ACS61012 “Machine Vision” Lab Assignment

The purpose of the lab sessions is to give you practical skills in machine vision and especially in image enhancement, image understanding and video processing. Machine vision is essential for a number of areas - autonomous systems, including robotics, Unmanned Aerial Vehicles (UAVs), intelligent transportation systems, medical diagnostics, surveillance, augmented reality and virtual reality systems.

The first labs focus on performing operations on images such as reading, writing calculating image histograms, flipping images and extracting the important colour and edges image features. You will become familiar how to use these features for the purposes of object segmentation (separation of static and moving objects) and for the next high-level tasks of stereo vision, object detection, classification, tracking and behaviour analysis. These are inherent steps of semi-supervised and unsupervised systems where the involvement of the human operators reduces to minimum or is excluded.

Your assignment consists of several subtasks listed below and described detail in the lab session parts. This is a brief description of all your tasks:

Task 1: Introduction to machine vision:

The aim of this task is for you to learn how to read images in different formats convert them from one format to another and analyse image histograms

- **Part I of this task:** Understanding different image formats, analysis of image histogram, You can use Images from the file **File: Lab 1 - Part I - Introduction to Images and Videos.zip** or your own image).
- **Part II of this task:** Different types of image noise/ image denoising, static object segmentation based on edge detection.

For the report from Task 1, you need to present results with:

- The Red, Green, Blue (RGB) image histogram of your own picture and analysis the histogram. The original picture should be shown as well (Lab session 1 – Part I)
- Results with different edge detection algorithms, e.g. Sobel, Prewitt and comment on their accuracy with different parameters. Visualise the results and draw conclusions (Lab session 1 – Part II).

[10 marks equally distributed between part I and part II]

Task 2: Optical flow estimation algorithm:

- Find corner points and apply the optical flow estimation algorithm. (file **Lab 2.zip** – image Gingerbread Man).

[5 marks]

- Track a single point with the optical flow approach (file: Lab 2.zip – the red square image).

[9 marks]

For the report, you need to:

- Presents results for the ‘Gingerbread Man’ tasks and visualise the results
- Visualise the track on the last frame and the ground truth track of ‘Red Square’ tasks

- Compute and visualise the root mean square error of the estimated track by the optical flow algorithm in comparison with the groundtruth values (the red square).

Task 3: Automatic detection of moving objects in a sequence of video frames
You are designing algorithms for automatic vehicular traffic surveillance. As part of this task, you need to apply two types of approaches: the basic frame differencing approach and the Gaussian mixture approach to detect moving objects.

Part I: with the frame differencing approach:

- Apply the frame differencing approach (Lab 3.zip file)

  For the report, you need to present results with:
- Image results of the accomplished tasks
- Analyse the algorithms performance when you vary the detection threshold.

[10 marks]

Part II: with the Gaussian mixture approach:

- Apply the Gaussian mixture model (file Lab 5.zip)

  For the report, you need to present results showing:
- The algorithm performance when you vary parameters such as number of Gaussian components, initialisation parameters and the threshold for decision making
- Detection results of the moving objects, show snapshots of images.

[10 marks]

Task 4: Treasure hunting:

- Application of the basic image processing techniques for finding "a treasure" in an image (**Lab 4.zip** file). There are three types of images – with easy (10 marks), medium (10 marks) and high level of difficulty (there are two treasures: the sun and the clove). In the third case you need to find both treasures.

  For the report, you need to present results with:
- The three different images showing the path of finding "the treasure"
- Explain your solution, present your algorithm and the related MATLAB code

[35 marks]

Task 5. Study and compare capsule Convolutional Neural Networks (CNNs) with the Siamese CNNs and YOLO CNN with respect to: their architecture, principle of operation, advantages, disadvantages and applications – with respect to tasks such as detection, classification and segmentation.

[21 marks]

A Well-written Report Contains:

- **A title page**, including your ID number, course name, etc., followed by a content page.
- **The main part**: description of the tasks and how they are performed, including results from all subtasks. For instance: "This report presents results on reading and writing images in MATLAB. Next, the study of different edge detection algorithms is presented and their sensitivity to different parameters…" You are requested to present in Appendices the MATLAB code that you have written to obtain these results. A very important part of your report is the analysis of the results. For instance, what does the image histogram tell you? How can you characterise the results? Are they accurate? Is there a lot of noise?

- **Conclusions** describe briefly what has been done, with a summary of the main results.
- **Appendix**: **Present and describe briefly in an Appendix the code only for tasks 2-4. Add comments to your code to make it approachable and easy to understand.**
- Cite all references and materials used. Write with own style and words to minimise and avoid similarities.

Report Submission

The deadline for your report is indicated on MOLE.

The advisable maximum number of words is 4000.

Please submit: 1) your course work report in a pdf format, and 2) the code (for all assignment tasks) in a zipped file via MOLE.

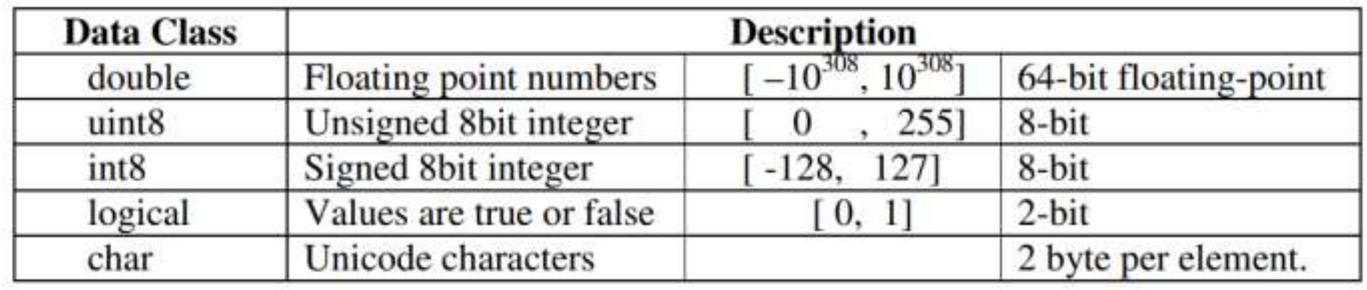
Lab Session 1 - Part I: Introduction to Image Processing

In this lab you will learn how to perform basic operations on images of different types, to work with image histograms and how to visualise the results.

Background Knowledge

A **digital image** is composed of ***pixels*** which can be thought of as small dots on the screen. We know that all numeric calculations in MATLAB are performed using *double* (64-bit) floating-point numbers, so this is also a frequent data class encountered in image processing. Some of the most common formats used in image processing are presented in Tables 1 and 2 given below.

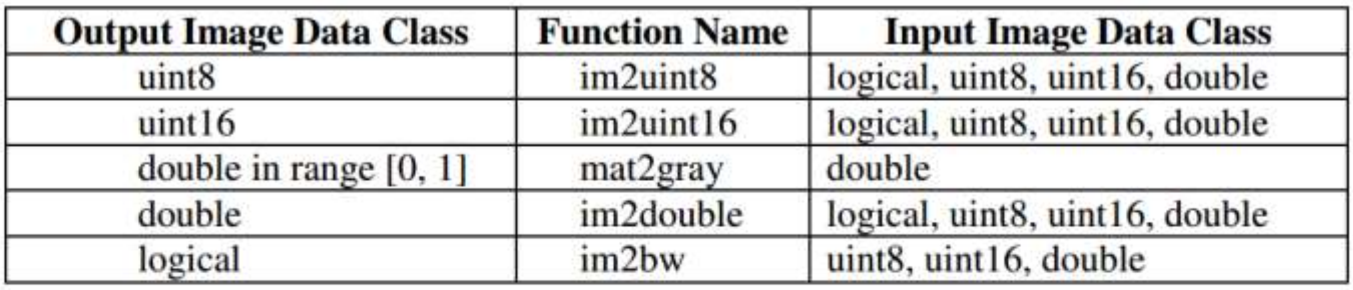
All MATLAB functions and capabilities work with double arrays. To reduce memory requirements, MATLAB supports storing image data in arrays of class uint8 and uint16. The data in these arrays is stored as 8-bit or 16-bit unsigned integers. These arrays require one eighth or one-fourth as much memory as data in double arrays.



Data Class	Description		
double	Floating point numbers	$[-10^{308}, 10^{308}]$	64-bit floating-point
uint8	Unsigned 8bit integer	[0 , 255]	8-bit
int8	Signed 8bit integer	[-128, 127]	8-bit
logical	Values are true or false	[0, 1]	2-bit
char	Unicode characters		2 byte per element.

Table 1. Data classes and their ranges

Most of the mathematic operations are not supported for types uint8 and uint16. It is therefore required to convert to double for operations and back to uint8/16 for storage, display and printing.



Output Image Data Class	Function Name	Input Image Data Class
uint8	im2uint8	logical, uint8, uint16, double
uint16	im2uint16	logical, uint8, uint16, double
double in range [0, 1]	mat2gray	double
double	im2double	logical, uint8, uint16, double
logical	im2bw	uint8, uint16, double

Table 2. Numeric formats used in image processing

Image Types

I. Intensity image (Grey scale image)

This form represents an image as a matrix where every element has a value corresponding to how bright/ dark the pixel at the corresponding position should be coloured. There are **two ways** to represent **the brightness of the pixel**:

1. The **double** class (or data type). This assigns a floating number ("a number with decimals") in the range -10^{308} to $+10^{308}$ for each pixel. Values of scaled class double are in the range [0,1]. **The value 0** corresponds to **black** and the value 1 corresponds to white.
2. The other class **uint8** assigns an integer between 0 and 255 to represent the intensity of a pixel. The value 0 corresponds to black and 255 to white. The class uint8 only requires roughly 1/8 of the storage compared to the class **double**. However, many mathematical functions can only be applied to the **double** class.

II. Binary image

This image format also stores an image as a matrix but can only colour a pixel black or white (and nothing in between): **0 – is for black** and a 1 – is for white.

III. Indexed image

This is a practical way of representing colour images. An indexed image stores an image as two arrays. The first matrix has the same size as the image and one number for each pixel. The second matrix is called the **colour map** and its size may be different from the image. The numbers in the first matrix is an instruction of what number to use in the colour map matrix.

IV. RGB image

This format represents an image with three matrices of sizes matching the image format. Each matrix corresponds to one of the colours red, green or blue and gives an instruction of how much of each of these colours a certain pixel should use. Colours are always represented with non-negative numbers.

Guidance on Performing Lab Session 1 – Part I

Demos in MATLAB

```
>> demo MATLAB % Opens a window from which you can select a demo for different tools
```

Workspace and saving results

To see the variables in the workspace:	**who, whos**
To clear the variables in the workspace:	**clear**
To save the variables in the workspace:	**save name_of_a_file.mat**
To load the data/ image from a file:	**load name_of_a_file.mat**

Examples of Reading images in MATLAB

```
>> clear all                % Clears the workspace in MATLAB

>> I = imread('Dog.jpg'); %
>> size(I)                  % Gives the size of the image
>> imshow(I);               % Visualises the image
>> Ig = rgb2gray(I);        % Converts a colour image into a grey level image
>> imshow(Ig)
```

1. The first line clears all variables from the workspace
2. The second line reads the image file into a 3 dimensional array (x, y, color). MATLAB can read many image file formats, so you do not have to worry about the details
3. Next, we will have information about the image size of the image
4. Visualise the colour image
5. This line converts an RGB image into a grey image. This is not necessary if the image is already a grey level image.
6. Visualise the grey image

Writing images in MATLAB

Images are written to disk using function imwrite, which has the following basic syntax:

imwrite(I,’filename’)

The string in filename must include a recognised file format extension (tiff, jpeg, gif, bmp, png or xwd).

```
>> imwrite(I,’Dog1.jpg’);    % The string contained in filename
```

Next, you can check the information about the graphics file, by using imfinfo.

Type: imfinfo Dog.jpg

Use the commands, whos and ls to visualise the variables in the workspace.

Changing the Image Brightness

Change the brightness of your image by adding a constant value to all pixel values, resp. by subtracting a constant value to all pixel values. For instance:

```
>> I_b = I – 100;
>> figure, imshow(I_b)
>> I_s = I + 100;
>> figure, imshow(I_s)
```

Flipping the image

Apply flipLtRt.m function (provided) to your image to flip an image. Visualise the results.

Detection of an area of a predefined colour

Change the colour of the white pixels of an image to yellow on the image *'duckMallardDrake.jpg'*:

```
% Color the duck yellow!

im= imread('duckMallardDrake.jpg');
imshow(im);
[nr,nc,np]= size(im);
newIm= zeros(nr,nc,np);
newIm= uint8(newIm);

for r= 1:nr
    for c= 1:nc
        if ( im(r,c,1)>180 && im(r,c,2)>180 && im(r,c,3)>180 )
            % white feather of the duck; now change it to yellow
            newIm(r,c,1)= 225;
            newIm(r,c,2)= 225;
```

```
                newIm(r,c,3)= 0;
            else  % the rest of the picture; no change
                for p= 1:np
                    newIm(r,c,p)= im(r,c,p);
                end
            end
        end
    end

    figure
    imshow(newIm)
```

Another example on finding an area of a predefined colour. Find the pixels indexes with the yellow colour on the image *'Two_colour.jpg'*.

```
im = imread('Two_colour.jpg'); % read the image
imshow(im);

% extract RGB channels separatelly
red_channel = im(:, :, 1);
green_channel = im(:, :, 2);
blue_channel = im(:, :, 3);

% label pixels of yellow colour
yellow_map = green_channel > 150 & red_channel > 150 & blue_channel < 50;
% extract pixels indexes
[i_yellow, j_yellow] = find(yellow_map > 0);
```

Visualise the results. Note that ***plot*** and ***scatter*** commands work with spatial coordinates.

```
% visualise the results
figure;
imshow(im); % plot the image
hold on;
scatter(j_yellow, i_yellow, 5, 'filled') % highlighted the yellow pixels
```

Conversion between different formats

1. Select your own image.
2. Read a colour image (***imread*** command). Convert the RGB colour image to grey and then to HSV format (***rgb2gray*** and ***rgb2hsv*** commands, respectively).
3. Convert your RGB image into a binary format (***im2bw*** command) and visualise the result. Use at least 3 more operations converting images from one format to another.

The conversion to a binary image is called ***binarisation***. Binarisation is based on a rough thresholding. The output binary image has values of **0 for black** for all pixels in the input image with luminance less than the threshold level and 1 (white) for all other pixels.

Understanding image histogram

1. Experiment on a grey scale image, calculate the histogram and visualise it. There are various ways to plot an image histogram: 1. imhist, 2. bar 3. stem 4. plot. Show results with them. What can you say about the objects/ images from the histograms?

Example code:

```
clear all
I = imread('image.jpg');
Im_grey = rgb2gray(I);
figure, imhist(Im_grey);
xlabel('Number of bins (256 by default for a greyscale image)')
ylabel('Histogram counts')
```

You can use the ***bar*** function to plot the image histogram, in the following way:

```
h = imhist(Im_grey);
h1 = h(1:10:256);
horz = 1:10:256;
figure, bar(horz,h1)
```

See the difference compared with what ***plot()*** function will give you:

```
figure, plot(h)
```

2. Calculate and visualise the histogram of an RGB image

In MATLAB you can only use the built in 'hist' on one channel at a time. One way to display the histogram of an image is to convert it into a grayscale format with ***rgb2gray*** and apply the ***imhist*** function. Another approach is to work with the RGB image in the following way. First, we convert the image into double and we can calculate for each channel:

```
r= double(I(:,:,1));
g = double(I(:,:,2));
b = double(I(:,:,3));
figure, hist(r(:),124)
title('Histogram of the red colour')
figure, hist(g(:),124)
title('Histogram of the green colour')
figure, hist(b(:),124)
title('Histogram of the blue colour')
```

Now repeat again the binarisation process after you choose the threshold value appropriately, based on the histogram that you observe. This threshold value must be normalised on the range [0, 1] to be used with the function ***im2bw.***

Example: If we choose the median value 128 of the full range [0, 255] as the threshold, then you can perform binarisation of image Im with the function.

```
ImBinary=im2bw(I,128/255);
```

Vary the threshold and comment on the results.

3. Calculate and visualise the histogram of an HSV image

For an HSV histogram you can use the same recommendation as for an RGB histogram, given above. Another way of calculating the histogram of in the HSV space is given below.

```
% Display the original image.
subplot(2, 4, 1);
imshow(rgbImage, [ ]);
title('Original RGB image');

% Convert to HSV color space
hsvimage = rgb2hsv(rgbImage);

% Extract out the individual channels.
hueImage = hsvimage(:,:,1);
satImage = hsvimage(:,:,2);
valueImage = hsvimage(:,:,3);

% Display the individual channels.
subplot(2, 4, 2);
imshow(hueImage, [ ]);
title('Hue Image');
subplot(2, 4, 3);
imshow(satImage, [ ]);
title('Saturation Image');
```

```
subplot(2, 4, 4);
imshow(valueImage, [ ]);
title('Value Image');

% Take histograms
[hCount, hValues] = imhist(hueImage(:), 18);
[sCount, sValues] = imhist(satImage(:), 3);
[vCount, vValues] = imhist(valueImage(:), 3);

% Plot histograms.
subplot(2, 4, 5);
bar(hValues, hCount);
title('Hue Histogram');
subplot(2, 4, 6);
bar(sValues, sCount);
title('Saturation Histogram');
subplot(2, 4, 7);
bar(vValues, vCount);
title('Value Histogram');

% Alert user that we're done.
message = sprintf('Done processing this image.\n Maximize and check out the
figure window.');
msgbox(message);
```

Include the results of understanding the RGB image histogram in your report.

Understanding image histogram – difference between one-colour and two-colour images

An image histogram is a good tool for image understanding. For example, image histograms can be used to distinguish a one-colour image (or an object in the image) from a two-colour image (or an object in the image):

1. Read *'One_colour.jpg'* and *'Two_colour.jpg'* (with ***imread***);
2. Convert both images into the greyscale format (with ***rgb2gray***);
3. Calculate and visualise the histograms for both images (with ***imhist***);

What is the differences between the histograms? Is it possible to decide according to the histograms, which image contains only one colour and which contains two colours?

Lab session 1 - Part II: Edge Detection and Segmentation of Static Objects

In this practical session, you will continue to study basic image processing techniques. You will enhance contrast of images. You will learn how to model different types of noise in images and how to remove the noise from an image. You will also learn approaches for edge detection and static objects segmentation.

Guidance on Performing Lab Session 1 – Part II

1. Read the image *'lena.gif'* (with ***imread***);

Enhancement contrast

2. Compute an image histogram for the image (***imhist***). Visualise the results. Analysing the histogram think about the best way of enhancement the image, recall the methods from the lectures;

3. Apply the histogram equalisation to the image (***histeq***). Visualise the results. Compute an image histogram for the corrected image. Visualise the results. Compare it with the original histogram. Does this method of enhancement actually enhance image quality?
4. Apply the gamma correction of the histogram to the image (***imadjust***). Visualise the results. Try different values for gamma and find the optimal one. Compute an image histogram to the corrected image. Visualise the results. Compare the histogram and the image with the original ones and the results of the histogram equalisation. Which method of enhancement performs better?

Images with different types of noise and image denoising

5. Synthesise two images from the image *'lena.gif'* with two types of noise – Gaussian and "salt and pepper" (***imnoise***). Visualise the results;
6. Apply the Gaussian filter to the Gaussian noised image (***imgaussfilt***). Find the optimal filter parameters values. Visualise the results;
7. Apply the Gaussian filter to the salt and pepper noised image (***imgaussfilt***). Make sure that no parameters values provide good results;
8. Apply the median filter to the salt and pepper noised image (***medfilt2***). Find the optimal filter parameters values. Visualise the results;

Static objects segmentation by edge detection

9. Find edges on the image *'lena.gif'* with the Sobel operator (***edge(…, 'sobel', …)***). Vary the threshold parameter value and draw conclusions about its influence over the quality of the segmented image. Visualise the results with the optimal threshold value;
10. Repeat the step 9 with the Canny operator (***edge(…, 'canny', …)***);
11. Repeat the step 9 with the Prewitt operator (***edge(…, 'prewitt', …)***);

Include the visualisation and your conclusions about static objects segmentation using edge detection (steps 9-11) in your report.

Lab Session 2: Object Motion Detection & Tracking

This lab session is focused on motion detection and tracking in video sequences. You will apply the optical flow algorithm to an object tracking by using corner points. The optical flow calculates the motion of image pixels from one frame to another.

You will apply the optical flow algorithm to the "interesting" corner points only since the numerical stability of the algorithm is guaranteed in these points only.
You need to find first the "interesting" points, and then apply an optical flow algorithm only to them.

Background Knowledge

Corner points

In many applications of image and video processing it is easier to work with "features" ("characteristic points" or "local feature points") rather than with all pixels of a frame. These "features" or "points" should differ from their neighbours in some area.
Corner points are an example of such features. **A corner point** is a point which surrounding points differ from the surroundings of its neighbours. Figure 7 shows an example of three types of points: 1) a top **corner point**, 2) an **edge point** and 3) a **point inside** the object (internal point).

- The **corner point** is surrounded with the solid line square and its neighbour point is surrounded by the dotted square. The corner point and its neighbour point have different surrounding areas.

- For the **edge point** its surrounding is the same as the surroundings of its neighbour point in one direction and it is different in any other direction.
- The **internal point** is surrounded by the same neighbourhood as all other near points around it.

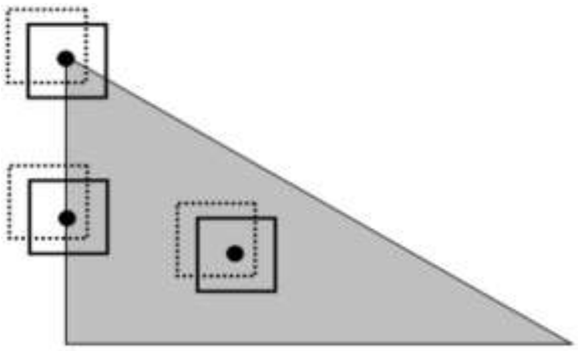


Figure 7. Illustration of the difference between corner, edge and internal points of an object. Please note that the analysed points are surrounded with a square and the dotted square indicates the area around neighbour points.

One of the most popular methods for detecting corner points is the Harris corner detector. It is used by default in the MATLAB function ***corner***.

The Optical Flow Approach

An optical flow is a vector field of apparent pixel motion between frames. Optical flow estimation is one of the widely methods for motion detection in robotics and computer vision. Given two images I_1 and I_2, optical flow estimation algorithms can find the vector field:

$$\{\boldsymbol{u}, \boldsymbol{v}\} = \{u_{i,j}, v_{i,j}\}_{i=1,j=1}^{N,M}$$

where $[N, M]$ is the image size. The vector field contains displacement vectors for each pixel. Pixel (x, y) from the image I_1 will have location $(x+u_{i,j}, y + v_{i,j})$ in the image I_2.

There are many different methods for optical flow estimation. The Lucas-Kanade algorithm is one of the most popular algorithms. This lab considers only the Lucas-Kanade algorithm. It has the following assumptions:

1. ***Brightness (colour) consistency.*** It means that pixels do not change their colour between frames.
2. ***Spatial similarity.*** It means that neighbours of each pixel have similar motion vectors.
3. ***Small displacement.*** This means that the displacement or motion vectors are small and a Taylor series expansion can be applied.

With these assumptions in place, the calculation of the optical flow reduces to solving an overdetermined linear system. This is done by the Least Square method. The conditions of the overdetermined linear system solution, lead to the Lucas-Kanade algorithm. You will apply the Lucas-Kanade algorithm to the “interesting” (“feature”) points only.

Tracking with the optical flow

Object tracking is the process of object localisation and association of its location on the current frame with the previous ones, building a trajectory for each object.
Optical flow estimation algorithms provide a tool to calculate a displacement vector from one frame to another. This information can be uses for tracking purposes. Indeed, if we determine the point of interest in the first frame, we can compute a displacement vector for

it for every successive frame, using an optical flow estimation algorithm. The combination of the positions of the points, computed by displacement vectors constitutes the trajectory of this point.

If we want to track a non-point object, we can find “interesting” points on the object, track them and use a median position of the “interesting” points as a position for the object. Since optical flow estimation algorithms are not perfect and can lose tracking points, one should reinitialise “interesting” points from time to time. At any time instant, the introduced “interesting” points should satisfy the following constrains:

- A point should not be far from the current median position of the object – it has to be inside the current bounding box;
- A point should be on the object – in your task you will use colour for this constraint;
- Each pair of tracking points has to differ from each other – if two points are too close to each other, one of them will be deleted.

As the result, we have the following algorithm:

1. Build a colour template of the object in the first frame.
2. If necessary (in your object detection task) read the next frame.
3. Detect “interesting” points of the object in the current frame. Make sure they are satisfying all the constraints, mentioned above.
4. Initialise tracks with detected and filtered “interesting” points.
5. Compute an optical flow for every “interesting” point between successive frames
6. Compute new positions of the tracks by adding the optical flow vectors to the current positions in the tracks.
7. Make sure the new positions of the tracks satisfy the second and third constraints, mentioned above. If not, delete those tracks.
8. Compute the median position of the new positions of the tracks. Move the bounding box to the new median position.
9. Make sure the new positions of the tracks are inside the bounding box. If not, delete those tracks.
10. Repeat steps 5-9. Introduce the new “interesting” points of the object in every *k* frames.

It is recommended to use *k = 5.*

Optical flow estimation and visualisation with MATLAB

From MATLAB 2015a there is an optical flow object for optical flow estimation – ***opticalFlowLK*** (http://uk.mathworks.com/help/vision/ref/opticalflowlk-class.html)
To estimate an optical flow you will use the command ***estimateFlow*** (http://uk.mathworks.com/help/vision/ref/opticalflowlk.estimateflow.html).

```
videoReader = VideoReader('…');
frameRGB = readFrame(videoReader);
frameGrey = rgb2gray(frameRGB);

opticFlow = opticalFlowLK('NoiseThreshold',0.009);
flow = estimateFlow(opticFlow,frameGrey);
```

You will use the following fields of the flow object:

- flow.Vx – the horizontal component of the velocity. size(flow.Vx) == size(frameGrey). flow.Vx(i, j) is the horizontal component of the velocity of the pixel (i, j).

- flow.Vy – the vertical component of the velocity. size(flow.Vy) == size(frameGrey). flow.Vy(i, j) is the vertical component of the velocity of the pixel (i, j).

You need the Computer Vision System toolbox from MATLAB.
For visualisation of the optical flow there are several options:

1. use the command ***plot*** (http://uk.mathworks.com/help/vision/ref/opticalflow.plot.html)
2. use the command ***quiver(u, -v, 0)***, where u, v are the horizontal and vertical displacements, respectively. Note, that it may take some time to visualise the results on your Figure.

*Moving a bounding box to a new position – help for the provided function

In the object tracking task you could move a bounding box around an object to a new position between frames. The function ***ShiftBbox*** could help perform this task.

The function ***ShiftBbox*** has two input arguments:

- **input_bbox** – the current bounding box in the format: input_bbox is a 1 x 4 vector The. input_bbox(1:2) are the spatial coordinates of the left top corner of the bounding box, input_bbox(3) is the horizontal size of the bounding box, input_bbox(4) is the vertical size of the bounding box;
- **new_center** – the new position of the centre of the bounding box in spatial coordinates

The function ***ShiftBbox*** has one output:

- **shifted_bbox** – the updated bounding box in the same format as the input_bbox argument. The centre of the updated bounding box is equal to the new_center input parameter

Guidance for performing Lab Session on Optical Flow

1. Find corner points (with the ***corner*** MATLAB function) on the images *'red_square_static.jpg'* and *'GingerBreadMan_first.jpg'*. Note that the command ***corner*** works with greyscale images. You need to convert first the input images to the greyscale format. Next, you can apply the function with different maximum number of corners. <u>Include the visualisation of the results in your report. You need to include the results only with one maximum number of corners value.</u>
2. Find optical flow of the pixels which moved from the image *'GingerBreadMan_first.jpg'* to the image *'GingerBreadMan_second.jpg'* (***opticalFlowLK***, ***estimateFlow***). Note that the function ***estimateFlow*** works with greyscale images. You need to convert the input images to greyscale format. <u>Include the visualisation of the found optical flow by any of the provided methods in your report.</u>
3. Perform tracking of a single point using the optical flow algorithm in the video *'red_square_video.mp4'*:
    a. Create an video reader object to read the 'red_square_video.mp4' video (***VideoReader***);
    b. Create an optical flow object (***opticalFlowLK***);
    c. Read the first frame (***readFrame***);
    d. Find left top point of the red square on the first frame (manually, you can use corner command to help);
    e. Add position of this point as the first position in the track;

f. Run the function ***estimateFlow*** with the first frame to initialise the optical flow object;
g. Read the next frame (***readFrame***);
h. *We know that Lucas-Kanade optical flow estimation works well only for "interesting" points. The estimateFlow function works with the current frame in comparison with the previous one. It means that we should use the "interesting" point from the current frame and not the point from the previous frame, which you detected in step c. This is the reason why we should find the nearest corner point for the position of the point of interest from the frame 1 to calculate an optical flow for it*
   Find corner points (***corner***) in frame 2;
i. Find the nearest corner point to your first position from the track;
j. Compute an optical flow (with the ***estimateFlow*** command) for this point (between frames 1 and 2);
k. Compute a new position of the point by adding the found velocity vector to the current position:

```
x_new = corner_x + flow.Vx(round(corner_y), round(corner_x));
y_new = corner_y + flow.Vy(round(corner_y), round(corner_x));
```

where corner_x and corner_y denote the coordinates of the nearest corner, *flow* is the optical flow object, the output of the estimateFlow function;

l. Add the new position of the point as the second position in the track;
m. Read the next frame (***readFrame***);
n. *As optical flow estimation is not perfect, your new point can differ from the actual corner. We also know that the Lucas-Kanade optical flow estimation algorithm works well only for "interesting" points. Hence, we should find the nearest corner point for our estimated position of the point of interest and calculate an optical flow for it.*
   Find corner points (with the ***corner*** function) in frame 3;
o. Find the nearest corner point to your second position from the track;
p. Compute an optical flow (with ***estimateFlow***) for this nearest point (between frames 2 and 3);
q. Compute a new position of the point by adding the found velocity vector to the current position;
r. Add the new position of the corner as the third position in the track;
s. Read the next frame (***readFrame***);
t. Find corner points (***corner***) in frame 4;
u. Find nearest corner point to your third position from the track;
v. Compute an optical flow (***estimateFlow***) for this nearest point (between frames 3 and 4) and so on.

Visualise the track on the last frame of the video. Visualise also the ground truth track (you are provided with the file **red_square_gt.mat**, which contains the correct track of the left top point of the red square in the variable *gt_track_spatial*. Note that the ground truth track is in spatial coordinates). Include the visualisation in your report. Please use the zoom-in functionality of the MATLAB figure in order to visualise well the estimation errors.

Compute the squared error of the estimated track by the optical flow algorithm in comparison with the ground truth values. Compute the error for each point of the track. Plot the results. Include the plot in your report. Make the conclusion about the accuracy of the method.

Lab Session 3: Background Subtraction

This lab session is focused on video processing, in particular on background subtraction methods for automatic object detection from static cameras. You will learn how the basic frame differencing algorithm for object detection works in a sequence of video frames, coming from an optical video camera. You will compare the results with the Gaussian mixture model for background subtraction.
For this lab session you will need to have the Computer Vision System Toolbox in MATLAB.

Background Knowledge

Background subtraction

As the name suggests, background subtraction is the process of separating out foreground objects (the moving objects) from the background (the static environment) in a sequence of video frames. The process can be performed off-line, but more commonly is needed in real time. Background subtraction is used in many emerging video applications, such as video surveillance, traffic monitoring, and gesture recognition for human-machine interfaces, to name a few.

Frame differencing approach

The frame differencing approach is the simplest form of background subtraction. It usually works on the video frames, after converting them from colour to greyscale format. Hence, the first thing to do is to convert the video frames arriving from the camera in RGB or HSV format to a greyscale format. Next, the current grey scale frame is simply subtracted from the previous frame, and if the difference in pixel intensity values for a given pixel is greater than a pre-specified threshold T_s, the pixel is considered as being a part of the foreground

$$|frame_i - frame_{i-1}| > T_s$$

The algorithm steps are listed below:

1. Convert the incoming frame *'fr'* to greyscale (here we assume a color RGB sensor)
2. Subtract the current frame from the background model *'bg_bw'* (in this case it is just the previous frame)
3. For each pixel, if the difference between the current frame and backround *'fr_diff (j,k)'* is greater than a threshold *'thresh'*, the pixel is considered part of the foreground.

Gaussian mixture model approach

In the Gaussian mixture model approach one builds the model of a background. It is assumed that the intensity of the background pixels is changeable. The distribution of the intensity is modelled as the mixture of Gaussian distributions. All the components of the mixture are scored based on the component weight in the mixture and its variance (respectively standard deviation). The components with the bigger scores are labelled as background, and others are labelled as foreground.
This approach is implemented in MATLAB, in the Computer Vision System Toolbox. You should use ***vision.ForegroundDetector*** object to perform foreground detection by the Gaussian mixture model approach.

Guidance for Performing Lab Session on Background Subtraction

Video reading/writing

1. Create the object to read the video 'car_tracking.mp4':

```
source = VideoReader('car-tracking.mp4');
```

Note that VideoReader supports the following file extensions: “.avi”, “.mj2”, “.mp4” or “.m4v” and others;

2. Create the object to write the video into the file ‘results.mp4’:
   ```
   output = VideoWriter('results.mp4', 'MPEG-4');
   ```

   Note that VideoWriter supports the following file extensions: “.avi”, “.mj2”, “.mp4” or “.m4v”;
3. Open the writer object in order to have an opportunity to write anything to the file:
   ```
   open(output);
   ```
4. Read a frame from the input file:
   ```
   frame = readFrame(source);
   ```
5. Visualise the frame:
   ```
   imshow(frame);
   ```
6. Write the frame to the output file:
   ```
   writeVideo(output, frame);
   ```
7. Make a loop to read and write frames. In order to read all frames you can check whether the reader object still has frames:
   ```
   while hasFrame(source)
       frame = readFrame(source);
       imshow(frame);
       writeVideo(output, frame);
   end
   ```
8. To finalise the output video close the video writer object:
   ```
   close(output);
   ```

Frame differencing algorithm for background subtraction

9. Open the script *“Frame_difference.m”*. You should vary the threshold parameter in the frame differencing algorithm for background subtraction and draw conclusions. The following steps explain the commands in the script;
10. Create and open a video reader object to read the video *‘car-tracking.mp4’* (with ***VideoReader***);
11. **Set the threshold parameter value.** Vary the threshold and comment on its influence over the detection process. Include your comments in your report, take snapshot video frames and give them in your report in order to support your conclusion.
12. Read the first frame as a background (with ***readFrame***) and convert it to the greyscale format (***rgb2gray***);
13. Loop on frames:
    a. Read a new frame (***readFrame***);
    b. Convert the new frame to the greyscale format (***rgb2gray***);
    c. Compute the difference between the current frame and the background frame;
    d. Create the frame with foreground mask, where the difference is bigger than the threshold the output pixel is white, otherwise is black;
    e. Update the background frame with the current one;
    f. Visualise the results;
    g. Write the foreground frame to the output video;
14. Close the video writer object

Gaussian mixture model algorithm for background subtraction

15. Open the script *“Gaussian_mixture_models.m”*. You need to vary the number of initial frames to learn the background model and the number of Gaussain components in the mixture of the Gaussian mixture model algorithm for background

subtraction and draw conclusions. The script is very similar to the previous one, so we highlight only differences:

a. You will use the foreground detector object from the Computer Vision System toolbox in MATLAB – ***vision.ForegroundDetector()***;
b. You need to vary two parameters:

    i. The number of initial frames for training a background model;
    ii. The number of Gaussians in the mixture;

c. To apply the foreground detector to a new frame use the command ***step***. It returns the foreground mask of the frame in the logical format.

Comment on how the parameters influence the detection performance in your report. Take snapshot video frames and show them in your report in order to support your conclusions.

MATLAB scripts

Frame_difference.m

```
clear all
close all

% read the video
source = VideoReader('car-tracking.mp4');

% create and open the object to write the results
output = VideoWriter('frame_difference_output.mp4', 'MPEG-4');
open(output);

thresh = 25;        % A parameter to vary

% read the first frame of the video as a background model
bg = readFrame(source);
bg_bw = rgb2gray(bg);           % convert background to greyscale

% --------------------- process frames ----------------------------------
% loop all the frames
while hasFrame(source)
    fr = readFrame(source);     % read in frame
    fr_bw = rgb2gray(fr);       % convert frame to greyscale
    fr_diff = abs(double(fr_bw) - double(bg_bw));  % cast operands as double to
avoid negative overflow

    % if fr_diff > thresh pixel in foreground
    fg = uint8(zeros(size(bg_bw)));
    fg(fr_diff > thresh) = 255;

    % update the background model
    bg_bw = fr_bw;

    % visualise the results
    figure(1),subplot(3,1,1), imshow(fr)
    subplot(3,1,2), imshow(fr_bw)
    subplot(3,1,3), imshow(fg)
    drawnow

    writeVideo(output, fg);           % save frame into the output video
end

close(output); % save video
```

Gaussian_mixture_models.m

```
clear all
close all

% read the video
source = VideoReader('car-tracking.mp4');

% create and open the object to write the results
output = VideoWriter('gmm_output.mp4', 'MPEG-4');
open(output);

% create foreground detector object
n_frames = 10;    % a parameter to vary
n_gaussians = 3;    % a parameter to vary
detector = vision.ForegroundDetector('NumTrainingFrames', n_frames,
'NumGaussians', n_gaussians);

% --------------------- process frames -----------------------------------
% loop all the frames
while hasFrame(source)
    fr = readFrame(source);     % read in frame

    fgMask = step(detector, fr);    % compute the foreground mask by Gaussian
mixture models

    % create frame with foreground detection
    fg = uint8(zeros(size(fr, 1), size(fr, 2)));
    fg(fgMask) = 255;

    % visualise the results
    figure(1),subplot(2,1,1), imshow(fr)
    subplot(2,1,2), imshow(fg)
    drawnow

    writeVideo(output, fg);           % save frame into the output video
end
close(output); % save video
```

Labs 4 and 5: Image Processing Techniques for Decision Making (Treasure Hunting)

In this practical session you will apply image processing techniques to perform autonomous decision making from images. You will develop an algorithm for finding the “treasure” in images, to replace a human operator.

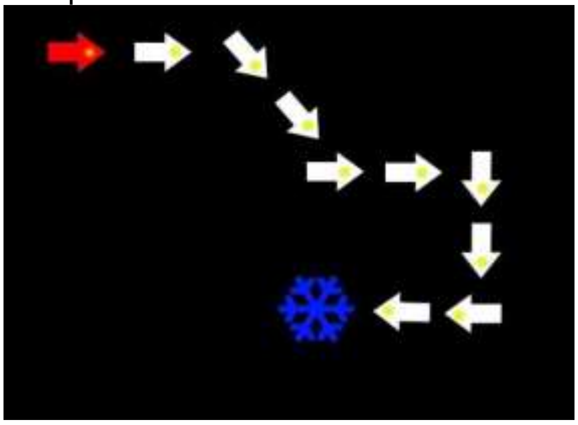


Figure 1 Image containing the treasure that needs to be found

You are provided with several images, where there are arrows and objects, such as on the Figure 1. The task is to develop an algorithm that will begin the treasure search process from the starting arrow (the red one in the presented image) and should follow arrows until reach one of the objects, which is the treasure:
You are provided with a starting script which you should complete with several functions.

Background Knowledge

Image plane (Recalling)

In MATLAB there are several types of coordinate systems for images, we will focus on two of them: pixel and spatial frames.
Pixel coordinates
When you read an image with *imread* command you get a 3D matrix (for an RGB image) or 2D matrix (for a grey or binary image). These are shown on Figures 2 and 3, respectively.
You can use the matrix elements (row and columns) to access pixel values. For example, to get an intensity level of the highlighting pixel on the right figure you can use:

```
im = imread('image')
```

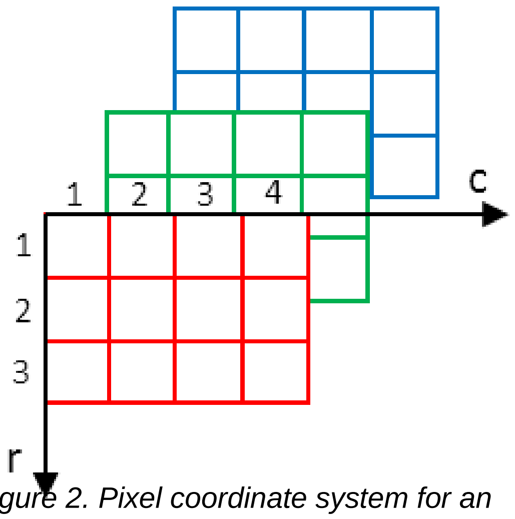


Figure 2. Pixel coordinate system for an RGB image

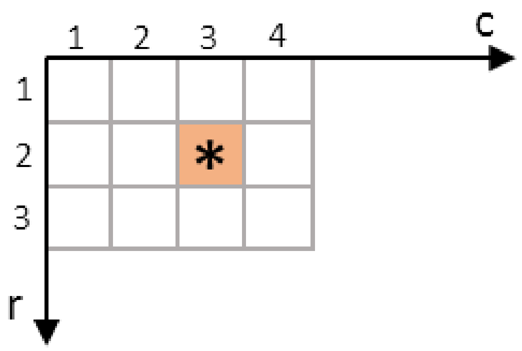


Figure 3. Pixel coordinate system for a grey scale image

```
im(2, 3)
```

Spatial coordinates

In a **spatial coordinate system**, **locations** in an image are positions on a plane and they are described as *x* and *y* coordinates (rather than by rows and columns as before). Figure 4 shows the spatial coordinate system for an image. Note that the centre of the pixel in the 2nd row and 3rd column (marked as *) has the spatial coordinates: x = 3, y = 2. For example, to plot this mark you can use the ***plot*** function which works with spatial coordinates):

```
im = imread('image')
imshow(im);
hold on
plot(3, 2, '*black')
hold off
```

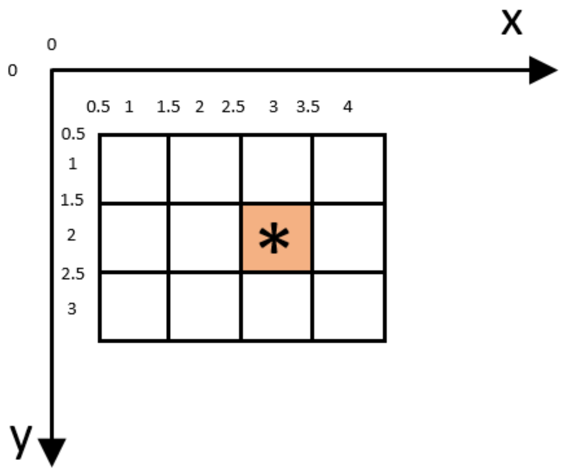


Figure 4. Spatial coordinate system for an image

Image binarisation

Often, in image analysis it is easier to work with binary images rather than with grey scale images or colour ones. In these cases usually white pixels (with label 1) correspond to the objects of interest and black pixels (with label 0) correspond to background. For image binarisation it is important to tune a threshold parameter. If the threshold is too high some objects or parts of objects can be lost, if the threshold is too low some parts of background can be labelled with 1.

In MATLAB the command ***im2bw*** performs the conversion of an image to a binary format, the operation which we call image binarisation.

Connected component analysis

Once you distinguish the objects of interest from the background (for example, by image binarisation), you may want to distinguish each object. One way to distinguish objects (if there is no occlusion between them) is to compute connected components of the binary image. The idea to add the same label to pixels that are connected (based on an image feature or other criterion). Pixels that are not connected with the current region will be assigned a different label. Two pixels are called “connected” if it is possible to build a path from one pixel to another, using only foreground pixels. By **foreground** we mean the object of interest and by **background** we denote the environment. Two successive pixels in the path must be neighbours. The neighbour areas can be different. For example, you can see on Figure 5 the 4-connected area (on the left) and the 8-connected area (on the right) where **the red pixel** is the current one, the **blue ones** are neighbours.

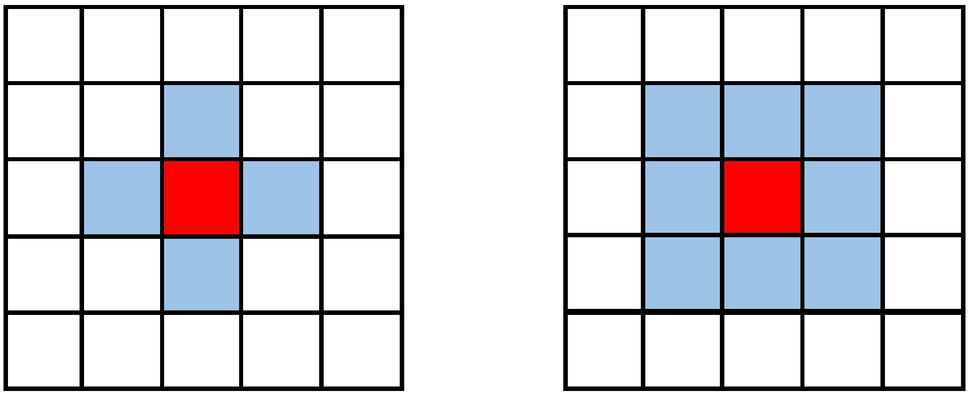
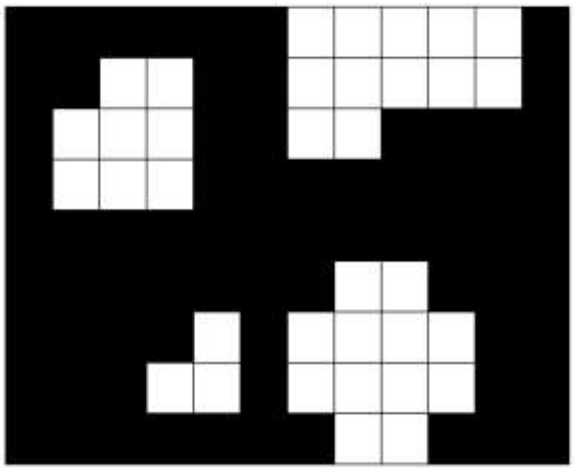
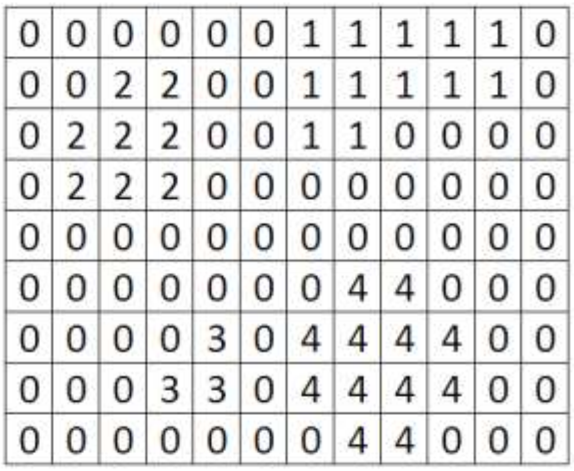


Figure 5. Examples of neighbourhoods used in connected component analysis. A 4-connected area is shown on the left hand side, and an 8-connected area on the right hand side.

A possible result of labelling of connected components is presented on Figure 6: on the left hand side is shown the input binary image, whereas the matrix on the right represents the labelled output.





0	0	0	0	0	0	1	1	1	1	1	0
0	0	2	2	0	0	1	1	1	1	1	0
0	2	2	2	0	0	1	1	0	0	0	0
0	2	2	2	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	4	4	0	0	0
0	0	0	0	3	0	4	4	4	4	0	0
0	0	0	3	3	0	4	4	4	4	0	0
0	0	0	0	0	0	0	4	4	0	0	0

Figure 6. The results of connected component analysis of a binary image. The input image is on the left hand side, the labelled output is on the right hand side.

In MATLAB, the command ***bwlabel*** performs connected component analysis of an input binary image and returns labelled output (Figure 6, right). For visualisation of the results one can use the ***label2rgb*** command.

Guidance on Performing Lab Sessions 4 and 5: Treasure Hunting

In this lab session you will need to complete the provided MATLAB script. You are already given most of the functions for performing the treasure hunting task. The parts, which are missing and which you will complete by yourself, are highlighted by the blue colour.

1. Open the script "*treasure_hunting.m*". In this script you have the needed commands, except for the highlighted with blue colour. Make sure you have the input images: *"Treasure_simple.jpg"*, *"Treasure_medium.jpg"* and *"Treasure_hard.jpg"*.
2. Convert all given images in binary format, using ***im2bw***. **You should find the appropriate threshold value**, so that the binarisation operation applies to all objects from the input images. Include the results of the binarisation of one of the images and the value of the threshold in your report.

3. Compute the connected components of your obtained binary images (with ***bwlabel***) and visualise (with ***label2rgb***) the detected connected components. You have this functionality in the provided script. You should not include this visualisation in your report.
4. Compute different characteristics of the connected components (with ***regionprops***). Visualise the bounding boxes of the objects (BoundingBox field from the **regionprops** function output, ***rectangle('Position', …)***). You have this functionality in the provided script. It is not required to include this visualisation in your report.
5. **Develop a function to distinguish arrows from other objects**. What does differ the arrows from the other objects? You may use any ideas from lectures, the previous lab session or common sense for your function. Include the *brief* description of main idea of your function in your report and the actual code of the function in the appendix of your report.
   ***Hint: arrows have points of different colour*.**
6. Find the starting red arrow. You have this functionality in the provided script.
7. **Develop a function to find the label of the next nearest object to which the current arrow indicates**. ***Hint 1****: to set a line it is enough to have two points.* ***Hint 2****: for each arrow you may extract the centroid point and the yellow one.* ***Hint 3****: a vector ($x_2 – x_1$, $y_2 – y_1$) points to the direction from the point (x_1, y_1) to the point (x_2, y_2).* You may have other ideas, you may not necessarily use the hints. Include the *brief* description and visualization of the main idea of your function in your report. Please include the code of the function in an Appendix of your report.
8. Apply your functions to find the treasure in the images. Visualise the treasure and the path to it from the starting arrow. You have this functionality in the provided script. Include your visualisation in your report for all images.
9. Other solutions of this task are possible. You will be awarded extra points if you show creativity and propose a different solution.

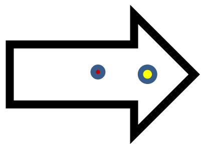
Additional Guidance on Performing the Treasure Hunting Task

1. Areas of the arrows are different from the treasures.
2. All arrows have a yellow dot inside, while treasures do not contain any yellow pixel.

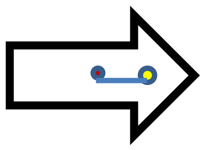
You may have other ideas, you may not necessarily use the hints. Include the *brief* description of the main idea of your function in your report. Please include the code of the function in an Appendix of your report.

In order to find the next object, you could consider these suggestions:

1. For each arrow you may extract the centroid point and the centroid of the yellow dot.

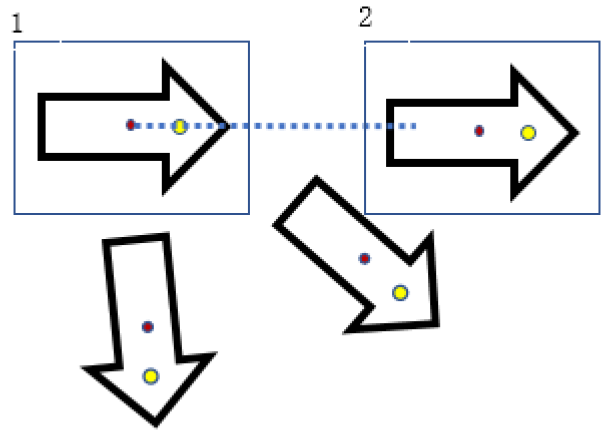


2. To set a line, it is enough to have two points.



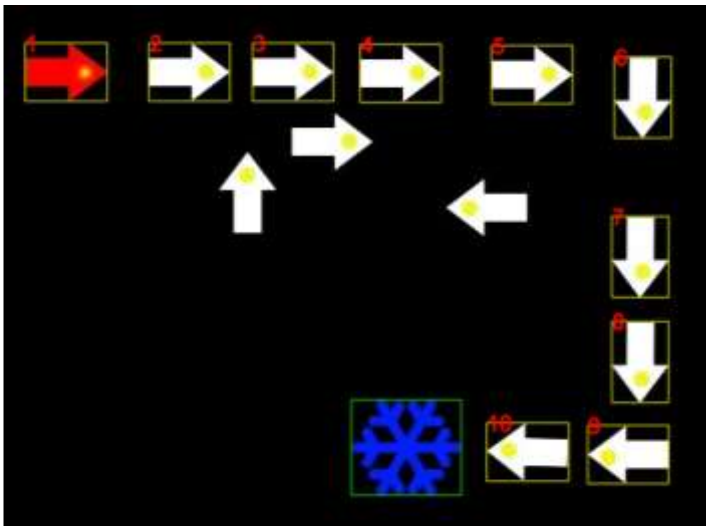
3. A vector (x2-x1,y2-y1) points the direction from the point (x1,y1) to the point (x2,y2). After extending the vector, you may find the vector enter the bounding box of another

arrow or the treasure. Therefore, that arrow is the next object.



You may have other ideas, you may not necessarily use these hints. Please include the *brief* description of the main idea of your function in your report. Please include the code of the function in an Appendix of your report.

The final result can look, for example, like that::



MATLAB script which needs to be completed

```
close all;
clear all;

%% Reading image
im = imread('Treasure_simple.jpg'); % change name to process other images
imshow(im);
pause;

%% Binarisation
bin_threshold = 0; % parameter to vary
bin_im = im2bw(im, bin_threshold);
imshow(bin_im);
pause;

%% Extracting connected components
con_com = bwlabel(bin_im);
imshow(label2rgb(con_com));
```

```
pause;

%% Computing objects properties
props = regionprops(con_com);

%% Drawing bounding boxes
n_objects = numel(props);
imshow(im);
hold on;
for object_id = 1 : n_objects
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'b');
end
hold off;
pause;

%% Arrow/non-arrow determination
% You should develop a function arrow_finder, which returns the IDs of the arror
objects.
% IDs are from the connected component analysis order. You may use any
parameters for your function.

arrow_ind = arrow_finder();

%% Finding red arrow
n_arrows = numel(arrow_ind);
start_arrow_id = 0;
% check each arrow until find the red one
for arrow_num = 1 : n_arrows
    object_id = arrow_ind(arrow_num);    % determine the arrow id

    % extract colour of the centroid point of the current arrow
    centroid_colour = im(round(props(object_id).Centroid(2)),
round(props(object_id).Centroid(1)), :);
    if centroid_colour(:, :, 1) > 240 && centroid_colour(:, :, 2) < 10 &&
centroid_colour(:, :, 3) < 10
    % the centroid point is red, memorise its id and break the loop
        start_arrow_id = object_id;
        break;
    end
end

%% Hunting
cur_object = start_arrow_id; % start from the red arrow
path = cur_object;

% while the current object is an arrow, continue to search
while ismember(cur_object, arrow_ind)
    % You should develop a function next_object_finder, which returns
    % the ID of the nearest object, which is pointed at by the current
    % arrow. You may use any other parameters for your function.

    cur_object = next_object_finder(cur_object);
    path(end + 1) = cur_object;
end

%% visualisation of the path
imshow(im);
hold on;
for path_element = 1 : numel(path) - 1
    object_id = path(path_element); % determine the object id
    rectangle('Position', props(object_id).BoundingBox, 'EdgeColor', 'y');
    str = num2str(path_element);
    text(props(object_id).BoundingBox(1), props(object_id).BoundingBox(2), str, 
'Color', 'r', 'FontWeight', 'bold', 'FontSize', 14);
end

% visualisation of the treasure
treasure_id = path(end);
rectangle('Position', props(treasure_id).BoundingBox, 'EdgeColor', 'g');
```