School of Computer Science and Communication, KTH

DD2423 Image Processing and Computer Vision:

Lab 3:
Image segmentation

In this lab you will test a number of popular methods for image segmentation, methods that deal with spatial coherence in different ways. Some methods do not take the spatial locations of pixels into consideration at all, whereas others do it explicitly using some neighbourhood system.

The *goal* of this lab is for you to understand the possibilities and limitations of image segmentation methods and gain practical experience from using such methods, so that you know how to approach new problems for which segmentation may be necessary.

As *prerequisites* to this lab you should have read the course material on image segmentation.

Reporting: When reporting this lab emphasis is placed on understanding how the algorithms work (in general), the strengths and weaknesses of the tested segmentation methods and to what extent and how they exploit spatial similarities for segmentation.

It is 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: Think more about what you are expected to learn, than what you are expected to do. Note that in many cases there are many possible correct answers to the same question. How would you characterize the methods you test? And when would you use them? **Before you present your results, make sure that you understand how each of the methods work.**

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. We will use `Matplotlib` to display images and graphs. Finally, `Pillow` will be used for loading, storing and filtering RGB images. Additional files dedicated to the course are kept in `Functions.py`.

1 Working with images in Pillow

With `Pillow`, colour images are internally stored in `Image` objects as three-dimensional arrays, with the last dimension being used for the colour space. You can access individual pixels, but it is fairly slow, not as fast as `NumPy`. Convenient and frequently used image functions are `open()` for reading images from disk, `save()` for storing images and `resize()` for changing the size of the image. Please read the help text for the individual functions to understand how they are used in practice. The commands

```
from PIL import Image
img = Image.open('filename.jpg')
```

can for example be used to read a JPEG file from disk. Typically, images are stored with each value represented by a byte. To keep the precision when you apply sequential filtering operations, the first thing to do is often to convert the image data to a `NumPy` array and then change the format to floating points using either the types `np.float64` or `np.float32`.

```
import numpy as np
I = np.asarray(img).astype(np.float32)
```

One should keep in mind though, that floating points will decrease the speed of computation. Thus if the format does not matter for a particular function, it might be preferable to keep the representation in bytes (`np.ubyte`). In some cases it can be practical to reshape the image representation from a 3D array of size $(H, W, 3)$ to one with the size $(WH, 3)$. This you can do with the command

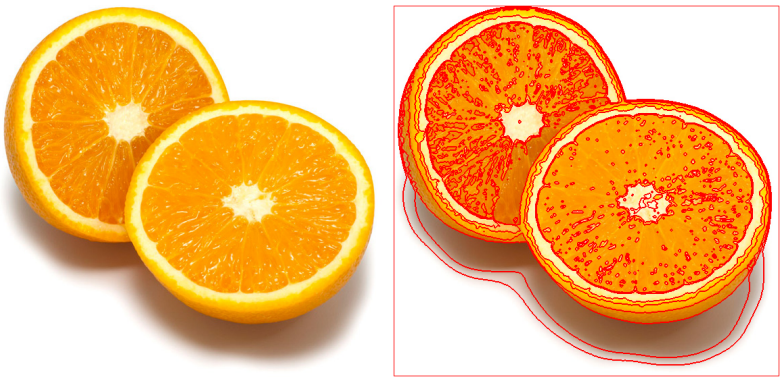
```
Ivec = np.reshape(I, (-1, 3))
```

and then use the same command to get the image back the original original $(H, W, 3)$ format, especially if you like to visualize the end result.

2 K-means clustering

Segmentation is often done in stages that involve different algorithms. In many cases it can be too time-consuming to work on individual pixels for a final segmentation. The solution is to create an *over-segmentation* and divide the image into many so called *superpixels*, i.e. groups of connected pixels that are so similar that they must originate from the same physical object in the scene. Each superpixel might contain from a handful of similar coloured pixels to tens of thousand. It all depends on the particular image.

For example, in the figure below the two halves of the orange have been divided into a larger number of superpixels. Even if many of the superpixels are small, there are many fewer pixels then there are original pixels, and no single superpixels covers parts of the two halves.



Figur 1: Two halves of an orange divided into superpixels using K-means clustering.

In a first exercise you will use the K-means clustering algorithm to cluster all pixel colours to a smaller set of colours, each cluster represented by its center colour, which is the mean of all pixels assigned to it. Often this procedure is used to change colour representation from 24 bits (8 bits per colour channel) to a palette of colours, where the colour of each pixel is given by an index that requires a fewer number of bits. However, it can also be used to create superpixels, which is its purpose here. The K-means clustering algorithms works like the algorithm below.

```
Let X be a set of pixels and V be a set of K cluster centers in 3D (R,G,B).
%  Randomly initialize the K cluster centers
%  Compute all distances between pixels and cluster centers
%  Iterate L times
%     Assign each pixel to the cluster center for which the distance is minimum
%     Recompute each cluster center by taking the mean of all pixels assigned to it
%     Recompute all distances between pixels and cluster centers
```

Create a function in Python that implements the K-means clustering algorithm

```
segmentation, centers = kmeans_segm(image, K, L, seed)
```

that given an `image`, the number of cluster centers `K`, number of iterations `L` and a `seed` for initializing randomization, computes a `segmentation` (with a colour index per pixel) and the `centers` of all clusters in 3D colour space. To compute pair-wise differences there is a convenient `SciPy` function, `spatial.distance_matrix()`, that can be used. The initialization of colours can be done in many different ways. Use an approach that you believe could solve the problem, with or without a seed used for randomization.

To try out your implementation you could use the file `kmeans_example.py` from the lab directory and modify it to suit your experiments. Note that this file uses two `Pillow` functions, `GaussianBlur()` and `filter()`, to first set up a Gaussian kernel for filtering and then apply this kernel to the image to blur out image noise. It also scales down the image using a `scale_factor`, to speed up the clustering process, if your CPU is too slow to handle the full-sized image.

Question 1: How did you initialize the clustering process and why do you believe this was a good method of doing it?

Try modifying the `K` and `L` parameters, the `scale_factor` and the amount of pre-blurring, to see how the changes affect the results. For your experiments there are a number of images in the lab directory that can be used for the purpose (`orange.jpg, tiger1.jpg, tiger2.jpg` and `tiger3.jpg`), but you are welcome to test it on your own images found on Internet or elsewhere.

Question 2: How many iterations `L` do you typically need to reach convergence, that is the point where no additional iterations will affect the end results?

Question 3: What is the minimum value for `K` that you can use and still get no superpixel that covers parts from both halves of the orange? Illustrate with a figure.

Try using parameters suitable for `orange.jpg` and see how these affect the tiger images.

Question 4: What needs to be changed in the parameters to get suitable superpixels for the tiger images as well?

3 Mean-shift segmentation

Typically, K-means clustering does not consider image positions for clustering. Spatially similar pixels can still be grouped together in an post-processing step, in which connected components are found, each component having a particular cluster colour. The segments could, however, stretch over large parts of the image, if colours are similar, leading to an unnecessary fragmentation of the result.

An alternative to K-means clustering, that usually does clustering both spatially and in colours, is mean-shift segmentation. With mean-shift segmentation a pixel can be represented by a point in a

five-dimensional space

$$\hat{x} = [x, y, r, g, b]^T,$$

where (x, y) is its image position and (r, g, b) the colour. The goal of mean-shift is to find an unknown number of modes (cluster centers) and then determine which pixel corresponds to which mode. This is done by maximizing a continuous density function

$$f(\hat{x}) = \frac{1}{n} \sum_{i=1}^{n} K(\hat{x} - \hat{x}_i),$$

where $K(\hat{x} - \hat{x}_i)$ is a kernel density function around each image point $\hat{x}_i$. The kernel here is given by a Gaussian function

$$K(\hat{x}) = \exp^{-\hat{x}^T D^{-1} \hat{x}/2},$$

where the variance is given by a diagonal matrix D with separate variances (bandwidths) for image positions and for colours,

$$D = \mathrm{d}iag(\sigma_s^2, \sigma_s^2, \sigma_c^2, \sigma_c^2, \sigma_c^2).$$

Starting from different pixels in the image, modes are found iteratively through gradient ascent, using an update function given by

$$\hat{x} \leftarrow \frac{\sum_{i=1}^{n} \hat{x}_i \; K(\hat{x} - \hat{x}_i)}{\sum_{i=1}^{n} K(\hat{x} - \hat{x}_i)}.$$

What is essentially computed is a weighted average of points close to the current point $\hat{x}$, a point that is then moved to the computed average. Depending on the starting pixel, the iterations will converge to different modes, given by the *basin of attraction* or each mode. Images are then segmented based on which mode the iterations will fall into, given each pixel in the images used as a starting point, a process that can be very time-consuming.

In this lab you will test an existing implementation of mean-shift (`mean_shift_segm()`). Note that to improve the speed of the mean-shift process, colours are internally represented as a linear combination of 32 colours, colours which are found using K-means clustering with your implementation of `kmeans_segm()`. So make sure that this function is correctly implemented. For your convenience there is an example file called `mean_shift_example.py` that can be modified to suit your analysis. It is recommended to change this file and for example use subplots and `imshow()` to get multiple outputs with different settings in the same window for easy comparison.

Run `mean_shift_segm()` with different bandwidths for image positions σ_s (`spatial_bandwidth`) and colours σ_c (`colour_bandwidth`), so that you get as large segments as possible, but segments that each do not cover more than one object in the scene.

Question 5: How do the results change depending on the bandwidths? What settings did you prefer for the different images? Illustrate with an example image with the parameter that you think are suitable for that image.

Question 6: What kind of similarities and differences do you see between K-means and mean-shift segmentation?

Figur 2: An example image of a tiger divided into multiple segments using mean-shift.

4 Normalized Cut

With mean-shift segmentation you cannot control in how many segments an image will be divided. It depends on how many modes (cluster centers) the method finds, which in turn depends on the bandwidths used and the particular image. In figure ground segmentation, however, you often like to limit the number of segments, preferably having one foreground segment and one background segment. A method that allows this is Normalized Cut.

Normalized Cut uses a neighbourhood system, here defined by a parameter `radius`, to connect pixels into a large graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with pixels on the vertices $\mathcal{V}$ and edges $\mathcal{E}$ between neighbouring pixels. Each edge is given a weight corresponding to the similarity between the two pixels that is connects. The method then tries to divide $\mathcal{V}$ into two subsets of vertices (pixels), A and B, such that

$$\text{Ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{assoc}(A, \mathcal{V})} + \frac{\text{cut}(A, B)}{\text{assoc}(B, \mathcal{V})}$$

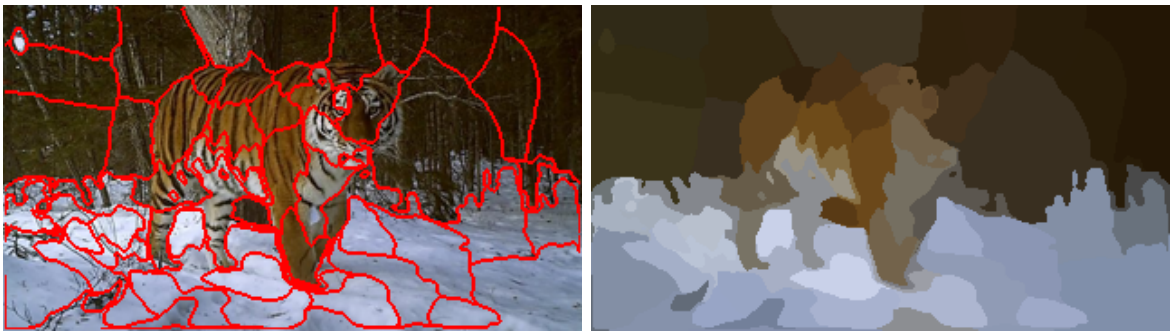
is minimized. Here $\text{cut}(A, B)$ is the sum of all edges that connect two vertices in A and B respectively, and $\text{assoc}(A, \mathcal{V})$ is the sum of all edges connected to any vertex in A. That is, Normalized Cut tries to minimize the similarities between pixels on the cut, while maximizing the similarities within each respective part of the cut.

You will perform experiments using an implementation of Normalized Cut called `norm_cuts_segm()`, where the file `norm_cuts_example.py` serves as an example of how to use it in practice. Instead of just splitting up an image into two pieces, A and B, this implementation can split up the two pieces into even smaller parts by applying a series of recursive calls. There are three parameters that control the recursive subdivision: `ncuts_thresh` that controls the maximum allowed value for $\text{Ncut}(A, B)$ for a cut to take place, `min_area` that controls the minimum size of a segment and `max_depth` that limits the depth of recursion.

The weights corresponding to the similarities between pixels are computed with a function similar to $K(\hat{x})$ that was used for mean-shift segmentation (see previous section), where you can control the colour bandwidth σ_c through the parameter `colour_bandwidth`. Test the effect of the different parameters on the images in the lab directory.

Question 7: Does the ideal parameter setting vary depending on the images? If you look at the images, can you see a reason why the ideal settings might differ? Illustrate with an example image using the parameters you prefer for that image.

Question 8: Which parameter(s) was most effective for reducing the subdivision and still result in a satisfactory segmentation?



Figur 3: An example image of a tiger divided into multiple segments using Normalized Cut.

Question 9: Why does Normalized Cut prefer cuts of approximately equal size? Does this happen in practice?

Try to increase the `radius` to include neighbouring pixels that are a bit further away from each other. This usually leads to a better segmentation, but at the cost of slower computations.

Question 10: Did you manage to increase `radius` and how did it affect the results?

5 Segmentation using graph cuts

Graph theoretic methods are frequently used for figure ground segmentation. A segmentation is typically found by setting up an energy based formulation, where the energy depends on how pixels are divided into foreground and background pixels. Each such combination results in a particular energy and the segmentation is sought that minimized the energy. This gives us a tool for defining what can be expected from a good segmentation.

Assume that you have a set of pixels $P \ni i$ and neighbours $N \ni (i, j)$. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is created with vertices $\mathcal{V}$ given by one vertex v_i per pixel, as well as one vertex v_s known as *source* and another vertex v_t as *drain*. All pixel vertices v_i are connected to its respective neighbours v_j with edges $e_{i,j}$. They are also connected to the source and drain with edges $e_{i,s}$ and $e_{i,t}$. In summary, vertices are given by

$$\mathcal{V} = (\{v_i\}, v_s, v_t), \quad i \in P$$

and edges by

$$\mathcal{E} = (\{e_{i,j}\}, \{e_{i,s}\}, \{e_{i,t}\}), \ i, j \in P, \ (i, j) \in N.$$

Also assume that you have set of unknown binary variables $X = \{x_i\}$, where the variable for each pixel denotes whether the pixel belongs to the foreground ($x_i = 1$) or the background ($x_i = 0$). Now we can set up an energy formulation, where the edges correspond to pixel and pair-wise energies,

$$E(X) = \sum_{i \in P} x_i e_{i,s} + \sum_{i \in P} (1 - x_i) e_{i,t} + \sum_{(i,j) \in N} [x_i \neq x_j] e_{i,j}.$$

If we minimize $E(X)$, using some suitable choice of edge energies, we get an optimal segmentation X^* of the image. Here $[x_i \neq x_j] = 1$, if $x_i \neq x_j$ is true, that is if pixels i and j belong to different segments (foreground or background), and $[x_i \neq x_j] = 0$ otherwise. Thus if two neighbours belong to the same segment, the corresponding edge energy $e_{i,j}$ does not contribute to $E(X)$. Then either $e_{i,s}$ or $e_{i,t}$ contribute, depending on whether i is a foreground or background pixel, as indicated by x_i.

Fortunately, the problem of minimizing $E(X)$ can be seen as a minimum cut problem, a problem for which there are many known algorithms. If you try to make a cut through the graph, such that

the source and drain are on different sides of the cut, you either cut the edge between a particular pixel and the source, or the opposing edge to the drain. You also cut between neighbouring pixels that belong to different segments. Each edge that is broken results in an energy that contributes to $E(X)$.

In this lab we will use a method suitable for high-speed parallel implementations known as Continuous Min-Cut. Instead of letting x_i only have values 0 or 1, the problem is relaxed so that x_i can have a value in the whole interval $(0, 1)$. In the end we can apply a threshold t (typically 0.5), such that $x_i > t$ means that pixel i belongs to the foreground, and background if $x_i < t$.

5.1 Setting up the energies

Energies can be seen as costs for deciding on a particular segmentation and rejecting other possibilities. If a certain pixel has a colour that resembles the background, but it is still segmented as foreground, then the cost (energy) should be high. The cost should also be high, if two neighbouring pixels have similar colours, but one is segmented as foreground and the other one as background.

Assume that we have two statistical models, one for the foreground and one for the background, that are described by the distributions $p(c|f)$ and $p(c|b)$ respectively. If backgrounds and foregrounds can be expected to be equally common, $p(f) = p(b)$, then the prior probability that a pixel with colour c_i is a foreground pixel can be written using Bayes' rule as

$$p(f|c_i) = \frac{p(c_i|f)p(f)}{p(c_i|f)p(f) + p(c_i|b)p(b)} = \frac{p(c_i|f)}{p(c_i|f) + p(c_i|b)} \tag{1}$$

and for the background $p(b|c_i) = 1 - p(f|c_i)$. The pixel-wise energies can then be defined as

$$e_{i,s} = -\log(p(b|c_i)) \quad \text{and} \quad e_{i,t} = -\log(p(f|c_i)).$$

This implies that if a pixel i matches the foreground model much better than the background model, the link cost to the source ($e_{i,s}$) will be high, whereas the link cost to the drain ($e_{i,t}$) will be close to zero. Thus the graph cut will most likely lead to the pixel being left connected to the source, and segmented as a foreground pixel.

However, the final segmentation will also depends on the similarities between neighbouring pixels. If two pixels are similar in colour the cost of separating the two pixels should be high. In this lab we will thus define the pair-wise edge costs as

$$e_{i,j} = \frac{\alpha\ \sigma}{|c_i - c_j| + \sigma}. \tag{2}$$

If the two pixels i and j have the same colour, the cost $e_{i,j}$ will be α, but then $e_{i,j}$ decays for decreasing similarity between the pixels. The speed of this decay is controlled by the parameter σ. The lower the value $e_{i,j}$ is, the cheaper the cost of separating the pixels will be.

This concludes the definitions of link costs. Using these link costs, a minimum cut of the graph can be found, a cut that in turn defines a segmentation. A problem that remains though is how to create the statistical models of foregrounds and backgrounds.

Mixture of Gaussians One way to create a statistical model of colours is to use a mixture of Gaussians, i.e. a weighted sum of multiple Gaussian distributions. Assume that we have N pixels in total and K Gaussian components. The probability of the colour of pixel i can then be written as

$$p(c_i) = \sum_k^K w_k g_k(c_i), \tag{3}$$

where w_k is a weight for the kth Gaussian component, a component defined by a distribution

$$g_k(c_i) = \frac{1}{\sqrt{(2\pi)^3|\Sigma_k|}} \exp^{-\frac{1}{2}(c_i-\mu_k)^T\Sigma_k^{-1}(c_i-\mu_k)},$$

with μ_k being the mean colour and Σ_k the 3×3 covariance matrix.

For example, if a foreground region contains a mixture of skin-coloured pixels and blue pixels that come from a blue T-shirt, you may have different components that model the different types of colours, instead of having one Gaussian that is supposed to model them all, which is difficult since the colours can be so different. The parameter w_k would then define how many pixels you have of each colour type and $g_k(c)$ defines the distribution of colours of each type.

Assuming that you have a set of pixels with colours $\{c_i\}$, you can find suitable weights w_k, mean colours μ_k and covariance matrices Σ_k through a processes called Expectation-Maximization. This is an iterative process that first computes how well each pixel can be modelled by the Gaussian components, and then updates the components with new parameters. The process is thus similar to K-means, but with K-means you do not have a covariance matrix for each colour cluster and a pixel as assigned to only one cluster each. With a mixture of Gaussians the assignment is soft, which means that it can be seen as a combination of colours. A pixel can thus be 80% skin-coloured and 20% blue for example.

Step 1: Expectation Given the current estimates of the parameters w_k, μ_k and Σ_k compute how much of pixel i comes from kth Gaussian component,

$$p_{i,k} = \frac{w_k g_k(c_i)}{\sum_k^K w_k g_k(c_i)}.$$

These values can be stored in a large matrix with one value for each of the N pixels and K components.

Step 2: Maximization Update each of the parameters in sequence w_k, μ_k and Σ_k. First the weights that tell you how much you have of each kind of colour

$$w_k \leftarrow \frac{1}{N} \sum_i p_{i,k},$$

then the mean colour of each components, which is a weighted average of all pixel colours,

$$\mu_k \leftarrow \frac{\sum_i p_{i,k} c_i}{\sum_i p_{i,k}}$$

and finally the covariance matrices, which is also a weighted average,

$$\Sigma_k \leftarrow \frac{\sum_i p_{i,k}(c_i - \mu_k)(c_i - \mu_k)^T}{\sum_i p_{i,k}}.$$

The two steps are then iterated until convergence, much similar to K-means clustering.

5.2 Programming exercise

There is a function that performs energy-based segmentation using graph cuts called `graphcut_segm()` and an example file (`graphcut_example.py`) that shows how to use it. Another function `mixture_prob()` that is needed by `graphcut_segm()` is missing though and needs to be implemented by you. This function uses a `mask` to identify pixels from an `image` that are used to estimate a mixture of `K` Gaussian components. The mask has the same size as the image. A pixel that is to be included has a mask value 1, otherwise 0. A call to the function can be written as

```
prob = mixture_prob(image, K, L, mask)
```

where `L` is the number of iterations that Expectation-Maximization is supposed to run. The output of the function is an image of probabilities (`prob`) that corresponds to $p(c_i)$ in Eq. (3) above. The whole function can me summarized as follows:

```
Let I be a set of pixels and V be a set of K Gaussian components in 3D (R,G,B).
%  Store all pixels for which mask=1 in a Nx3 matrix
%  Randomly initialize the K components using masked pixels
%  Iterate L times
%     Expectation: Compute probabilities P_ik using masked pixels
%     Maximization: Update weights, means and covariances using masked pixels
%  Compute probabilities p(c_i) in Eq.(3) for all pixels I.
```

Making a really fast implementation is not easy. One should try never to loop over all pixels, but exploit efficient matrix and vector operations in `NumPy`. Note that the function returns an image of probabilities (`prob`), but not the weights, means and covariances of the components. You may thus represent these the way you prefer. Note that to initialize μ_k, you may use K-means that you have already implemented and set Σ_k to an equal value for all components, with w_k set to the fraction of pixels assigned to each K-means cluster. Here are some additional recommendations.

NumPy Tip 1: It is preferable to store all colours as a large $(WH, 3)$ matrix with the three colour channels stored in different columns, instead of a 3D array of size $(H, W, 3)$. For example,

```
Ivec = np.reshape(I, (-1, 3)).astype(np.float32)
```

would create such a $(WH, 3)$ array from an image `I` and convert all values to 32-bit floating points.

NumPy Tip 2: In `NumPy` you can apply operations on arrays of different sizes, using something called broadcasting (link), as long as `NumPy` can figure out how to repeat operations over different dimensions to make the sizes fit. This is true if, for each dimension, the sizes are either equal for both arrays or the sizes in that dimension is equal to one for one of the arrays. For example,

```
diff = Ivec - mean
```

can be used to subtract a mean colour (given by a 1×3 matrix) from pixels stored in a $N \times 3$ matrix. However, note that if the mean is a vector of size 3, you first need reshape it into an `NumPy` array of size $(1, 3)$. You might also need to do a transpose, `mean.T`, to get the ordering of dimensions right.

5.3 Experiments

Imagine an image editing software that includes the facility to segment out image objects. Somehow the user must tell the software which object in the image he or she wants to cut out, since there could be many different objects to choose from. Selecting an object by simply drawing a rectangle around it and use this to create a mask, is a quick and easy way to initialize segmentation.

The function `graphcut_segm()` is implemented with such a selection procedure in mind. Thus instead of using a mask as an input, it uses a rectangular `area` defined in the following format:

```
area = [ minx, miny, maxx, maxy ]
```

First it uses the area within the rectangular area to compute a foreground model using your implementation of `mixture_prob()` and then it inverts the mask to find a similar background model for the pixels outside the area. The resulting probabilities are then used to set up the energies and compute a segmentation by finding a minimum cut of the graph with the energies on its links.

When testing the software you may change the area in `graphcut_example.py` to see how different areas affect the end result. Note that if too many background pixels are included in the rectangular area, parts of the background will eventually be segmented as foreground, because the colour models will be polluted by pixels of the wrong kind.

Figur 4: An example of prior foreground probabilities and final segmentation using Graph Cut

If `mixture_prob()` is correctly implemented it should be possible to obtain the results in Figure 4. The left image shows the prior foreground probabilities, $p(f|c_i)$ in Eq. (1), that are returned from `graphcut_segm()`. By thresholding these one may get a prior segmentation, i.e. a segmentation without the pair-wise costs between pixels taken into consideration. In contrast, the image to the right shows the posterior segmentation, i.e. when pair-wise costs have been included.

In `graphcut_example.py` you may also change the number of Gaussian components (`K`), the maximum cost of an edge α (`alpha`) and how much the edge cost decays for decreasing similarity between neighbouring pixels σ (`sigma`), as defined in Eq. (2).

Question 11: Does the ideal choice of `alpha` and `sigma` vary a lot between different images? Illustrate with an example image with the parameters you prefer.

Question 12: How much can you lower `K` until the results get considerably worse?

Question 13: Unlike the earlier method Graph Cut segmentation relies on some input from a user for defining a rectangle. Is the benefit you get of this worth the effort? Motivate!

Question 14: What are the key differences and similarities between the segmentation methods (K-means, Mean-shift, Normalized Cut and energy-based segmentation with Graph Cuts) in this lab? Think carefully!!

Laboration 3 in DD2423 Image Analysis and Computer Vision

. . . . . .

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

. . . . . .

Approved on (date) Course assistant