Image Processing in MATLAB

Exercises

1 Introduction

During this exercise, you will become familiar with image processing in MATLAB. Afterwards you will hopefully have a better feel for the connection between an algorithm described as mathematical formulas (as e.g. in the book *Computer Vision: Algorithms and Applications*) and how such an algorithm might be implemented.

MATLAB is a bit different from other programming environments, such as C, C++, and Pascal. In other environments, processing of data stored in arrays is usually performed using explicit iterations over the array elements (i.e. using for loops, etc). In MATLAB, however, you usually do not have to perform iterations over the array elements, since most image processing operations can be described as element-wise operations, linear combinations, or calls to MATLAB functions. There actually exists a MATLAB for construct, but it is recommended that it is avoided, since the code will usually be a lot more concise and descriptive without it. Besides, for constructs are a lot slower than most alternatives in MATLAB.

This exercise is also meant as an opportunity for you to refresh your knowledge of Fourier transforms, and the MATLAB environment. At the same time you will also learn some simple image-processing tricks.

1.1 Preparations

Before the exercise you should have read through this exercise guide, and do not miss the mini-reference of useful MATLAB operations in section A.

1.2 Start Matlab

The first thing we need to do in MATLAB is to add the course to the path. This needs to be done each time you start MATLAB.

>> initcourse TSBB15

2 Matlab

This part demonstrates some general features of MATLAB.

2.1 The most useful function: the help function

Whenever you want to know something about some command in MATLAB, use help <command name>. It will display information on what that command does, what arguments it may be supplied with and what is returned. If you have not used it before, give it a try.

>> help find
>> help plot
>> help help

More extensive help is available by using the command doc. Try

>> doc plot
Question: How would you make a plot with a magenta dash-dotted line with hexagrams at each data point?

2.2 Identifier overloading

Enter the following commands in the given order, watch the result of each command.

```matlab
>> sin(1)
>> sin = 42
>> sin(1)
>> which sin
>> clear sin
>> which sin
```

Question: What happens? What could happen if you would have a file named `sin.m` in your current directory?

2.3 Loops and vector operations

Compare the three different ways of computing \( \sin(x_i \cdot y_i) \) present in the file `timedloops.m`. Look at the contents of the file to figure out what it does. Run the script to find out if there are any differences in calculation time.

The `toc` command shows the elapsed time since the latest `tic`. If you are really interested in how much time is spent in different parts of your program you may have a look at the `profile` command.

```matlab
>> type timedloops
>> timedloops
```

Question: What differs between the three alternatives? What should you consider when making calculations on large vectors?

2.4 Indexing and masks

In order to use the vector operations you may have to extract parts of matrices. In MATLAB, indexing can be done in most usual and unusual ways. The first of the following commands creates a nine by nine matrix \( A \) filled with random values. Try to predict the result of the other commands. An explanation of the colon operator can be found by typing `help colon`.

```matlab
>> A = rand(9,9)
>> A(2, 7)
>> A(3:6, 1:2:5)
>> A([1 1 5 5 3 1], :)
>> A(end:-2:1, 1:2:end)
>> A(17)
```
>> A(:,)
>> b = [1 7 3]
>> A(b,5) = 97
>> A(4, 7:9) = b
>> A(A<0.5) = 0
>> mask = A==0
>> A(mask) = 1:sum(mask(:))

Indexing arrays with higher dimension can be done in a corresponding way. Variables may be used as indices. Notice the order of the coordinates, (row, column). To find the minimum value of a three-dimensional array \( B \) you may use \( \min(B(:)) \) instead of \( \min(\min(\min(B))) \). The last examples selects elements from \( A \) using a matrix with logical values the same size as \( A \).

### 2.5 Reshaping and repeating matrices

Have a look at the \texttt{reshape} and \texttt{repmat} commands. What do the following commands do?

\begin{verbatim}
>> A = 1:64
>> reshape(A, 16, 4)
>> reshape(A, [4 4 4])
>> reshape(A, 4, [], 2, 2)
>> B = [1 7 2 49]
>> repmat(B, 10, 2)
>> repmat(B, 4:-1:2)
\end{verbatim}

### 2.6 Matlab function

Write a function called \texttt{imagebw} which shows a greyscale image. This function should take two parameters \texttt{imagebw(im,type)}, where \texttt{im} is the image that should be displayed and \texttt{type} controls the visualization. The function should change to a grey colormap. If \texttt{type} is 0 the colors should be mapped such that 0 becomes black and 255 becomes white with different shades of gray in between. If \texttt{type} is different from zero, \( \min(\texttt{im}) \) should map to black and \( \max(\texttt{im}) \) should map to white. Also create a colorbar to show the used range. See section A.7 and section A.8.

### 3 Test-pattern generation

We start by generating two simple gradients.

\begin{verbatim}
>> [x,y]=meshgrid(-128:127,-128:127);
>> subplot(1,2,1);imagebw(x,1)
>> subplot(1,2,2);imagebw(y,1)
\end{verbatim}

Now, use these arrays to generate an image of the function \( r = \sqrt{x^2 + y^2} \).

Question: Write down the commands you use here:

\begin{verbatim}
>> p1=cos(r/2);
>> subplot(1,2,1);imagebw(p1,1)
\end{verbatim}

We will now use this function to generate our first test image:

\begin{verbatim}
>> p1=cos(r/2);
>> subplot(1,2,1);imagebw(p1,1)
\end{verbatim}

Generate the Fourier transform of this pattern as variable \( \texttt{P1} \), and plot it.
Question: What does the Fourier transform look like? Explain why:

Question: What is the difference between \texttt{fftshift} and \texttt{ifftshift}

We will now create a test pattern with a slightly more interesting frequency content. Use the variable \( r \) to create the function \( p_2 = \cos(r^2/200) \).

Plot the image \( p_2 \), and its Fourier transform \( P_2 \).

Question: The function \( p_2 = \cos(r^2/200) \) appears to be rotationally symmetric, and its Fourier transform should then be rotational symmetric. Why then isn’t the Fourier transform of \( p_2 \) rotationally symmetric?

To get a slightly better result, we now cut off the outermost periods of the \( \cos() \) signal:

\[
\text{>> } p2 = p2.*(r.^2/200<22.5*\text{pi});
\]

You should now get a somewhat more correct representation of the transform \( P_2 \), but note that the superimposed ringings are still there.

To generate a test image with a controlled frequency content is actually not that easy. The main problem is to create a test image which locally contains one frequency and has a nice behavior in the transition from the dc part to the low frequency part.

### 4 Filtering

We will now low-pass filter our test image. We start with filtering in the Fourier domain. Here the following variables will probably come handy:

\[
\text{>> } u = x/256*2*\text{pi};
\text{>> } v = y/256*2*\text{pi};
\]

Question: How do you generate a low-pass filter in the Fourier domain?

Assuming that the Fourier domain spans the range \([-\pi, \pi]\), use the method you describe above to generate an ideal LP-filter with the cut-off frequency at \( \rho = \pi/4 \) (Hint: A scaled \( r \), and logical expressions are useful here.).

Use your filter on the transform \( P_2 \) of your test image \( p_2 \).

Question: How is filtering in the Fourier domain computed?
Compute the inverse Fourier transform of the result from your filtering, and compare this with the original image p2.

**Question**: Since the image p2 contains radially increasing frequency, one could expect that the result was attenuated outside a certain radius, and largely unaffected in the image centre. Why didn’t this happen here?

We will now apply a box filter in the spatial domain. Instead of multiplication, the filtering now consists of a convolution:

```matlab
>> lp=ones(1,9)/9;
>> p2fs=conv2(lp,lp',p2);
>> imagebw(p2fs,1)
```

**Question**: This is not good either. Why does this filter behave differently in different parts of the image?

Now, try a Gaussian filter instead:

```matlab
>> sigma=3;
>> lp=exp(-0.5*([-6:6]/sigma).^2);
>> lp=lp/sum(lp);
```

**Question**: Why does this filter work better? (Hint: Look at the filter lp'*lp.)

### 5 Multi-dimensional arrays

We will now read a colour image from disk:

```matlab
>> A=double(imread('paprika.png'))/255;
```

Which gives the same result as `im2double` in IMAGE PROCESSING TOOLBOX:

```matlab
>> A=im2double(imread('paprika.png'));
```

The `double` conversion, and the division by 255 simply converts the image data from integer values in the range [0, 255] to double precision floating point numbers in the range [0, 1]. Check the size of the resultant array:

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

The first dimension of A indexes the y coordinates of the individual pixels, the second dimension indexes the x coordinates, and the third dimension indexes the RGB (red, green, blue) components of this image. Now we will have a look at the components:
Question: Which component is which?

- Convert this image to a grey-scale image \( Ag \), using the command `rgb2gray`, and plot it where the colour image now is. Compare the grey-scale image with the RGB components.

5.1 Complex valued images

We will now combine the results of two Gaussian derivative filterings into one complex-valued field describing image orientation. Computing the derivative of a discrete signal is in general impossible, since it is the limit of a difference quotient, and we only have a sampled signal. It is however possible to compute a *regularised derivative*, i.e. the derivative convolved with a smoothing kernel:

\[
\frac{\partial}{\partial x} (f * g(\sigma))(x) = (f * \frac{\partial}{\partial x} g(\sigma))(x) = (f * \frac{-x}{\sigma^2} g(\sigma))(x)
\]

Where \( g = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \). Thus, the regularised 1D derivative filter becomes:

\[
\text{df} = \frac{1}{\sigma^2} \{[-6:6].*lp; \}
\]

Using \( lp \) and \( df \), we can compute regularised partial derivatives along the \( x \) and \( y \) axes. Apply them on the test pattern \( p2 \). Now we store the results of filtering with these kernels as a complex field, and visualise the result using the `gopimage` command:

\[
\text{fx} = \text{conv2}(lp,df',p2,'same');
\text{fy} = \text{conv2}(df,lp',p2,'same');
\text{z} = \text{fx} + \text{1i*fy};
\text{subplot}(1,2,1);\text{gopimage}(z);\text{axis image}
\]

Each complex value in \( z \) can now be seen as a vector \((\text{Re}(z) \quad \text{Im}(z))^T \) that approximates the image gradient.

Use the zoom tool to locate pixels of different colours, then extract the corresponding complex values from \( z \). For instance you could try:

\[
\text{z(139,92)}
\]

Question: Describe what the colours mean:

Now, double the argument of the gradient image:

\[
\text{z2} = \text{abs(z)}.*\exp(\text{1i*2*angle(z)});
\text{subplot}(1,2,2);\text{gopimage(z2);axis image}
\]

Question: Compare the results, and explain why the double-angle representation is useful.
Have a look at the gradient magnitude, by computing the absolute value of the z variable.

**Question:** Why are there periodic variations in the magnitude?

## 5.2 2D tensor images

If we want to estimate the local orientation, the periodic variation in magnitude we saw above is not desirable. One way to remove the periodic variations is to average the double angle representation $z^2$, why is it not possible to average $z$? Another way to avoid this problem is to replace the complex valued image $z$ with a tensor image $T$. Each position in the tensor image $T$ contains the outer product of the vector $f = (f_x, f_y)^T$, i.e. $T(x) = f(x)f(x)^T$. If we perform averaging in the tensor representation, we can smear out the magnitude such that it is largely independent of the local signal phase.

$$T = \begin{pmatrix} f_x \\ f_y \end{pmatrix} \begin{pmatrix} f_x & f_y \end{pmatrix} = \begin{pmatrix} f_x^2 & f_x f_y \\ f_y f_x & f_y^2 \end{pmatrix} \quad \text{and} \quad T_{lp}(x) = (T * g_x * g_y)(x)$$

Since $T$ is symmetric, we only need to store three of the components in MATLAB.

Now generate the tensor and look at it:

```matlab
t = zeros(256,256,3);
t(:,:,1)=fx.*fx;
t(:,:,2)=fx.*fy;
t(:,:,3)=fy.*fy;
```

Next we apply Gaussian smoothing with the appropriate standard deviation $\sigma$ on all the tensor components. Try to find a $\sigma$-value that gives just enough smoothing to remove the magnitude variation.

```matlab
lp=exp(-0.5*([-10:10]/sigma).^2);
tlp=zeros(256,256,3);
tlp(:,:,1)=conv2(lp,lp',T(:,:,1),'same');
tlp(:,:,2)=conv2(lp,lp',T(:,:,2),'same');
tlp(:,:,3)=conv2(lp,lp',T(:,:,3),'same');
```

**Question:** Write down the required $\sigma$ value:

Next we apply Gaussian smoothing with the appropriate standard deviation $\sigma$ on all the tensor components. Try to find a $\sigma$-value that gives just enough smoothing to remove the magnitude variation.

```matlab
lp=exp(-0.5*([-10:10]/sigma).^2);
tlp=zeros(256,256,3);
tlp(:,:,1)=conv2(lp,lp',T(:,:,1),'same');
tlp(:,:,2)=conv2(lp,lp',T(:,:,2),'same');
tlp(:,:,3)=conv2(lp,lp',T(:,:,3),'same');
```

**Question:** Write down the required $\sigma$ value:

**Question:** The actual local orientation property is an angle modulo $\pi$. How is this angle represented in the tensor?

## 5.3 Images as vectors

This last exercise emphasizes the fact that we can treat an image as a vector instead of a multidimensional matrix. Treating an image as a vector has the advantage that we only need to keep track of one index instead of several. Load `mystery_vector.mat` to obtain an image stored as a vector. The exercise is to
find the maximum value and the corresponding position of this value given in image coordinates. Finally reshape the vector to the original image. The original image consists of 320 columns and 240 rows.

**HINT:** One useful command is `ind2sub`.

**Question:** What is the position of the maximum value in image coordinates?

Next, we shall see that we can implement convolution as matrix multiplication. Use the $3 \times 3$ sobel-x convolution kernel on the image obtained from `mystery_vector.mat`. Compare the result obtained by ordinary convolution with the result obtained by the matrix multiplication.

**HINT:** One useful command is `convmtx2`.

### 6 Debugging

Try the debugging possibility described in section A.11 on a function of your choice. The `imagebw` function may be used.

**Question:** What is the difference between `step`, `step in` and `step out`?

In the file area there are some erroneous functions you are going to debug. Use the the command `copyfile('/site/edu/bb/ComputerVision/matlab/debug/*','~/TSBB15_lesson/')` to copy the files in the debug directory to your home area, and edit the files locally (you may get an error but the files will still be copied). The expected result from running the `display_test_image` script is shown in figure 1.

**Question:** What has to be changed in the script and/or the functions in order to get the expected result? This may be a bit tricky as there may be many different ways of fixing the program. For this exercise, all comments in the code are assumed to be valid and correct.

![Figure 1: Expected result after running display_test_image script file.](image)
A Some useful Matlab operations

We will now have a look at some MATLAB skills that you should know about before this exercise.

A.1 Transpose and conjugate
First we create a complex-valued array:

```matlab
>> A=[cos(pi/3)+1i*sin(pi/3) cos(-pi/3)+1i*sin(-pi/3)]
```

Conjugate transpose of a matrix is computed as:

```matlab
>> A'
```

Plain transpose is computed as:

```matlab
>> A.'
```

The `permute` command is a generalization of the transposition operation to multidimensional arrays.

A.2 Array creation

We will now create some MATLAB arrays. Try these commands:

```matlab
>> A=ones(3,4)
>> A=zeros(5,2)
>> A=[-3:4]'*ones(1,3)
```

The size of an array may be verified using the command `size`:

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

To simplify the creation of coordinate arrays MATLAB has the command `meshgrid`. Try the following:

```matlab
>> [X,Y]=meshgrid(1:7,1:4);
```

This will generate two arrays looking like this:

\[
X = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 2 & 3 & 4 & 5 & 6 & 7
\end{pmatrix}
\quad \text{and} \quad
Y = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 & 4 & 4
\end{pmatrix}
\]

A.3 Indexing

MATLAB arrays are indexed as matrices, i.e. row first, then column. Try for instance:

```matlab
>> A=[1:3]'*ones(1,3)
>> A=A+3*(A'-1)
>> A(3,1)
```

This will extract the element at the third row, first column of \( A \).

A.4 Element-wise operations

Most MATLAB functions that apply to scalars also work on vectors, matrices, and other multi-dimensional arrays. Try for instance:

```matlab
Mathematical operations such as:

\* / + - ~

that apply both to scalars, vectors, and matrices can be specified to work element-wise by prefixing them with a dot "\.". Try for instance:

```
>> plot(A,A.^2)
```

### A.5 Fourier transforms

Discrete Fourier transforms (DFTs), and inverse discrete Fourier transforms, can be computed through the commands `fft2`, `ifftshift`, `ifft2`, `fftshift`. i.e. a signal, $p$, and its Fourier transform, $P$, are related as:

```
>> p=fftshift(ifft2(ifftshift(P)));
>> P=fftshift(fft2(ifftshift(p)));
```

The commands `fftshift` and `ifftshift` are needed, since the MATLAB Fourier transform is defined with the DC component as the first element, and we want to view the Fourier transforms with DC in the centre of the image. `fftshift` swaps half-planes of the signal along each dimension. `ifftshift` undoes the effect of `fftshift`. `ifftshift` is needed in case the half-planes are of different sizes, for equal sizes, `fftshift` is its own inverse.

Note that there is a difference between the functions `fft2` and `fft`, although both accept 2D input, only the first calculates the 2D fourier transform. Further, if the signal is not even, the result of the transform will be complex.

### A.6 Logical expressions

In MATLAB logical expressions have numerical values. If a statement is `true` it has the value 1, if it is `false` it has the value 0. For instance, compare the plots:

```
>> x=-2*pi:.1:2*pi;
>> plot(x,1.5*cos(x))
>> hold on;plot(x,cos(x)>0)
```

### A.7 Visualisation

Visualisation of image data in MATLAB can be performed using the `image` command. i.e.

```
>> A=double(imread('paprika.png'));
>> figure(1)
>> image(A/255)
```

The `image` command work on either single band images, or RGB images, but it assumes that the intensity levels in the single band case range from 1 to the number of rows in the current colormap, and 0 to 1 in the RGB case. This is valid for images stored as floating point numbers, for images stored as unsigned integers, the ranges are different. The `imagesc` command rescales the data to fit the current colormap before showing the image. Alternatively, a second input argument can be used to specify this rescaling (see `doc imagesc`). The values and the current colormap can be visualized next to the image by calling `colorbar` after plotting the image.

Single band images by default use a colormap that moves through shades of blue, green, yellow, and red as the intensity increases. In the scope of this course, however, a grey-scale palette will probably be
more useful. The colormap command can be used to set and retrieve the current colormap. To generate a gray colormap with 256 levels, use gray(256). Setting a gray colormap can thus be accomplished by the following.\footnote{If you do not like the gray color-map, you may try another like hsv, hot, bone, copper, pink, white, flag, lines, colorcube, vga, jet, prism, cool, autumn, spring, winter or summer.}

```matlab
>> colormap(gray(256))
```

Another feature of MATLAB is that it tries to stretch images to fill the figure window. Normally however, we want square pixels. This can be obtained by the following command:

```matlab
>> axis image
```

Two more variants of the axis command are worth mentioning. axis ij will cause the vertical axis to point downwards (the matrix convention, and MATLAB default), while axis xy will cause the vertical axis to point upwards.

For this course we have also provided a command named gopimage that visualises vector data. You will use this command in this exercise.

Note: Certain combinations of image data types, image plot functions and colorbar have shown visually strange results without any error message from MATLAB. In such case, check that the image is of data type double.

### A.8 M-functions

M-functions make working in MATLAB a lot easier, since they allow you to group a set of routine operations into a function. Each M-function should reside in a separate text-file, accessible from the MATLAB path. An M-function can be written in a text editor, such as emacs or the built-in editor in MATLAB (try edit imagebw). Here is a simple example:

```matlab
function [o1,o2]=sumodiff(i1,i2)

% This function computes the
% sum and the difference of its arguments.

o1=i1+i2;
o2=i1-i2;

end
```

where i1, i2 are inputs and o1, o2 are outputs. All lines starting with a % sign are interpreted as comments.

The M-function should be saved as a file with the same base name as the function name, and the extension “.m”. The function above for instance should be saved as sumodiff.m.

It is actually possible to add additional functions after the main function in a file. These additional functions can only be accessed from other functions in the same file. They can for example be used to solve subproblems.

Concerning variables, the variables defined in the base workspace are not accessible in the functions. At the same time, variables in the function that are not returned to the caller can not be accessed from the command line. Each function also has its own workspace and local variables are not preserved between function calls.

The commands help and type are useful for finding out how a function works. help prints the first contiguous block of comment lines. For instance:

```matlab
>> help sumodiff
```
This function computes the sum and the difference of its arguments.

This is a simple but useful way to add on-line help for the functions you write. The type command simply prints all lines in the M-file. For instance:

`>> type sumodiff`

```matlab
function [o1,o2]=sumodiff(i1,i2)

% This function computes the
% sum and the difference of its arguments.

o1=i1+i2;
o2=i1-i2;
```

A.9 M-scripts

M-scripts are similar to functions, text files with the extension “.m” but without the function keyword in the beginning. Executing a script has the same effect as typing all the commands in the command window. A script uses the same set of variables as the command window or calling function.

A.10 Editor cell mode and partial code execution

The built-in text editor supports dividing a script or function into cells. A cell is limited by a double comment token, for instance:

```matlab
%% % this is the first cell
A = 7;
disp(A);

%% % this is the second cell
A = A + 4;
disp(A);
```

The commands in the active cell can be executed by pressing Ctrl+Enter. By using Shift+Ctrl+Enter, the commands in the active cell are executed and focus is shifted to the next cell. Further, pressing the F9 key will execute any selected text in the editor.

A.11 Debugging

MATLAB has some tools for debugging. It is possible to create breakpoints in the functions as well as stepping through code and watching local variables. It is also possible to do calculations and create plots of local variables while in debugging mode.

Breakpoints can be set by using the built-in editor. Lines where breakpoints can be set are marked with a dash next to the line number. By clicking on the dash, the breakpoint is set and marked with a red circle. The breakpoint may be cleared by clicking on the red circle.

When a function is invoked, the execution is halted at the first breakpoint reached while running the program.\(^2\) From this state, it is possible to step through the function and step into or over any calls to other functions. Variables can be inspected by hovering with the pointer over the variable name in the editor.

In addition to just looking at the variable values, the command window is changed to the function workspace so it is possible to use all MATLAB commands. Plotting local variables or viewing partial result images may be helpful.

\(^2\)If the breakpoint is located inside an if-clause that never is executed, the program will not stop.
A.12 The Matlab path

The commands available at the MATLAB prompt reside in one of the directories stored in the MATLAB path. To find out where a command resides, you can use the command `which`:

```matlab
>> which imagesc
```

To find out which directories are contained in the MATLAB path, simply type `path` to find out the value of this variable.

You can add a directory to the path using:

```matlab
>> addpath <new-path>
```

and remove a directory from the path using:

```matlab
>> rmpath <unwanted-path>
```