Computer Vision
WS 2015/2016

Prof. Bastian Leibe<leibe@vision.rwth-aachen.de>
Stefan Breuers<breuers@vision.rwth-aachen.de>

Exercise 3: Structure Extraction, Mean-Shift Segmentatnon, Segmentation with Graph Cuts

due before 2015-12-07

Important information regarding the exercises:

- In the archive for this exercise you will find the functions `apply.m` that should be used for displaying your results. You should also use it to test your implementation and see if the results make sense. Answers are to be submitted within `answers.m` Do **not** modify the apply files in any way.
- Please do **not** include the data files in your submission!
- Please submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Please submit your solutions via the L^{2}P system.

Please note:

- The exercise is not mandatory.
- There will be no corrections. If you want to verify your solutions, use the provided `apply` functions.
- Nevertheless, we encourage you to work on the exercises and present your solutions in the exercise class. For this regard the above submission rules.

Question 1: 2D Structure Extraction

In this exercise, we will implement a Hough transform in order to detect parametric curves, such as lines or circles. In the following, we shortly review the motivation for this technique.

Consider the point $p = (\mathtt{x}, \mathtt{y})$ and the equation for a line $y = mx + c$. What are the lines that could pass through p? The answer is simple: all the lines for which m and c satisfy $\mathtt{y} = m\mathtt{x} + c$. Regarding $(\mathtt{x}, \mathtt{y})$ as fixed, the last equation is that of a line in (m, c)-space. Repeating this reasoning, a second point $p' = (\mathtt{x}', \mathtt{y}')$ will also have an associated line in parameter space, and the two lines will intersect at the point $(\tilde{m}, \tilde{c})$, which corresponds to the line connecting p and p'.

In order to find lines in the input image, we can thus pursue the following approach. We start with an empty accumulator array quantizing the parameter space for m and c. For each edge pixel in the input image, we then draw a line in the accumulator array and increment the corresponding cells. Edge pixels on the same line in the input image will produce intersecting lines in (m, c)-space and will thus reinforce the intersection point. Maxima in this array thus correspond to lines in the input image that many edge pixels agree on.

In practice, the parametrization in terms of m and c is problematic, since the slope m may become infinite. Instead, we use the following parametrization in polar coordinates:

$$\mathtt{x} \cos\theta + \mathtt{y} \sin\theta = \rho$$

This produces a sinusoidal curve in (ρ, θ)-space, but otherwise the procedure is unchanged.

The following subquestions will guide you through the steps of building a Hough transform.

```
function houghSpace = houghtransform(imgEdge, nBinsRho, nBinsTheta)
```

a) Build up an accumulator array `acc` for the parameter space (ρ, θ). θ ranges from $-\pi/2$ to $\pi/2$, and ρ ranges from $-D$ to D, where D denotes the length of the image diagonal. Use `nBinsRho` and `nBinsTheta` as the number of bins in each direction. Initially, the array should be filled with zeros.

   For each edge pixel in the input image, create the corresponding curve in (ρ, θ) space by evaluating Eq.(1) for all values of θ and increment the corresponding cells of the accumulator array. Visualize the resulting hough space by displaying it as a 2D image. (Please remember to use the apply-functions.)

b) Write a function `nonmaxsup2d` which suppresses all points in the Hough space that are not local maxima. This can be achieved by looking at the 8 direct neighbors of each pixel and keeping only pixels whose value is greater than the value of the largest neighbor. This function is simpler than the non-maximum suppression from the Canny Edge Detector since it does not take into account local gradients.

```
function imgResult = nonmaxsup2d(imgHough)
```

   Write a function `findhoughpeaks` that takes the result of `houghtransform` as an argument, finds the extrema in Hough space using `nonmaxsup2d` and returns all points (ρ_i, θ_i) for which the corresponding Hough value is greater than `thresh`. Use the return variable `vRho` for ρ_i components, and `vTheta` for the θ_i components.

```
function [vRho, vTheta] = findhoughpeaks(houghSpace, thresh)
```

   Try your implementation on the images `gantrycrane.png` and `circuit.png`. Do you find all the lines?

c) (Bonus Question) The Hough transform is a general technique that can not only be applied to lines, but also to other parametric curves, such as circles. In the following, we will show how the implementation can be extended to finding circles.

   A circle can be parameterized by the following equation:

$$(\mathtt{x} - a)^2 + (\mathtt{y} - b)^2 = r^2.$$

   Unfortunately, the computation and memory requirements for the Hough transform increase exponentially with the number of parameters. While a 3D search space is still just feasible, we can dramatically reduce the amount of computation by integrating the gradient direction in the algorithm.

   Without gradient information, all values a, b lying on the cone given by Eq. (1c) are incremented. With the gradient information, we only need to increment points on an arc centered at (a, b):

$$\begin{aligned} a &= x + r\cos\phi & (1) \\ b &= y + r\sin\phi, & (2) \end{aligned}$$

   where ϕ is the gradient angle returned by the edge operator.

   Create a function `houghcircle` which implements the Hough transform for circles. Try your implementation for a practical application of counting coins in an image. You can use the images `coins1.png` and `coins2.png` for testing.

```
function [vA, vB, vR] = houghcircle(imgEdge, nBinsA, nBinsB, nBinsR,
    thresh)
```

d) (Bonus Question) Download the "Haribo classification" demo from the class webpage and adapt this demo code to execute the functions you wrote in the previous questions. Can you build an online coin classification and counting system? (Hint: you may need to include a reference shape in the picture in order to obtain the absolute scale).

Question 2: Mean-Shift Clustering

The Mean Shift algorithm clusters an n-dimensional data set (i.e., each data point is described by a feature vector of n values) by associating each point with a peak of the data set's probability density. For each point, Mean Shift computes its associated peak by first defining a spherical window at the data point of radius r and computing the mean of the points that lie within the window. The algorithm then shifts the window to the mean and repeats until convergence, i.e., until the shift is less than a threshold (here: 0.1). At each iteration the window will shift to a more densely populated portion of the data set until a peak is reached, where the data is equally distributed in the window.

Debug and test your algorithm using the provided dataset `pts.mat` which contains a set of 3D points that belong to two 3D Gaussian distributions. Plot your results with the provided function `plot3Dclusters`.

a) Implement the peak searching processes as the function

```
function peak = findpeak(data, idx, r)
```

where `data` is an $n \times p$ matrix containing p data points; each point is defined by an n-dimensional column vector of feature values; `idx` is the column index of the data point for which we wish to compute its associated density peak; and `r` is the search window radius.

b) Implement the `meanshift` function, which calls `findpeak` for each point and then assigns a label to each point according to its peak. This function should have the syntax:

```
function [labels, peaks] = meanshift(data, r)
```

where `labels` is a $1 \times p$ vector that has an entry for each data point of data storing its associated cluster label, and `peaks` is a matrix (with p columns) storing the density peaks found using `meanshift` for each data point. Peaks should be compared after each call to `findpeak` and similar peaks should be merged. For your implementation, consider two peaks to be the same if the distance between them is $\leq \mathtt{r}/2$. Also, if the peak associated with a data point is found to already exist in `peaks`, then for simplicity its computed peak is discarded and it is given the label of the already existing peak in `peaks`.

c) As described so far, the Mean Shift algorithm is too slow to be used for image segmentation where each pixel is a data point. Therefore, you should incorporate the following two speedups into your implementation. Upon finding a peak, the first speedup will be to associate each data point that is at a distance $\leq \mathtt{r}$ from the peak with the cluster defined by that peak. This speedup is known as the "basin of attraction" and is based on the intuition that points that are within one window size distance from the peak will, with high probability, converge to that peak. Incorporate this speedup into your implementation of `meanshift` and call the new function `meanshift_opt`:

```
function [labels, peaks] = meanshift_opt(data, r)
```

The second speedup is based on a similar principle, where points that are within a distance of $\mathtt{r}/c$ of the search path are associated with the converged peak, where c is some constant value. Use $c = 4$ for this assignment. To realize the second speedup, you will need to modify `findpeak` as follows:

```
function [peak, cpts] = findpeak_opt(data, idx, r)
```

where `cpts` is a vector storing a 1 for each point that is within a distance $\mathtt{r}/4$ from the path, and a 0 otherwise.

Mean-shift can be computationally quite expensive when implemented in Matlab. Use the following hints in order to speed up your implementation.

- Since MATLAB is optimized for matrix operations, not loops, try to avoid using loops in your functions whenever possible. Instead, use matrix manipulation. For example, if you want to select all points that have been labeled "1," instead of writing a for loop you could write:

```
currdata = data(:, labels == 1);
```

which uses logical indexing to return the elements of labels that match the Boolean expression, which in this case is `labels == 1`. Then `data(:, labels == 1)` selects those columns whose indexes are listed in the resulting logical vector.

- If you are trying to find points in a matrix `A` that are closer than `r` to the point of index `idx` in `A`, you can do it with a for loop this way:

```
for i = 1:size(A, 2)
        R(i) = norm(A(:, i) - A(:, idx));
end
find(R<r)
```

But this is way too slow in MATLAB. The following is much faster:

```
n = size(A, 2)
find(r > sqrt(sum((A - repmat(A(:, idx), 1, n)).^2, 1)))
```

d) Choose different values for the radius and compare the results. Which radius gives good results and how did you find it? How much faster is the optimized version? Does it change the result? If yes, is the result worse?

Question 3: Mean-Shift Image Segmentation

Next, build upon your implementation so that it can be used to perform image segmentation. Note that although very efficient implementations exist, the simplified implementation of Mean Shift we implement here may take several minutes to run. Debug using small images.

a) Implement the function

```
function segIm = meanshift_segment(im, r)
```

where `im` is an input image or, more generally, an image feature matrix, and `r` is the parameter associated with the Mean Shift algorithm. The output segmented image is then constructed using the cluster labels and peak values. That is, the output image is constructed by assigning a different color value (from the peak value) to each label and coloring pixels in the output image accordingly.

b) Note that Mean Shift clusters use the Euclidean distance metric. Unfortunately, Euclidean distance in RGB color space does not correlate well to perceived difference in color by people. For example, in the green portion of the spectrum large distances are perceived as the same color, whereas in the blue part of the spectrum a small distance may represent a large change in perceived color (see Figures 6.13 and 6.14 of Forsyth and Ponce). For this reason you should use the non-linear LUV color space. In this space Euclidean distance better models the perceived difference in color. In `meanshift_segment` cluster the image data in LUV color space by first converting the RGB color vectors to LUV using the provided MATLAB function `rgb2luv`. Then convert the resulting cluster centers back to RGB using the provided MATLAB function `luv2rgb`.

```
function seg_img = meanshift_segment_luv(img, r)
```

Segment the images sunset.png and terrain.png using `r = 5` and `10`. Given an input image file, you will need to read the image into MATLAB and then convert it into the matrix form required by data. If the feature vector is simply the 3 LUV color values at a pixel, then you will have to create a $3 \times p$ matrix where p is the number of pixels in the input image. If we want to include spatial position information as well, we can define the feature vector as a 5D vector specifying the LUV and x, y coordinates of each pixel. For each value of `r`, run your algorithm using (1) just the LUV color values (i.e., a 3D feature vector), and (2) LUV+position values (i.e., a 5D feature vector).

```
function seg_img = meanshift_segment_luv_pos(img, r)
```

c) Experiment a bit with different parameter settings. What effect does varying `r` and the feature vector seem to have on the resulting segmentations? What effect does adding position information have on the resulting segmentations? What are the advantages and disadvantages of using each type of feature vector? Can you suggest any extensions that might avoid many of the situations where single regions are over-segmented into multiple regions? Also run your algorithm on at least one other test image using your own choice of `r` and feature vector definition. One source for possible test images is the dataset of images available at `http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/`.

Question 4: Image Segmentation with Graph Cuts

This exercise will show you to how to implement a segmentation (foreground vs. background) using Markov Random Fields. In general, Markov Random Fields are a graphical model that incorporates the structure of the input and output variables into its calculation. In this exercise that means we do not use only a model for foreground and background but also take into account the neighbor relations of the pixels. Both models are combined in a so called energy function, which is then minimized to obtain an optimal model. Formally the problem is described as following:

Given a structured input (here: image) $\mathcal{Y} = \{Y_j | j \in I\}$ find the output (here: labeling) $\mathcal{X} = \{X_j | j \in I\}$, such that

$$\widehat{\mathcal{X}} = \operatorname{argmin}_{\mathcal{X}} E(\mathcal{X}, \mathcal{Y})$$

with E defined as

$$E(\mathcal{X}, \mathcal{Y}) = \sum_{j \in I} \psi_u(X_j, Y_j) + \sum_{\substack{i \in I \\ j \in I}} \psi_p(X_i, X_j, Y_i, Y_j)$$

The set I contains all possible indices. The so called unary potential ψ_u models the probability of foreground and background. The pairwise potential ψ_p incorporates the dependencies between pixels. To speed up the computation, the pairwise model is usually restricted to neighboring pixels and is therefore set to zero if X_i and X_j are not direct neighbors in a 4-neighborhood. The unary potential is written as

$$\psi_u(X_j, Y_j) = -w_u \cdot \log p(X_j | Y_j)$$

with an appropriate model p for the foreground and the background and a weighting factor w_u. The pairwise potential is then written as

$$\psi_p(X_i, X_j, Y_i, Y_j) = w_p \cdot \begin{cases} 1, & \text{if } X_i = X_j \\ 0, & \text{otherwise} \end{cases} .$$

A Graph Cut method is used to find the optimal solution $\widehat{\mathcal{X}}$ of the energy function. A framework[1] is provided so you do not have to implement the min-cut algorithm. You just have to pass the different parts of the energy function and the framework computes a local minimum of the energy function. There are two functions in the framework: `GCMex_compile`, which compiles the framework and `GCMex_test`, which checks the compiled binaries for correct behavior. You should not need to compile the framework since binaries are already available.

a) Implement a function to model the probability of a color given either the foreground or the background. Foreground and background are each initially estimated by a user defined box in the image. Usually gaussian mixture models are used for this task. However, to keep this exercise simple, we only want to use relative frequencies of colors (i.e. histograms) in the respective region of the image defined by the box.

[1] `http://vision.ucla.edu/~brian/gcmex.html`

```
function histogram = calculate_histogram(img, mask, n_bins)
```

The function should take as input the image `img` with values in $[0, 255]$. Additionally, it should receives a `mask` the same size as the image. The mask is 1 at the positions of the image where the histogram will be computed, zero otherwise. The final parameter `n_bins` describes how many bins are used in the histogram in each dimension. The function should return a 3-dimensional matrix with the relative frequencies of the color (r, g, b) within the region of the image defined by the mask. The histogram should be normalized. Initialize all bins with a small value (10^{-3}) to counteract relative frequencies which are zero (This method is called additive smoothing). Explain what you see in the marginal histograms.

b) The next step in the segmentation process is the estimation of a probability map. For each pixel of the image we want to estimate the probability of it belonging to the foreground. This will be used as basis for the unary potential.

```
function foreground_map = foreground_pmap(img, fg_histogram,
    bg_histogram)
```

The function should take the image `img` and the two histograms `fg_histogram`, `bg_histogram` estimated from the foreground region and the background region respecively. It should return a matrix the same size as the image containing the probability for each pixel to belong to the foreground. Use an equal prior probability for foreground and background. Explain what you see in the probability map. To estimate the required probability $p(c|[r, g, b])$ from the computed histograms, a class prior $p(c)$ of 0.5 is used, which means that both foreground and background are equally likely.

$$p(c|[r, g, b]) = \frac{p(c) \cdot p([r, g, b]|c)}{p([r, g, b])} = \frac{p(c) \cdot p([r, g, b]|c)}{\sum_{\tilde{c}} p(\tilde{c}) \cdot p([r, g, b]|\tilde{c})}$$

c) Use the previously computed probability map to compute the unary potential for both foreground and background.

```
function potentials = unary_potentials(probability_map, unary_weight)
```

This function shall use the `probability_map` and a scalar weighting factor to compute the unary potentials. It should return a matrix of the same size as the probability matrix.

Additionally, create a function to compute the prefactor of the pairwise potential for two specific pixels. The function signature is the following:

```
function score = pairwise_potential_prefactor(img, x1, y1, x2, y2,
    pairwise_weight)
```

where `img` is the image, (`x1`, `y1`), (`x2`, `y2`) the points in the image and the last parameter the weight for the pairwise potential. Keep in mind that this prefactor does not depend on both labels and is therefore independent of $\mathcal{X}$!

d) Using the functions from the previous task, implement a function to compute a sparse matrix of all pairwise potential prefactors for the image in a 4-neighborhood. That means only the top, bottom, left and right pixels should be connected to a given pixel. At (i, j) the resulting matrix should contain the pairwise potential prefactor for pixels i and j. Note that you have to use a linearized index instead of (x, y)-coordinates. A conversion function will be supplied:

```
function idx = coord_transform(height, x, y)
        idx = ((x-1) * height) + y;
end
```

A sparse matrix is used to reduce the memory consumption since most of the pixels are not neighbors and their pairwise potential therefore zero. A sparse matrix can be created by

```
pairwise_matrix = sparse(from, to, cost)
```

where all parameters `from`, `to`, `cost` are vectors of the same size.
With this information, implement a function to compute the sparse matrix with the following signature:

```
function pairwise_matrix = pairwise_potential_prefactor_matrix(img,
    pairwise_weight)
```

Now you can execute the optimization. How is the result? Why is the segmentation the way it is?

e) (Bonus Question) Modify the pairwise potential to incorporate a contrast sensitive term. This pairwise potential is called contrast sensitive Potts model.

$$\psi_p(X_i, X_j, Y_i, Y_j) = w_p \cdot \exp\Big(-w_d ||Y_i - Y_j||^2\Big) \cdot \begin{cases} 1, & \text{if } X_i = X_j \\ 0, & \text{otherwise} \end{cases}.$$

What does this modification change in the resulting segmentation?

f) (Bonus Question) Iterating the labeling process is a method to improve the final labeling. Implement a method to execute the optimization process iteratively. Use the previously computed labeling as initial segmentation and estimate new models for foreground and background on these.

```
function labeling = iterate_opt(img, foreground_mask, n_bins,
    unary_weight, pairwise_pot, n_iter)
```

How does the labeling change? Do you have an explaination why?

g) (Bonus Question) Extend the given framework to allow the user to select an additional foreground region. Then recalculate the foreground and background model according to the combined selection and segment the image again. Iterate this process until the user is satisfied with the result.

Please turn in your solutions including all relevant files before 2015-12-07!