5. Multi-scale derivatives: implementations

Three people were at work on a construction site. All were doing the same job, but when each was asked what the job was, the answers varied. "Breaking rocks," the first replied. "Earning my living," the second said. "Helping to build a cathedral," said the third.
-Peter Schultz

In order to get a good feeling for the interactive use of *Mathematica*, we discuss in this section three implementations of convolution with a Gaussian derivative kernel (in 2D) in detail:
1. implementation in the spatial domain with a 2D kernel;
2. through two sequential 1D kernel convolutions (exploiting the separability property);
3. implementation in the Fourier domain.
Just blurring is done through convolution with the zero order Gaussian derivative, i.e. the Gaussian kernel itself.

5.1 Implementation in the spatial domain

Mathematica 4 has a fast implementation of a convolution: **`ListConvolve[kernel, list]`** forms the convolution of the kernel **`kernel`** with **`list`**. This function is N-dimensional, and is internally optimized for speed. It can take any *Mathematica* expression, but its greatest speed is for **`Real`** (floating) numbers. We first define the 1D Gaussian function **`gauss[x,σ]`**:

```
<< FrontEndVision`FEV`;
Unprotect[gauss];
```

$$\texttt{gauss[x\_, σ\_ /; σ > 0] :=}\ \frac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{x^2}{2\sigma^2}}\ ;$$

We explain in detail what happens here:
The function **`gauss[x_, σ_]`** is defined for the variables **`x_`** and **`σ_`**. The underscore **`_`** means that **`x_`** is a **`Pattern`** with the name **`x`**, it can be anything. This is one of the most powerful features in *Mathematica*: it allows pattern matching. In the appendix a number of examples are given. The variable **`σ_`** has the condition (indicated with **`/;`**) that σ should be positive. If this condition is not met, the function will not be evaluated. The function is defined with delayed assignment (**`:=`** in stead of **`=`** for direct assignment). In this way it will be evaluated only when it is called. The semicolon is the separator between statements, and in general prevents output to the screen, a handy feature when working on images.

The function **`gDc[im,nx,ny,σ]`** implements the same function in the spatial domain. The parameters are the same as above. This function is much faster, as it exploits the internal function **ListConvolve**, and applies Gaussian derivative kernels with a width truncated to +/- 4 standard deviations, which of course can freely be changed.

```
gDc[im_, nx_, ny_, σ_ /; σ > 0] := Module[{x, y, kernel},
  kernel = N[Table[Evaluate[
        D[gauss[x, σ] * gauss[y, σ], {x, nx}, {y, ny}]],
       {y, -4 * σ, 4 * σ}, {x, -4 * σ, 4 * σ}]];
    ListConvolve[kernel, im, Ceiling[Dimensions[kernel] / 2]]];
```

Module[{vars}, ...] is a construct to make a block of code where the vars are shielded from the global variable environment. The derivative of the function **gauss[]** is taken with **D[f,{x,nx},{y,ny}]** where **nx** is the number of differentiations to **x** and **ny** the number of differentiations to **y**. The variable **kernel** is a **List**, generated by the **Table** command, which tabulates the function **gauss[]** over the range $\pm 4\sigma$ for both **x** and **y**. The derivative function must be evaluated with **Evaluate[]** before it can be tabulated. The function **N[]** makes the result a numerical value, a **Real** number.

ListConvolve is an optimized internal *Mathematica* command, that *cyclically* convolves the kernel **kernel** with the image **im**. The **Dimensions[]** of the kernel are a **List** containing the x- and y-dimension of the square kernel matrix. Finally, the upwards rounded (**Ceiling**) list of dimensions is used by **ListConvolve** to fix that the kernel starts at the first element of **im** and returns an output image with the same dimension as the input image.

```
im = Table[If[x^2 + y^2 < 7000, 100, 0], {x, -128, 127}, {y, -128, 127}];
Block[{$DisplayFunction = Identity},
  p1 = ListDensityPlot[#] & /@ {im, gDc[im, 1, 0, 1]}];
Show[GraphicsArray[p1], ImageSize -> 350];
```

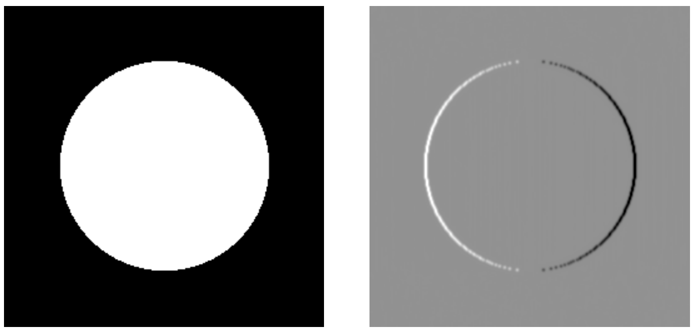

Figure 5.1 The derivative to x (right) at scale $\sigma = 1$ pixel on a 256^2 image of a circle (left).

The wider the kernel, the more points we include for calculation of the convolution, so the more computational burden we get. When the kernel becomes wider than half of the domain of the image, it becomes more efficient to apply the Fourier implementation discussed below. This trade-off has been worked out in detail by Florack [Florack2000a].

5.2 Separable implementation

The fastest implementation exploits the separability of the Gaussian kernel, and this implementation is mainly used in the sequel:

```
Options[gD] = {kernelSampleRange → {-6, 6}};
gD[im_List, nx_, ny_, σ_, (opts___)?OptionQ] :=
 Module[{x, y, kpleft, kpright, kx, ky, mid, tmp},
  {kpleft, kpright} = kernelSampleRange /. {opts} /. Options[gD];
  kx = N[Table[Evaluate[D[gauss[x, σ], {x, nx}]],
    {x, kpleft * σ, kpright * σ}]];
  ky =
    If[nx == ny, kx, N[Table[Evaluate[D[gauss[y, σ], {y, ny}]],
    {y, kpleft * σ, kpright * σ}]]]; mid = Ceiling[Length[#1] / 2] & ;
  tmp =
    Transpose[ListConvolve[{kx}, im, {{1, mid[kx]}, {1, mid[kx]}}]];
  Transpose[ListConvolve[{ky}, tmp, {{1, mid[ky]}, {1, mid[ky]}}]]];
```

The function `gD[im, nx, ny, σ, options]` implements first a convolution per row, then transposes the matrix of the image, and does the convolution on the rows again, thereby effectively convolving the columns of the original image. A second `Transpose` returns the image back to its original orientation. This is the default implementation of multi-scale Gaussian derivatives and will be used throughout his book.

```
im = Table[If[x^2 + y^2 < 7000, 100, 0], {x, -128, 127}, {y, -128, 127}];
Timing[imx = gD[im, 0, 1, 2]][[1]]

0.031 Second

Block[{$DisplayFunction = Identity},
  p1 = ListDensityPlot[#] & /@ {im, imx}];
Show[GraphicsArray[p1], ImageSize -> 260];
```

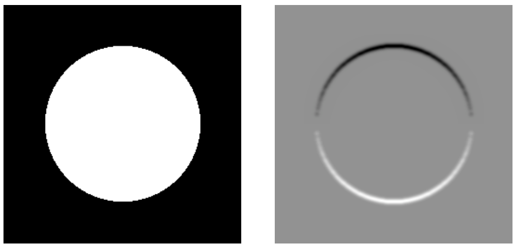

Figure 5.2 The derivative to y (right) at scale $\sigma = 2$ pixels on a 256^2 image of a circle (left).

▲ Task 5.1 Write a *Mathematica* function of the separable Gaussian derivative kernel implementation for 3D. Test the functionality on a 3D test image, e.g. a sphere.

5.3 Some examples

Convolving an image with a single point (a delta function) with the Gaussian derivative kernels, gives the kernels themselves., i.e. the pointspread function. E.g. here is the well known series of all Cartesian partial Gaussian derivatives to 5th order:

```
spike = Table[0., {128}, {128}]; spike[[64, 64]] = 1.;
Block[{$DisplayFunction = Identity},
  array = Table[Table[ListDensityPlot[gD[spike, m - n, n, 20],
      PlotLabel -> "∂x = " <> ToString[m - n] <> ", ∂y = " <> ToString[n]],
     {n, 0, m}], {m, 0, 5}]];
Show[GraphicsArray[array], ImageSize → 330];
```

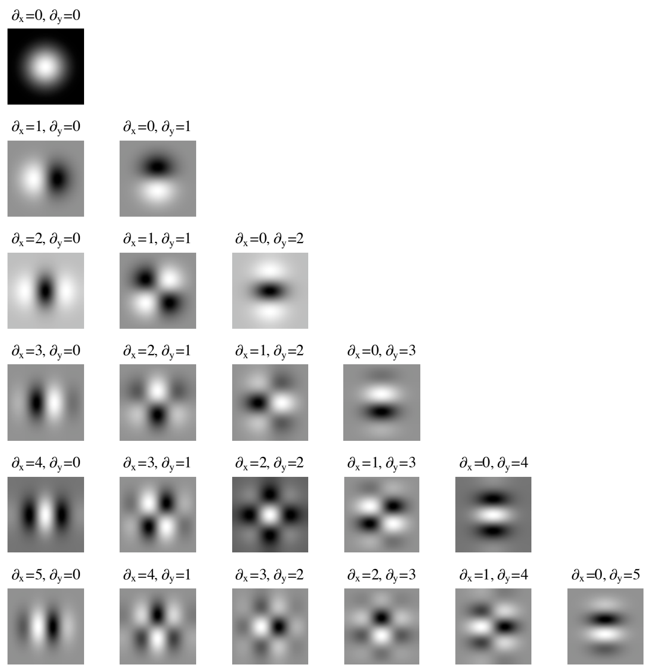

Figure 5.3 Gaussian partial derivative kernels up to 5th order.

`$DisplayFunction` is the internal variable that determines how things should be displayed. Its normal state (it default has the value **`Display[$Display,#1]&`**) is to send PostScript to the output cell. Its value is temporarily set to **`Identity`**, which means: no output. This is necessary to calculate but not display the plots.

We read an image with **`Import`** and only use the first element **`[[1,1]]`** of the returned structure as this contains the pixeldata.

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

We start with just blurring at a scale of $\sigma = 3$ pixels and show the result as 2D image and 3D height plot:

```
DisplayTogetherArray[
  {ListDensityPlot[gD[im, 0, 0, 3]], ListPlot3D[gD[im, 0, 0, 3],
    Mesh -> False, BoxRatios → {1, 1, 1}]}, ImageSize → 500];
```

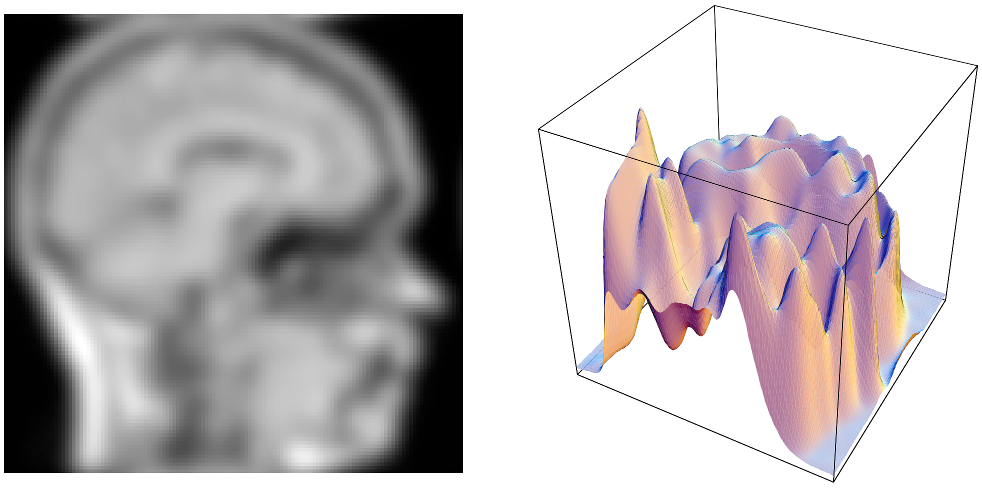

Figure 5.4 Left: a blurred MR image, resolution 128^2, $\sigma_{\text{blur}} = 3$ pixels. Right: The intensity surface as a height surface shows the blurring of the surfaces.

A movie of a (in this example) logarithmically sampled intensity scale-space is made with the **`Table`** command. Close the group of cells with images by double-clicking the group bracket. Double-clicking one of the resulting images starts the animation. Controls are on the bottom windowbar.

```
ss = Table[ListDensityPlot[gDf[im, 0, 0, E^τ], ImageSize -> 150],
  {τ, 0, 2.5, .25}];
```

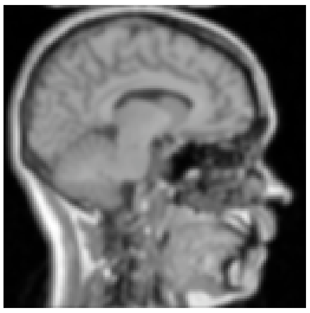

Figure 5.5 Animation of a blurring sequence, with exponential scale parametrization. Double-click the image to start the animation (only in the electronic version). Controls appear at the lower window bar.

This animation is only available in the electronic version. Here are the images:

```
Show[GraphicsArray[Partition[ss, 5]], ImageSize -> 450];
```

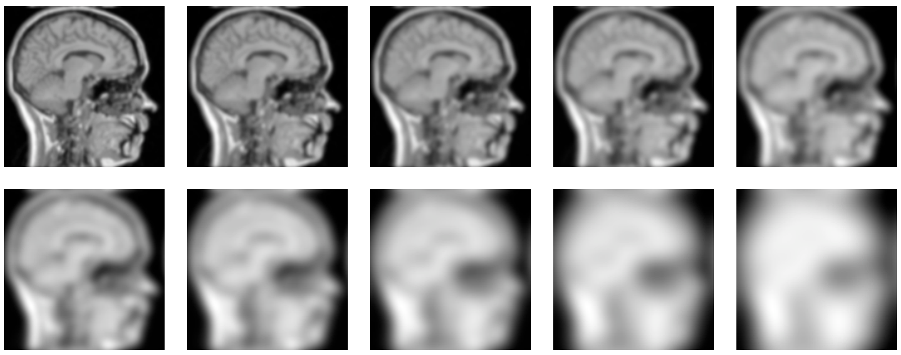

Figure 5.6 Frames of the animation of a blurring sequence above.

The sequence can be saved as an animated GIF movie (e.g. for use in webpages) with:

```
Export["c:\\scalespace.gif", ss, "GIF"];
```

The gradient of an image is defined as $\sqrt{L_x{}^2 + L_y{}^2}$. On a scale $\sigma = 0.5$ pixel for a 256^2 CT image of chronic cocaine abuse (EuroRAD teaching file case #1472, www.eurorad.org):

```
im = Import["Cocaine septum.gif"][[1, 1]];
DisplayTogetherArray[{ListDensityPlot[im],
   grad = ListDensityPlot[√(gD[im, 1, 0, .5]^2 + gD[im, 0, 1, .5]^2)]},
  ImageSize -> 370];
```

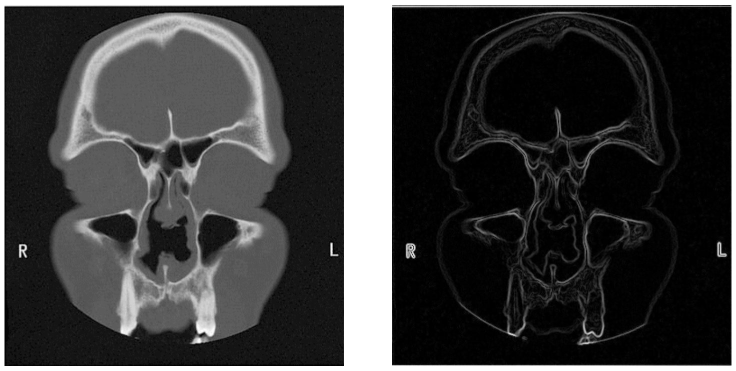

Figure 5.7 The gradient at a small scale $\sigma = 0.5$ pixels. Due to the letters R and L in the image with steep gradients the gradient image is not properly scaled in intensity. Note the completely missing septum in this patient (From www.eurorad.org, EuroRAD authors: D. De Vuyst, A.M. De Schepper, P.M. Parizel, 2002).

To change the window/level (contrast/brightness) settings one can change the displayed range of intensity values:

```
Show[grad, PlotRange → {0, 20},
  DisplayFunction -> $DisplayFunction, ImageSize -> 150];
```

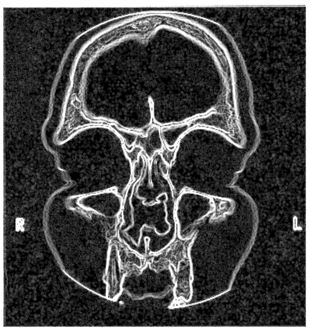

Figure 5.8 The gradient at a small scale $\sigma = 0.5$ pixels, now with an intensity window of 0 (black) to 30 (white).

We can also transfer the image into its histogram equalized version, by substituting its grayvalues by the values given by its cumulative lookup table:

```
Unprotect[heq];
heq[im_List] := Module[{min, max, freq, cf, lcf, maxcf, lut, int},
   min = Min[im]; max = Max[im];
   freq = BinCounts[Flatten[im], {min, max, (max - min) / 256}];
   cf = FoldList[Plus, First[freq], Drop[freq, 1]];
  maxcf = Max[cf]; lcf = Length[cf];
  lut = Table[N[{(i - 1) / lcf, cf[[i]] / maxcf}], {i, 1, lcf}];
  lut[[lcf]] = {1., 1.};
   int = Interpolation[lut]; max int[(im - min) / (max - min)]];
```

```
ListDensityPlot[
  heq[√(gD[im, 1, 0, .5]^2 + gD[im, 0, 1, .5]^2)], ImageSize -> 150];
```

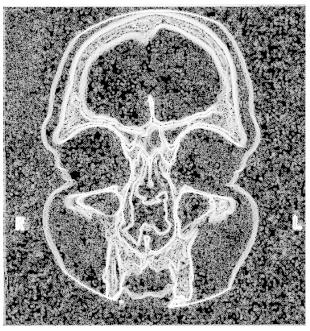

Figure 5.9 Histogram equalization of the gradient image of figure 5.7. By many radiologists this is considered too much enhancement. 'Clipped' adaptive histogram equalization admits different levels of enhancement tuning [Pizer1987].

The cumulative lookup table is applied for the intensity transform. Small contrasts have been stretched to larger contrasts, and reverse. We next compare the histograms of the gradient image with the histogram of the histogram-equalized gradient image. The total histogram of this image is indeed reasonably flat now.

```
grad = √(gD[im, 1, 0, .5]^2 + gD[im, 0, 1, .5]^2) ; DisplayTogetherArray[
  Histogram[Flatten[#]] & /@ {grad, heq[grad]}, ImageSize → 380];
```

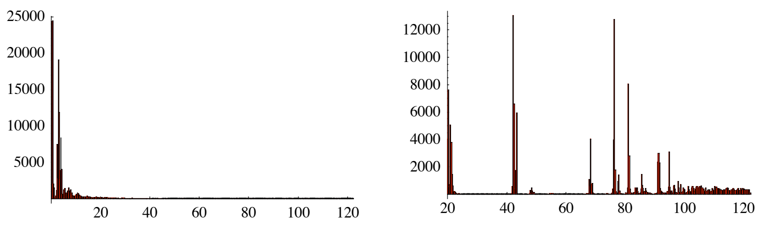

Figure 5.10 Left: Histogram of the gradient image of figure 5.7. Right: Histogram of the histogram-equalized gradient image. Note the equalizing or marked stretching of the histogram.

To conclude this introduction to multi-scale derivatives, let us look at some edges detected at different scales. It is clear from the examples below that the larger scale edges denote the more 'important' edges, describing the coarser, hierarchically higher structure:

```
im = Import["Utrecht256.gif"][[1, 1]];
DisplayTogetherArray[
  ListDensityPlot[√(gD[im, 1, 0, #]^2 + gD[im, 0, 1, #]^2)] & /@ {.5, 2, 5},
  ImageSize -> 400];
```

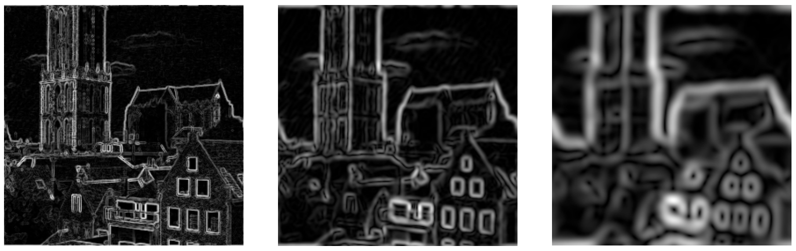

Figure 5.11 Gradient edges detected at different scales (σ = 0.5, 2, 5 pixels resp.). The coarser edges (right) indicate hierarchically more 'important' edges.

Other sources of different scales for edges are shadows and diffuse boundaries [Elder1996].

5.4 N-dim Gaussian derivative operator implementation

One of the powerful capabilities of *Mathematica* as a programming language is the relative ease to write numerical functions on N-dimensional data. In scale-space theory often high dimensional data occur: 3D and 3D-time medical images, such as 3D cardiovascular time sequences, orientation bundles (see chapter 16 where an extra dimension emerges from the inclusion of orientation as the output of *measurements* by oriented filters), high dimensional feature spaces for texture analysis, etc. Here is the separable implementation for N-dimensions:

```
Unprotect[gDn]; gDn[im_, orderlist_, σlist_, opts___?OptionQ] :=
 Module[{gaussd, dim = Length[Dimensions[im]], out = N[im], l, r, gder, x,
   kernel, cat, mid, lc, tl, td}, td = Dimensions /@ {orderlist, σlist};
  tl = Length /@ td; {l, r} = kernelSampleRange /. {opts} /. Options[gD];
```

$$\texttt{gaussd} = \frac{1}{\#2\sqrt{2\pi}}\left(-\frac{1}{\#2\sqrt{2}}\right)^{\#1}\texttt{HermiteH}\left[\#1,\ \frac{x}{\#2\sqrt{2}}\right]e^{-\frac{x^2}{2\,\#2^2}}\ \texttt{\&;}$$

```
  gder = Table[N[gaussd[#1, #2]], {x, Floor[l #2], Ceiling[r #2]}] &;
  kernel = RotateRight[MapThread[gder, {orderlist, σlist}]];
  mid = (Ceiling[Length[#1] / 2] &) /@ kernel;
  cnt = Append[Table[1, {dim - 1}], mid[[#1]]] &;
  lc =
   Transpose[ListConvolve[Nest[List, kernel[[#2]], dim - 1], #1, {cnt[#2], cnt[#2]}],
     RotateRight[Range[dim]]] &; Do[out = lc[out, i], {i, dim}]; out]
```

The function makes use of the possibility to **Nest** functions to large depth, and the universality of the **ListConvolve** function. The function is fast. Note the specification of orders and scales as lists, and note the specific, *Mathematica*-intrinsic ordering with the fastest running variable last: {z,y,x}.

Example: **gDn[im,{0,2,1},{2,2,2}]** calculates $\frac{\partial^3 L}{\partial x\,\partial y^2}$ of the input image **im** at an isotropic scale of $\sigma_z = \sigma_y = \sigma_x = 2$ pixels.

Here is the time it takes to calculate the first order derivative in 3 directions at scales of 1 pixel of a 128^3 random array (more than 2 million pixels, 1.7 GHz, 512 MB, Windows XP):

```
im = Table[Random[], {128}, {128}, {128}];
Timing[gDn[im, {1, 1, 1}, {1, 1, 1}]] // First

5.094 Second
```

This gives help on how to call the function:

```
? gDn

gDn[im,{...,ny,nx},{...,σy,σx},options] calculates the Gaussian
  derivative of an N-dimensional image by approximated spatial
  convolution. It is optimized for speed by 1D convolutions per
  dimension. The image is considered cyclic in each direction.
  Note the order of the dimensions in the parameter lists.
     im = N-dimensional input image [List]
     nx = order of differentiation to x [Integer, nx ≥ 0]
     σx = scale in x-dimension [in pixels, σ > 0]
     options = <optional> kernelSampleRange: range of kernel
  sampled in multiples of σ. Default: kernelSampleRange->{-6,6}

 Example: gDn[im,{0,0,1},{2,2,2}] calculates the x-
  derivative of a 3D image at an isotropic scale of σz=σy=σx=2.
```

5.5 Implementation in the Fourier domain

The spatial convolutions are not exact. The Gaussian kernel is truncated. In this section we discuss the implementation of the convolution operation in the Fourier domain.

In appendix B we have seen that a convolution of two functions in the spatial domain is a multiplication of the Fourier transforms of the functions in the Fourier domain, and take the inverse Fourier transform to come back to the spatial domain. We recall the processing scheme (e.g. with 1D functions):

$$\begin{array}{ccccc} f(x) & = & h(x) & \otimes & g(x) \\ \updownarrow & & \updownarrow & & \updownarrow \\ F(\omega) & = & H(\omega) & . & G(\omega) \end{array}$$

The $\updownarrow$ indicates the Fourier transform in the downwards direction and the inverse Fourier transform in the upwards direction. $f(x)$ is the convolved function, $h(x)$ the input function, and $g(x)$ the convolution kernel.

The function **`gDf[im,nx,ny,σ]`** implements the convolution of the 2D image with the Gaussian derivative for 2D discrete data in the Fourier domain. This is an exact function, no approximations other than the finite periodic window in both the x- and y-direction. We explicitly give the code of the functions here, so you see *how* it is implemented, the reader may make modifications as required. All information on (always capitalized) internal functions is on board of the *Mathematica* program in the Help Browser (highlight+key F1), as well as on the 'the Mathematica book' internet pages of Wolfram Inc.

Variables: **`im`** = 2D image (as a **`List`** structure)
`nx, ny` = order of differentiation to **`x`** resp. **`y`**
σ = scale of the Gaussian derivative kernel, in pixels

The underscore in e.g. **`im_`** means **`Blank[im]`** and stands for *anything* (a single element) which we name **`im`**. **`im_List`** means that **`im`** is tested if it is a **`List`**. If not, the function **`gDf`** will not be evaluated.

```
Unprotect[gDf]; Remove[gDf];

gDf[im_List, nx_, ny_, σ_] :=
  Module[{xres, yres, gdkernel},
        {yres, xres} = Dimensions[im];
    gdkernel =
    N[Table[Evaluate[D[1/(2 π σ^2) Exp[-(x^2 + y^2)/(2 σ^2)], {x, nx}, {y, ny}]], {y,
      -(yres - 1) / 2, (yres - 1) / 2}, {x, -(xres - 1) / 2, (xres - 1) / 2}]];
      Chop[N[√(xres yres) InverseFourier[Fourier[im]
      Fourier[RotateLeft[gdkernel, {yres / 2, xres / 2}]]]]]];
```

A **Module[{vars}, ...code...]** is a scope construct, where the vars are private variables. The last line determines what is returned. The assignment with **:=** is a delayed assignment, i.e. the function is only evaluated when called. The dimensions of the input image are extracted (note the order!) and the Gaussian kernel is differentiated with the function **D[gauss,{x,nx},{y,ny}]** and symmetrically tabulated over the x- and y-dimensions to get a kernel image with the same dimensions as the input image.

We have now a 2D **List** with the kernel in the center. We shift **gdkernel** with **RotateLeft** over half the dimensions in x- and y-direction in order to put the kernel's center at the origin at {0,0}. We could equally have shifted in this symmetric case with **RotateRight**. We then take the **Fourier** transform of both the image and the kernel, multiply them (indicated by a space) and take the **InverseFourier** transform.

Because we have a finite Fourier transform, we normalize over the domain through the factor $\sqrt{\mathbf{xres\ yres}}$. The function **N[]** makes all output numerical, and the function **Chop[]** removes everything that is smaller then 10^{-10}, so to remove very small round-off errors.

```
im = Import["mr256.gif"][[1, 1]]; imx = gDf[im, 1, 0, 1];
ListDensityPlot[imx, ImageSize -> 240];
```

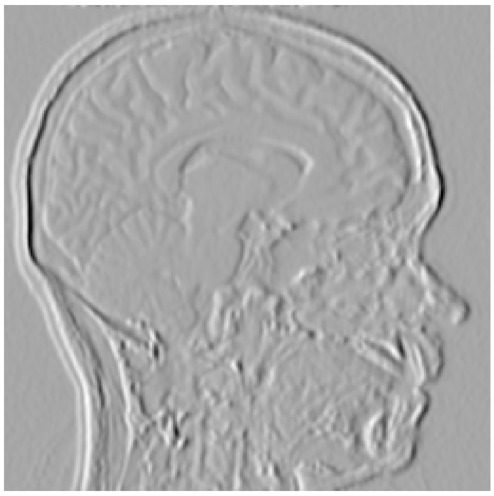

Figure 5.12 First order Gaussian derivative with respect to x at scale $\sigma = 1$ pixel, calculated through the Fourier domain. Resolution 256^2 pixels.

The *Mathematica* function Fourier is highly optimized for any size of the data, and uses sophisticated bases when the number of pixels is not a power of 2.

This function is somewhat slower that the spatial implementation, but is exact. Here is a vertical edge with a lot of additive uniform noise. The edge detection at very small scale only reveals the 'edges of the noise'. Only at the larger scales we discern the true edge, i.e. when the scale of the operator applied is at 'the scale of the edge'.

```
im5 = Table[If[x > 128, 1, 0] + 13 Random[], {y, 256}, {x, 256}];
DisplayTogetherArray[
 Prepend[ListDensityPlot[Sqrt[gDf[im5, 1, 0, #]^2 + gDf[im5, 0, 1, #]^2] & /@
   {2, 6, 12}, ListDensityPlot[im5]], ImageSize -> 500];
```

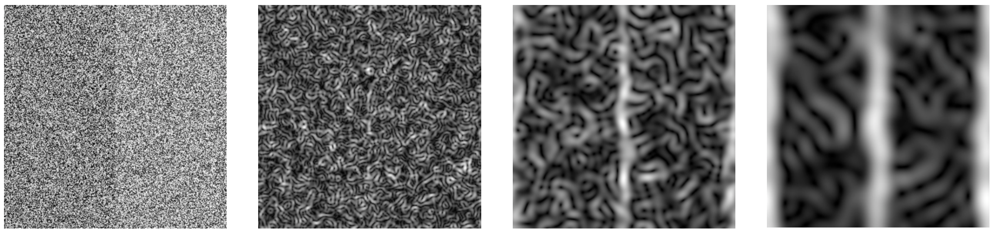

Figure 5.13 Detection of a very low contrast step-edge in noise. Left: original image, the step-edge is barely visible. At small scales (second image, $\sigma = 2$ pixels) the edge is not detected. We see the edges of the noise itself, cluttering the edge of the step-edge. Only at large scale (right, $\sigma = 12$ pixels) the edge is clearly found. At this scale the large scale structure of the edge emerges from the small scale structure of the noise.

▲ Task 5.2 The Fourier implementation takes the Fourier transform of the image and the Fourier transform of a calculated kernel. This seems a waste of calculating time, as we know the *analytical* expression for the Fourier transform of the Gaussian kernel. Write a new *Mathematica* function that takes this into account, and check if there is a real speed increase.

▲ Task 5.3 The spatial implementation has different speed for different size kernels. With increasing kernel size the number of operations increases substantially. How?

▲ Task 5.4 Compare for what kernel size the choice of implementation is computationally more effective: Fourier or spatial domain implementation. See also [Florack 2000a].

There are two concerns we discuss next: what to do at the boundaries? And: the function is slow, so how to speed it up?

5.6 Boundaries

```
DisplayTogetherArray[
  Show /@ Import /@ {"Magritte painting boundary.gif", "Magritte.jpg"},
  ImageSize -> 340];
```

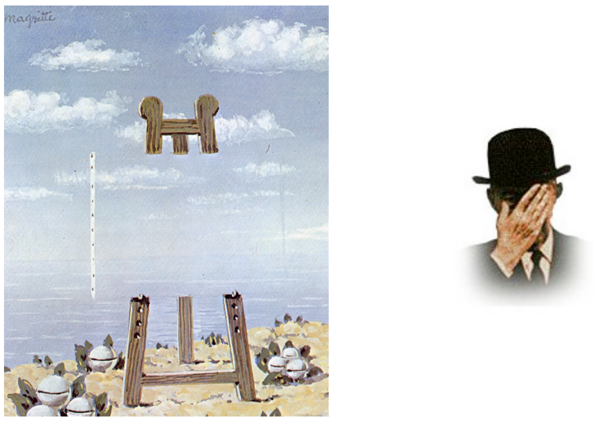

Figure 5.14 It is important to consider what happens at the boundaries of images. It matters what we *model* outside our image. Painting by René Magritte (right: self-portrait, 1898-1967).

At the boundary of the image artefacts may appear when we do convolutions with (by nature) extended kernels. Here is an example: two linear intensity ramps give a constant output when we calculate the first derivative to x, but we see both at the left- and right-hand side strong edge responses, for the Fourier implementation as well as for the spatial implementation:

```
im = Table[If[y > 64, x - 64, 64 - x], {y, 128}, {x, 128}];
DisplayTogetherArray[ListDensityPlot[#] & /@
   {im, gDf[im, 1, 0, 3], gD[im, 1, 0, 3]}, ImageSize -> 400];
```

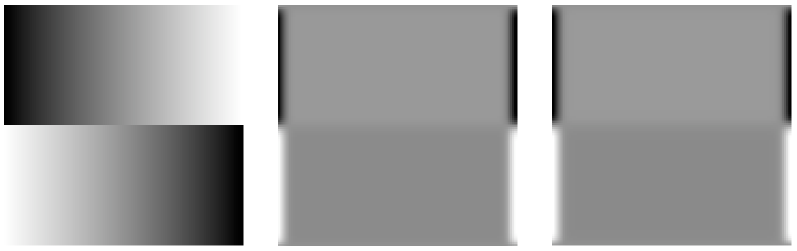

Figure 5.15 Boundary effects due to the periodicity of the Fourier domain.

This is due to the fact that both in the Fourier domain as the spatial domain implementation of the convolution function the image is regarded as repetitive. A Fourier transform is a *cyclic* function, i.e. $\mathcal{F}(\omega) = \mathcal{F}(\omega + n\,2\,\pi)$. In 2D: $\mathcal{F}(\omega_x, \omega_y) = \mathcal{F}(\omega_x + n_x\,2\,\pi, \omega + n_y\,2\,\pi)$. The boundary effects in the image above are due to the strong edge created by the neighboring pixels at both ends. One can regard the domain of the image as a window cut-out from an infinite tiling of the plane with 2D functions. Figure 4.1 shows a tiling with 20 images, each 64^2 pixels:

```
im = Import["mr64.gif"][[1, 1]];
tiles = Join @@ Table[MapThread[Join, Table[im, {5}]], {4}];
ListDensityPlot[tiles, ImageSize -> 280];
```

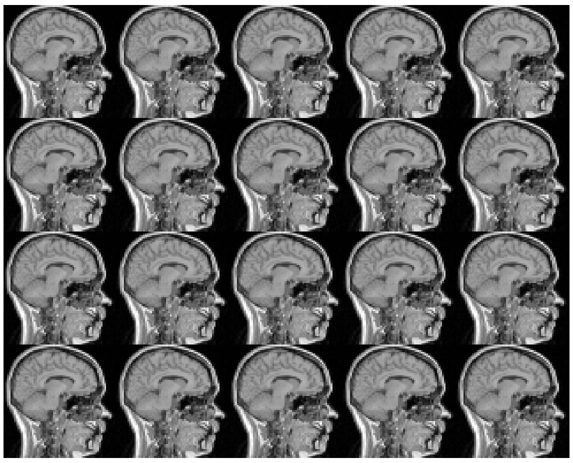

Figure 5.16 A section from the infinite tiling of images when we consider a cyclic operation. The *Mathematica* function **MapThread** maps the function **Join** on the rows of the horizontal row of 5 images to concatenate them into long rows, the function **Join** is then applied (with **Apply** or @@) on a table of 4 such resulting long rows to concatenate them into a long vertical image.

```
Clear[a, b, c, d, e, f, h];
MapThread[h, {{a, b, c}, {d, e, f}}]

{h[a, d], h[b, e], h[c, f]}

Apply[h, {{a, b, c}, {d, e, f}}]

h[{a, b, c}, {d, e, f}]

h @@ {{a, b, c}, {d, e, f}}

h[{a, b, c}, {d, e, f}]
```

It is important to realize that there is no way out to deal with the boundaries. Convolution is an operation with an *extended* kernel, so at boundaries there is always a choice to be made. The most common decision is on repetitive tiling of the domain to infinity, but other choices are just as valid. One could extend the image with zero's, or mirror the neighboring image at all sides in order to minimize the edge artefacts. In all cases information is put at places where there was *no* original observation. This is no problem, as long as we carefully describe how our choice has been made. Here is an example of mirrored tiling:

```
im = Import["mr128.gif"][[1, 1]]; Off[General::spell];
imv = Reverse[im]; imh = Reverse /@ im; imhv = Reverse /@ Reverse[im];
                                              ⎛ imhv  imv  imhv ⎞⎞
mirrored = Join @@ ⎛MapThread[Join, #] & /@ ⎜ imh   im   imh  ⎟⎟;
                                              ⎝ imhv  imv  imhv ⎠⎠
ListDensityPlot[mirrored, ImageSize -> 270];
```

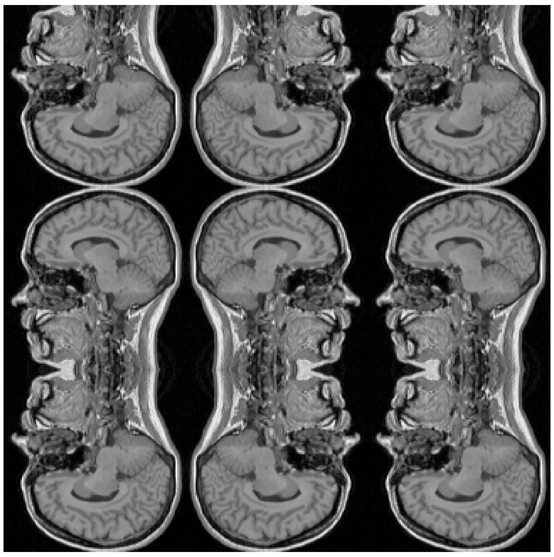

Figure 5.17 A section from the infinite tiling of images when we consider a mirroring operation. Note the rather complex mirroring and concatenation routines for these 2D images.

▲ Task 5.5 Rachid Deriche [Deriche 1992] describes a fast *recursive* implementation of the Gaussian kernel and its derivatives. Make a *Mathematica* routine for recursive implementation.

▲ Task 5.6 A small truncated kernel size involves less computations, and is thus faster. Blurring with a large kernel can also be accomplished by a concatenation of small kernels, e.g. a blurring step with $\sigma = 3$ pixels followed by a blurring step with $\sigma = 4$ pixels gives the same result as a single blurring step with $\sigma = 5$ pixels ($\sigma_1{}^2 + \sigma_2{}^2 = \sigma_{\text{new}}{}^2$). What is faster, a large kernel, or a cascade series of smaller kernels? Where is the trade-off?

5.7 Advanced topic: speed concerns in *Mathematica*

This section can be skipped at first reading.

Mathematica is an interpreter, working with symbolic elements, and arbitrary precision. For this reason, care must be taken that computation times do not explode for large datasets. When proper measures are taken, *Mathematica* can be fast, close to compiled C++ code. In this section we discuss some examples of increasing the speed of the operations on larger datasets.

It pays off to work numerically, and to *compile* a function when it is a repetitive operation of simple functions. *Mathematica*'s internal commands are optimized for speed, so the gain here will be less. We discuss the issue with the example of the generation of a discrete Gaussian derivative kernel. The timings given are for a 1.7 GHz 512 MB PC and *Mathematica* 4.1 under Windows XP.

First of all, exact calculations are slow. Most internal *Mathematica* functions can work both with symbolic and numerical data. These internal functions are fully optimized with respect to speed and memory resources for numerical input. Here is a simple example:

```
σ = 1; m = Table[Exp[-(x^2 + y^2)/(2 σ^2)], {x, -4, 4}, {y, -4, 4}];
Timing[Eigenvalues[m]]
Timing[Eigenvalues[N[m]]]
Timing[Chop[Eigenvalues[N[m]]]]
```

$\{3.156\ \text{Second},\ \{0, 0, 0, 0, 0, 0, 0, 0, 1 + \frac{2}{e^{16}} + \frac{2}{e^{9}} + \frac{2}{e^{4}} + \frac{2}{e}\}\}$

$\{0.\ \text{Second},\ \{1.77264,\ 5.59373\times10^{-17},\ -4.27232\times10^{-17},\ -1.82978\times10^{-18},\ 3.22688\times10^{-22},\ -6.5072\times10^{-24},\ -7.47864\times10^{-34},\ 1.05492\times10^{-35},\ 0.\}\}$

```
{0. Second, {1.77264, 0, 0, 0, 0, 0, 0, 0, 0}}
```

In the sequel we will develop a very fast implementation for the convolution of a 2D image with a Gaussian derivative in the Fourier domain (see section 4.3). Most of the time is spent in the creation of the 2D Gaussian kernel, e.g. for 256^2:

```
{xres, yres} = {256, 256}; σ = 3;
Timing[
  kernel = Table[1/(2 π σ^2) Exp[-(x^2 + y^2)/(2 σ^2)], {y, -(yres - 1) / 2, (yres - 1) / 2},
     {x, -(xres - 1) / 2, (xres - 1) / 2}]][[1]]

4.859 Second
```

Mathematica keeps values as long as possible in an exact representation. Here is the pixelvalue at (30, 47):

```
kernel[[30, 47]]
```

$$\frac{1}{18\, e^{32689/36}\, \pi}$$

An additional disadvantage is that the Fourier transform on such symbolic expressions also takes a long time:

```
Timing[fft = Fourier[kernel]] // First

8. Second
```

It doesn't make much difference when we enter the data as **Real** values (to be done with the insertion of a decimal point in a number, or through the function **N**):

```
{xres, yres} = {256., 256.}; σ = 3.; pi = N[π];
Timing[
  gdkernel = Table[1/(2 π σ^2) Exp[-(x^2 + y^2)/(2 σ^2)], {y, -(yres - 1) / 2, (yres - 1) / 2},
     {x, -(xres - 1) / 2, (xres - 1) / 2}]][[1]]
```

```
5.797 Second
```

The output is now a number, not a symbolic expression:

```
gdkernel[[30, 48]]
```

$6.379323933059 \times 10^{-393}$

But still, we have no gain in speed. This is because the internal representation is still in 'arbitrary precision' mode. The smallest and largest number that can be represented as a **Real** is:

```
$MinMachineNumber
$MaxMachineNumber
```

2.22507×10^{-308}

1.79769×10^{308}

We have smaller values in our pixels! As soon as *Mathematica* encounters a number smaller or larger then the dynamic range for **Real** numbers, it turns into arbitrary precision mode, which is slow. A good improvement in speed is therefore gained through restricting the output to be in this dynamic range. In our example the parameter for the exponential function **Exp** should be constrained:

```
{xres, yres} = {256., 256.}; σ = 3.; pi = N[π];
Timing[gdkernel = Table[1/(2 π σ^2) Exp[If[-(x^2 + y^2)/(2 σ^2) < -100, -100, -(x^2 + y^2)/(2 σ^2)]],
    {y, -(yres - 1) / 2, (yres - 1) / 2},
    {x, -(xres - 1) / 2, (xres - 1) / 2}]] // First
```

```
2.594 Second
```

Most of the internal commands of *Mathematica* do a very good job on real numbers.

A further substantial improvement in speed can be obtained by *compilation* of the code into fast internal assembler code with the function **Compile[{args}, ...code..., {decl}]**. This generates a pure function, that can be called with the arguments **{args}**. This function generates optimized code based on an idealized register machine. It assumes approximate real or inter numbers, or matrices of these. The arguments in the argumentlist need to have the proper assignment (**_Real**, **_Integer**, **_Complex** or **True**/**False**).

The assignment **`_Real`** is default and may be omitted, so **`{x, _Real}`** is equivalent to **`{x}`**. An example to calculate the factorial of a sum of two real numbers:

```
gammasum = Compile[{{x, _Real}, {y, _Real}}, (x + y) !]
```

```
CompiledFunction[{x, y}, (x + y) !, -CompiledCode-]
```

```
gammasum[3, 5]
```

```
40320.
```

We now check if the compiled code of our Gaussian kernel gives a speed improvement:

```
gdkernel = Compile[{xres, yres, σ}, xresh = (xres - 1)/2; yresh = (yres - 1)/2;
   p = Table[1/(2 π σ^2) Exp[If[-(x^2 + y^2)/(2 σ^2) < -100, -100, -(x^2 + y^2)/(2 σ^2)]],
      {y, -yresh, yresh}, {x, -xresh, xresh}], {{x, _Real},
      {y, _Real}, {xresh, _Real}, {yresh, _Real}, {p, _Real, 2}}];
```

```
Timing[gdkernel[256, 256, 3]] // First
```

```
2.532 Second
```

In version 4.2 of *Mathematica* we see no improvement, running the example above, the kernel has been optimized for these calculations. In earlier versions you will encounter some 60% improvement with the strategy above. See the Help browser (shift-F1) for speed examples of the Compile function. We now add the symbolic operation of taking derivatives of the kernel. We force direct generation of the polynomials in the Gaussian derivatives with the Hermite polynomials, generated with **`HermiteH`**. The symbolic functions are first evaluated through the use of the function **`Evaluate`**, then compiled code is made:

```
gdkernel = Compile[{xres, yres, σ, {nx, _Integer}, {ny, _Integer}},
   xresh = (xres - 1)/2; yresh = (yres - 1)/2;
   p = Table[Evaluate[
      (-1/(σ Sqrt[2]))^(nx+ny) HermiteH[nx, x/(σ Sqrt[2])] HermiteH[ny, y/(σ Sqrt[2])]
      1/(σ^2 2 π) Exp[If[-(x^2 + y^2)/(2 σ^2) < -100, -100, -(x^2 + y^2)/(2 σ^2)]]],
      {y, -yresh, yresh}, {x, -xresh, xresh}], {{x, _Real},
      {y, _Real}, {xresh, _Real}, {yresh, _Real}, {p, _Real, 2}}];
```

```
Timing[gdkernel[256, 256, 3, 10, 10]] // First
```

```
4.25 Second
```

Larger kernels are now no problem anymore, e.g. for 512^2:

```
Timing[t = gdkernel[512, 512, 3, 2, 1]] // First

7.593 Second
```

We adopt this function for our final implementation. Because the output is a matrix of real numbers, also the Fourier transform is very fast. This is the time needed for the Fast Fourier Transform on the 512^2 kernel just generated:

```
Timing[Fourier[t]] // First

0.172 Second
```

To complete this section we present the final implementation, available throughout the book in the context **FEV`**, which is loaded by default in each chapter.

In the compiled function also complex arrays emerge, such as the result of `Fourier[]` and `InverseFourier[]`. The compiler is told by the declarations at the end that anything with the name **Fourier** and **InverseFourier** working on something (_) should be stored in a complex array with tensorrank 2, i.e. a 2D array. Study the rest of the details of the implementation yourself:

```
Unprotect[gDf]; gDf[im_, nx_, ny_, σ_] :=
 Module[{}, {xres, yres} = Dimensions[im]; gf[im, nx, ny, σ, xres, yres]];

gf =
  Compile[{{im, _Real, 2}, {nx, _Integer}, {ny, _Integer}, σ, xres, yres},
   Module[{x, y}, xresh = (xres - 1)/2; yresh = (yres - 1)/2;
    p = RotateLeft[Table[1/(2 π σ^2) Evaluate[(-1/(σ Sqrt[2]))^(nx+ny) HermiteH[nx,
          x/(σ Sqrt[2])] HermiteH[ny, y/(σ Sqrt[2])] ⅇ^If[-(x^2+y^2)/(2 σ^2)<-200,-200,-(x^2+y^2)/(2 σ^2)]],
        {y, -yresh, yresh}, {x, -xresh, xresh}], {xres/2, yres/2}]];
   Sqrt[xres yres] Chop[Re[InverseFourier[Fourier[im] Fourier[p]]]],
   {{x, _Real}, {y, _Real}, {xresh, _Real}, {yresh, _Real}, {p, _Real, 2},
    {Fourier[_], _Complex, 2}, {InverseFourier[_], _Complex, 2}}];
```

5.8 Summary of this chapter

Mathematica is fast when:
- it can use its internal kernel routines as much as possible. They have been optimized for speed and memory use;
- it can calculate on numerical data. Use the function `N[...]` to convert infinite precision representations like `Sin[3/7]` to numerical data;
- it is working in the representation range of real numbers. Otherwise it enters the infinite precision mode again;
- the function is compiled with the function `Compile[...]`;