Introduction to MATLAB and Digital Image Filtering

Contents

The goals of this laboratory session are *i*) to get students familiarized with MATLAB, the scientific computing environment used in this course, and *ii*) to show how to perform basic image processing operations using such an environment; specifically, the operation of image filtering introduced in lecture 3.

1 Introduction to MATLAB

MATLAB (MATrix LABoratory) is a matrix-oriented language for technical computing. It is not only used for computation, but also for visualization and programming in an easy-to-use environment. It is an interpreted language (not compiled) that was conceived to provide easy access to matrix and linear algebra software that was written in FORTRAN. One of the main features of MATLAB is that it is oriented toward numerical computing, instead of symbolic computing (as e.g., Maple software, Mathematica). The software comes in the form of a core program and additional libraries or *toolboxes*. A toolbox is a collection of MATLAB functions (called M-functions or M-files) that extend the capability of the core environment to solve specific topic problems.

MATLAB is optimized to be relatively fast when performing array operations, so it is important to take this into account to write suitable instructions, for example, to avoid unnecessary `'for'` loops that process individual array elements.

1.1 References:

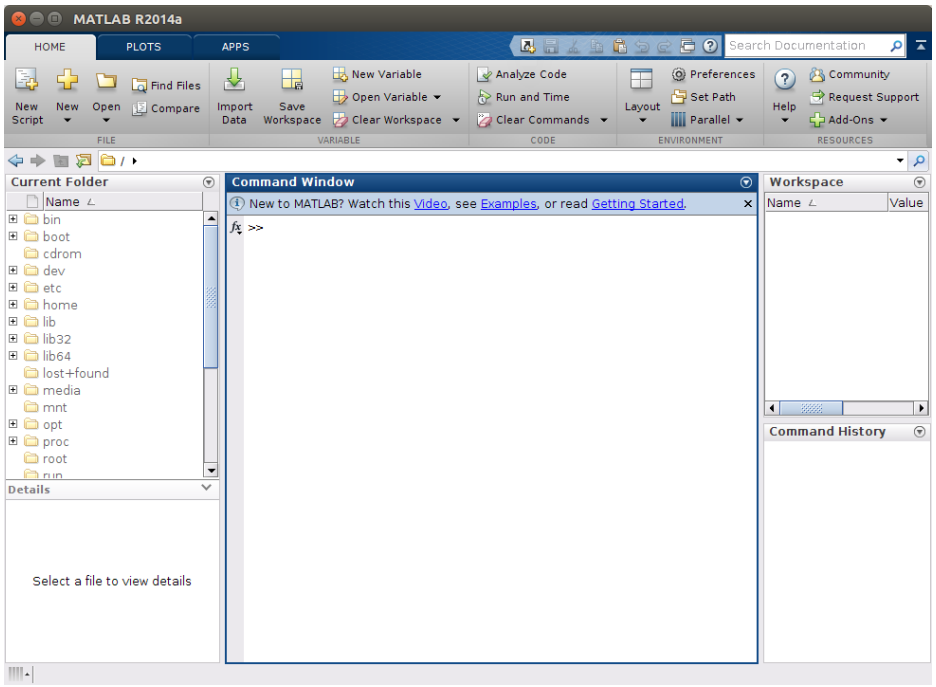
Some useful references that provide a more detailed introduction to MATLAB are available at the following URLs. It is recommended that those of you not familiar with MATLAB invest some effort in getting acquainted by practicing after this first l a b session.

Tutorials: https://maths.dundee.ac.uk/dfg/MatlabNotes.pdf
https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-matlab.pdf
Many MATLAB tutorials available on https://www.youtube.com/watch?v=OHxR8iMHDWw

Cheatsheets: https://n.ethz.ch/~marcokre/download/ML-CheatSheet.pdf
http://web.mit.edu/18.06/www/Spring09/matlab-cheatsheet.pdf
http://sites.nd.edu/gfu/files/2019/07/cheatsheet.pdf

1.2 How to start and quit a MATLAB session

Start a MATLAB session by double-clicking in the MATLAB shortcut icon (or by typing matlab in a Unix terminal). The MATLAB desktop appears, and it consists of several tools:



- the Command Window (where commands are input to the program, interactively),
- the Command History (shows all previously executed statements. It is not displayed in the latest MATLAB version),
- the Workspace (with all the variables in the current session and their properties),
- the Current Folder (and file browser),
- the Help Browser,
- and the Start button (for quick access to tools and more, in previous MATLAB versions).

An alternative method to quit the session is by typing 'exit' or 'quit' in the command-line prompt:
>> exit

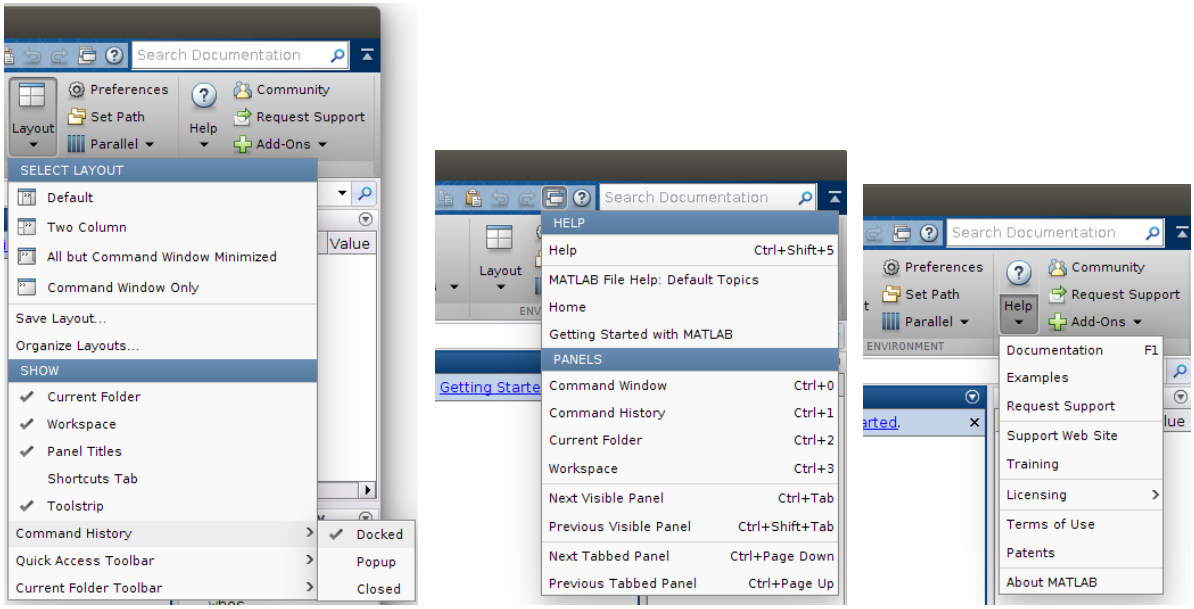


Figure 1: Layout configuration button (left); Panel shortcuts (center), and Help menu (right).

1.3 MATLAB help facilities

Type help to find a list of MATLAB help topics, type at the prompt:

```
>> help
```

To see the 'Elementary matrices and matrix manipulation' topic, type:

```
>> help elmat
```

To see the functions in the Image Processing Toolbox, type:

```
>> help images
```

To see the help for a standard plot of (x,y) data, type:

```
>> help plot
```

For window based help, from the MATLAB prompt type helpwin, or, e.g., helpwin elmat.

For a friendlier and extended documentation, you may type

```
>> doc plot
```

The following MATLAB commands are very useful to determine the available MATLAB variables, and to clear variables. To see the names of all existing variables in the current workspace, type

```
>> who
```

To see more details of the variables, type

```
>> whos
>> whos variable_name
```

To delete variables from the current workspace, use the Clear Workspace button or type

```
>> clear variable_name
>> clear    (this deletes all variables, clearing the workspace)
```

The commands `save` and `load` can be used to save (part of a) workspace into a .mat file and to load it from disk. Try

```
>> save myWorkspace.mat
>> load myWorkspace.mat
```

The command to print matlab related files in a folder is

```
>> what folder_name
```

The commands '`diary on`', '`diary off`' may be used to store in a text file called '`diary`', in the current folder, the statements written and the output generated in the command window. The written (input) statements are also available in the Command History panel.

Other useful tips: auto-completion (tab key) and review of previous commands (up and down keys).

1.4 Using MATLAB as a calculator

MATLAB can be used as a calculator simply by typing mathematical commands at the prompt. All of the usual arithmetic expressions are recognized. For example, type

```
>> 2+3
```

at the prompt and press return, and you should see

```
ans = 5
```

The variable '`ans`' (answer) contains the result of the last command executed if it is not assigned to a given variable. The basic arithmetic operators are `+ - * /` and `^` ('to the power of', e.g. 2^3=8). Parentheses ( ) can also be used. The order precedence is the usual: Parentheses are evaluated first, then powers, then multiplication and division, and finally addition and subtraction.

Writing a semicolon (;) at the end of the statement prevents from printing the output in the Command Window.

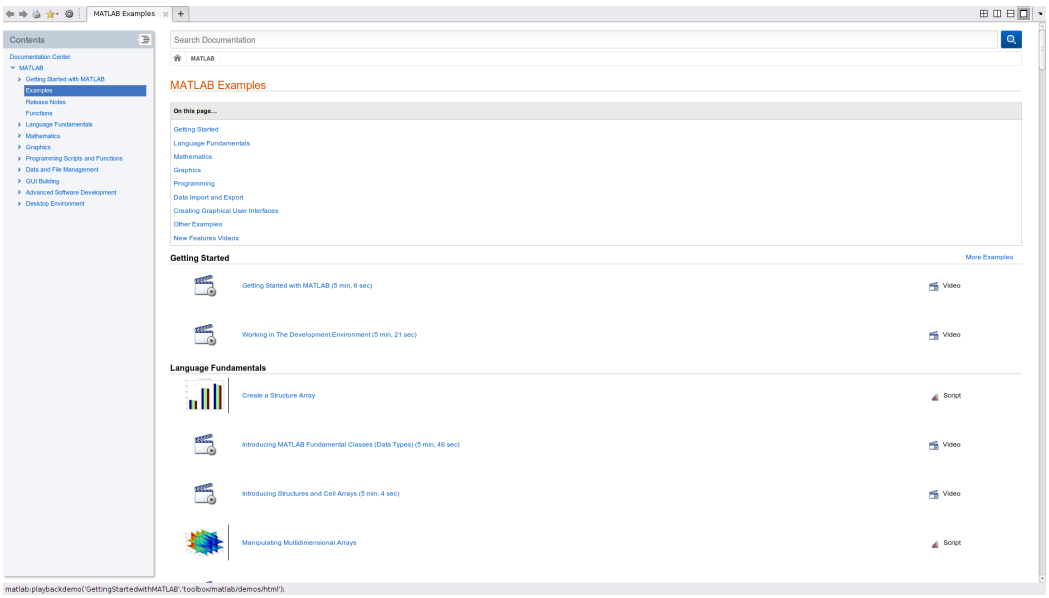
1.5 Working with matrices

The default data type in MATLAB is matrix in double floating-point precision. Then, another numeric type commonly used in image processing is `uint8` (since intensities are represented with 8 bits, in the range 0..255, thus unsigned integer).

To get started using MATLAB, type in the command window:

```
>> demo
```

And go through the excellent examples to learn the environment and some basic commands.



Small matrices can be loaded in MATLAB's memory by specifying its entries, row-wise and separated by brackets. For example, matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is loaded as:

```
>> A = [1,2,3; 4,5,6; 7,8,9];
```

Commas are unnecessary if entries are separated by a blank space. The semicolon inside the brackets signals the row breaks. Larger matrices (or vectors) with structure are more easily coded using some built-in commands: zeros, ones, zeros, repmat, linspace, meshgrid, diag, toeplitz, etc.

It is also important to learn how to access the elements of a matrix using indices. For example,

```
>> A(2,3)
```

returns the entry in the 2nd row and 3rd column of A. Multiple entries can be selected using the colon operator ':' . For example,

```
>> A(1,:)
```

selects the first row of A, whereas

```
>> A(:,1)
```

selects the first column of A. Multiple rows or columns can be selected using the statement `first_index:step:last_index`. For example, (assuming a large enough matrix A)

```
>> B=A(1:2:7,:)
```

selects rows {1,3,5,7} of A, creating another matrix (B) only with these rows. This statement is very useful in creating vectors of uniformly spaced elements. For example, the command

```
>> x = 0:0.2:1;
```

creates a row matrix with elements between 0 and 1 in steps of 0.2, i.e., matrix x = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0].

To obtain the dimensions of a matrix, type:

```
>> size(A)
```

Familiarize yourself with the functions in the *Elementary matrices and matrix manipulation* and *Matrix functions - Numerical Linear Algebra* topics, and the operations that can be carried out with them.

```
>> help elmat
>> help matfun
```

For example, arithmetic operations such as matrix addition, subtraction and multiplication are encoded in usual operators +, - and *, respectively. The inverse of a (square) matrix is requested using the `inv` command, e.g., `inv(A)`, or with the backslash command \ if followed by another matrix. The transpose of a matrix is specified with a dot followed by the single quote: `A.'` . The single quote, `A'`, computes the transpose and complex-conjugate of the matrix (MATLAB also supports complex data types). For example:

```
>> A_sym = (A+A.')/2
```

computes the symmetric part of matrix A and assigns it to variable called `A_sym`.

One thing worth mention is the difference between array and matrix operations. The former are specified with a dot whenever they differ. For example: `A.*B` computes the product of matrices A and B entry-wise, whereas `A*B` computes the matrix product (the usual one in the matrix ring).

There are many useful Linear Algebra functions, such as those that compute determinants (`det`), norms (`norm`), eigenvalues and eigenvectors (`eig`), solutions of linear systems (`linsolve`), etc.

Also, take a look at the functions/commands available in the *Image Processing Toolbox* by typing

```
>> help images
```

The functions that we will use for image processing in this session will be mentioned in the second part of these notes, where they are used.

1.6 MATLAB scripts

MATLAB programs can be executed interactively via the command line or sequentially via .m files called scripts. A script is just an .m file containing a sequence of commands that will be executed as if they were written in the prompt line. However, scripts are highly recommended because they are a convenient storage for several instructions and because they execute faster than using the command line.

In .m files, comments may be added to MATLAB code by preceding the comment with a percent (%) symbol:

```
sample code % Your descriptive comment goes here.
```

1.7 MATLAB programming: functions

Matlab comes with built-in functions. However, the user can program its own functions (in regular text files with .m extension) to process data in a customized manner. Check the MATLAB documentation on how to write your own functions with different number of input and output arguments:

```
>> doc function
```

2 Basics of Digital Image Processing

2.1 Digital Images

Monochrome (grayscale) images can be modeled by two-dimensional functions $f : \mathbb{R}^2 \to \mathbb{R}$. The amplitude of f at spatial coordinates (x, y), i.e., $f(x, y)$, is called the *intensity* or *gray level* at that point, and it is related to a physical quantity by the nature of the image acquisition device; for example, it may represent the energy radiated by a physical source. We will deal with bounded (i.e., finite) quantities, and so $|f| < \infty$. Common practice is to perform an affine transformation (substituting $f \leftarrow af + b$ for all (x, y) by means of some suitable constants a, b) so that f takes values in a specified interval, e.g., $f \in [0, 1]$.

A *digital image* can be modeled as obtained from a continuous image f by a conversion process having two steps: sampling (digitizing the spatial coordinates x, y) and quantization (digitizing the amplitude f). Therefore, a digital image may be represented by an array of numbers, $M = (m_{ij})$, where i, j and m can only take a finite number of values, e.g., $i = \{0, 1, \ldots, W - 1\}$, $j = \{0, 1, \ldots, H-1\}$ and $m = \{0, 1, \ldots, L-1\}$ for some positive integers W, H, L (Width, Height and number of gray Levels)[2]. That is, a digital image is a 2-D function whose coordinates and amplitude values are discrete (e.g., integers). Specifically, $m_{ij} = q\big(f(x, y)\big) = q\big(f(x_0 + i \cdot \Delta x,\, y_0 + j \cdot \Delta y)\big)$, where Δx and Δy are the sampling steps in a grid with spatial coordinates starting at some location (x_0, y_0), and $q : \mathbb{R} \to \{0, \ldots, L - 1\}$ is the input-output function of the quantizer (stairway shape).

Common practice for grayscale images is to use 1 byte to represent the intensity at each location (i, j) (i.e., picture element or "pixel"). Since 1 byte = 8 bits, the number of possible gray levels is $L = 2^8 = 256$, and so intensities range from $i = 0$ (black) to $i = L - 1 = 255$ (white). However, to numerically operate with grayscale images, it is convenient to convert the data type of the image values from integers to real numbers, i.e., from 8 bits to single or double precision. Once operations are finished, it may be convenient to convert back to 8 bit format for storage of the resulting image, thus producing a quantization of the data values. Medical images are usually represented with a larger depth (10-12 bits) to mitigate the occurrence of visual artifacts due to the quantization process.

Color images can be represented, according to the human visual system, by the combination of three monochrome images (with the amount of red (R), green (G) and blue (B) present in the image), and so, each pixel is represented by 3 bytes, which provides a means to describe $2^{3\times 8} \approx 16.8$ million different colors. Many of the techniques developed for monochrome images can be extended to color images by processing the three component images individually.

The number of bits required to store a (grayscale) digital image is $b = W \cdot H \cdot \log_2(L)$, and so compression algorithms (such as JPEG) are essential to reduce the storage requirement by exploiting and removing redundancy of the image.

Spatial resolution is a measure of the smallest discernable detail in an image, and it is usually given in dots (pixels) per unit distance. That is, it is the density of pixels over the image, but informally, the image size ($W \times H$ pixels) is regarded as a measure of spatial resolution (although it should be stated with respect to physical units - cm, etc.). It is common to refer to the number of bits used to quantize intensity as the *intensity resolution.*

2.2 Loading an image

To read an image from disk, use the command imread:

```
>> A = imread('example.tif');
```

To see all the supported file formats and options of this vuilt-in function, you may type

```
>> doc imread
```

Once variable A has been created, we may see its properties by typing:

[2] Actually, in MATLAB indices i and j start at 1 instead of at 0 (and consequently end in W, H, respectively). The latter is used in other languages such as C/C++, java, etc.

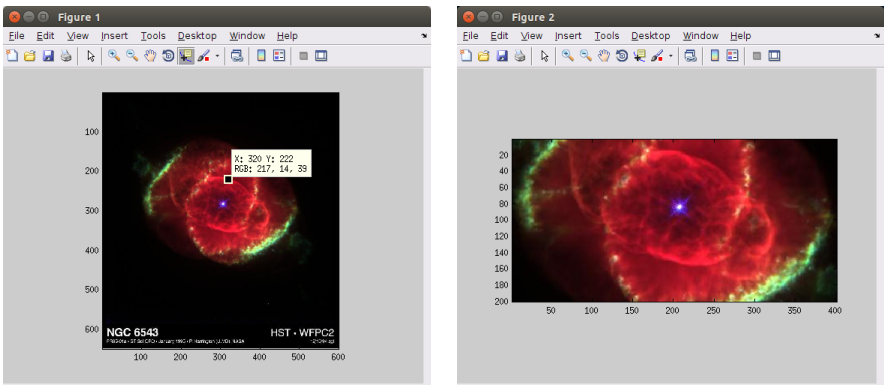


Figure 2: Figures with images 'example.tif' (left) and a part of it (right).

```
>> whos A
```

It will return that A is a variable of type `uint8` (unsigned integer represented by 8 bits = 1 byte) and it is a multi-dimensional array/matrix of size $650 \times 600 \times 3$, i.e., an image of 650×600 pixels and three components or bands, thus, it is a color image (in RGB format).

To know more about the attributes of an image, you may use the command imageinfo, which is part of the Image Processing Toolbox:

```
>> imageinfo('example.tif')
```

2.3 Displaying an image and coordinate axes convention

To visualize an image without loading it in the workspace (not creating a variable), you may type:

```
>> imageview('example.tif')
```

To visualize the image that has been previously loaded in variable A, you may type:

```
>> figure, imagesc(A)
```

This will create a new window and display a color image. By default, it will also show the tick marks in both axes, in pixel units, and in the "matrix" axes mode (`>> axis ij`), as opposed to the "Cartesian" axes mode implied by (`>> axis xy`). In the "matrix" axes mode, the coordinate system origin is at the upper left corner, with the positive i axis extending downward and the positive j axis extending to the right. This is a conventional representation from the way CRT displays worked.

To properly set the aspect ratio of the figure, use

```
>> axis equal tight
```

If you have the Image Processing Toolbox, type

```
>> figure, imshow(A,[])
```

to visualize the image in variable A. You may also try (`>> axis on`) to visualize the axes; (`>> axis off`) to hide them.

If the image was an indexed image that stores colors as an array of indices into a colormap (e.g., a GIF image), you could use (`>> colormap(gray)`) to display a binary or grayscale image.

To find out the intensity value at pixel (j, i), type:

```
>> A(222,320,:)
```

It will return three numbers R=217, G=14, B=39. Hence the color is approximately red. This is depicted in Fig. 2.

You may select and visualize part of an image using indices into the array. For example, to visualize the pixels of A in the rectangular region between rows $i \in [200, 400]$ and columns $j \in [100, 500]$, type:

```
>> B = A(200:400,100:500,:);
>> figure, imagesc(B)
```

This is also shown in Fig. 2.

Figure 3: Effect on image quality of reducing the spatial resolution of the image, i.e., the number of samples (pixels) used to represent the underlying continuous intensity signal.

You may close a figure window from command line by typing
`>> close(fig_number)`
where fig_number is the number in the top bar of the figure window.
To close all figure windows, type
`>> close all`

2.4 Writing an image to disk

To write an array into a file with an image format, we use the function imwrite:
`>> imwrite(B,'SavedImage.tif')`
This will create a file called SavedImage.tif in the working folder.
You may want to check the supported image format files by typing
`>> doc imwrite`

2.5 Resolution

Spatial resolution. Given a well-sampled digital image, the effect of reducing the spatial resolution while keeping the number of intensity levels constant is coded in the script `run_spatial_resolution.m`. To run the script, type
`>> run_spatial_resolution`
in the command window. You may have to move to the folder containing the corresponding .m file or may have to add such folder to the MATLAB path. The output produced on the input image is shown in Fig. 3. The image quality degrades as it is represented with fewer number of pixels. The coarsening effect or "pixelization" is evident when an insufficient number of pixels is used; such number of samples cannot capture the fine details present in the scene.

Intensity resolution. The script `run_intensity_resolution.m` contains the MATLAB statements to simulate the effect of reducing the intensity resolution while keeping the number of samples constant. To run the script, type in the command window:
`>> run_intensity_resolution`
The output produced on a given image with 256 intensity levels is depicted in Fig. 4. Can you discern the differences between the resulting images? How many intensity levels does the human

Figure 4: Effect on image quality of reducing the intensity resolution of the image, i.e., the number of gray levels used to represent the underlying continuous intensity signal.

visual system need to represent the scene faithfully? Observe that, as the number of levels decreases, false contours appear in smooth intensity regions. A more evident example is given in Fig. 5.

2.6 Filtering

2.6.1 1-D Filtering. Convolution

The convolution of two sequences of numbers ("signals") $a[n]$ and $b[n]$, $n \in \mathbb{Z}$, is symbolized by $c = a \star b$ and calculated according to

$$c[n] = \sum_{k=-\infty}^{\infty} a[k]b[n-k]. \tag{1}$$

The convolution is commutative ($a \star b = b \star a$), so a and b can be swapped in the previous formula. In practice, we use sequences of finite length, so the summation in (1) is carried over a finite number of products. In MATLAB, the convolution operation is implemented in the conv command:

```
>> help conv
>> c = conv(a,b)
```

A demonstration of the convolution is given in script `run_convolution.m` (illustrated in Fig. 6), where two signals a, b are defined and their convolution is computed. It also shows a movie of the way that each coefficient of the result is computed. In the command window, type

```
>> run_convolution
```

The convolution can be interpreted as a method to compute the coefficients of the product of two polynomials. Let $p(x) = x^2 - 2x + 3$ and $q(x) = 2x + 5$, whose coefficients are represented by vectors $a = [1, -2, 3]$ and $b = [2, 5]$, then the coefficients of the product polynomial $p(x)q(x)$ are given by `conv`(a, b). Check it!

Linear filtering is implemented by linear, shift-invariant systems, whose output consists of the convolution of the input signal (a) and the impulse response of the filter (b), i.e., $c = a \star b$. For images, we will be using the 2-D convolution implemented in the `filter2` and `imfilter` commands in MATLAB, but let us first take a look at their 1-D version[3]:

[3]Warning: the conv and filter commands do not provide exactly the same results for finite duration signals; there

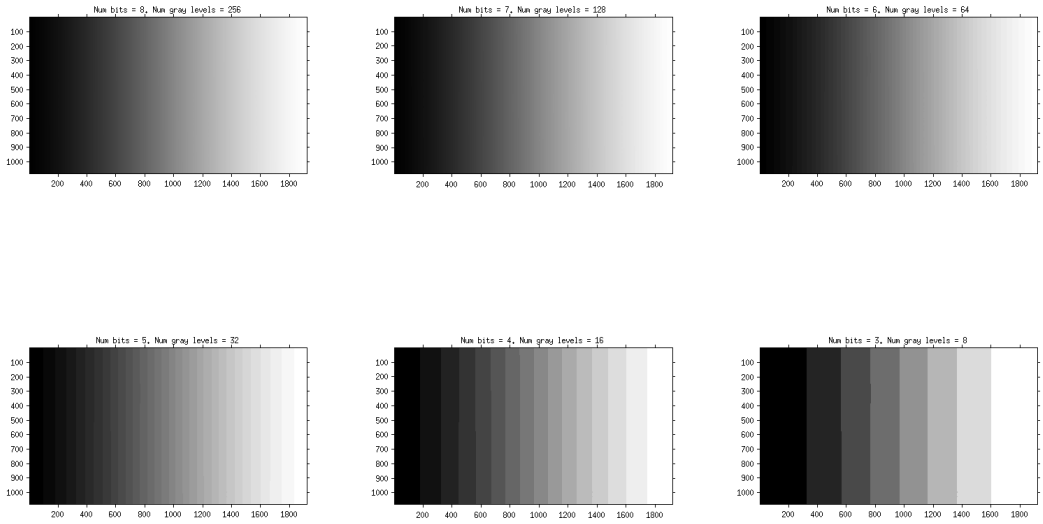


Figure 5: Effect on image quality of reducing the intensity resolution of the image, i.e., the number of gray levels used to represent the underlying continuous intensity signal.

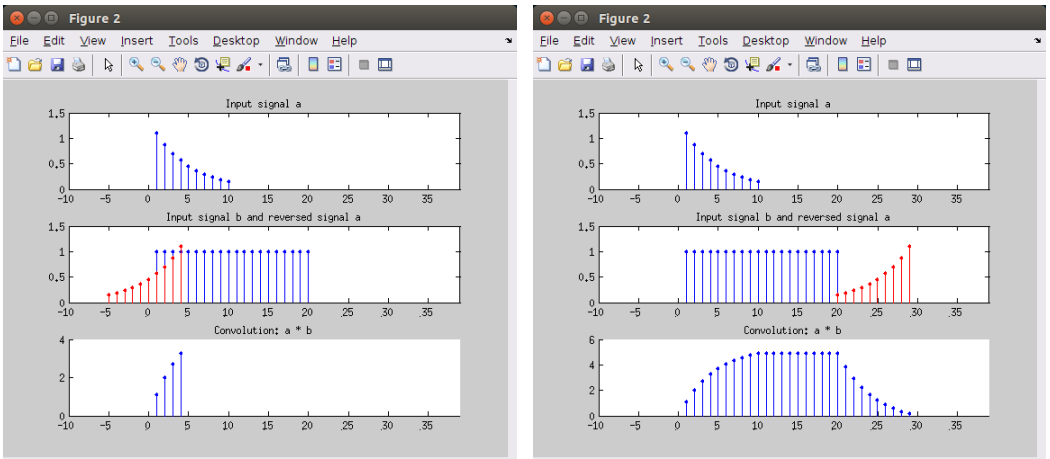


Figure 6: Example of 1-D convolution computation. One signal (middle plots, in blue) is multiplied by a reversed and shifted version (middle, red) of the other signal (top); the sum of the product gives one sample of the output signal (bottom).

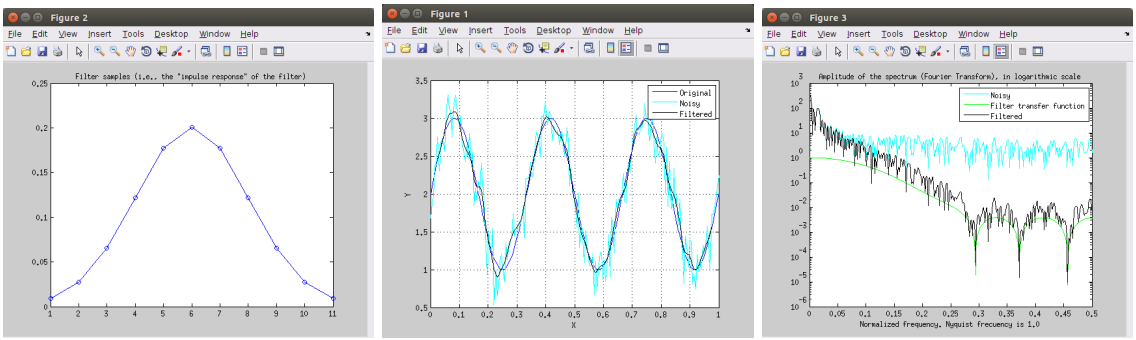


Figure 7: Example of 1-D filtering. Left: impulse response of the filter (Gaussian approximation); Center: noisy and filtered signals; Right: frequency domain interpretation.

`>> help filter`

`Y = filter(B,A,X)` filters the data in vector X with the filter described by vectors A and B to create the filtered data Y. In our case, we will use finite impulse response (FIR) filters, which have $A = 1$. Variables B, X and Y in the description of the filter function stand for a, b, c in our convolution notation, respectively. We have prepared script that demonstrates how to use the filter function to remove noise from a signal. Its output is depicted in Fig. 7.

`>> run_filter1D`

An input signal (a sine wave with a constant non-zero offset, called the DC component) corrupted by noise is passed through a low-pass filter with Gaussian shape (an approximation of it, since Gaussians are continuous and have infinite support). During the convolution performed by the filter function, each sample of the input signal is replaced by a Gaussian weighted average of its neighbors. The output of the filter is cleaner (smoother) than the input since high frequency noise has been attenuated by the filter.

2.6.2 2-D Filtering.

The convolution operation can be extend to two-dimensional discrete signals, i.e., monochrome images. Following the notation in the course slides, the convolution of two digital images h, f is given by $g = h \star f$ and calculated as

$$g[i,j] = \sum_u \sum_v h[u,v] f[i-u, j-v]. \tag{2}$$

Often, h represents the filter (and is called "kernel" or "mask") and f the input image. In this case, due to the commutativity of the convolution, it is standard to think of g as computed by reversing and shifting h instead of f in (2). Each output pixel is computed by a weighted sum of its neighbors in the input image, and if h is a filter with a kernel of size $n \times n$ pixels, this is the size of the neighborhood, and it implies n^2 multiplications.

Separable filters are of special importance since they save a lot of computations. A filter is separable if its kernel can be written as the product of two independent 1-D kernels, i.e., $h[u,v] = p[u]q[v]$. That is, the matrix $h[u,v]$ can be written as the (exterior) product of two vectors $p[u]$ and $q[v]$. This reduces the cost of convolution; the filtered image can be obtained as a cascade of the two 1-D convolutions, with $2n$ multiplications per pixel instead of the n^2 count in the general, non-separable case.

2.6.3 Applications: Blurring and Noise Reduction

So far we have seen how to read and display images as well as how to filter 1-D signals. With sample code from the scrips `run_Spatial_resolution.m` and `run_filter1D.m` let, you are prepared to

is a difference in the size of the output due to the initial conditions of the filter, but for our purposes, they produce the same result.

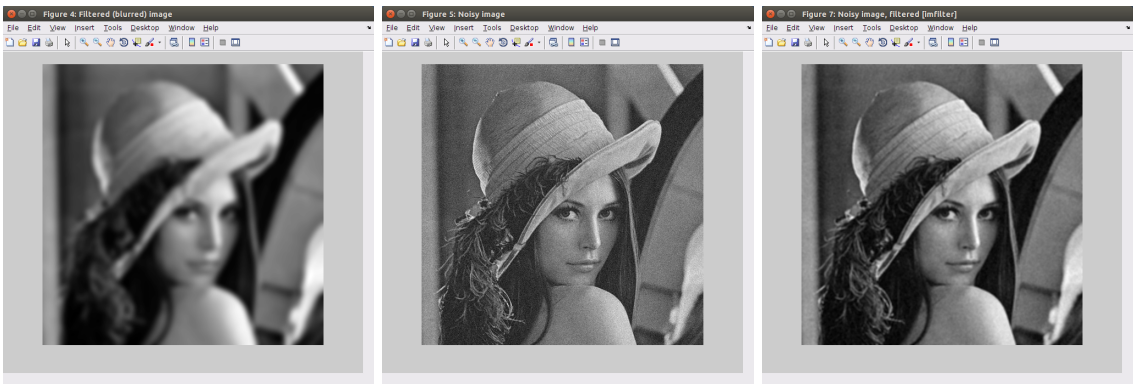


Figure 8: Left: blurred image; Center: noisy image; Right: filtered ("de-noised") image.

implement (2-D) image filtering.

First, let us load an image and pass it through a low-pass filter, for example, a Gaussian kernel $h_\sigma[u, v]$, which is an approximation of the (2-D) Gaussian function:

$$\begin{aligned} \mathbb{G}(x, y; \sigma) &= \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right), \\ h_\sigma[u, v] &= \mathbb{G}(u, v; \sigma), \quad u, v \in \{-W, \ldots, W\}, \end{aligned} \tag{3}$$

where W is half of the kernel size in each dimension.

Is this filter separable? Justify your answer.

This sample code might be useful in preparing your own script:

```
h_size = 11;
h_sigma = 5;
h = fspecial('gaussian', h_size, h_sigma);
figure, mesh(h);
f = imread('lena.jpg'); % if not grayscale, use rgb2gray()
g = imfilter(double(f), h, 'replicate');
figure('Name','Original image'), imshow(f,[])
figure('Name','Filtered image'), imshow(g,[])
```

You may want to check the documentation of the `imfilter` function to understand how it works.

Comparing the original and the filtered image, can you describe the process that took place?

Try other parameters of the filter (kernel size and σ), and see the corresponding output. For example, try $\sigma = 2$ and $\sigma = 5$ with 31×31 kernel.

Experiment with other types of filters. For example, a rectangular filter (in both dimensions), using the '`average`' option in function `fspecial`. Which one gives better visual results?

Next, add Gaussian noise to the original image (i.e., variations in intensity drawn from a Gaussian normal distribution [`>> help randn`]) and filter the noisy image with a Gaussian kernel, as before. Try, for example, a standard deviation of noise $\sigma_n = 15$ intensity levels, and try to vary the parameters of the filter to remove noise while preserving the details of the underlying clean image.

A sample script for this section is given in file `run_filter2D_noise_removal.m`, whose output is illustrated in Fig. 8.

The script also contains an example of salt & pepper noise removal (random occurrences of black and white pixels) using a linear filter (Gaussian) and a non-linear filter (median filter); see Fig. 9. Observe that linear smoothing filters do not alleviate salt and pepper noise whereas the median filter does an excellent job!

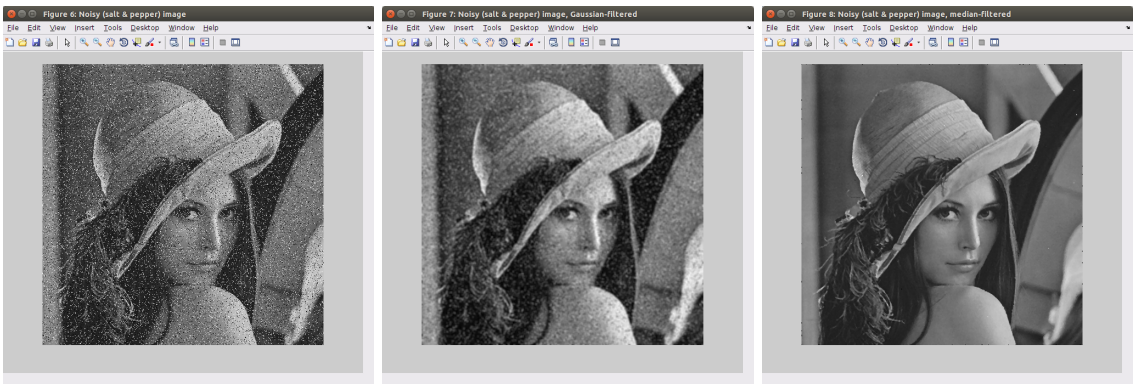


Figure 9: Salt & pepper noise (left): Gaussian (center) vs. median filter (right) noise removal.

2.6.4 Applications: Edge Detection.

Edge detection using first derivatives. Sobel filter. Linear filtering can be used to compute numerical approximations to the partial derivatives of an image. Among the filter kernels included in the function `fspecial`, check the one given by the option '`sobel`'. [`>> help fspecial`]

Are Sobel filters separable? What is the interpretation of the Sobel masks?

Use the following sample code to compute the vertical gradient image ($f_y = \partial f/\partial y$, shown in Fig. 10, top-center), which highlights horizontal edges. Since kernels to compute derivatives have signed values, the result of the convolution will also be signed (values at a pixel can be positive or negative). Because functions `imshow(A,[])` or `imagesc(A)` display the minimum value as black and the maximum as white, an image with positive and negative values will show zero intensities as middle gray.

```
f = imread('tiger.jpg')
h_Sobel = fspecial('sobel')
f_y = imfilter(double(f), h_Sobel, 'replicate');
figure('Name','Original image'), imshow(f,[])
figure('Name','Filtered image'), imshow(f_y,[])
```

Write similar lines of code to compute and display the horizontal gradient ($f_x = \partial f/\partial x$, shown in Fig. 10, top-right), which highlights vertical edges. Hint: use the transpose of the previous kernel.

At each location (x, y), the gradient of the image is given by a 2-vector $\nabla f = (f_x, f_y)$. Thus, two matrices/images are needed to describe the gradient for all pixels. To detect edges, we look for peaks in the gradient magnitude $\|\nabla f\| = \sqrt{f_x^2 + f_y^2}$, also called the edge strength (see Fig. 10, bottom-left). In MATLAB, it is easy to compute this operation for all pixels using element-wise operations, thus avoiding writing a 'for' loop:

```
>> f_grad = sqrt( f_x.^2 + f_y.^2 )
```

The information of the direction of the gradient vectors can be combined with that of the gradient magnitude visually in a single image using color coding. In the color wheel in Fig. 10, bottom-right, we use the angle of the gradient ($\theta = \angle \arctan(f_y/f_x)$) to represent different *hue* values (green, yellow, red, etc.), and the edge strength (radius) to represent different *saturation* values. Zero gradients are colored in white. The resulting color coded gradient is shown in Fig. 10.

You may compare your solution with that in the script `run_edge_detection.m`

Edge detection using first derivatives. Derivative-of-a-Gaussian filters. To mitigate the noise amplification effect caused by differentiation it is common to smooth the image prior to differentiation. An alternative method to compute image derivatives is by using the properties of the convolution operator, and combining both linear operations (smoothing and differentiation) in a single one. That is, if h_σ and h_∂ are smoothing and differentiation filters, respectively, and f is an input signal (image), then the output signal is $g = h_\partial \star (h_\sigma \star f) = (h_\partial \star h_\sigma) \star f$ due to the associativity of

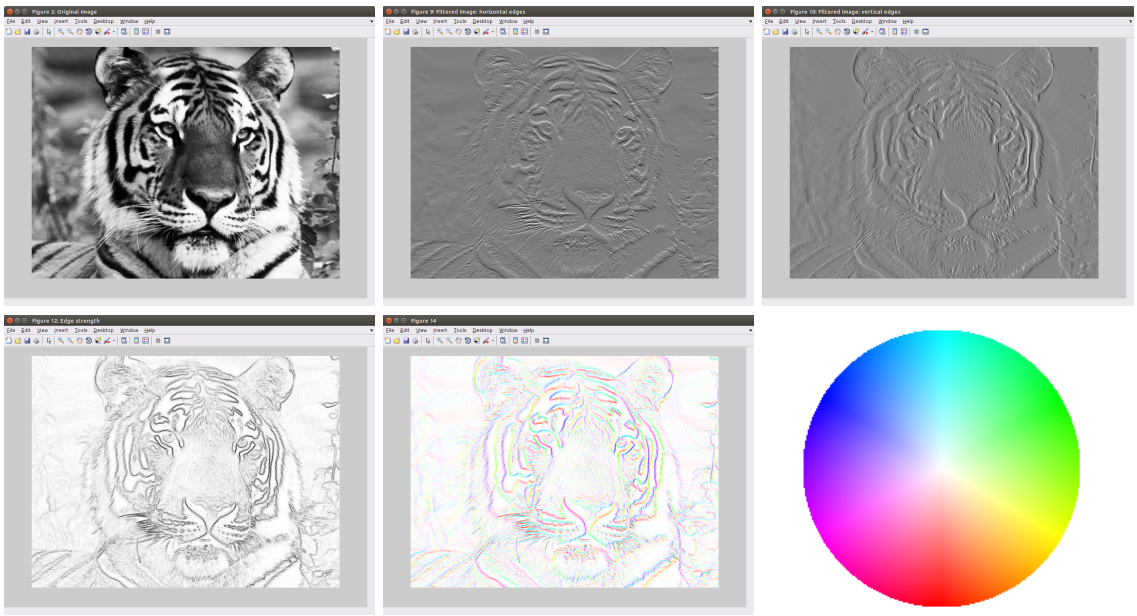


Figure 10: Edge detection via linear filtering. Top row: (left) original image f; (center) approximation to $\partial f/\partial y$, horizontal edges; (right) approximation to $\partial f/\partial x$, vertical edges. Bottom row: (left) gradient magnitude $\|\nabla f\|$ in negative grayscale [0=white, 1=black]; (center) color coded ∇f, both strength and angle; (right) color scheme used (hue = gradient angle, saturation = gradient strength).

the convolution. Hence, a single filter, $\tilde{h} = h_\partial \star h_\sigma$ suffices to perform both operations. The filter kernel is obtained by sampling the "continuous world" form of the filter: for a Gaussian smoother, from (3),

$$\begin{aligned} \mathbb{G}_x(x, y; \sigma) &= \frac{\partial}{\partial x}\mathbb{G}(x, y; \sigma) \propto -x \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \\ \tilde{h}_u[u, v] &= \mathbb{G}_x(u, v; \sigma), \quad u, v \in \{-W, \ldots, W\}, \end{aligned}$$

and similarly for the derivative in the orthogonal direction, $\tilde{h}_v$ is obtained from $\mathbb{G}_y$.

The next lines of MATLAB code create the kernels just described

```
sigma = 2;
W = ceil(3*sigma); % Half of the kernel size
[u,v] = meshgrid(-W:W, -W:W);
% Derivative in u direction emphasizes vertical edges
dG_v = -u .* exp(-(u.^2 + v.^2)/(2*sigma^2));
% Derivative in v direction emphasizes horizontal edges
dG_h = -v .* exp(-(u.^2 + v.^2)/(2*sigma^2));
```

Given the above filter kernels, use the `imfilter` command to process an input image, which will produce (approximations to) the partial derivatives of the smoothed image. Then, you may follow the usual approach of searching for peaks in the gradient to detect edges. The filtered images as well as the resulting gradient (edge information) are shown in Fig. 11. Because we used $\sigma = 2$, more smoothing than that in the Sobel filters is present, and so the surviving edges look stronger but worse localized (spread out) than those in Fig. 10. You may compare your solution with that in the script `run_edge_detection.m`

Edge detection using second derivatives. LoG filter. The `fspecial` function also has an option to return a Laplacian of Gaussian (LoG) filter, which can be used to detect edges by searching for zero-crossings of the filtered image: $\Delta I_\sigma = \nabla^2 I_\sigma$. The following MATLAB code demonstrates this application (see Fig. 12):

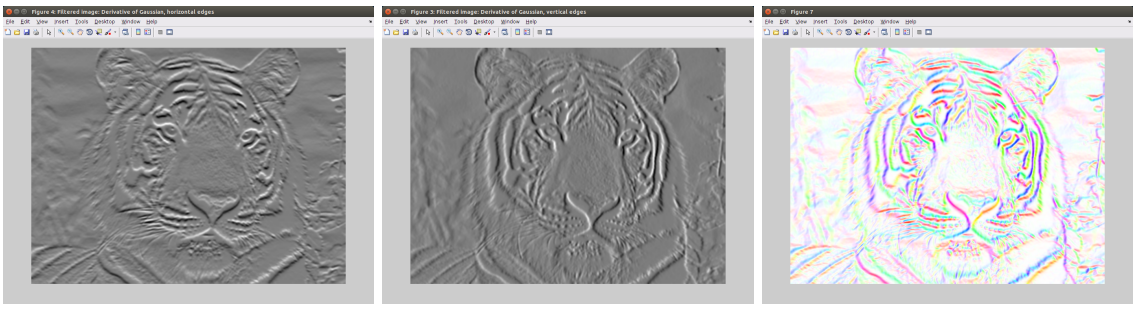


Figure 11: Edge detection via linear filtering using Derivatives of a Gaussian. Top row: (left) approximation to $\partial f_\sigma/\partial y$ with $f_\sigma = h_\sigma \star f$, horizontal edges; (center) approximation to $\partial f_\sigma/\partial x$, vertical edges; (right) color coded ∇f_σ, both strength (saturation) and angle (hue). cf. Fig. 10.

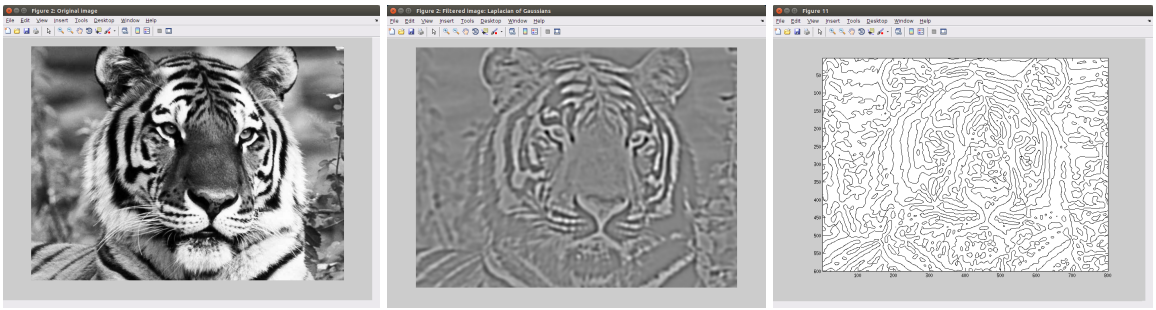


Figure 12: Edge detection according to zero-crossings of the second derivative of the image. Input image $I \equiv f$ (left); Output of an LoG filter, $\Delta I_\sigma = \nabla^2 I_\sigma$ (center); Zero-crossings of ΔI_σ, signaling the location of image edges.

```
h_LoG = fspecial('log',25,5);
figure('color','white'), mesh(h_LoG), axis vis3d % Visualize filter
f_LoG = imfilter(f,h_LoG,'replicate');
figure('Name','Filtered image: Laplacian of Gaussians')
imshow(f_LoG,[])
% Find edges by zero crossings of the output
figure('Name','Zero crossings')
contour(f_LoG,[0,0],'Color','k'), axis ij equal tight
```

The LoG filter can be approximated by the difference of two Gaussian filters with different standard deviations: $h_{\text{LoG}} \approx h_{\sigma_2} - h_{\sigma_1}$, when $\sigma_2/\sigma_1 \approx 1.6$ and $\sigma_{\text{LoG}}/\sigma_1 \approx 1.24$ in the calls to the `fspecial` command (see Fig. 14).

MATLAB's edge detection function. There is a built-in function in the Image Processing Toolbox that automatically extracts image edges by filtering, thresholding and thinning (non-maxima suppression). It implements several filters (Sobel, LoG, etc.) and the Canny method, one of the most popular in image processing. The following code demonstrates this:

```
BW = edge(f,'sobel',20);
BW = edge(f,'log',2.0);
BW = edge(f,'canny',0.25);
% Evaluate one of the lines above along with:
figure, imshow(-BW,[])
```

You may change the parameters (threshold and width σ in the Canny method), and see the results. This allows to choose whether to detect "large-scale" or fine edges.

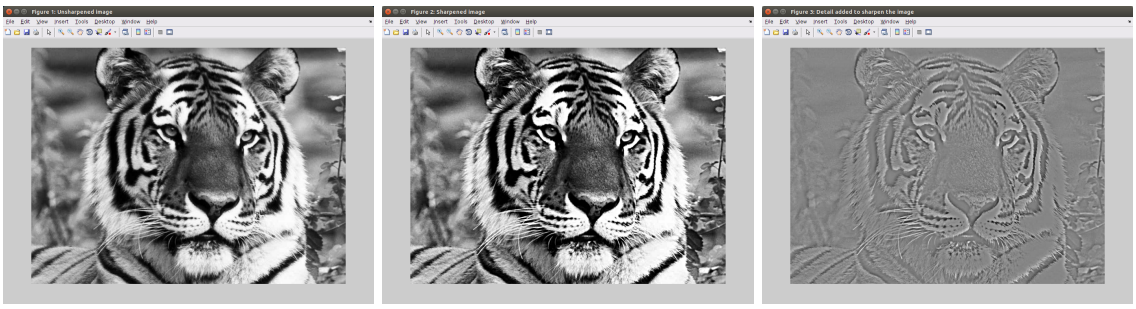


Figure 13: Image sharpening via filtering. Left: input image; center: enhanced (sharpened) image; right: detail image that was added to the input to produce the output: $f_{\text{detail}} = f - h_\sigma \star f$, where h_σ is a Gaussian (smoothing) filter.

2.6.5 Applications: Sharpening

The unsharp mask filter. Another application of linear filtering is image enhancement by sharpening, i.e., to increase fine details (differences with respect to local averages), accentuating edges. The simplest way to design a filter that achieves this goal is by interpreting the output image as the sum of the input and a scaled detail image, the latter consisting in the details that are removed by smoothing $f_{\text{detail}} = f - h_\sigma \star f$, that is,

$$f_{\text{sharp}} = \underbrace{f}_{\text{input}} + \alpha \underbrace{(f - h_\sigma \star f)}_{\text{detail}}.$$

The sharpening filter has the impulse response $h_{\text{sharp}} = \delta + \alpha(\delta - h_\sigma) = (1+\alpha)\delta - \alpha h_\sigma$, $(\alpha > 0)$ where δ (Delta) is implemented with a matrix that is zero everywhere except at the central entry, which is 1.

In MATLAB, the code to build the above sharpening filter is:

```
alpha = 0.6; % Weight of the detail added to the original image
sigma = 4; % Standard deviation of Gaussian filter (in pixels)
W = ceil(3*sigma); % Half of the kernel size
h_size = 2*W+1;
h_smooth = fspecial('gaussian', h_size, sigma);
h_sharpen = -alpha * h_smooth;
h_sharpen(W,W) = h_sharpen(W,W) + (1+alpha);
```

Use the above lines to filter an image; then visualize the result. You would have to specify the visualization interval with `imshow(f_sharp,[0,255])` to avoid losing intensity contrast due to the increased range of the output image.

You may change the amount of added detail by modifying alpha, and may change the spatial scale of the added details by varying sigma.

Using the above example, write code to create a filter whose output is the detail image that is added to sharpen the given image. Show the detail image.

The results of this experiment are depicted in Fig. 13.

Laplacian-of-Gaussian edge sharpening. Observe that the detail image in Fig. 13 is very similar to the negative output of an LoG filter ($-\Delta I_\sigma$), shown in Fig. 12. You may design and implement other sharpening filters by subtracting a weighted Laplacian from the input image. The filter would be described by $h_{\text{sharp}} = \delta - \beta\, h_{\text{LoG}}$, where $\beta > 0$ and h_{LoG} is the filter kernel of the Laplacian of Gaussians (function `fspecial`).

Image sharpening code is provided in the script `run_sharpening.m`

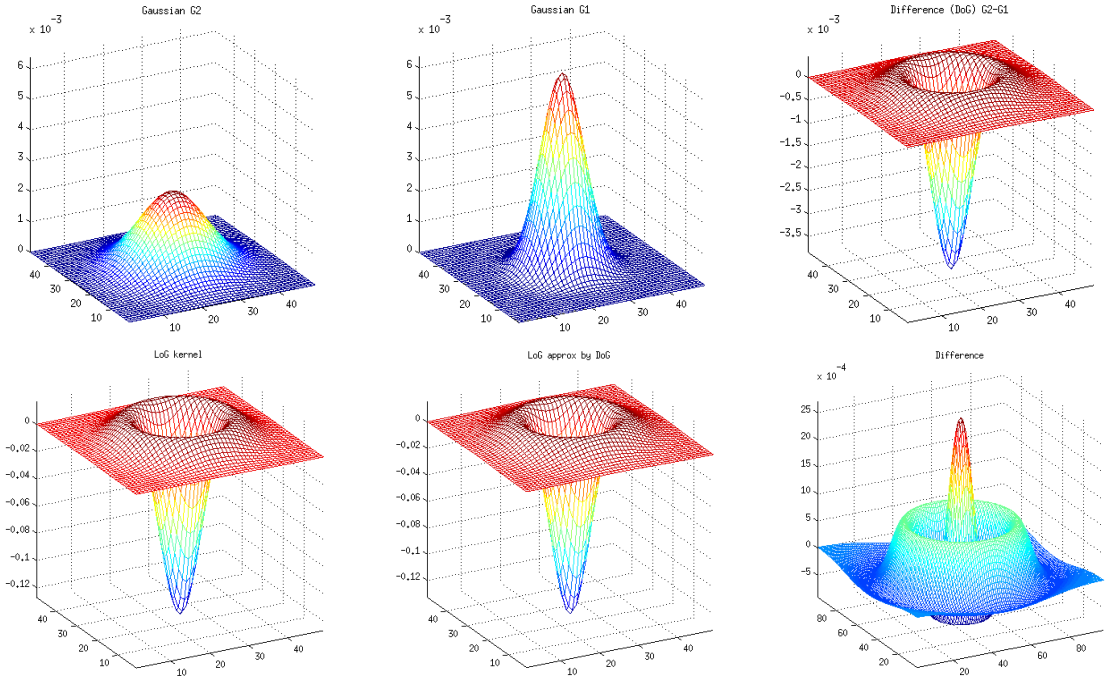


Figure 14: LoG approximation by DoG. Top row: Gaussian kernel h_{σ_2}, with $\sigma_2 = 1.6\sigma_1$ (left); Gaussian kernel h_{σ_1}, with $\sigma_1 = 5$ (center); Difference of Gaussians (DoG) kernel $h_{\sigma_2} - h_{\sigma_1}$ (right). Bottom row: Laplacian of Gaussian (LoG) kernel with $\sigma_{\text{LoG}} = 1.24\sigma_1$ in the `fspecial` command (left) and normalized to unit Frobenius norm; approximation to LoG given by the DoG kernel $h_{\sigma_2} - h_{\sigma_1}$ with $\sigma_2/\sigma_1 = 1.6$ and normalized to unit norm (center); approximation error given by the kernel difference (right).