Convolution and Local Image Structure

Lab Exercise 2 for Beeldverwerken

Informatics Institute
University of Amsterdam
The Netherlands

December 3, 2013

1 Convolution

First read Chapter 6.2.1 up to and including 6.2.3 of the Lecture Notes. That gives you a background in the convolution principles. You can also play with "The Joy of Convolution" at `http://www.jhu.edu/signals/discreteconv2/`, to see 1D convolution in JAVA action for some simple 'images' and kernels. For convenience, we repeat the definition of the convolution (discrete version):

$$(f * g)(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{x} - \mathbf{y})\, g(\mathbf{y}).$$

This is like (6.16), but formulated more generally so it can be used in d-dimensional space.

Theory Questions

1.1. **(1 points)** By hand, perform the 1D convolution of the signals $g = \{0\ 0\ 0\ 0\ \underline{1}\ 1\ 1\ 1\ 1\}$ and $f = \{1\ \underline{2}\ 1\}$. Note the origin $x = 0$ (underlined). Explain how you handle border cases.

1.2. **(1 points)** By hand, perform the 1D convolution of the signals $f = \{0\ 0\ 0\ 0\ \underline{1}\ 1\ 1\ 1\ 1\}$ and $g = \{1\ \underline{2}\ 1\}$. Note the origin $x = 0$ (underlined). Explain how you handle border cases.

1.3. **(1 points)** By hand, perform the 1D convolution of the signals $f = \{0\ 0\ 0\ 0\ \underline{1}\ 1\ 1\ 1\ 1\}$ and $g = \{-1\ \underline{1}\}$. Note the origin $x = 0$ (underlined). Explain how you handle border cases.

1.4. **(2 points)** Prove that convolution is commutative (you can use the discrete or continuous convolution formula for this, they are structurally identical). As a consequence, you could also use as definition

$$(f \star g)(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{y})\, g(\mathbf{x} - \mathbf{y}).$$

(**Hint:** Note that the commutativity property depends crucially on the minus sign in the argument, i.e., the reflection of one of the functions!)

1.5. **(2 points)** Prove that convolution is associative (you can use the discrete or continuous convolution formula for this, they are structurally identical).

Now study Appendix A,[1] which derives an important result: *any translation invariant linear image operation can be written as a convolution.* The bottom line of that is: convolutions are an important class of transformations. It is therefore worth getting to know this class a bit better. In this exercise, we will first play a bit with them in general, and then return to the general subject of Chapter 6: characterising and detecting local structures in images.

1.1 Convolution Examples and Implementation

Write the following operations as a convolution by specifying the convolution kernel, or indicate why this is impossible:

Theory Questions

1.6. **(1 points)** The identity operation on an image.

1.7. **(1 points)** Multiplying the image intensity by 3.

1.8. **(1 points)** Translating the image over the vector $[-3, 1]^T$ (where positive x and y correspond to right and down, respectively).

1.9. **(1 points)** Rotating an image over an arbitrary angle around the origin.

1.10. **(1 points)** Taking the average in a 3×3 neighborhood.

1.11. **(1 points)** Taking the median in a 3×3 neighborhood.

1.12. **(1 points)** Computing the minimum value in a 5×5 neighborhood.

1.13. **(1 points)** Performing motion blur, as if the camera moved horizontally to the right over 5 pixels during recording.

1.14. **(1 points)** Optical blur: every point in the image is imaged vaguely, for instance spread out like a Gaussian with a standard deviation of 3 pixels.

1.15. **(1 points)** Unsharp masking of an image (see Section 2.4, Figure 2.10).

1.16. **(1 points)** Taking an approximation of the derivative in the x-direction. (**Hint:** is the kernel like $\frac{1}{2}\{1\ \underline{0}\ -1\}$ or like $\frac{1}{2}\{-1\ \underline{0}\ 1\}$?)

1.17. **(1 points)** Taking the second derivative in the x-direction. (**Hint:** You could think of this as taking a first derivative twice - how does that help to design its kernel?)

1.18. **(1 points)** Zooming in on an image to 4 times the size, and performing bilinear interpolation to obtain the intermediate points.

1.19. **(1 points)** Thresholding an image so that all pixels with an intensity lighter than 0.5 become 1, and all others 0.

[1] Augment the last equation on page A-1 to the more convenient (and check why this is allowed!):

$$f(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{y}) T_{\mathbf{x}} \delta(\mathbf{y}) = \sum_{\mathbf{y} \in E} f(\mathbf{y}) T_{\mathbf{y}} \delta(\mathbf{x}).$$

Change the last equation on page A-2 to the more correct:

$$L(f)(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{y})(T_{\mathbf{y}} w)(\mathbf{x}) = \sum_{\mathbf{y} \in E} f(\mathbf{y}) w(\mathbf{x} - \mathbf{y}).$$

The discrete convolution operator for a 2D image is available in Matlab with the functions `conv2` and `imfilter`. The examples in this assignment use the `imfilter` function. Convolving an image `F` with kernel `G` resulting in an image `H` is done with the function call

```
H = imfilter(F, G, 'conv', 'replicate');
```

⋆ **(4 points)** Read the documentation of `imfilter` to understand the meaning of the arguments. Use it to perform three of the operations you explained above, and show the results.

1.2 Gaussian Convolution

The Gaussian blur example above turns out to be very important for image processing: it can be made the basis for a consistent theory of 'scale of details' in images, and within that context give a good way to take derivatives of images, to arbitrary order. This is compactly explained in Section 6.2.4 through 6.2.7, so read those now.

The basic idea should be clear: to compute with an image at a certain scale, we blur it first. Then we take derivatives at that scale. But since both blurring and taking derivatives are convolutions, we can make an operator that takes a derivative at the desired scale immediately. That strategy is independent of the kind of kernel you might use for smoothing. But it turns out that the Gaussian kernel has the unique properties (among all kernels) of not introducing extraneous detail with further blurring. We do not have the math skills to prove that (check the original papers by Jan Koenderink or his disciple Bart ter Haar Romeny if you are interested).

As implementers, we are fortunate that this Gaussian kernel happens to be so important, because it has a number of attractive computational properties. We explore those now, but do so using functions in the continuous domain. Discretization will wait till our implementation in the next section.

For scale σ and dimension d, the Gaussian kernel is given in Section 6.2.4 as:

$$G_\sigma(\mathbf{x}) = \frac{1}{(\sqrt{2\pi}\sigma)^d}\, e^{-\frac{\|\mathbf{x}\|^2}{2\sigma^2}},$$

where d is the dimension of the Gaussian, $\mathbf{x}$ is a d-dimensional vector (*e.g.* $(x, y)^T$ in the case $d = 2$) and σ is a parameter, called the *scale*, which determines the width (and height) of the Gaussian curve. We will be mostly interested in $d = 1$ (to match you calculus intuition) and $d = 2$ (for our regular images).

Theory Questions

1.20. **(2 points)** In 1D, show that (or convince yourself that) the Gaussian has smooth derivatives of arbitrary order.

1.21. **(2 points)** Why should 'taking a derivative' be a convolution?

1.22. **(2 points)** The derivatives of the convolution of an image intensity function with a Gaussian kernel (i.e., of a blurred image) can be calculated by convolving the original image with the derivative of the Gaussian, instead. The result is therefore often called the *Gaussian derivative* of the image. But this depends on two fundamental properties of the convolution: associativity and commutativity. Show precisely how these properties are involved.

1.23. **(2 points)** In 2D, show that the formula given in Section 6.2.5 for the derivative holds:

$$\frac{\partial G_\sigma}{\partial x}(x) = -\frac{x}{\sigma^2} G_\sigma(x).$$

1.24. **(2 points)** Also in 2D, show that (see Section 6.2.5)

$$\frac{\partial^2 G_\sigma}{\partial x^2}(x) = (\frac{x^2}{\sigma^4} - \frac{1}{\sigma^2}) G_\sigma(x).$$

1.25. **(2 points)** Show *analytically* that the Gaussian smoothing of a Gaussian smoothed image is a Gaussian smoothed image. Make this specific: if you first smooth with G_{σ_1} and then with G_{σ_2}, what is the resulting smoothing scale? (**Hint:** the result should be familiar from the statistics of normal distrbuttions. Why?)

1.26. **(2 points)** Show *analytically* that the 2D Gaussian G_σ is *separable*, i.e., smoothing with it can be performed by an x-smoothing and then an y-smoothing. What are the scales of those 1D smoothings?

1.27. **(2 points)** Show *analytically* that all derivatives of the 2D Gaussian functions are separable as well.

Now that we have the theoretical properties nailed down, we can start implementing.

2 Implementation of Gaussian derivatives

In image processing, we cannot deal with functions on a continuous domain. Images don't *have* values between the pixels. Also, we cannot have functions of infinite extent (since they would take too long to compute with).

You therefore need to discretize the Gaussian kernel in order to use it for image processing. The following Matlab code will help you do so.

Listing 1: Sampling the Gaussian Kernel

```
% create appropriate ranges for x and y
x = -M:M;
y = -N:N;
% create a sampling grid
[X, Y] = meshgrid(x, y);
% determine the scale
sigma = S;
% calculate the Gaussian function
G = ??;
```

In the last line you can calculate the Gaussian kernel `G` in terms of the coordinates `X` and `Y` and the scale `S`. Note, that for squaring every value in an array `X` you have to use `X.^2`

When performing a convolution with a Gaussian kernel, you *always* have to select a scale. The bigger the scale, the broader your Gaussian curve will be. As a consequence the resulting image becomes more blurred.

⋆ **(10 points)** Write a function `Gauss(sigma)` that returns the sampled two-dimensional Gaussian function with the specified scale. Choose an appropriate size for the sampling grid

(*i.e.* the values `M` and `N` in the above code fragment). Explain your choice!

⋆ **(4 points)** Use `sum(sum(Gauss(sigma)))` on your function with different `sigma`. Which result do you expect? Why? Do you get this result in the Matlab calculation? How should you correct your function to avoid this effect?

⋆ **(1 points)** Plot the Gaussian kernel using `mesh(Gauss(3))` and include the plot in your hand-in.

• **(5 points)** What is the "physical" unit of the scale parameter σ?

⋆ **(2 points)** The time it takes to perform a convolution with your Gaussian kernel can be measured as:

```
tic;
H = imfilter(F, Gauss(sigma), 'conv', 'replicate');
elapsedTime = toc;
```

To get an accurate impression of the runtime you might want to execute this many times for each sigma. Create an image `F` and make a plot of the elapsed time as a function of the scale `sigma`.

⋆ **(4 points)** What is the order of the computational complexity in terms of the scale?

• **(3 points)** Verify your computation on the scale of a blurred blur above, by comparing the result of two consecutive Gaussian convolutions with a single one of the appropriate scale.

⋆ **(10 points)** The convolution with a *separable* function is easy. First you convolve along the columns (the first axis in Matlab) using the kernel $G_1(y, \sigma)$, followed by a convolution along the rows using $G_1(x, \sigma)$.

Let `Gauss1(sigma)` be the function that returns a sampled 1D Gaussian function in an $1 \times M$ array, *i.e.* a row vector. Then the convolution along the columns is calculated with

```
imfilter(F, Gauss1(sigma)', 'conv', 'replicate');
```

Note the *transpose* (with the `'` Matlab operator) of the Gaussian kernel to obtain a $M \times 1$ kernel.

Write the `Gauss1` function. Again, make an appropriate choice for the size of the kernel. Also make sure, the call `sum(Gauss1(sigma))` returns the expected value.

⋆ **(6 points)** Calculate the 2D Gaussian convolution using the separability property and time the convolution as a function of scale. Give the plot of time as a function of scale. What is the order of computational complexity now?

⋆ **(10 points)** Write a function `gD(f, sigma, xorder, yorder)` that uses the function `Gauss1` and Matlab function `imfilter` to calculate the Gaussian derivatives of an image `f`. The variables `xorder` and `yorder` are the orders of differentiation in the x- and y-direction, respectively. You should make use of the fact that the derivative of a Gaussian function is equal to the multiplication of the Gaussian function with a polynomial function in x and y. Only differentiation up to order 2 is needed.

Your report should include the code for the function `gD` and (as always) *enough documentation to make it understandable.*

⋆ **(3 points)** Visualize the 2-jet of the image `cameraman.tif` in your report.

Hint: To display all occurring values in an image use `imshow(image, [])`. See the Matlab Help for details.

3 The Canny Edge Detector

Now we have the tools to implement the various detectors based in scale space theory. We will focus on the Canny edge detector, but you can implement any of the other local structure detectors conveyed in Sections 6.1. Read that first.

You will use a simple synthetic image to test study the *gradient*, a vector-valued image that indicates the direction and magnitude of changes in the intensity, at a chosen scale. Consider the following function:

$$f(x, y) = A \sin(Vx) + B \cos(Wy) \tag{1}$$

A discrete version of this function can be made by sampling it (as you did with the Gaussian, above).

```
x = -100:100;
y = -100:100;
[X, Y] = meshgrid(x, y);
A = 1; B = 2; V = 6*pi/201; W = 4*pi/201;
F = A * sin(V*X) + B * cos(W*Y);
imshow(F, [], 'xData', x, 'yData', y);
```

Note that `X`, `Y` and `F` are arrays of size 201×201. Also note the use of `[]` in the function call to `imshow` to display `F` correctly.

⋆ **(3 points)** Calculate *analytically* the following derivatives of the function (1): f_x, f_y, f_{xx}, f_{yy} and f_{xy} and give the results in your report.

⋆ **(2 points)** Generate the images `Fx` and `Fy`, which are sampled versions of the functions f_x and f_y, that you have calculated *analytically* before.

⋆ **(4 points)** The values `Fx(i, j)` and `Fy(i, j)` define the x- and y-coordinates of the gradient vector of `F` in point `(i, j)`. The following Matlab code plots the gradient vectors on top of image `F`:

```
xx = -100:10:100;
yy = -100:10:100;
[XX, YY] = meshgrid(xx, yy);
Fx = ??;
Fy = ??;
imshow(F, [], 'xData', x, 'yData', y);
hold on;
quiver(xx, yy, Fx, Fy, 'r');
hold off;
```

Put the resulting plot (image overlayed with vectors) in your report.

⋆ **(5 points)** Write Matlab code which uses `gD` and `imfilter` to create images `Gx` and `Gy`, sampled versions of the x- and y-derivatives of `F`. Plot the image `F` and overlay a quiver using `Gx` and `Gy` instead of `Fx` and `Fy`.

Hint1: Use a scale value of 1 for this.

Hint2: You will have to "subsample" the images `Gx` and `Gy` for the sake of readability. If you do this right you can see, that this results in exactly the same plot as before.

- **(5 points)** Use the rotation function that you wrote in Exercise 1 to rotate `F` by a *small* angle (10 degrees or so). Do the calculation of `Gx` and `Gy` on the rotated version and overlay gradient vectors on it. Give a plot of that in your report and convince yourself, that the gradient vectors point in the same direction *with respect to the image*, *i.e.* they have rotated with it.

  Hint1: Due to rotation and interpolation, the vectors will probably not point *exactly* in the same direction, because they are not calculated at the very same points of the image. But the plot is enough to get the idea.

  Hint2: You might have to scale the gradient vector arrows to make them more clearly visible. Matlab uses the longest occurring arrow to automatically choose a scaling factor. By the rotation you probably introduced additional edges (at which the gradient vectors are long). This makes the other arrows appear very small. Use *e.g.* `quiver(..., 5)` to make all arrows five times as large.

- **(3 points)** Remember that the coordinate in the (local) gradient direction of an image is usually called w, see Section 6.1.5 of the Lecture Notes. *Analytically* derive the formula for the derivatives f_w and f_{ww}. Use the directional derivative formula given above to determine these derivatives of an image f in the direction of its gradient vector.

  Hint: Make sure you *normalize* the gradient vector, before you take the directional derivative. Your result should give you f_w in terms of f_x and f_y. The formula for f_{ww} contains higher order derivatives, as well.

⋆ **(10 points)** Implement the Canny edge detector using the function `gD` that you implemented before.

  The result of the function `e = canny(image, sigma)` should be an image where the value `e(i, j)` equals the gradient norm of the original image at that point, but only if `(i, j)` is an edge point. Otherwise `e(i, j)` should be zero.

  Hint: To find the zero crossings in an image you could write a simple function that looks in each 3×3 neighbourhood of a pixel, whether there are both negative and positive values on opposite sides of the central pixel. This is not a perfect solution, but it does the job quite well. (Or you could use the hint in Section 6.2.7 to emphasize the positive and negative regions.)

⋆ **(3 points)** Test your implementation on the image `cameraman.tif` (or an image of your choice). Show the results in your report. If you print out your result, please print the inverted image (so that most of the image is white with black edges on it.)

- **(5 points)** If you have time left, implement the *corner detector* of Section 6.1.7 and use it on an image of your choice.

MAX POINTS: 139