School of Computer Science and Communication, KTH

DD2423 Image Analysis and Computer Vision:

Lab 2:
Edge detection & Hough transform

In this lab you will implement a differential geometry based edge detector that works on multiple scales, and apply the results from this operator to detect lines with the Hough transform.

The *goal* of this lab is for you to understand how differential geometry based edge detection and the Hough transform works as well as gain practical experience in applying these operations to real data and learn the characteristics of the methods, including how important the choice of scale is for edge detection.

As *prerequisites* to this lab you should have read the course material on differential operators, edge detection and Hough transform. You should also preferably have completed and presented the results from Lab 1.

Reporting: When reporting this lab emphasis is placed on: For parts 1-4 the experimental results and interpretations/explanation of these, are the most central. For parts 5-6 you should also have created well functioning and structured functions in `Python`, that respectively perform edge detection and accumulation in the Hough domain.

You are recommended to use the command `subplot` to assemble multiple results into figures, thus simplifying the interpretation of these. In the answer sheet write down summarizing conclusions and compile results for the explicitly stated exercises and questions.

Advice: Read through *all* the lab instructions before you begin and prepare some principle solutions before you start writing the code to Exercise 6. Read through the hints and practical suggestions in section 6.1. Perform some simple experiments to analyze the behavior for different coordinate systems used to access pixels in images and curves respectively.

Python packages The labs rely on a couple of python packages that you need to have installed, before you can begin. Most important is `NumPy` that includes fundamental functions for linear algebra and representation of matrices (and images). Selected functions in `SciPy` will also be used. Finally, we will use `Matplotlib` to display images and graphs. Additional files dedicated to the course are kept in `Functions.py`. At the head of your script for this lab you can include the following lines:

```
import numpy as np
from scipy.signal import convolve2d, correlate2d
import matplotlib.pyplot as plt
from Functions import *
```

1 Difference operators

Create two difference operators `deltax()` and `deltay()` that approximate first order partial derivatives in two orthogonal directions. You may choose to use either the simple difference operator, central differences, Robert's diagonal operator or the Sobel operator. Then load the (`NumPy` based) image `few256` from the course library with

```
tools = np.load("Images-npy/few256.npy")
```

and compute discrete derivation approximations using

```
dxtools = convolve2d(tools, deltax(), 'valid')
dytools = convolve2d(tools, deltay(), 'valid')
```

Show the results on screen.

Question 1: What do you expect the results to look like and why? Compare the size of `dxtools` with the size of `tools`. Why are these sizes different?

2 Point–wise thresholding of gradient magnitudes

Based on the results above, compute an approximation of the gradient magnitude

```
gradmagntools = np.sqrt(dxtools**2 + dytools**2)
```

and show the results on screen. Compute the histogram of this image (using `numpy.histogram()`) and use this to guess a threshold that yields reasonably thin edges when applied to `gradmagntools`. Show the thresholded image with

```
showgrey((gradmagntools > threshold).astype(int))
```

for different values of *threshold*. Perform the same study on the image `godthem256`, whose content include image structures of higher complexity. The work is facilitated considerably if you write a function

```
def Lv(inpic, shape = 'same'):
    Lx = convolve2d(inpic, dxmask, shape)
    Ly = convolve2d(inpic, dymask, shape)
    return np.sqrt(Lx**2 + Ly**2)
```

where `dxmask` and `dymask` are `NumPy` arrays with suitable masks that approximate the derivatives. Preferably, combine these operations with smoothing of the image using a Gaussian convolution step prior to computing the gradient magnitude. Apply the function you developed in previous lab, alternatively use `discgaussfft()`. Note: For `Lx` and `Ly` always to be of the same size, the same needs to be true also for `dxmask` and `dymask`.

Question 2: Is it easy to find a threshold that results in thin edges? Explain why or why not!

Question 3: Does smoothing the image help to find edges?

3 Differential geometry based edge detection: Theory

One way of extracting thin edges is by considering points for which the gradient magnitude reaches local maxima in gradient direction. As it has been mentioned during the lectures, this edge definition can be expressed in differential geometry terms as the second order derivative being equal to zero and the third order derivative being negative. Introduce in each point a local coordinate system (u, v) such that the v direction is parallel to the gradient direction and the u direction is perpendicular to this. Then an edge can be defined as

$$\begin{cases} L_{vv} = 0, \\ L_{vvv} < 0, \end{cases}$$

where L_{vv} and L_{vvv} denote the second and third order derivatives of the smoothened intensity function L in the v direction. Typically, we get L from the original image function f by convolving it with a Gaussian kernel g

$$L(\cdot;\ t) = g(\cdot;\ t) * f$$

where

$$g(x, y;\ t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/(2t)}.$$

To express the edge definition above in partial derivatives of the smoothened intensity function L with respect to the Cartesian coordinate directions, we write the gradient vector at an arbitrary point (x_0, y_0) as

$$\nabla L = \left(\begin{array}{c} L_x \\ L_y \end{array} \right) \Big|_{(x_0,y_0)} = |\nabla L| \left(\begin{array}{c} \cos\varphi \\ \sin\varphi \end{array} \right) \Big|_{(x_0,y_0)}$$

In terms of

$$\cos\varphi = \frac{L_x}{\sqrt{L_x^2 + L_y^2}} \qquad \sin\varphi = \frac{L_y}{\sqrt{L_x^2 + L_y^2}}$$

the derivatives in the u and v directions can be written as

$$\partial_u = \sin\varphi\, \partial_x - \cos\varphi\, \partial_y, \quad \partial_v = \cos\varphi\, \partial_x + \sin\varphi\, \partial_y.$$

After expansion the explicit expressions of L_{vv} and L_{vvv} assume the following forms

$$L_{vv} = \frac{L_x^2 L_{xx} + 2L_x L_y L_{xy} + L_y^2 L_{yy}}{L_x^2 + L_y^2},$$

$$L_{vvv} = = \frac{L_x^3 L_{xxx} + 3L_x^2 L_y L_{xxy} + 3L_x L_y^2 L_{xyy} + L_y^3 L_{yyy}}{(L_x^2 + L_y^2)^{3/2}}.$$

Since only the sign of the derivatives are of interest for the edge definition, we can avoid dividing by the gradient magnitude $L_v = |\nabla L| = \sqrt{L_x^2 + L_y^2}$ and instead define edges the following way:

$$\begin{cases} \tilde{L}_{vv} = L_v^2 L_{vv} & = L_x^2 L_{xx} + 2L_x L_y L_{xy} + L_y^2 L_{yy} = 0, \\ \tilde{L}_{vvv} = L_v^3 L_{vvv} & = L_x^3 L_{xxx} + 3L_x^2 L_y L_{xxy} + 3L_x L_y^2 L_{xyy} + L_y^3 L_{yyy} < 0. \end{cases}$$

4 Computing differential geometry descriptors

Write two functions `Lvvtilde()` and `Lvvvtilde()` that compute the differential expressions $\tilde{L}_{vv}$ and $\tilde{L}_{vvv}$. Preferably, design this function in the same way as the function `Lv()` in exercise 2. You need to define masks corresponding to the discrete derivative approximations of all partial derivatives up to the order of three. The easiest way to do this is by using the central differences of the first order derivatives

$$\delta_x = \left(\tfrac{1}{2},\ 0,\ -\tfrac{1}{2}\right) \qquad \delta_y = \begin{pmatrix} \frac{1}{2} \\ 0 \\ -\frac{1}{2} \end{pmatrix}$$

and simple difference approximations of the second order derivatives

$$\delta_{xx} = (1,\ -2,\ 1) \qquad \delta_{yy} = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}.$$

Approximations of higher order derivatives can then be created by applying these in cascade.

The notation here is a form of operators and means that every difference operator should be interpreted as linear filtering with a mask. The concatenation of two operators thus becomes a filtering operation of their corresponding masks. In `SciPy` this concatenation of two masks is performed with the functions `convolve2d()` or `correlate2d()`, with the argument `'shape'` set to `'same'`. For example,

$$\delta_{xxx} = \delta_x\, \delta_{xx}, \qquad \delta_{xy} = \delta_x\, \delta_y, \qquad \delta_{xxy} = \delta_{xx}\, \delta_y.$$

When you create masks for these computational molecules, think of how the argument `shape` of the `SciPy` functions `convolve2d()` and `correlate2d()` treats those image points for which some of the points of the filter masks will be placed outside the available amount of image data. For the image boundaries to be treated consistently for all derivative approximations, you should let all filter masks have the same size (e.g. 5×5), even if some masks then will contain a large number of zeros.

In order to make sure that your filter masks have the expected effect, you should analyze their performances on some reference data. The easiest way to do this with `NumPy` is to define some matrices with x and y coordinates according to

```
[x, y] = np.meshgrid(range(-5, 6), range(-5, 6))
```

and study the results of for example the operations

```
print(convolve2d(x**3, dxxxmask, 'valid'))
print(convolve2d(x**3, dxxmask, 'valid'))
print(convolve2d(x**2*y, dxxymask, 'valid'))
```

When these difference approximations are applied to polynomials the following should hold

$$\delta_x(x^n) = nx^{n-1} + (\text{lower order terms}), \tag{1}$$

$$\delta_{x^n}(x^n) = n! \tag{2}$$

$$\delta_{x^{n+k}}(x^n) = 0 \tag{3}$$

$$\delta_{x^n}(y^k) = 0 \tag{4}$$

Make sure that this is the case! The most common reason for failure in the following exercises, is that this step has not been properly done. Check in particular the sign of each derivative.

Experiments: When you feel convinced that all the subcomponents work, load the image

```
house = np.load("Images-npy/godthem256.npy")
```

and compute the zero crossings of $\tilde{L}_{vv}$ on a number of different scales by writing

```
showgrey(contour(Lvvtilde(discgaussfft(house, scale), 'same')))
```

where you appropriately try *scale* = 0.0001, 1.0, 4.0, 16.0 and 64.0.

Question 4: What can you observe? Provide explanation based on the generated images.

Study the sign of the third order derivative in the gradient direction by loading the image

```
tools = np.load("Images-npy/few256.npy")
```

and show the result of

```
showgrey((Lvvvtilde(discgaussfft(tools, scale), 'same')<0).astype(int))
```

for the same values of *scale*. What is the effect of the sign condition in this differential expression?

Question 5: Assemble the results of the experiment above into an illustrative collage with the `subplot` command. Which are your observations and conclusions?

Question 6: How can you use the response from $\tilde{L}_{vv}$ to detect edges, and how can you improve the result by using $\tilde{L}_{vvv}$?

5 Extraction of edge segments

So far we have only considered the result of computing differential operators at different scales and looked at their zero crossings and sign variations. In this exercise we will collect these components and write a function

```
edgecurves = extractedge(inpic, scale, threshold, shape)
```

that computes the edges of an image at arbitrary scale and returns these lists of edge points. If the parameter `threshold` is given, this value shall be used as threshold for the gradient magnitude computed at the same scale. Otherwise, no thresholding shall be used and all edge segments will be returned.

Two functions are provided in the course library. The first one extracts level curves in a given image and rejects points based on the sign of the second input argument. The second function thresholds these curves with respect to the sign of another image.

```
curves = zerocrosscurves(zeropic, maskpic1)
curves = thresholdcurves(curves, maskpic2)
```

These two functions represent curves as tuples of two `NumPy` arrays with Y- and X-coordinates respectively. For more information take a look at the functions `zerocrosscurves()` and `hresholdcurves()` in `functions.py`. The mask images are assumed to be boolean `NumPy` arrays with `True` for those pixels fulfilling some condition.

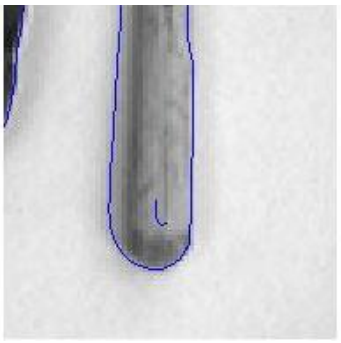
The exercise thus consists of, based on the theory described in section 3 and the analogy with the experiments you performed in section 4, computing suitable differential quantities that can be used as

input data for the functions `zerocrosscurves()` and `thresholdcurves()`, so as to determine edges thresholded with respect to the gradient magnitude at arbitrary scale. When your function works satisfactory, use it to extract edges from the images `house` and `tools`. First try different scales e.g. 0.0001, 1.0, 4.0, 16.0 and 64.0. For each image find the best scale and adjust the threshold accordingly. Preferably, show the results on screen with

```
overlaycurves(image, edgecurves)
```

that overlays the curves on top of the original image. Make sure that the results are satisfactory. Save a suitable subset of the resulting images. Use this as a basis for your lab report and later exercises.

Note! As a reference, your output images should look something like:



Question 7: Present your best results obtained with `extractedge()` for `house` and `tools`.

6 Hough transform

In this exercise you will write a function `houghline()` that uses the Hough transform to determine linear approximations of a given number of curves, based on parametrizations of the type

$$x \cos\theta + y \sin\theta = \rho.$$

where x and y are suitably defined (e.g. centered) image coordinates, and ρ and θ have appropriately selected definition areas with respect to the choices above. Your function should be declared as

```
[ linepar, acc ] = houghline(curves, magnitude, nrho, ntheta, threshold, \
                             nlines, verbose)
```

where

- `linepar` is a list of (ρ, θ) parameters for each line segment,
- `acc` is the accumulator matrix of the Hough transform,
- `curves` are the edge points from which the transform is to be computed,
- `magnitude` is an image with one intensity value per pixel (in exercise 6.2 you will here give the gradient magnitude as an argument),
- `nrho` is the number of accumulators in the ρ direction,
- `ntheta` is the number of accumulators in the θ direction,

- `threshold` is the lowest value allowed for the given magnitude,
- `nlines` is the number of lines to be extracted,
- `verbose` denotes the degree of extra information and figures that will be shown.

Based on this function and `extractedge()` you will then write a function `houghedgeline()` that first performs an edge detection step and then applies a Hough transform to the result. As a suggestion this function might have a declaration similar to

```
[linepar, acc] = houghedgeline(pic, scale, gradmagnthreshold, nrho, ntheta, \
                               nlines, verbose)
```

where the (additional) arguments have the following meaning

- `pic` is the grey-level image,
- `scale` is the scale at which edges are detected,
- `gradmagnthreshold` is the threshold of the gradient magnitude.

To visualize the processing steps, you ought to show the important partial results on the screen. Preferably, control the amount of partial results that is produced by varying the argument `verbose` of the function, such that the value zero results in a "silent" function and the amount of information displayed during execution increases with an increasing value of `verbose`.

To visualize the final result you should draw the lines overlaid on the original image. The easiest way to do this is to first used `showgrey()` and then the `plot()` function in `Matplotlib`. If you have enough time, add the possibility to draw only those line segments that correspond to points that are close to the edge elements.

6.1 Hints and practical advice

- Think through the parametrization used for the edge segments (the choice of origin and definition areas for ρ and θ). Especially, make sure that you make use of the quantized accumulator space reasonably efficiently, and that discontinuities in the parametrization do not lead to any destructive effects.
- If you strike any programming problems, remember to divide the problem into smaller subproblems and test these partial modules separately. For example, you can be greatly helped by writing test routines that draw a number of lines given values of ρ and θ.
- The interesting quantization of ρ corresponds to accumulator cells with $\Delta\rho$ in the neighborhood of one pixel, while the interesting quantization of θ typically corresponds to accumulator cells with $\Delta\theta$ of about a degree.
- Large values of $\Delta\theta$ considerably speed up the Hough transform. This fact is of value when testing and debugging the function. Of the same reason, test and debug the function using *small* images.
- You most easily detect local maxima in the accumulator by calling the function `locmax8()` in the course library. The following code segment may be used to detect the local maxima and create an index vector from these:

```
pos, value, _ = locmax8(acc)
indexvector = np.argsort(value)[-nlines:]
pos = pos[indexvector]
```

Thereafter you can extract the index values in the accumulator that correspond to the `nlines` responses with a call like

```
for idx in range(nlines):
  thetaidxacc = pos[idx, 0]}
  rhoidxacc = pos[idx, 1]}
```

- Quantization in accumulator space: For large values of $\Delta\rho$ and $\Delta\theta$ the lines will be of low accuracy. However, small values may result in multiple responses for the same line. If you unable to achieve enough accuracy without getting too many multiple responses, it might be advantageous to smoothen the histogram with a call to `discgaussfft()` before detecting local maxima.

- The output format of the results: The easiest way to visualize the results is by generating three points of each extracted line. Note that `Matplotlib` automatically cuts off those parts of a line that are located outside the image when the function `plot()` is used, which means that you do not explicitly have to compute where the line begins and ends. You can generate lines that illustrate the results as follows:

  ```
  x0 = < fill in >
  y0 = < fill in >
  dx = < fill in >
  dy = < fill in >
  plt.plot([x0-dx, x0, x0+dx], [y0-dy, y0, y0+dy], 'r-')
  ```

  In other words you have to convert each line defined by (ρ, θ) back to Cartesian coordinates (x, y). For convenience we decompose each line into 2 segments originated in (x_0, y_0), defined by 3 points in a curve which forms a single line on the screen. Think of how you specified your coordinate system in the Hough space! When you draw the lines, you might use `plt.xlim()` and `plt.ylim()` to display only what is inside the image.

- Test run the algorithm on *small* images. Exploit the interactivity of `Matplotlib`, as well as the graphics facilities, if you need to debug your code.

  Before you apply the algorithm to real images, make sure that it gives the correct results on simpler test images, such as

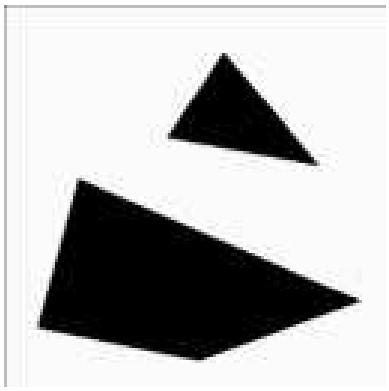
  ```
  testimage1 = np.load("Images-npy/triangle128.npy")
  smalltest1 = binsubsample(testimage1)

  testimage2 = np.load("Images-npy/houghtest256.npy")
  smalltest2 = binsubsample(binsubsample(testimage2))
  ```

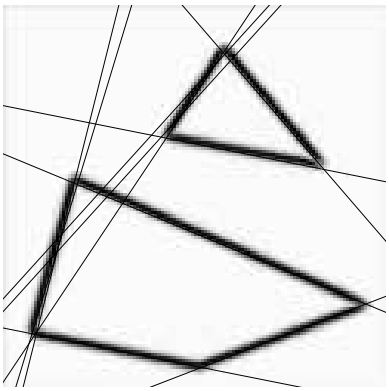
Apply your function `houghedgeline()` to the simple test image `triangle128`.

Question 8: Identify the correspondences between the strongest peaks in the accumulator and line segments in the output image. Doing so convince yourself that the implementation is correct. Summarize the results in one or more figures.

Then apply the same function to the test image `houghtest256`. If the implementation is correct, the results should be similar to those below: Show the results of your function `houghedgeline()`



Original image



Hough transform (10 lines)

applied to the images `few256`, `phonecalc256` and `godthem256`.

Question 9: How do the results and computational time depend on the number of cells in the accumulator?

6.2 Choice of accumulator incrementation function

A natural choice of accumulator increment is letting the increment always be equal to one. Alternatively, let the accumulator increment be proportional to a function of the gradient magnitude

$$\Delta S = h(|\nabla L|)$$

where $|\nabla L|$ is the gradient magnitude and h is some monotonically increasing function. Furthermore, you may in some cases use the local gradient direction.

Question 10: How do you propose to do this? Try out a function that you would suggest and see if it improves the results. Does it?

6.3 A howto

When you develop function `houghline()`, the follwing steps may help:

```
def houghline(curves, magnitude, nrho, ntheta, threshold, nlines, verbose):

# Allocate accumulator space

# Define a coordinate system in the accumulator space

# Loop over all the edge points

    # Check if valid point with respect to threshold

    # Optionally, keep value from magnitude image

    # Loop over a set of theta values

        # Compute rho for each theta value

        # Compute index values in the accumulator space

        # Update the accumulator

# Extract local maxima from the accumulator

# Delimit the number of responses if necessary

# Compute a line for each one of the strongest responses in the accumulator

# Overlay these curves on the gradient magnitude image

# Return the output data [linepar, acc]
```

Laboration 2 in DD2423 Image Analysis and Computer Vision

.....

Student's personal number and name (filled in by student)

.....

Approved on (date) Course assistant