Exercise 2: Recognition using histograms, convolution, and image filtering

Machine perception

2017/2018

Create a folder `exercise2` that you will use during this exercise. Unpack the content of the `exercise2.zip` that you can download from the course webpage to the folder. Save the solutions for the assignments as the *Matlab/Octave* scripts to `exercise2` folder. In order to complete the exercise you have to present these files to the teaching assistant. Some assignments contain questions that require sketching, writing or manual calculation. Write these answers down and bring them to the presentation as well. The tasks that are marked with ★ are optional. Without completing them you can get at most 75 points for the exercise (the total number of points is 100 and results in grade 10). Each optional exercise has the amout of additional points written next to it.

Introduction

This exercise contains two assignments. In the first one you will familiriaze yourself with several methods of histogram comparison on the domain of image retrieval. In the second assignment you will get to know convolution that is a basic operation in image processing.

Assignment 1: Global approach to image description

In the previous exercise we have looked at a single channel histogram construction. In this assignment we will upgrade this knowledge to multi-dimensional histograms and use histograms as global image descriptors to compare images.

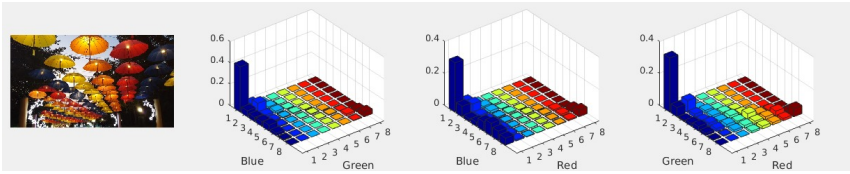
(a) We will start with an implementation of function `myhist3` that computes a 3-D histogram from three channel image (note that we assume that the image is in the RGB color space but if used with caution the function should also work for images in other color spaces). The resulting histogram is stored in a 3D matrix.

```
function H = myhist3(img, bins)
    % when computing cell indices add a small factor to avoid overflow
    % problems
    idx = floor(double(img) * bins / (255 + 1e-5)) + 1;
    H = zeros(bins,bins,bins);
    % increment the appropriate cell of the H(R,G,B) for each pixel in the image
    for i = 1:size(img, 1)
        for j = 1:size(img, 2)
            R = idx(i, j, 1);
            G = idx(i, j, 2);
            B = idx(i, j, 3);
```

```
            H(R,G,B) = H(R,G,B) + 1;
        end
    end
    % normalize the histogram (sum of cell values should equal to 1)
    H = H / sum(sum(sum(H)));
```

Analyze the code and understand what a value in a single cell of a histogram represents and how a cell in the histogram for each pixel is determined.

(b) Test the `myhist3` function on the image `umbrellas.jpg` from the first exercise. Load the image, and calculate the histogram. Since 3-D histograms are difficult to visualize on a 2-D screen, visualize the three marginal 2-D histograms that you obtain by summing up[1] the histogram matrix across a given dimension and visualizing the result using `bar3` function.



(c) ★ (10 points) Improve the performance of function `myhist3` by getting rid of a double for loop. Try calculating the 1-D index of a cell in advance and only iterate over a single vector of indices or go even further and try to get rid of the for loop by using `accumarray`. This task will only be accepted if you can show that the performance has indeed improved using timing functions `tic` and `toc` and you will show that you get the same results with both functions. Note that this task only makes sense performance-wise if you are using Octave or an older version of Matlab, newer versions of Matlab have been optimized to make the difference in the implementation almost negligible.

(d) You will now implement a function for histogram comparison that is capable of performing comparison of two histograms using several distance measures that are defined upon histograms. This function accepts three arguments, two histograms and a string that identifies of the distance used. It returns a computed distance value. Use the following skeleton as a starting point and implement the distance measures that are described below. Start with the L_2 measure.

```
function d = compare_histograms( h1, h2, dist_name )
 switch dist_name
    case 'l2'
        % TODO: d = ...
    case 'chi2'
        % TODO: d = ...
    case 'hellinger'
        % TODO: d = ...
    case 'intersect'
        % TODO: d = ...
    otherwise
        error('Unknown distance type!'); % This is how you throw an exception.
 end
```

[1] Read the documentation on function `sum` to see how to sum in different dimensions. Function `squeeze` will also come in handy.

The L_2 type distance is the Euclidean distance. Each histogram with N bins is considered as a point in N-dimensional space. For histograms h_1 in h_2 (that have equal number of bins) the L_2 distance is defined as:

$$L_2 = [\sum_{i=1:N} (h_1(i) - h_2(i))^2]^{\frac{1}{2}}. \quad (1)$$

In *Matlab/Octave* the equation above can be implemented using vector operations as:

```
d = sqrt( sum( (h1-h2).^ 2 ) ) ;
```

Implement also the following distance measures that have been presented at the lectures and are even more suitable for histogram comparison:

- Chi-square distance $\chi^2 = \frac{1}{2} \sum_{i=1:N} \frac{(h_1(i)-h_2(i))^2}{h1(i)+h2(i)+\varepsilon_0}$, where ε_0 is a very small constant value (e.g., $1e-10$) that is used to avoid pathological cases of division by zero.
- Intersection $I = 1 - \sum_{i=1:N} \min(h_1(i), h_2(i))$
- Hellinger distance $H = (\frac{1}{2} \sum_{i=1:N} (h_1(i)^{\frac{1}{2}} - h_2(i)^{\frac{1}{2}})^2)^{\frac{1}{2}}$

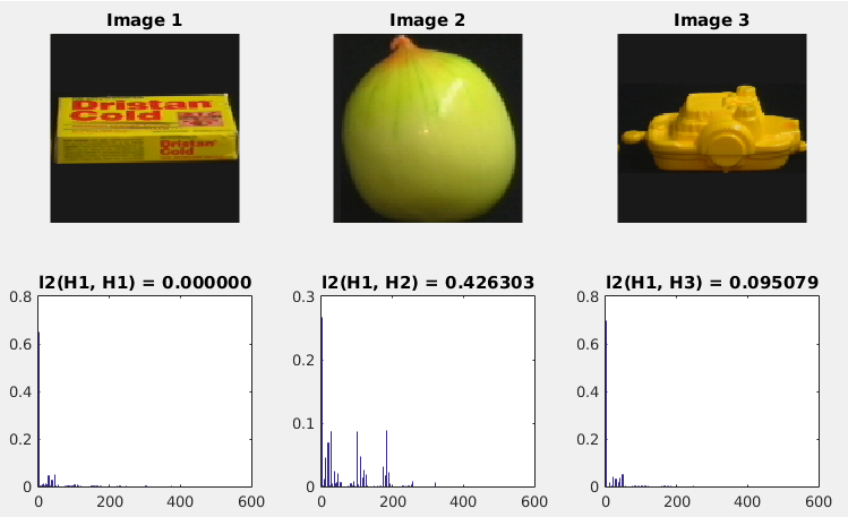
This is an excellent opportunity to learn to use vector operations in *Matlab/Octave* so that you will not have problems with this in the future. All the distance measures have to be implemented without explicit loops.

(e) Test your function with the following images:

- `images/object_01_1.png`,
- `images/object_02_1.png`,
- `images/object_03_1.png`.

Compute a $8 \times 8 \times 8$-bin 3-D histogram for each image. Vectorize all three histograms (convert them to 1-D histograms by reshaping the matrix). Using the `subplot` function display all three images in one figure together with their corresponding histograms (e.g. in a 2×3 grid). Compute the L_2 distance between histograms of object 1 and 2 as well as L_2 distance between histograms of objects 1 and 3. Remember that each histogram has to be normalized, so that the sum of its bins will be 1. This should be automatically done by the `myhist3` function.

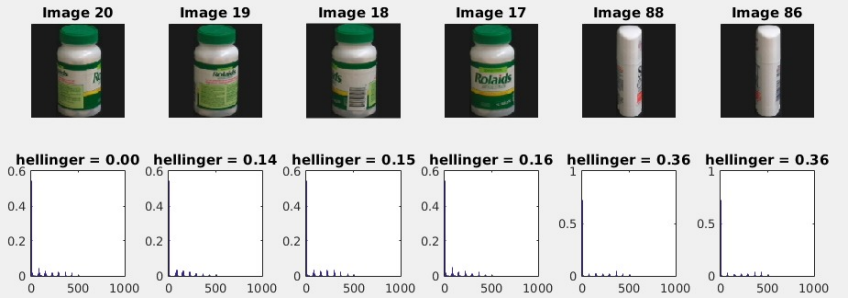
Question: Which image (`images/object_02_1.png` or `images/object_03_1.png`) is more similar to image `object_01_1.png` considering the L_2 distance? How about the other three distances? We can see that all three histograms contain a strongly expressed component (one bin has a much higher value than the others). Which color does this bin represent?



(f) Now everything is ready to implement a simple image retrieval system using histograms and their mutual distances. Implement a function `load_histogram_database` that receives two input arguments, a path to a directory with images and the number of histogram bins. The function loads images, computes a RGB histogram[2], transforms this 3D histogram to 1D histogram and returns a 2D matrix of all histograms. The resulting matrix is assembled in a way that the i-th row contains a histogram of the i-th image. Use the following snippet as a starting point:

```
function [histograms, files] = load_histogram_database(directory, bins)
files = cell(30 * 4, 1); % We will use cell array to store filenames
% initialize matrix for histograms
histograms=zeros(30 * 4, bins^3);
% calculate histogram for each image
for i = 1:30 % Iterate objects
    for j = 1:4 % Iterate orientations
        image = (i-1) * 4 + j;
        files{image} = fullfile(directory, sprintf('object_%02d_%d.png', i, j));
        % TODO: load image, extract histogram
        histograms(image, :) = % TODO: change 3D matrix to 1D vector and save it here
    end;
end;
```

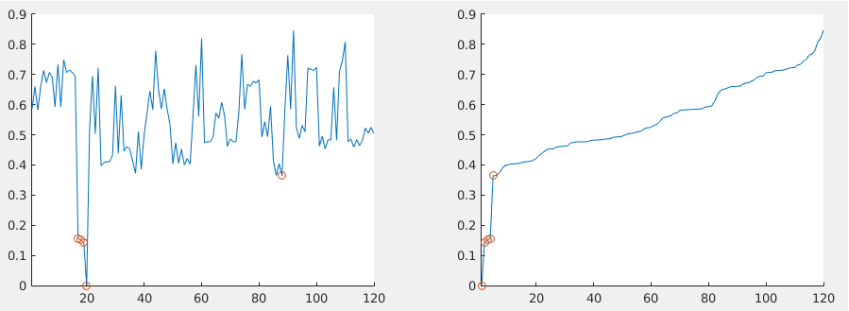
Use function `load_histogram_database` to get image histograms. Take histogram of image 20 in the list and compute histogram distances to all other images in the list (their histograms). Display the reference image and the first five images according to similarity (use function `sort` to order the sequence according to the similarity values). Plot the corresponding histograms in the same figure as well. Visualize results for all four distance measures that you have implemented.



[2]using function `myhist3(img1,bins)`

Question: Use 8 bins per color channel – which distance measure is in your opinion best suited for the specific task (elaborate your answer)? How does the retrieved sequence change if we change the number of histogram bins (e.g. 16 or 32)? Does the time required to perform the operation also change?

(g) A handy tool for distance visualization is to plot a graph that displays image indices on the x axis and a distance to the reference image on y axis. Using this kind of visualization you can see if the most similar images are indeed more similar to the reference image than the rest of the image set. Write a script that visualizes a sorted and unsorted sequence of distances this way. In both cases highlight the top five matches by drawing circles around their corresponding points (see the documentation of function `plot` for more information).



(h) ★ (5 points) One of the problems that this simple retrieval system has is strong influence of the dominant colors that are present in all images and therefore carry no discriminative information. Analyze the presence of various colors by summing up image histograms bin-wise and displaying the resulting histogram. Which bin dominates in this combined histogram? To address this we will implement a simple frequency-based weighting technique, similar to the ones that are employed in document-retrieval systems. Use the frequency histogram that you have computed to determine weighting factors for each bin. The idea is that the bins, that are strongly represented in all image histograms get lower weight and the bins that are less represented (and therefore could be more discriminative) get a higher weight[3]. Multiply each histogram bin by bin with its corresponding weight (and do not forget to normalize the histogram after that). Then compare the retrieval process for the weighted and unweighted histograms (check which are the most similar images, and what are their distances to the reference image). Write down your observations. In which ways did the weighting improve the retrieval results?

Assignment 2: Convolution

At the lectures you have looked at linear filtering operation, that is based on convolution. For easier understanding, we will first implement convolution on a 1D signal. Convolution of a kernel $g(x)$ with an signal $I(x)$ is defined with the following equation:

[3]One way of computing this weight is to use exponential function $w_i = e^{-\lambda F(i)}$, where $F(i)$ represents a frequency of the i-th bin and λ is a scaling constant that you have to set. There are also other ways of computing weights that you can experiment with.

$$I_g(x) = g(x) \star I(x) = \int_{-\infty}^{\infty} g(u) I(x-u) du, \tag{2}$$

or in the case of discrete signals (e.g. images) as

$$I_g(i) = g(i) \star I(i) = \sum_{-\infty}^{\infty} g(j) I(i-j) dj. \tag{3}$$

A nice visualization of convolution can be seen on the Wikipedia convolution entry website[4]. As the kernel is in practice of finite size, the sum, in the equation above runs only from the *leftmost element* to the *rightmost element* of the kernel. For example, let say that we have a kernel of size $N + 1 + N$ elements. In i-th element of the signal $I(i)$ the value of the convolution is defined (as a *Matlab/Octave* expression) as `I_g(i) = sum(I(i-N:i+N).*g)`. Intuitively this means that the kernel is *overlayed* upon the signal with central alignment in the i-th element, multiplied element-wise with the corresponding elements in the image which are then summed together into a response value.

(a) Compute the convolution for between the signal and kernel below $(k \star f)$ by hand.

$f =$

0	1	1	1	0	0.7	0.5	0.2	0	0	1	0

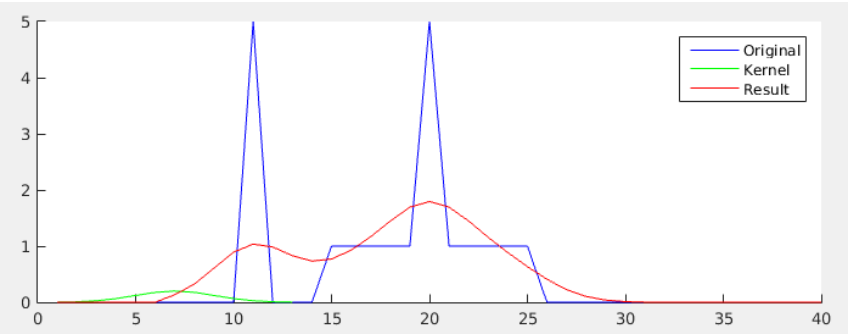
$k =$

0.5	1	0.3

(b) Implement function `simple_convolution.m`, that uses 1D signal I and a symmetrical kernel g of size $2N+1$, and computes convolution I_g. For the sake of simplicity you can start computing the convolution at $i = N + 1$ and finish with $i = |I| - N$. This means that the convolution will not be computed for the first and the last N elements of the signal I. Test the implementation using a script that loads the signal (file `signal.txt`) and the kernel (file `kernel.txt`) from the disk using function `load` and perform the convolution operation. Plot the input signal, kernel and the result of the convolution to the same figure.

```
function Ig = simple_convolution(I, g)
N = (length(g) - 1) / 2;
Ig = zeros(1, length(I));
for i = N+1:length(I)-N
i_left = max([1, i - N]);
i_right = min([length(I), i + N]);
Ig(i) = sum(g .* I(i_left:i_right));
end
```

Question: Can you recognize the shape of the kernel? What is the sum of the elements in the kernel?



[4] `http://en.wikipedia.org/wiki/Convolution`

(c) Compute the convolution again, but this time use `Ig = conv(I, g, 'same')` that is already included with *Matlab/Octave*[5]. In what way does the result differ from the result of the function `simple_convolution(I, g)`? What is the cause of this?

(d) Now, look at the frequently used *Gaussian kernel* what is defined as

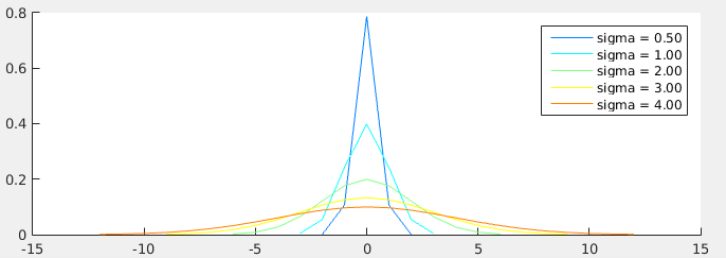
$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{x^2}{2\sigma^2}). \tag{4}$$

An important property of the Gaussian function is that its value becomes very small for $|x| > 3\sigma$. A Gaussian kernel is therefore frequently bound to the size $2 * 3\sigma + 1$.

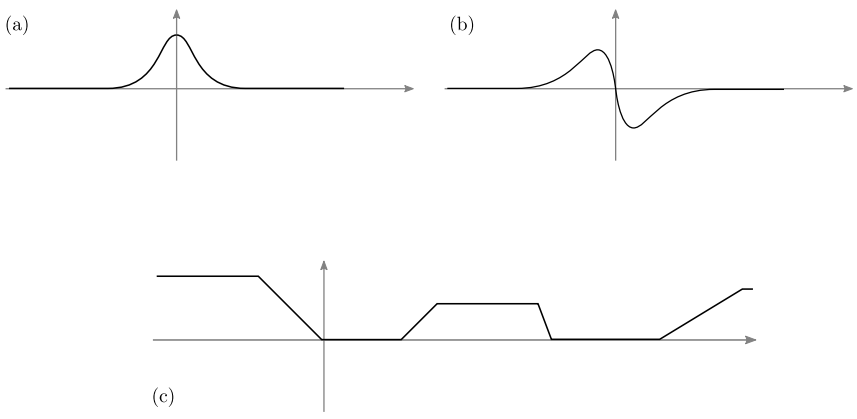
Implement a function `gauss` that is given a parameter `sigma`, and returns a corresponding Gaussian kernel.

```
function [g, x] = gauss(sigma)
x = -round(3.0*sigma):round(3.0*sigma);
g = ... % <-- !!! TODO: calculate Gaussian kernel for values of x (in one line)
g = g / sum(g) ; % normalisation
```

Generate kernel for $\sigma = 2$, and make sure that the sum of its elements is 1 and that it is similar in shape to the kernel, stored in file `kernel.txt`. Plot Gaussian kernels for values of $\sigma = 0.5, 1, 2, 3, 4$ on the same figure.

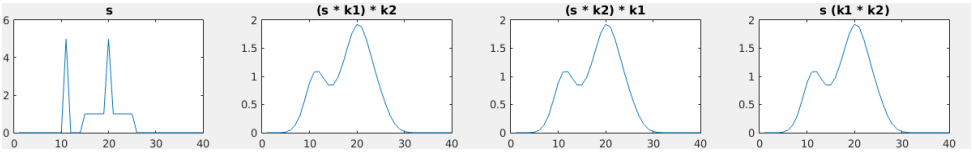


Question: The figure below shows two kernels (a) and (b) as well as signal (c). Sketch (do not focus on exact proportions of your drawing but rather on the understanding of what you are doing) the resulting convolved signal of the given input signal and each kernel. You can optionally also implement a convolution demo based on the signals and your convolution code, but the important part here is your understanding of the general idea of convolution.



[5] Check the documentation to find the meaning of the argument `'same'`

(e) The main advantage of convolution in comparison to correlation is associativity of operations. This allows us to pre-convolve multiple kernels instead of performing convolution with each one of them. Test this property by loading the signal from file `signal.txt` and then performing two consecutive convolutions on it, first one with a Gaussian kernel k_1 with $\sigma = 2$, the second time with kernel $k_2 = [0.1, 0.6, 0.4]$. In the second try switch the order of kernels. Finally, use the same input signal and convolve it with a kernel that is a product of convolution $k_1 * k_2$. Plot all three results and compare them.



Assignment 3: Image filtering

In this assignment you will learn how to use convolution on 2-D signals to perform image filtering that can be used to smooth, sharpen, or remove certain types of noise from the image. You will then also experiment with filters that are not implemented with convolution.

(a) An important property of the Gaussian kernel that has also been mentioned at the lectures is its separability in multiple dimensions. A Gaussian kernel in 2D space can be written as

$$\begin{aligned} G(x,y) &= \frac{1}{2\pi\sigma}\exp(-0.5\frac{x^2+y^2}{\sigma^2}) && (5)\\ &= [\frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{x^2}{2\sigma^2})][\frac{1}{\sqrt{2\pi}\sigma}\exp(-\frac{y^2}{2\sigma^2})] && (6)\\ &= g(x)g(y), && (7) \end{aligned}$$

that is a product of two 1D Gaussian kernels, each one in its own dimension. If we now again write a continuous convolution

$$\begin{aligned} G(x,y) * I(x,y) &= \int_u \int_v g(u)g(v)I(x-u,y-v)dudv && (8)\\ &= \int_u g(u)(\int_v g(v)I(x-u,y-v)du)dv && (9)\\ &= g(x) * [g(y) * I(x,y)], && (10) \end{aligned}$$

we can see that we get the same result if we filter a 2D signal using a single 2D Gaussian kernel or if we do it using two 1D Gaussian kernels that represent the separated components of the former 2D Gaussian kernel. This technique can be used to translate a slow nD filtering operation to a fast sequence of 1D filtering operations.

Write a function `gaussfilter.m`, that generates a Gaussian filter and applies it to an 2D image taking into account that the kernel is separable. Instead of the `conv` function use its analogy in 2D space, `conv2`. Generate a 1D kernel and use it to filter the image in the first dimension (`Ib = conv2(I, g, 'same')`) and then in the second dimension simply by transposing the kernel (`Ig = conv2(Ib, g' ,'same')`). Test the function by using the code snippet below that loads a reference image `lena.png`, transforms it to grayscale and corrupts it with a Gaussian noise (pixel value offset by a random number, sampled from a Gaussian distribution), as well as the *salt-and-pepper* noise (a portion of randomly selected pixels set to the lowest, black, or highest, white, value). Then the code uses the `gaussfilter` function (with $\sigma = 1$) to filter both corrupted images.

```
A = rgb2gray(imread('lena.png'));
Icg = imnoise(A,'gaussian', 0, 0.01); % Gaussian noise
figure;
subplot(2,2,1); imshow(Icg); colormap gray;
axis equal; axis tight; title('Gaussian noise');
Ics = imnoise(A,'salt & pepper', 0.1); % Salt & pepper noise
subplot(2,2,2) ; imshow(uint8(Ics)); colormap gray;
axis equal; axis tight; title('Salt and pepper');
Icg_b = gaussfilter(double(Icg), 1);
Ics_b = gaussfilter(double(Ics), 1);
subplot(2,2,3) ; imshow(uint8(Icg_b)); colormap gray;
axis equal; axis tight; title('Filtered') ;
subplot(2,2,4) ; imshow(uint8(Ics_b)); colormap gray;
axis equal; axis tight; title('Filtered');
```

Question: Which noise is better removed using the Gaussian filter?

(b) Another useful filter that you have heard about at the lectures is sharpening filter.

Look for its formulation in the slides, implement it and test it on image `museum.jpg`. What do you notice?

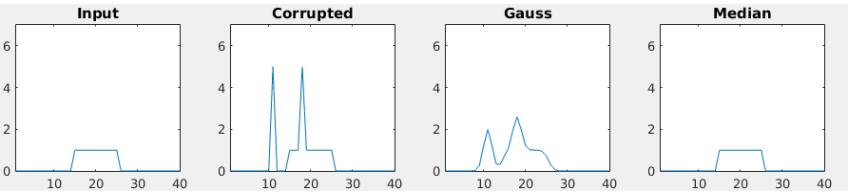
(c) Implement a nonlinear *median* filter that has also been mentioned at the lectures. In comparison Gaussian filter computes a locally weighted average values to compute a weighted mean of the signal, the median filter sorts the signal values in the given filter window and uses the middle value of the sorted sequence (a median) as a local result. Implement a simple median filter as a function `simple_median`, that takes an input signal I and a width of the filer W as an input and returns a filtered signal.

```
function Ig = simple_median(I, W)
% TODO
```

Use the snippet below to test the function. It generates a 1D step signal and corrupts it using the *salt-and-pepper* noise and then filters it using the Gaussian and median filter that you have implemented. Set the parameters of the filters that they would achieve the best reconstruction result.

```
x = [zeros(1, 14), ones(1, 11), zeros(1, 15)]; % Input signal
xc = x; xc(11) = 5; xc(18) = 5; % Corrupted signal
figure;
subplot(1, 4, 1); plot(x); axis([1, 40, 0, 7]); title('Input');
subplot(1, 4, 2); plot(xc); axis([1, 40, 0, 7]); title('Corrupted');
g = gauss(1);
x_g = conv(xc, g, 'same');
x_m = simple_median(xc, 5);
subplot(1, 4, 3); plot(x_g); axis([1, 40, 0, 7]); title('Gauss');
subplot(1, 4, 4); plot(x_m); axis([1, 40, 0, 7]); title('Median');
```

Question: Which filter performs better at this specific task? In comparison to Gaussian filter that can be applied multiple times in any order, does the order matter in case of median filter? What is the name of filters like this?



(d) ★ (5 points) Implement a 2-D version of the median filter and test it on an image, corrupted by a Gaussian noise as well as the *salt-and-pepper* noise. Compare the results with the Gaussian filter for multiple noise intensities as well as filter sizes. Compare (analytically estimate) what is the computation complexity of the

Gaussian filter operation and what is the computational complexity of the median filter using the $O(\cdot)$ notation (in the median filter we use the *quicksort* algorithm to perform sorting.

(e) ★ (5 points) Implement the hybrid image approach that was presented at lectures. To do this you also have to implement the Laplacian filter, filter two images (one with Gaussian and one with Laplacian filter) and combine them together. You can use images `cat1.jpg` and `cat2.jpg` as a reference since they are both of the same size. Do not convert them to grayscale, simply apply the same operations to all three channels and display the result as a color image.