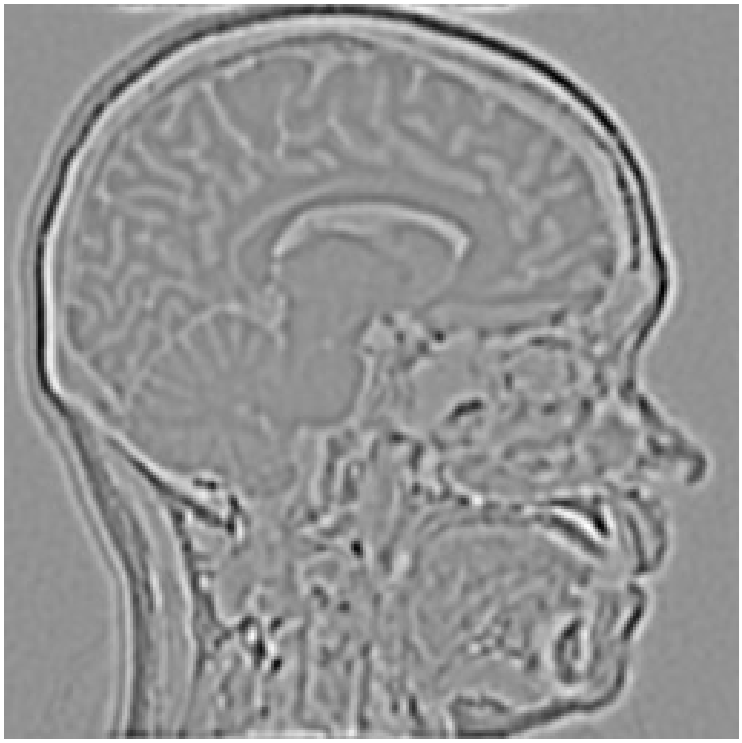
```
In[1]:= << MathVisionTools`
```

The symbolic second-order Gaussian derivative of Cos(*x*) with respect to *x* at scale *t*.

```
In[2]:= GaussianDerivative[{t, 2}][Cos][x]

Out[2]= -e^-t Cos[x]
```

The numeric Laplace operator applied to an image.

```
In[3]:= img = Import["mr256.jpg"][[1, 1]];

In[4]:= edges = GaussianDerivative[{1.2, 2}, {1.2, 0}][img] +
          GaussianDerivative[{1.2, 0}, {1.2, 2}][img];

In[5]:= ListDensityPlot[edges, Mesh → False, Frame → False, PlotRange → All];
```



```
In[6]:= Clear[img, edges]
```

GeometryDrivenDiffusion Package

Edwin Bennink
Technische Universiteit Eindhoven

■ Prefix

```
TimeStamp["MathVisionTools`GeometryDrivenDiffusion`"] = {2005, 1, 12, 13, 15, 0}
```

The preamble of the package containing all kinds of remarks about authorship, contemns, and copyright.

```
(*:Name: MathVisionTools`GeometryDrivenDiffusion` *)

(*:Author: Edwin Bennink*)

(*:Email address: H.E.Bennink@student.tue.nl*)

(*:Context: MathVisionTools`GeometryDrivenDiffusion`*)

(*:Package Version: 1.0 *)

(*:Mathematica Version: 5.0 *)

(*:Copyright: Copyright 2004, Technische Universiteit Eindhoven*)

(*:Title: GeometryDrivenDiffusion *)

(*:Summary: This package implements Geometry Driven Diffusion *)

(*:Keywords: Geometry Driven, Diffusion, Perona, Malik, Euclidean Shortening *)

(*:Requirements: None *)

(*:Source: None *)

(*:History: Version 1.0 by Edwin Bennink, November 2004 *)

(*:Limitations: *)

(*:Discussion: *)

(*:To do: *)
```

■ Begin Package

Declaring the package and unlocking all symbols defined in the code.

```
BeginPackage["MathVisionTools`GeometryDrivenDiffusion`",
 {"MathVisionTools`GaussianDerivative`"}]

Unprotect[
 GeometryDrivenDiffusion,
 EuclideanShorten,
 PeronaMalik1,
 PeronaMalik2,
 ForwardEuler,
 RungeKutta,
 DiffusionEquation,
 Parameters,
 Method
]
```

■ Online Help and Options

```
GeometryDrivenDiffusion::usage =
  "GeometryDrivenDiffusion[data, t, dt] simulates the diffusion of luminance
    within a 2D or 3D data-image using numerical integration and renders
    the final image. t is the diffusion time, dt the integration step size.
    \nDifferent diffusion equations and integration methods can be chosen
    by specifying the Options DiffusionEquation, Parameters and Method.";

DiffusionEquation::usage =
  "DiffusionEquation is and option to GeometryDrivenDiffusion, which
    specifies what diffusion equation to use. Possible options are
    EuclideanShorten, PeronaMalik1 and PeronaMalik2. It is also possible
    to specify any function of the form ∂L/∂t = F[L, {p1, p2, …}], in
    which L is the image data and {p1, p2, …} is a list of parameters.";

Parameters::usage = "Parameters is and option to GeometryDrivenDiffusion,
   which specifies the parameters of the DiffusionEquation."

If[
  Not[StringMatchQ[Method::usage, "*GeometryDrivenDiffusion*"]],
  Method::usage = Method::usage <>
    "\nPossible options for GeometryDrivenDiffusion are ForwardEuler (
      standard Forward Euler) and RungeKutta (4th order Runge-Kutta)."];

Options[GeometryDrivenDiffusion] =
  {DiffusionEquation → EuclideanShorten, Parameters → {0.8, ∞}, Method → ForwardEuler};
```

■ Package Code

Starting the private context of the package.

```
Begin["`Private`"]
```

■ Diffusion Equations (DiffusionEquation)

■ Euclidean shortening (EuclideanShorten)

```
EuclideanShorten[data_, param_] := Module[{lx, ly, lz, lxx, lxy, lyy, σ},
  If[Length[param] ≥ 1, σ = param[[1]],
   Message[EuclideanShorten::parameters]; Throw[$Failed]];
  Switch[ArrayDepth[data],
   2,
   {lx, ly, lxx, lxy, lyy} = GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}][data] & /@
     {{1, 0}, {0, 1}, {2, 0}, {1, 1}, {0, 2}};
   (lxx ly^2 - 2 lx lxy ly + lyy lx^2)/(lx^2 + ly^2 /. {0. → 1.}),
   3, {lx, ly, lz, lxx, lxy, lxz, lyy, lyz, lzz} =
    GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}, {0.5 σ^2, #[[3]]}][data] & /@
     {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {2, 0, 0},
      {1, 1, 0}, {0, 1, 1}, {0, 2, 0}, {0, 1, 1}, {0, 0, 2}};
   (-2 ly lyz lz + lyy lz^2 - 2 lx (lxy ly + lxz lz) + lxx (ly^2 + lz^2) + ly^2 lzz + lx^2 (lyy + lzz))/
    (lx^2 + ly^2 + lz^2 /. {0. → 1.}),
   _, Message[EuclideanShorten::depth]; Throw[$Failed]]]

EuclideanShorten::depth = "This DiffusionEquation accepts 2D or 3D data only.";
EuclideanShorten::parameters =
  "This DiffusionEquation needs at least 1 parameter, {σ}.";
```

■ Perona & Malik with $c = e^{-\frac{|\nabla L|^2}{k^2}}$ (PeronaMalik1)

```
PeronaMalik1[data_, param_] := Module[{lx, ly, lxy, lxx, lyy, σ, k},
  If[Length[param] ≥ 2, σ = param[[1]]; k = param[[2]],
   Message[PeronaMalik1::parameters]; Throw[$Failed]];
  Switch[ArrayDepth[data],
   2,
   {lx, ly, lxx, lxy, lyy} = GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}][data] & /@
     {{1, 0}, {0, 1}, {2, 0}, {1, 1}, {0, 2}};
   If[k == ∞, lxx + lyy, (E^(-(lx^2 + ly^2)/k^2) ((k^2 - 2 lx^2) lxx - 4 lx lxy ly + (k^2 - 2 ly^2) lyy))/k^2],
   3, {lx, ly, lz, lxx, lxy, lxz, lyy, lyz, lzz} =
    GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}, {0.5 σ^2, #[[3]]}][data] & /@
     {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {2, 0, 0},
      {1, 1, 0}, {0, 1, 1}, {0, 2, 0}, {0, 1, 1}, {0, 0, 2}};
   If[k == ∞, lxx + lyy + lzz, 1/k^2 (E^(-(lx^2 + ly^2 + lz^2)/k^2) ((k^2 - 2 lx^2) lxx + k^2 lyy -
        2 ly^2 lyy - 4 ly lyz lz - 4 lx (lxy ly + lxz lz) + k^2 lzz - 2 lz^2 lzz))],
   _, Message[PeronaMalik1::depth]; Throw[$Failed]]]

PeronaMalik1::depth = "This DiffusionEquation accepts 2D or 3D data only.";
PeronaMalik1::parameters =
  "This DiffusionEquation needs at least 2 parameters, {σ, k}.";
```

■ Perona & Malik with $c = \frac{1}{1+\frac{|\nabla L|^2}{k^2}}$ (PeronaMalik2)

```
PeronaMalik2[data_, param_] := Module[{lx, ly, lxy, lxx, lyy, σ, k},
  If[Length[param] ≥ 2, σ = param[[1]]; k = param[[2]],
   Message[PeronaMalik2::parameters]; Throw[$Failed]];
  Switch[ArrayDepth[data],
   2,
   {lx, ly, lxx, lxy, lyy} = GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}][data] & /@
     {{1, 0}, {0, 1}, {2, 0}, {1, 1}, {0, 2}};
   If[k == ∞, lxx + lyy, (k^2 (-4 lx lxy ly + lxx (k^2 - lx^2 + lx^2) + lx^2 lyy + (k^2 - ly^2) lyy))/(k^2 + lx^2 + ly^2)^2],
   3, {lx, ly, lz, lxx, lxy, lxz, lyy, lyz, lzz} =
    GaussianDerivative[{0.5 σ^2, #[[1]]}, {0.5 σ^2, #[[2]]}, {0.5 σ^2, #[[3]]}][data] & /@
     {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {2, 0, 0},
      {1, 1, 0}, {0, 1, 1}, {0, 2, 0}, {0, 1, 1}, {0, 0, 2}};
   If[k == ∞, lxx + lyy + lzz, 1/(k^2 + lx^2 + ly^2 + lz^2)^2
     (k^2 (k^2 lyy - ly^2 lyy - 4 ly lyz lz + lyy lz^2 - 4 lx (lxy ly + lxz lz) +
        lxx (k^2 - lx^2 + ly^2 + lz^2) + k^2 lzz + ly^2 lzz - lz^2 lzz + lx^2 (lyy + lzz)))],
   _, Message[PeronaMalik2::depth]; Throw[$Failed]]]

PeronaMalik2::depth = "This DiffusionEquation accepts 2D or 3D data only.";
PeronaMalik2::parameters =
  "This DiffusionEquation needs at least 2 parameters, {σ, k}.";
```

■ Integration Schemes (Method)

■ Forward Euler (ForwardEuler)

```
ForwardEuler[data_, t_, dt_, f_, param_] :=
 Module[
  {im = data},
  Do[im += f[im, param] dt, {Ceiling[t / dt]}];
  im
 ]
```

■ 4^{th} order Runge-Kutta (RungeKutta)

```
RungeKutta[data_, t_, dt_, f_, param_] :=
 Module[
  {im = data},
  Do[im +=
    Plus @@ ({1/6, 2/6, 2/6, 1/6} FoldList[f[im + #1 #2, param] &, f[im, param], {1/2, 1/2, 1}])
     dt, {Ceiling[t/dt]}];
  im
 ]
```

■ Main Function (GeometryDrivenDiffusion)

```
GeometryDrivenDiffusion[data_, t_, dt_, (opts___)?OptionQ] :=
 Catch[
  Module[
   {im = data, d, m},
   {d, param, m} = {DiffusionEquation, Parameters, Method} /. {opts} /.
     Options[GeometryDrivenDiffusion];
   Off[GaussianDerivative::scalefail];
   im = Chop[m[im, t, dt, d, param]];
   On[GaussianDerivative::scalefail];
   im
  ]
 ]
```

■ Closing Package

Closing the package and locking all the symbols.

```
End[]

SetAttributes[
  {GeometryDrivenDiffusion,
   EuclideanShorten,
   PeronaMalik1,
   PeronaMalik2,
   ForwardEuler,
   RungeKutta,
   DiffusionEquation,
   Parameters,
   Method},
  {Protected, ReadProtected}
 ];

EndPackage[]
```

MathVisionTools

Mathematical Prototyping in Medical Image Analysis

Markus A. van Almsick, Bart M. ter Haar Romeny, Edwin H. Bennink

Eindhoven University of Technology
Department of Biomedical Engineering
Biomedical Image Analysis Group

Initialization

```
<< MathVisionTools`;
<< Graphics`;

SetOptions[ListDensityPlot, Mesh → False, Frame → False, PlotRange → All];

SetDirectory["C:\\Documents and Settings\\All Users\\Application
    Data\\Mathematica\\Applications\\FrontEndVision\\Images"];
```

Introduction

Medical image analysis today is based on serious mathematics. More and more methods involve PDE's, linear algebra, complex transforms, optimization theory etc.. Higher mathematics finds its way into sophisticated and efficient applications. The flow of development of such algorithms passes typically through three stages:

1. The *design* stage (also called rapid prototyping): This is the creative stage and needs careful exploration of each processing steps, analysis ofsensitivity parameters, and thorough understanding of the mathematical, physical, and statistical concepts involved. In the design stage, algorithms are developed and tested on relatively small, but in some cases high-dimensional datasets. A high-level programming language can facilitate these tasks.

2. The *validation* stage: Speed and memory issues for clinical validation come into focus. Here, C++/C or Java routines are exploited, supported by specialized libraries, such as VTK and OpenGL.

3. The clinical *implementation* stage: in the final step the approved algorithms are molded into real clinical implementation where the methods are fixed and highly optimized for speed and memory usage, e.g. by low level implementation languages and hardware support (graphics cards, DSP boards, parallel processing).

This paper introduces the software environment *MathVisionTools* that is suited for the design stage, where the emphasis lies on mathematical modeling. One finds here a niche in the world of medical image analysis software that has not yet been claimed.

We are well aware of the long list of computer vision libraries available [1]. Efforts with a too small user base tend to disappear when the original author(s) discontinue the development. The open source code and data initiative, well

supported by MICCAI [2] and Nature Biology [3], provides a good impetus for lasting developments by giving the endeavor a critical mass. Next to users contribution, a large supporting body is essential, such as government grants (NLM/NIH ITK, Kitware VTK), research institute support (MEVIS MevisLab, Mayo Clinic Analyze) and industrial support (Wolfram's *Mathematica*, MathWorks Matlab, ITTVIS IDL) to just name a few.

This article is meant to promote the *MathVisionTools* software and to convince colleges of its advantages thereby strengthening the vitality of the user base and enlarging the scope of our and hopefully your software tools in the future.

High level algorithm design

Where to start? This question stands at the beginning of every large software development and in many cases the answer has been "from scratch". This is no longer an acceptable choice as software engineering evolves over time. To reach the stars, one has to stand on the shoulders of giants. Today's giants are high-level computer languages, libraries, integrated development environments, applications with plug-in capabilities and so on.
Thus, the question needs to be rephrased. What software platform is a good foundation for rapid prototyping in medical imaging and what are the criteria? We begin with the latter:

- Full symbolic manipulation capabilities for the mathematical design phase,
- Fast numerical functionality for the validation of algorithms,
- Full, high quality, and interactive graphic rendering,,
- Availability of advanced extensions such as wavelets, neural networks, PDE solvers, optimization etc. [8],
- Easy to learn and to use to keep the training under typically one week,
- Code interpretation instead of compilation for a fast coding - testing - debugging cycle,
- Integration of code and text in a WYSIWYG user-interface for easy code maintenance and proper documentation,
- Free choice of the programming paradigm, such as functional programming and rule-based pattern matching,
- Powerful commands to obtain very short code,
- Platform independence for wide acceptance,
- I/O routines for a wide range of data formats, including all standard image formats, DICOM, STL, VRML etc.,
- Internet support via WebService, MathML, XML and alike,
- Availability of GUI design (e.g. Java) to bridge the transition into the validation phase,
- Easy integration of external routines and libraries written in C, C++, Fortran, Java, etc.,
- Solid and professional industrial support, with good long term perspectives,
- Large user community, with news-groups, discussion forum and dedicated international and technical conferences.

Our answer to the above questions has been and still is the computer algebra software *Mathematica*, a high level computer language with a large collection of powerful commands and libraries. *Mathematica* (Wolfram Inc., Champaign, USA, www.wolfram.com) has developed to a degree that all the above requirements are met. We have implemented *Mathematica* as our major prototyping software since the start of our group in Sept. 2001, and have gained great speed in the development of new mathematical algorithms in computer vision. *Mathematica* has improved at great speed, notably since version 4 and especially version 5 (the current version is 5.2). For some this fact may have come unnoticed as they may have abandoned the program in its early years, when it was slow and memory hungry. Those who look again will find that many routines are now faster than the competition (e.g. the Inverse of a dense matrix with 1000 x 1000 real numbers takes 1.5 seconds on a 1.5 GHz 1GB labtop) and even rival dedicated graphics software. For full details of *Mathematica* see the Wolfram Technology Guide (http://www.wolfram.com/technology/guide/index.html).

We specifically did not choose Matlab, despite its large user base. We needed a full *symbolic* engine in combination with a seamless integrated *fast numeric* engine. With *Mathematica* this is the case. Function names are consistent and not abbreviated expediting the training process. Another important advantage is the professional front-end, enabling us to write and combine the derivation and implementation of source code very much like a scientific paper. Researcher's

and student's documentation naturally accompany the short code, which is an essential element for research groups. Experience has shown that long and poorly documented code is unreadable for successors and destined to be thrown away. Naturally, this paper is written as a *Mathematica* 'notebook' in this front-end.

Level	Language	Purpose
High	*Mathematica*	Design, Education
Medium	C++, C, Java VTK, OpenGL	Evaluation, Visualization
Low	C#, C++	Graphics cards

There is an active *Mathematica* user's community [4] (there are 2 million licenses worldwide), with a newsgroup (comp.soft-sys.math.mathematica), technology conferences and national user meetings (www.wolfram.com/news/), and an *International Mathematica Symposium* series [6] (edition 2008 will be organized by us in Maastricht, the Netherlands). Wolfram hosts the famous *MathWorld* website [5], a leading mathematics reference on the internet with templates of advanced *Mathematica* code fragments for developer's *Mathematica* notebooks.

This paper presents *MathVisionTools*, a new *Mathematica*-library or add-on with advanced tools for mathematical image analysis and algorithm design. This library is open for international collaboration.

MathVisionTools

MathVisionTools [9, 10] is a *Mathematica* Add-On for the fast prototyping of biomedical image analysis algorithms. This growing library of high-level imaging tools has been initiated in 2004 [10] and is managed by the Biomedical Image Analysis group of the Department of Biomedical Engineering at Eindhoven University of Technology, Eindhoven, the Netherlands. The purpose of *MathVisionTools* is to provide the developer with more and more powerful commands and to host reusable code fragments and routines that are specifically needed for image analysis and processing.

Starting point has been multi-scale differential calculus of images within the framework of scale space theory [19, 13, 14, 17]. Extensions have been I/O routines for biomedical image formats such as DICOM and Kretz ultrasound. Fourier transformations in polar coordinates and invertible transformations into orientation bundles (filter responses on the Euclidean group-manifold) have been the latest addition. Dedicated applications for biomedical imaging such as multi-modality image registration, multi-scale optic flow detection on 2D-time image sequences, and methods for computer-aided diagnosis are currently developed in ongoing research projects, based on routines available in *MathVisionTools.*

Mathematica and *MathVisionTools* also play a key role in the education of our Biomedical Engineering students [11, 12]. The strategy of 'Here is a classical paper, read it, understand it, and make an implementation in *Mathematica* in a few days' works amazingly well, even for 3 months projects. For a range of examples see http://www.bmi2.bmt.tue.nl/image-analysis/Education/index.html (Master & Internship). See also the student example exhibited by Wolfram: http://library.wolfram.com/infocenter/Conferences/5756/ . A large subset of routines and commands in *MathVisionTools* stems from the many functions released in textbooks on multi-scale image analysis [13, 14]. The course "Front-End Vision and Multi-Scale Image Analysis" (http://www.bmi2.bmt.tue.nl/image-analysis/education/courses/FEV/course/index.html) is a popular national course in the Netherlands and is completely given in *Mathematica* [13].

Many Digital Image Processing toolkits are rather basic and contain the elementary operations on images, such as filters, geometric transformations, histogram operations, mathematical morphology and edge detection. Beyond this, there is a need for an efficient design toolkit with high-level building blocks for doing advanced mathematics on images. To emphasize this issue we provide a short list of exemplary needs for mathematical methodologies:

- evolutionary and energy minimizing methods in image enhancement,
- high order robust differential geometry (invariants),
- calculations on matrix valued images (DTI, hessian),
- multi-scale methods (deep structure singularities),
- multi-orientation methods (perceptual grouping, tensor voting, stochastic completion fields),
- texture analysis and statistical pattern recognition tools,
- statistical pattern recognition techniques (!!! not the same as above?),
- dynamic shapes (snakes, balloons),
- active shape and appearance models,
- retrieval methods for similar images in huge databases,
- robust analysis of optic flow dynamics,
- etc.

Mathematica is ideally suited to handle this wide range of mathematics on images. It is intrinsically multi-dimensional. It embeds any programming style, but is primarily designed for functional programming, where it as an interpreter language exhibits its greatest speed. A clear advantage of functional programming is also the typically short code, often resembling closely the verbal English statement of the problem. A full debugger/profiler is now available [7]. It is easy to install C++ routines into *Mathematica* code and vice-versa through the *MathLink* protocol (co-compile C/C++ code with *MathLink*.h).

Mathematica reads and writes any type of image, including medical DICOM images. In the package *MathVisionTools* we have developed many special I/O routines to convert a variety of vendor-specific data into DICOM format, such as BioRAD microscope data, high field small-bore small animal MRI data (FDF format), data in ANALYZE format etc.

Mathematica is not freeware. This may hamper its proliferation and the proliferation of *MathVisionTools*. However, the commercial embedding ensured ongoing R&D for the sake of *Mathematica,* its add-ons, and its users. It is a highly professional software environment, not just for image analysis, which is supported by a well established, mid-size company. The costs for image analysis researchers can be spread by a full campus or institute license, which is now available to most large universities worldwide. In Eindhoven a universal license for all departments permits installation of *Mathematica* on all computers of the university, including the 9600 laptops supplied with 50% funding to all TU/e students, including home use. The cost per computer is thus reduced to a few euros. This requires a decision at top university /institute level. See for the TU/e situation: http://w3.tue.nl/en/services/dienst_ict/organisatie/groepen/wins/campus_software/.

Example I: Differential Calculus on Images

According to scale-space theory [13, 19, 20] and tempered distribution theory [14] it is well known that a regularized way to take (high-order) derivatives of discrete data $L[x, y]$ is by convolution with a Gaussian derivative kernel $G_t^{(n_x,n_y)}$.

$$\partial_x L[x, y] = \iint G_t[x-\tilde{x}, y-\tilde{y}]\ (\partial_x L[\tilde{x}, \tilde{y}])\, d\tilde{x}\, d\tilde{y} = \iint G_t^{(1,0)}[x-\tilde{x}, y-\tilde{y}]\ L[\tilde{x}, \tilde{y}]\ d\tilde{x}\, d\tilde{y}.$$

The above equation opens an operational way to apply practically all tools of differential calculus to discrete images. *MathVisionTools* contains an extensive `GaussianDerivative` package to calculate partial derivatives to any order, both symbolically on many function or numerically on any *n*-dimensional data set, under a variety of boundary condi-

tions, in a fast and highly optimized way, either via spatial convolution (`ListConvolve`) or via multiplication in Fourier space. Here are three examples that built onto Gaussian derivatives.

■ Example: Gauge Derivatives

A famous class of differential invariants is the N-jet of intrinsic image derivatives, expressed in the local, first-order, gradient-defined coordinate system {*v*, *w*}, the so-called gauge derivatives [13]. The following short *Mathematica* code renders the gauge derivatives of any order (*nv*-times with respect to the unit-gradient coordinate *v* and *nw*-times with respect to the orthogonal coordinate *w*) of a 2D function $f[x, y]$ in terms of {*x*, *y*}-coordinates.

```
GaugeDerivative[nv_, nw_][f_] :=
 Module[
  {Lx, Ly, v, w}, w = {Lx, Ly} / Sqrt[Lx^2 + Ly^2]; v = {{0, 1}, {-1, 0}}.w;
  Nest[(v.{∂x#, ∂y#} &), Nest[(w.{∂x#, ∂y#} &), f, nw], nv] /.
    {Lx → D[f, x], Ly → D[f, y]} // Simplify
 ]
```

```
GaugeDerivative[0, 1][L[x, y]]
```

$$\sqrt{L^{(0,1)}[x, y]^2 + L^{(1,0)}[x, y]^2}$$

"Ridgeness" is given by L_{vv}:

```
GaugeDerivative[2, 0][L[x, y]]
```

$$\left(L^{(0,2)}[x, y]\, L^{(1,0)}[x, y]^2 - 2\, L^{(0,1)}[x, y]\, L^{(1,0)}[x, y]\, L^{(1,1)}[x, y] + L^{(0,1)}[x, y]^2\, L^{(2,0)}[x, y]\right) \Big/ \left(L^{(0,1)}[x, y]^2 + L^{(1,0)}[x, y]^2\right)$$

For easier reading, one can convert the above expression into subscript notation via pattern matching in a single statement:

```
% /. f_^(nx_,ny_)[x_, y_] :> f_StringJoin[Table["x",{nx}],Table["y",{ny}]]
```

$$\frac{-2\, L_x\, L_{xy}\, L_y + L_{xx}\, L_y^2 + L_x^2\, L_{yy}}{L_x^2 + L_y^2}$$

The next statement exploits *Mathematica*'s powerful pattern matching to *replace* (with the operator /.) any occurrence of an analytical derivative into a discrete convolution operator. Thus, in one line we write the complete discrete implementation for *any* order:

```
ListGaugeDerivative[t_, nv_, nw_][img_] :=
 GaugeDerivative[nv, nw][L[x, y]] /.
  L^(nx_,ny_)[x, y] :> GaussianDerivative[{t, nx}, {t, ny}][img]
```

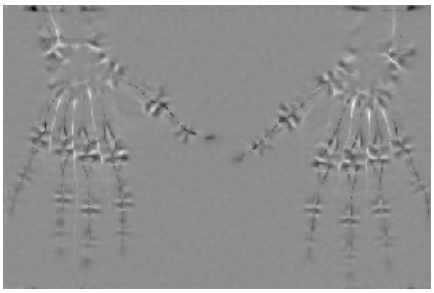
The gradient L_w of an X-ray image at scale t=2:

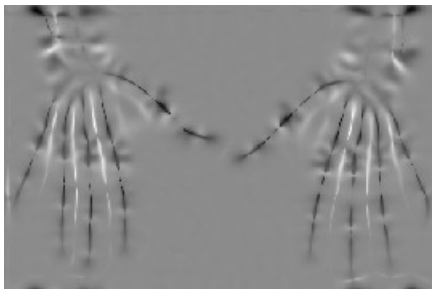
```
im = Import["hands.gif"][[1, 1]];
```

```
ListDensityPlot[ListGaugeDerivative[2, 0, 1][im]];
```

Ridges L_{vv} at scale $t = 2$ and $t = 8$:

```
DisplayTogetherArray[ListDensityPlot /@
   {ListGaugeDerivative[2, 2, 0][im], ListGaugeDerivative[8, 2, 0][im]}];
```





■ Example: Gaussian Deblurring

To deblur Gaussian blur [18], one can extrapolate an image *L[x,y,t0]* at scale *t0* towards smaller *t* via a Taylor expansion [2][3]. First, we import an MRI image of a head from the first frame of a DICOM dataset. (!!! its a GIF file here, not DICOM .. do you have a DICOM version)

```
img = Import["mr128.gif"][[1, 1]];
```

We then perform a Taylor expansion of $L[x, y, t]$ with respect to scale t_0 around t_0 and substitute the derivatives ∂_t by the Laplacian operator $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$, which is equal to ∂_t according to the diffusion equation $\frac{\partial L}{\partial t} = \frac{\partial^2 L}{\partial x^2} + \frac{\partial^2 L}{\partial y^2}$ that governs linear scale space. Thus, we obtain a differential deblurring operator.

```
BlurExpansion[order_] := Series[L[x, y, t], {t, t0, order}] /.
  L^(0,0,n_)[x, y, t0] :> Nest[(∂x,x# + ∂y,y#) &, L[x, y, t0], n]; BlurExpansion[4]
```

$$L[x, y, t0] + (L^{(0,2,0)}[x, y, t0] + L^{(2,0,0)}[x, y, t0])\,(t - t0) + \frac{1}{2}(L^{(0,4,0)}[x, y, t0] + 2\,L^{(2,2,0)}[x, y, t0] + L^{(4,0,0)}[x, y, t0])\,(t - t0)^2 + \frac{1}{6}(L^{(0,6,0)}[x, y, t0] + 3\,L^{(2,4,0)}[x, y, t0] + 3\,L^{(4,2,0)}[x, y, t0] + L^{(6,0,0)}[x, y, t0])\,(t - t0)^3 + \frac{1}{24}(L^{(0,8,0)}[x, y, t0] + 4\,L^{(2,6,0)}[x, y, t0] + 6\,L^{(4,4,0)}[x, y, t0] + 4\,L^{(6,2,0)}[x, y, t0] + L^{(8,0,0)}[x, y, t0])\,(t - t0)^4 + O[t - t0]^5$$

We substitute the `GaussianDerivative` command for the symbolic derivatives of $L[x, y, t]$ to obtain an instantly working deblurring command.

```
Deblur[img_, order_, gt_, Δt_] := Normal[BlurExpansion[order]] /.
  {L[x, y, t0] :↦ img,
   L^(nx_,ny_,0)[x, y, t0] :↦ GaussianDerivative[{gt, nx}, {gt, ny}][img],
   (t - t0) → (Δt - gt)}
```

gt denotes the scale, that is inflicted by the Gaussian derivatives. *Δt* stands for the scale interval, by which to deblur. First, we generate a blurred image at scale $t = 2$.

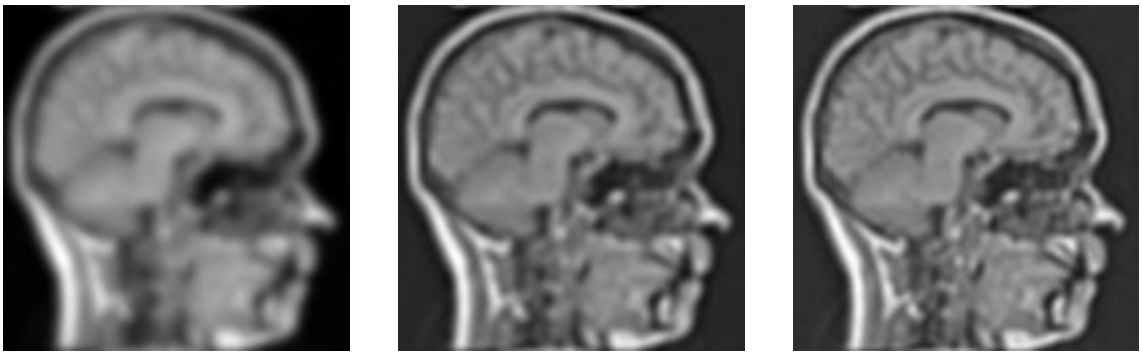
```
blur = GaussianDerivative[{2, 0}, {2, 0}][img];
```

Then, we deblur it by $\Delta t = -2$.

```
deblur4 = Deblur[blur, 4, 2.0, -2];
deblur8 = Deblur[blur, 8, 2.7, -2];
```

The result speaks for itself. Note, that we can rerun this example for any order of Taylor expansion generating the *Mathematica* code on the fly.

```
DisplayTogetherArray[ListDensityPlot /@ {blur, deblur4, deblur8}];
```



Example II: Geometry-driven diffusion

The Perona & Malik equation (edge preserving smoothing evolution PDE [16, 15]) for 3D is given by

$$\frac{\partial L}{\partial s} = \vec{\nabla}.\left(e^{-\frac{|\vec{\nabla}L|^2}{k^2}}\,\vec{\nabla}L\right).$$

The exponential term is the conductivity and is a decreasing function of the gradient magnitude. After loading a standard *Mathematica* package to obtain the commands `Grad` for the gradient and `Div` for the divergence, this formula for 3 dimensions is easily expanded and implemented for discrete images.

```
<< Calculus`VectorAnalysis`;
SetCoordinates[Cartesian[x, y, z]];

Div[E^-(Grad[L[x, y, z]]^2/k^2) Grad[L[x, y, z]]] // ExpToTrig // FullSimplify
```

$$\frac{1}{k^2}\left(e^{-\frac{L^{(0,0,1)}[x,y,z]^2}{k^2}}\left(k^2-2\,L^{(0,0,1)}[x,y,z]^2\right)L^{(0,0,2)}[x,y,z]+ e^{-\frac{L^{(0,1,0)}[x,y,z]^2}{k^2}}\left(k^2-2\,L^{(0,1,0)}[x,y,z]^2\right)L^{(0,2,0)}[x,y,z]+ e^{-\frac{L^{(1,0,0)}[x,y,z]^2}{k^2}}\left(k^2-2\,L^{(1,0,0)}[x,y,z]^2\right)L^{(2,0,0)}[x,y,z]\right)$$

As we have seen before, the derivatives are conveniently converted into subscript notation via pattern matching in a single statement.

```
% /. L^(nx_,ny_,nz_)[x, y, z] :> L_StringJoin[{Table["x",{nx}], Table["y",{ny}], Table["z",{nz}]}]
```

$$\frac{e^{-\frac{L_x^2}{k^2}}\,(k^2 - 2\,L_x^2)\,L_{xx} + e^{-\frac{L_y^2}{k^2}}\,(k^2 - 2\,L_y^2)\,L_{yy} + e^{-\frac{L_z^2}{k^2}}\,(k^2 - 2\,L_z^2)\,L_{zz}}{k^2}$$

Via pattern matching we can convert these derivatives to discrete Gaussian derivative operators as explained in the previous section, and we can applied them in e.g. a forward Euler, Runge-Kutta or AOS [22] iteration scheme.

Advanced mathematics

MathVisionTools is *not* a package that implements the basic image processing routines, as are available in many other packages. It is dedicated to be a design tool for advanced mathematical reasoning and experimentation, and it is destined to fill that niche. Current directions of research and development in *MathVisionTools* are invariant stochastic processes of medical image data in scale space and on the Euclidean manifold to obtain perceptual grouping measures.

The appendices list examples of code in *MathVisionTools* and some pages from the Help Browser in *Mathematica* for *MathVisionTools* routines. We just scratched the surface of the vast possibilities in image calculus that is now accessible through *MathVisionTools* and *Mathematica*. Applications currently being built in *MathVisionTools* are:

- multi-scale optic flow detection on 2D-time cardiac image sequences [29];
- multi-scale singularity techniques for robust scene retrieval in CAD [25];
- active shape and active appearance algorithms for cardiac shape variability analysis [26];
- detection of stellate tumors in mammography (based on [30]);
- geometry-driven diffusion techniques for enhancement (edge-preserving smoothing) [13, 15];
- invertible orientation-spaces for dim contour enhancement (orientation scores [27]);
- perceptual grouping of elongated structures through stochastic completion fields [21];
- 2D tensor voting with steerable filters for catheter detection in low-dose fluoroscopy[23];
- 3D radial basis functions interpolation for neuronavigation atlas matching [24];
- and many more in development.

Conclusion and call

Mathematica has proven to be an ideal prototyping tool in advanced image analysis. The routines and commands collected in *MathVisionTools* accelerate the process of developing new algorithms. We call for participation by interested institutes and companies.
Participation will be on an exchange basis. The code will be made available to partners that contribute code to the project.

Please contact:

Prof. Bart M. ter Haar Romeny, Dipl. Ing. Markus van Almsick, H.E. Bennink
Eindhoven University of Technology
Department of Biomedical Engineering
Den Dolech 2, WH2.106
5600 MB Eindhoven, the Netherlands
Tel. +31-40-2475537

Fax +31-40-2472740
Email: B.M.terHaarRomeny@tue.nl, M.v.Almsick@tue.nl, H.E.Bennink@student.tue.nl
URL BMIA: http://www.bmi2.bmt.tue.nl/image-analysis/
URL MathVisionTools: www.mathvisiontools.net

References

[1] Computer Vision Homepage - Software: http://www.cs.cmu.edu/afs/cs/project/cil/www/v-source.html.

[2] MICCAI conferences: http://www.miccai.org.

[3] Nature Biology: Need for open source and data: http://www.nature.com/nbt/journal/v22/n8/full/nbt0804-1037.html;jsessionid=7D3D4BCDAFD331B20-E0226093857ABAA

[4] *Mathematica* users: http://www.mathematica-users.org/webMathematica/wiki/wiki.jsp.
See also: http://forums.wolfram.com/.

[5] MathWorld: http://mathworld.wolfram.com/

[6] International *Mathematica* Symposium: http://internationalmathematicasymposium.org/IMS2006/

[7] *Mathematica* WorkBench (debugger/profiler): http://www.wolfram.com/news/workbenchprerelease.html

[8] *Mathematica* packages: http://www.wolfram.com/products/field_specific.html

[9] MathVisionTools website: http://www.bmi2.bmt.tue.nl/image-analysis/Research/Software/Mathematica/AddOns/MathVisionTools/index.html

[10] M. A. van Almsick, B. M. ter Haar Romeny, "MathVisionTools, the design of a new framework for biomedical image analysis", Wolfram Developers Conference 2004, Urbana-Champaign.

[11] B. M. ter Haar Romeny, "Computer Vision and Mathematica 4", Computing and Visualization in Science, vol. 5, no. 1, pp. 53-65, Springer, 2002. PDF (1.7MB), Mathematica 4 Notebook (3.1MB).

[12] B.M. ter Haar Romeny, M.A. van Almsick, "Rapid prototyping of biomedical image analysis applications with Mathematica". Proc. Medicon 2004, Ischia, Italy. MS-Word, PDF.

[13] B.M.ter Haar Romeny,"Front-End Vision and Multi-Scale Image Analysis. Multi-Scale Computer Vision Theory and Applications, written in Mathematica". Springer, 2003.

[14] L. M. J. Florack, "Image Structure", Dordrecht: Kluwer Academic Publishers, 2001.

[15] B. M. ter Haar Romeny (ed.), "Geometry-driven diffusion in computer vision". Dordrecht: Kluwer Academic Publishers, 1994.

[16] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion", IEEE Tr. on Pattern Analysis and Machine Intelligence, vol. 12, pp. 629-639, July 1990.

[17] B. M. ter Haar Romeny, L. M. J. Florack, "Front-End Vision, a Multiscale Geometry Engine". Proc. First IEEE International Workshop on Biologically Motivated Computer Vision (BMCV2000), May 15-17, 2000, Seoul, Korea. Lecture Notes in Computer Science vol. 1811, pp. 297-307, Springer-Verlag, Heidelberg, Germany, 2000. PDF (1.7MB)

[18] B. M. ter Haar Romeny, L. M. J. Florack, M. de Swart, J. Wilting, and M. A. Viergever, "Deblurring Gaussian blur," in Proceedings Mathematical Methods in Medical Imaging II, vol. 2299, (San Diego, CA), pp. 139-148, SPIE, July, 25-26 1994.

[19] J.J.Koenderink,"The structure of images",Biological Cybernetics,vol.50,pp.363-370,1984.

[20] L.M.J.Florack, B.M.ter Haar Romeny, J.J.Koenderink and M.A.Viergever, "Linear scale-space", Journal of Mathematical Imaging and Vision,vol.4,no.4,pp.325-351,1994.

[21] M.A. van Almsick, R. Duits, E.M. Franken, B.M. ter Haar Romeny, From Stochastic Completion Fields to Tensor Voting , Lecture Notes in Computer Science, 3753, 124-134, 2005.

[22] J. Weickert, B. M. ter Haar Romeny, and M. A. Viergever, "Efficient and reliable schemes for nonlinear diffusion filtering," IEEE Transactions on Image Processing, vol. 7, no.3, pp. 390-410, 1998.

[23] E.M. Franken, M.A. van Almsick, P.M.J. Rongen, L.M.J. Florack, B.M. ter Haar Romeny, Steerable Tensor Voting, in ASCI Conference 2005; Heijen, the Netherlands, 65-72, (2005)

[24] J. Korbeeck, B. Janssen, E. H. Bennink, A. Jansen, M. Koppert, R. Lahaije, T. Plantenga, B. M. ter Haar Romeny, Warping a neuro-anatomy atlas on 3D MRI data with Radial Basis Functions. Proc. 8th Intern. *Mathematica* Symposium, Avignon, France, 2006.

[25] B. Platel, E. Balmachnova, L.M.J. Florack, F.M.W. Kanters, B.M. ter Haar Romeny, Using Top-Points as Interest Points For Image Matching, in Deep Structure, Singularities, and Computer Vision; Editors: O. Fogh Olsen, L. M. J. Florack and A. Kuijper, Maastricht, Netherlands, 211 - 222, 2005.

[26] H.C. van Assen, M.G. Danilouchkine, A.F. Frangi, S. Ordás, J.J.M. Westenberg, J.H.C. Reiber, B.P.F. Lelieveldt, “SPASM: a 3D-ASM for Segmentation of Sparse and Arbitrarily Oriented Cardiac MRI Data, Med. Image Analysis, 10(2), 286-303, 2006.

[27] R. Duits, M. Duits, M.A. van Almsick, L.M.J. Florack, A new reconstruction from orientation bundle functions as an application of generalized wavelet theory, in Proc. Early Cognitive Vision Workshop 2004, Isle of Skye, 2004.

[28] B. M. ter Haar Romeny, B. Titulaer, S. Kalitzin, G. Scheffer, F. Broekmans and E. te Velde, "Computer assisted human follicle analysis for fertility prospects with 3D ultrasound", Proceedings Intern. Conf. on Information processing in Medical Imaging (IPMI '99), vol. 1613, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 1999. PDF.

[29] A. Suinesiaputra, L. M. J. Florack, J. J. M. Westenberg, B. M. ter Haar Romeny, J. H. C. Reiber, and B. P. F. Lelieveldt. "Optic flow computation from cardiac MR tagging using a multiscale differential method-a comparative study with velocity-encoded MRI". In Proc. MICCAI 2003, Montréal, Canada, Lecture notes in computer science, 2878, 483-490, 2003.

[30] N. Karssemeijer, G. M. te Brake, "Detection of Stellate Distortions in Mammograms", IEEE Tr. on Medical Imaging, vol. 15, no. 5, 611-619, Oct. 1996. PDF

MathVisionTools: OrientationBundleTransform

■ `OrientationBundleTransform[`*img*`,`*t*`,`α`,`*n*`]` generates an orientation bundle of image *img* with a Kalitzin kernel of scale *t* with *n* orientations. α is the ratio between angular and radial units.

■ `OrientationBundleTransform` is the convolution of an image with the orientation kernel, which is done via multiplication in Fourier space: $\mathcal{F}(\psi(r,\varphi))(\omega) * \mathcal{F}(image)(\omega) = \sum_{m=0}^{\infty} \sqrt{\frac{t^m}{m!}}\, e^{i\,m\,\omega_\varphi}\, {\omega_r}^{\alpha\,m} \left(e^{-t\,\frac{\omega_r^{2\,\alpha}}{2}} * \mathcal{F}(image)(\omega)\right)$.

■ The following options can be given:

`DirectionalFourierFilter {1,1,1,…}`	assigns a `List` of weighting coefficients $\{w_0,\ w_1,\ w_2\}$.

■ See also: `InverseOrientationBundleTransform`, and `DirectionalFourierFilter`.

■ New in Version 2.

Further Examples

This loads the *MathVisionTools* and the Graphics`Graphics` packages.

```
In[1]:= << Graphics`Graphics`
        << MathVisionTools`
```

Loading a 2D-image.

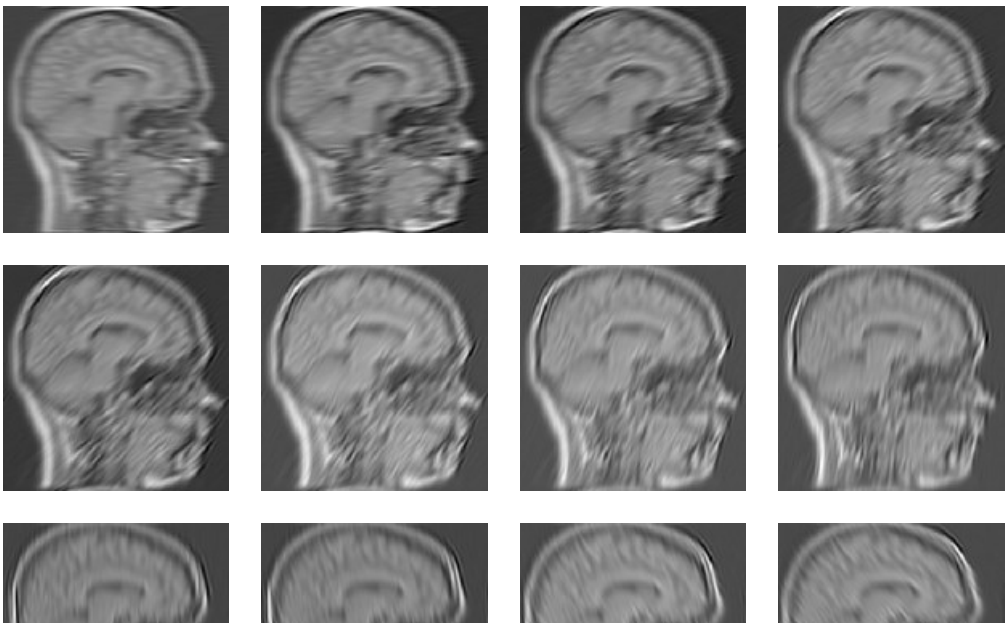
```
In[3]:= img = Import["mr256.jpg"][[1, 1]];
```

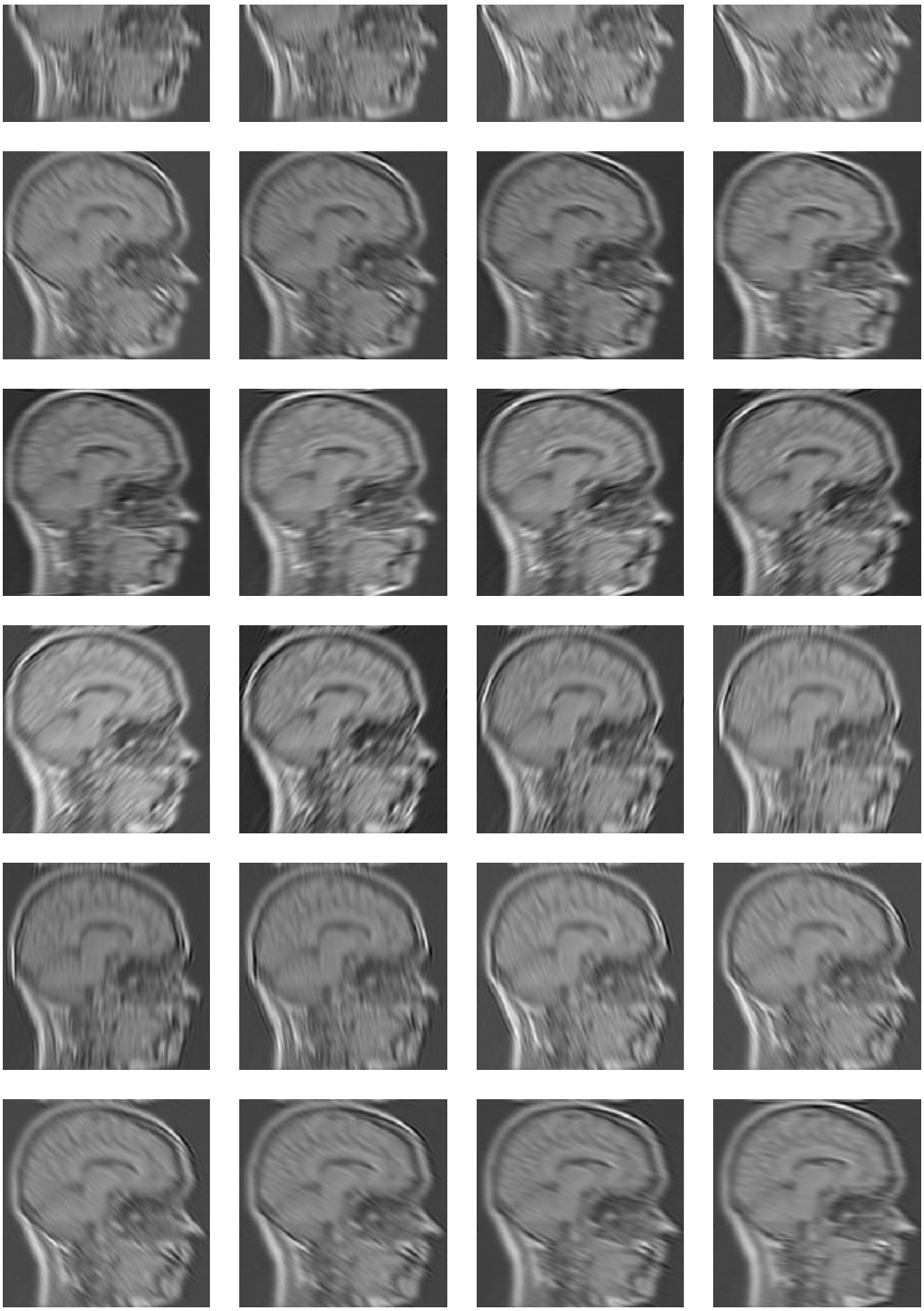
Generating an orientation bundle.

```
In[4]:= bndl = OrientationBundleTransform[img, 16, 1.2, 32];
```

Displaying the orientation bundle.

```
In[5]:= DisplayTogetherArray[
          Partition[Map[ListDensityPlot[#, Mesh → False, Frame → False] &, Re[bndl]], 4],
          ImageSize → 400];
```

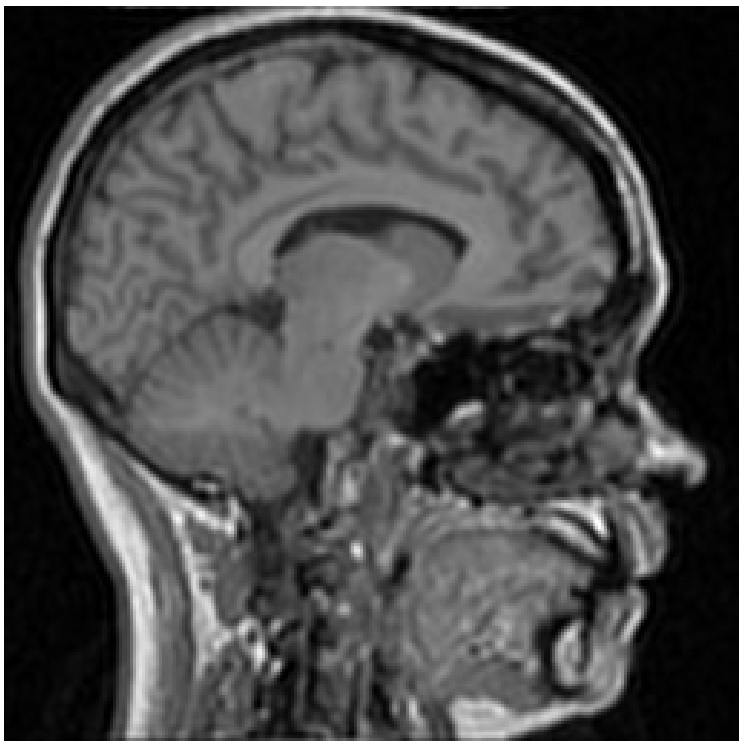




Reconstructing the original image from the orientation bundle.

```
In[6]:= newimg = InverseOrientationBundleTransform[bndl, 16, 1.2];
```

```
In[7]:= ListDensityPlot[newimg, Mesh → False, Frame → False, PlotRange → {0, 255}];
```



Reconstructing a modified orientation bundle.

```
In[8]:= newimg = InverseOrientationBundleTransform[Im[bndl]^4, 16, 1.2];
```

```
In[9]:= ListDensityPlot[Re[newimg], Mesh → False, Frame → False, PlotRange → All];
```

```
In[10]:= Clear[img, bndl, newimg]
```