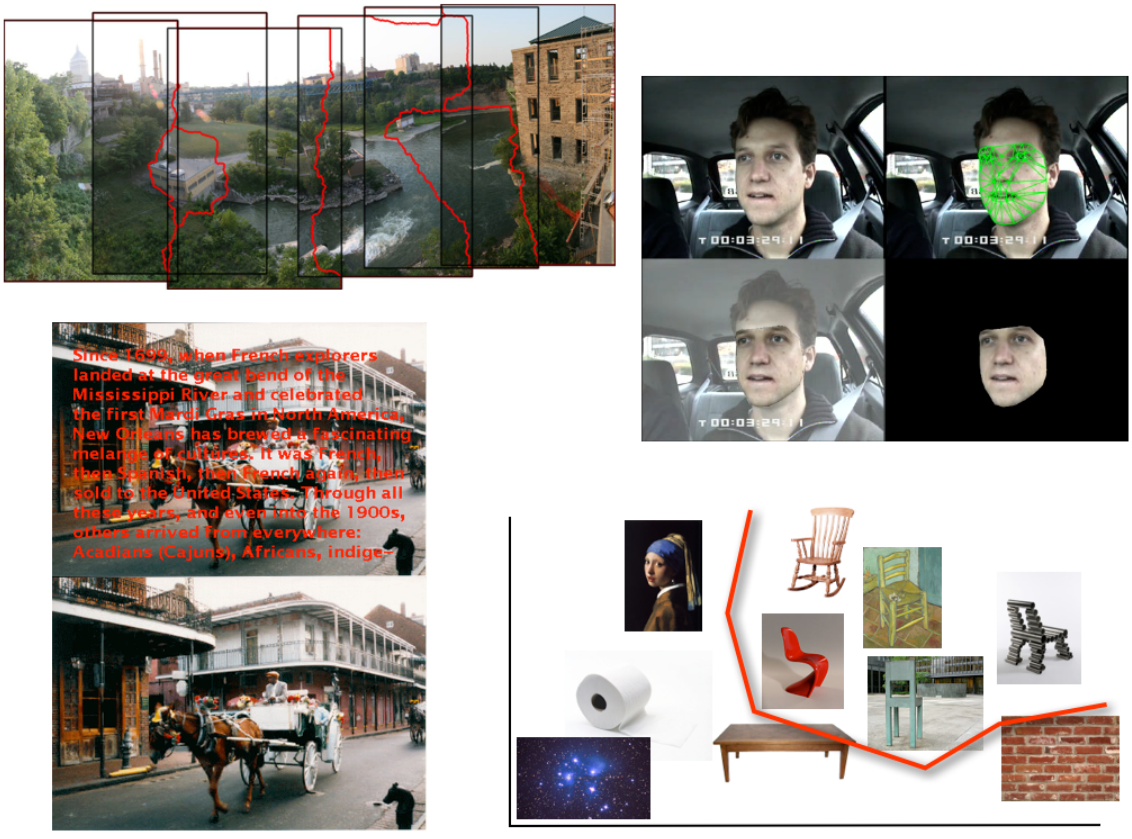
# IN4393 Computer Vision: Assignments

Delft University of Technology, April–June 2017




TU Delft

# 1 Feature point detection

In these exercises, you will develop and experiment with Matlab code for the automatic detection of feature points.

**Exercise 1.1. Harris corner detector**

For this exercise, you will first load an image and normalize it:

```
>> I = imreadbw('images/im.jpg');
```

The Harris corner detector inspects the eigenvalues of the autocorrelation matrix $\mathbf{A}$ at every location in the image. Recall that the autocorrelation matrix $\mathbf{A}$ a a particular image location is given by

$$\mathbf{A} = w \star \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{1}$$

where $I_x$ denotes the horizontal image gradient and $I_y$ the vertical image gradient, and $w$ is a local weighting function.

**Step 1.** Write Matlab code that approximates the horizontal and vertical image derivatives by convolving the image with an `[-2 -1 0 1 2]` filter, and run this code on image `I`. Inspect the resulting image gradients.

**Step 2.** Write Matlab code that generates a $15 \times 15$-pixel filter $w$ that contains a centered Gaussian kernel with bandwidth $\sigma = 2$. Make sure that the elements of the filter sum up to one.

**Step 3.** Write Matlab code that computes $w \star I_x^2$ for every location in the image. Store the results in a single matrix `wIx2`. Note that this step can be implemented efficiently via a two-dimensional convolution.

**Step 4.** *Bonus question:* How could you decompose the two-dimensional filter $w$? Can you use this decomposition to implement the previous step more efficiently?

**Step 5.** Construct the matrices `wIy2` and `wIxIy` that contain $w \star I_y^2$ and $w \star I_x I_y$, respectively.

Note that we now have computed and stored all three unique elements of the autocorrelation matrix $\mathbf{A}$ at every location in the image. The Harris corner detector inspects the eigenvalues $\lambda_1$ and $\lambda_2$ of this autocorrelation matrix. The eigenvalues of a $2 \times 2$-matrix $\mathbf{X} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}$ can be computed efficiently as follows:

$$\lambda_1 = \frac{a+c}{2} + \sqrt{b^2 + \left(\frac{a-c}{2}\right)^2} \tag{2}$$

$$\lambda_2 = \frac{a+c}{2} - \sqrt{b^2 + \left(\frac{a-c}{2}\right)^2} \tag{3}$$

**Step 6.** Compute the eigenvalues $\lambda_1$ and $\lambda_2$ for all locations in the image. Inspect the two eigenvalue-fields using the `imagesc` function. What do you see?

**Step 7.** Implement the Harris feature function: $\det(\mathbf{A}) - \alpha \operatorname{trace}(\mathbf{A})^2 = \lambda_1 \lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$. Set $\alpha = 0.06$ and compute the value of the Harris feature function at each location in the image,

and store the result in a matrix `harris_im`. Inspect the resulting matrix `harris_im` using the `imagesc` function. What do you notice?

**Step 8.** Corners are located at sufficiently high local maxima in the matrix `harris_im`. These local maxima can be identified using non-maxima suppression. Use the function `nonmaxsuppts` to perform such non-maxima suppression, and plot the identified corner points onto the input image. Experiment with different values of the threshold `thresh` to find a good value for it.

**Step 9.** Combine all the Matlab code you have written to construct a function `[r, c] = harris(im)` that performs Harris corner detection.

### Exercise 1.2. Rotation and scale invariance

Load the image `img.jpg` using the `imreadbw` function and display it using `imshow`. Use the `[r,c] = harris(im)` function you implemented in the previous exercise to detect corners in the image. Plot the detected corners by typing:

```
>> hold on; scatter(r, c, 12); hold off
```

Rotate the image $90°$ by simply transposing it (using the `'` operator). Re-run your Harris corner detector and plot the results in the same way as before. Did the detector find different corners? Why or why not?

Now resize the image to half its size using the `imresize` function (use bicubic interpolation), and re-run your Harris corner detector. Did the detector find different corners? Why or why not?

### Exercise 1.3. Scale-invariant feature transform

In the following exercise, you will become acquainted with the scale-invariant feature transform (SIFT). You will be using the SIFT-implementation by Andrea Vedaldi (UCLA). Make sure that the `'sift/'` folder is added to your Matlab path, by executing:

```
>> addpath('sift');
```

If the SIFT-implementation gives strange errors, you may need to recompile the software for your platform. You can do so by typing:

```
>> sift_compile
```

You can load and normalize an image a follows:

```
>> I = imreadbw('images/im.jpg');
>> I = I - min(I(:));
>> I = I / max(I(:));
```

Load the images `'images/img1.jpg'` and `'images/img2.jpg'`, normalize the images, and inspect both images using the `imshow` function. Confirm that the two images depict the same object under two different viewpoints.

You can perform SIFT on one of the two images using the following function:

```
>> [frames, descr, gss, dogss] = sift(I, 'Threshold', 0.05);
```

First, we will inspect the difference-of-Gaussian responses that were computed by SIFT:

```
>> plotss(dogss); colormap gray;
```

What do the rows in the plot correspond to? And the columns? Which subplots reveal fine-scale features? And which subplots reveal coarse-scale features? Are there features that show up in all subplots?

Next, we inspect the identified SIFT feature points:

```
>> imagesc(I); colormap gray; axis off
>> h = plotsiftframe(frames); set(h, 'LineWidth', 2, 'Color', 'g');
>> h = plotsiftframe(frames); set(h, 'LineWidth', 1, 'Color', 'k');
```

What do the center points of the circles correspond to? And the line inside the circles? And the size of the circles? In what kind of image regions (homogeneous, edge, corner, *etc.*) does the SIFT detector find feature points?

Make the image twice as small and re-run the SIFT detector. Do you find the same feature points?

Perform SIFT feature point detection on both the images that you have loaded into Matlab. Plot the detected keypoints side-by-side (using the `subplot` function). Can you find some matching feature points by visual inspection?

# 2 Feature point descriptors and matching

**Exercise 2.1. Computing SIFT descriptors**

This exercise builds on the results from the exercises in the previous section.

Load the images `img1.jpg` and `img2.jpg`, perform SIFT feature point detection, and compute the descriptors for the two images.

```
>> I1 = imreadbw('img1.jpg');
>> I2 = imreadbw('img2.jpg');
>> [frames1, descr1] = sift(I1, Threshold, 0.05);
>> [frames2, descr2] = sift(I2, Threshold, 0.05);
```

Inspect the feature points that were found. Next, inspect some of the SIFT feature descriptors, `descr1`, that were extracted from the first image:

```
>> for i=1:9
>>  subplot(3, 3, i); plotsiftdescriptor(descr1(:,i)); axis off
>> end
```

What are these plots showing?

**Exercise 2.2. Matching SIFT feature points**

This exercise re-uses the descriptors `descr1` and `descr2` that you computed previously.

Implement a function `matches = siftmatch_nn(descr1, descr2, threshold)` that returns an $N \times 2$-matrix that contains $N$ pairs of descriptor indices that match according to a thresholded nearest neighbor criterion. In other words, find the nearest neighbor in `descr2` for each descriptor in `descr1`, but only accept the match whenever the Euclidean distance between the two descriptors is smaller than `threshold`.

Perform the following commands to compute and visualize the matches you found:

```
>> matches = siftmatch_nn(descr1, descr2, threshold);
>> plotmatches(I1, I2, frames1(1:2,:), frames2(1:2,:), matches);
```

Try different values of `threshold`, and try to find a value that works well. Roughly what percentage of feature points is correctly matched? Is it easy to spot some mismatched feature points? Can you think of a way to reduce the rate of mismatches?

The standard SIFT algorithm uses an approximate nearest neighbor search using kd-trees to match feature points. Match the two sets of SIFT feature descriptors using the standard SIFT algorithm and show the resulting matches by executing:

```
>> descr1 = uint8(512 * descr1);
>> descr2 = uint8(512 * descr2);
>> matches = siftmatch(descr1, descr2);
>> plotmatches(I1, I2, frames1(1:2,:), frames2(1:2,:), matches);
```

How do these matches compare to the matches you found with the nearest neighbor algorithm you implemented yourself? Why is it important to use a RANSAC-algorithm when trying to find the transformation that relates the two camera viewpoints?

### Exercise 2.3. Panography

In this exercise, you will develop a simple algorithm to construct a panography of two images.

Load the two images `bigsur1.jpg` and `bigsur2.jpg` as follows:

```
>> I1 = im2double(imread('bigsur1.jpg'));
>> I2 = im2double(imread('bigsur2.jpg'));
```

As before, you can use the `sift`-function to extract SIFT feature points and descriptors:

```
>> [frames1, descr1] = sift(rgb2gray(I1), 'Threshold', 0.05);
>> [frames2, descr2] = sift(rgb2gray(I2), 'Threshold', 0.05);
```

Match the descriptors extracted from the two images using the `siftmatch` function, and show the results using the `plotmatches` function. Try different values of the SIFT `'Threshold'` until you are satisfied with the number of matches you obtained.

You can compute the translation that is represented by the `i`-th match as follows:

```
>> translation = frames1(1:2, matches(1, i)) - ...
                 frames2(1:2, matches(2, i));
```

Note that the first element of `translation` contains the translation in the $x$-direction, whilst the second element of `translation` contains the translation in the $y$-direction.

When constructing a panography, we are only translating the images relative to each other. For translations, the difference between two corresponding points $\mathbf{x}$ and $\mathbf{x}'$ is linear in the parameters of the transformation $\mathbf{p}$:

$$\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x} = J(\mathbf{x})\mathbf{p}. \tag{4}$$

Noting that $\mathbf{x}' = \mathbf{P}\mathbf{x}$ with $\mathbf{P}$ being the $2 \times 3$ transformation-matrix corresponding to translations, show that this relation is indeed linear.

The optimal parameters $\mathbf{p}^*$ that minimize the sum of squared errors are given by:

$$\mathbf{p}^* = \left( \sum_{i=1}^{N} J(\mathbf{x}_i) J^{\mathrm{T}}(\mathbf{x}_i) \right)^{-1} \sum_{i=1}^{N} J^{\mathrm{T}}(\mathbf{x}_i) \Delta\mathbf{x}_i, \tag{5}$$

where $N$ is the number of points $\mathbf{x}_i$ for which you have found a SIFT match $\mathbf{x}'_i$. Fill in the translation Jacobian in the above equation and simplify the expression (*i.e.*, work out the inverse). What do you notice about the solution $\mathbf{p}^*$?

The translation that minimizes the sum of squared errors between corresponding points (after translation) is given by the mean of the translations of the corresponding points. Compute the mean translation `mean_trans` from the translations of the matched points.

You can use the function `show_panography(im1, im2, translation, 'hard')` to overlay the two images, translation `im2` with the translation `translation`. The function `show_panography(im1, im2, translation, 'soft')` will produce a similar result, but overlays the two images using alpha-blending.

Inspect the panography that is specified by the mean translation `mean_trans`. Do you think this produces a good result? Why (not)?

A RANSAC-algorithm works roughly as follows:

```
% Initialize inliers and error of best model
inliers = matches
best_err = Inf

% Perform RANSAC iterations
for iter=1:10

      % Fit model on current inliers
      model = fit_model(inliers)

      % Compute error of current model on the inliers
      err = compute_error(inliers, model)

      % Determine which observations are inliers
      inliers = do_fit_model_well(matches, model)

      % Store best model so far
      if err < best_err
          best_err = err;
          best_model = model;
      end
end
```

Implement a simple RANSAC algorithm to estimate the parameters of the translation. Use the sum of the standard deviations of the translations in both directions as a measure for the goodness of the model fit: if this standard deviation is low, then apparently all translations are quite similar, and are parameter estimate is very good. To determine whether or not a pair of matched points is an inlier, check whether the translation between the pair of points is within one standard deviation from the current parameter estimate.

Visualize the results of your RANSAC-algorithm using the `show_panography` function. Are they better than the least-squares estimate? Why? What should still be improved to obtain a nicely stitched image?

# 3 Image stitching

**Exercise 3.1. Fitting a homography**

Recall the definition of a homography between two corresponding feature points obtained from two different images:

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}^{\mathrm{T}}\tilde{\mathbf{x}}, \tag{6}$$

where $\tilde{\mathbf{x}} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{bmatrix}$ and $\tilde{\mathbf{x}}' = \begin{bmatrix} \tilde{x}' \\ \tilde{y}' \\ \tilde{w}' \end{bmatrix}$ are represented using homogeneous coordinates.

The function `fit_homography` fits a homography between two sets of $N$ corresponding feature points by performing the following minimization:

$$\min_{\tilde{\mathbf{H}}} \sum_{n=1}^{N} \| f(\mathbf{x}_n; \tilde{\mathbf{H}}) - \mathbf{x}'_n \|^2, \tag{7}$$

where the function $f(\mathbf{x}_n; \tilde{\mathbf{H}})$ applies the homography encoded by $\tilde{\mathbf{H}}$ on point $\mathbf{x}_n$. Is this minimization a linear or a non-linear least squares problem? What optimization algorithm would you use to perform the minimization?

Inspect the code of the `fit_homography`. In line 17–23, the homography is initialized. Line 38–73 implements an iterative procedure that performs the actual minimization. Can you tell which optimization algorithm is implemented here?

In line 43–45, a part of the code is missing. The missing part of code takes as input the points `x` and `y` (which are represented in Cartesian coordinates!) and (1) computes the homography represented by the matrix `H'` on these points, and stores the result in vectors `x_prime` and `y_prime` in Cartesian coordinates; and (2) stores the normalization weights of the homogeneous coordinates of `x_prime` and `y_prime` in a row vector called `Z`. Implement the missing part of the code.

Load the two images `images/im89.jpg` and `images/im90.jpg`, extract SIFT feature points from both images, and match both sets of feature points. Use the following command to fit a homography based on these matches:

```
>> H = fit_homography(p1_x, p1_y, p2_x, p2_y, matches);
```

This command assumes that the $x$- and $y$-coordinates of both sets of feature points are stored separately (in `p1_x`, `p1_y`, `p2_x`, and `p2_y`, respectively), and that matches is an $2 \times N$ matrix as produced by the `siftmatch` function. Inspect the matrix `H` you obtained. Do you understand what its entries mean?

The function `show_homography` is a simple implementation that visualizes the learned homography. Inspect the results of the following two commands:

```
>> show_homography(im2, im1, H);
>> show_homography(im1, im2, inv(H));
```

What does the inverse of the homography-matrix encode?

Are the results you obtained satisfactory? Why are the results much worse than you (presumably) expected? And what can we do about this?

### Exercise 3.2. Image stitching

Add the folders `sift` or `maxflow` to your Matlab path.

The function `find_homography` implements an algorithm that extracts SIFT keypoints, matches the keypoints, and then fits a homography using a RANSAC algorithm using the Levenberg-Marquardt algorithm of which you just finished the implementation. (Note: `find_homography` calls `fit_homography` in its inner loop!)

Load the same two images as in the previous exercise. Find the homography between the two images, and visualize it:

```
>> H = find_homography(im1, im2);
>> show_homography(im1, im2, inv(H));
```

Did the RANSAC algorithm improve over the results you obtained earlier? Are there any exposure differences between the two images?

The function `show_nice_homography` implements a seam-finding algorithm based on graph cuts. Use the following command to perform seam finding and the display the result:

```
>> show_nice_homography(im1, im2, inv(H), 'simple');
```

Compare the result with your previous result? Can you identify the seam? Where is the seam located?

Feathering reduces the effect of exposure differences between the two images by taking a weighted average of the two images around the seam. Use the following command to use feathering when producing the stitched image:

```
>> show_nice_homography(im1, im2, inv(H), 'feathering');
```

Does feathering sufficiently correct for exposure differences? Why (not)?

One can also compute the gradient of the two images, use the identified seam to combine the two gradient images, and reconstruct the combined image from the combined gradient image. The latter step is performed using an algorithm that finds a two-dimensional function of which the gradient is as close as possible to the gradient that is used as input (in the squared error sense).

Run this algorithm, and inspect the results

```
>> show_nice_homography(im1, im2, inv(H), 'gradient');
```

Is combining the images in the gradient domain a better or a worse way to correct for exposure differences?

In case you want to perform more experiments: in the `images/` folder, you find some more image pairs that you can use to experiment with the image-stitching software.

# 4 Face recognition

### Exercise 4.1. Eigenfaces

To start with the first exercise, `cd` into the `eigenface` folder.

You can use the `files = dir('*.tiff')` function to get a list of all TIFF-files in a folder. This function returns a struct-array; you can obtained the $i$-th filename via `files(i).name`.

Load all the images from the Jaffe data set, which are located in the `images/` folder. Convert each image to a row vector (using `x = x(1:end)`), and store all these row vectors in an $N \times D$ matrix `X`, with $N$ representing the number of images and $D$ representing the number of pixels per image.

To compute eigenfaces, you need to apply principal components analysis (PCA) to the matrix `X`. PCA performs an eigendecomposition of the covariance matrix of the data.

**Question 1.** How many images $N$ does your data set have, and how many pixels per image $D$? What is the rank of the covariance matrix?

Using the `eig` and `cov` functions, compute the eigenvalues of the covariance matrix of `X`. This may take a few minutes, so store the resulting eigenvectors and eigenvalues in a MAT-file so that you do not have to re-compute them over and over again.

The eigenvalues $\lambda$ are stored in a diagonal matrix; extract the diagonal using the `diag` function. Use the `sort` function to sort the eigenvalues (and the corresponding eigenvectors) in descending order.

Normalize the eigenvalues in $\lambda$ so that they sum up to one (*i.e.*, divide by the sum). Each normalized eigenvalue now measures the percentage of the variance that is explained by the corresponding eigenvectors. Select the principal components that together explain $90\%$ of the variance in the face data.

**Question 2.** How many principal components did you obtain?

Note that your principal components have length $D$, *i.e.* they correspond to pixels in the face images. Therefore, you can visualize the $i$-th principal component (eigenface) by reshaping it to the size of the original images:

```
>> imagesc(reshape(W(:,i), [64 64])); axis off equal tight; colormap gray
```

**Question 3.** Inspect all eigenfaces. What are the main variations between the different faces? Are these variations due to texture variations or due to shape variations? (Or both?)

### Exercise 4.2. Training an active appearance model

In the next three exercises, you will experiment with Matlab code for active appearance models. To start with the exercise, `cd` into the `aam` folder.

The AAM implementation uses a Viola & Jones-style face detector to initialize the AAM fits. This implementation is located in the `/initialization` folder, which you need to add to the Matlab path. You also need to add PRTools to the Matlab path. Type:

```
>> addpath('initialization');
>> addpath('prtools');
```

We also need to make sure that Matlab can find the images used for training the active appearance model. This is done by setting a variable called `base_folder`:

```
>> base_folder = 'ARF';
```

We can now inspect the data we will use to train the active appearance model. The data consists of a collection of images that are annotated with feature points. Inspect the data by typing:

```
>> show_annotations(base_folder);
```

By pressing a keyboard button, you can scroll through the dataset. **Question 1.** How many feature points are annotated in this dataset? In a trained AAM, how many variables will a single shape component have?

To train an AAM with 10 shape principal components and 30 texture principal components on all images in the data set, type the following:

```
>> no_shape_pcs = 10;
>> no_texture_pcs = 30;
>> [shape_model, texture_model] = learn_model(base_folder, ...
                                  no_shape_pcs, 1, no_texture_pcs);
>> shape_model.transf_mult = 'ARF';
```

The training procedure may take up to a minute to complete. You may choose to first save your trained model by typing:

```
>> save 'my_model.mat'
```

This allows you to re-use the trained model later (by typing: `load 'my_model.mat'`). You can inspect the trained active appearance model, by typing:

```
>> show_model(shape_model, texture_model);
```

This creates two figures, one showing the shape model, and another showing the appearance model.

**Question 2.** What type of transformation does the first shape component? And the second? And the third and fourth?

**Question 3.** What variations does the fifth shape component appear to model? Are these variations more or less common than the variations described by, say, the tenth shape component?

**Question 4.** What variations does the first texture component appear to represent? Looking at all texture components, what face regions exhibit the most variation?

**Exercise 4.3. Fitting an active appearance model**

Load an image from the dataset:

```
>> [images, points, israw] = get_file_lists(base_folder);
>> im = imread([base_folder '/images/' images(1).name]);
>> imshow(im);
```

Fit our AAM model to the image we just loaded:

```
>> [p, lambda, err, ind, diverged, precomp] = fit_model([], [], ...
                              im, shape_model, texture_model, 'color');
```

This command may take half a minute to complete. The vector p is the vector with shape loadings, whereas the vector lambda is the vector with texture loadings. What size do these vectors have?

Inspect the AAM fit, by typing:

```
>> show_result_fit([], [], im, p, lambda, 1, shape_model, texture_model);
```

This creates two figures that show the shape fit and the appearance fit, respectively.

**Question 1.** How was the shape fit computed from the parameters `p` and `lambda`? And the appearance fit?

The `fit_model` function also returned a variable `precomp`, and some variables we do not need. The variable `precomp` is a structure that contains all information that could be pre-computed for use in the inverse compositional algorithm. We can use it when we fit our model to new images.

Fit the AAM to another image, thereby using the precomputed data, and inspect the AAM fit:

```
>> im = imread([base_folder '/images/' images(2).name]);
>> [p, lambda] = fit_model([], [], im, shape_model, ...
                                texture_model, 'color', precomp);
>> show_result_fit([], [], im, p, lambda, 1, shape_model, texture_model);
```

Was the AAM fitting process faster or slower this time?

**Question 2.** Think about the differences between the Lucas-Kanade and the inverse compositional algorithm. What pre-computed variables does the precomp variable presumably contain?

**Exercise 4.4. Using the AAM fit parameters as features**

We can now fit our AAM on all images in the dataset, and store the AAM fit parameters to use them as features. You can do so by typing:

```
>> p = cell(length(images), 1);
>> lambda = cell(length(images), 1);
>> diverged = repmat(true, [length(images) 1]);
>> for i=1:length(images)
>>    disp(['Image ' num2str(i) ' of ' num2str(length(images)) '...']);
>>    im = imread([base_folder '/images/' images(i).name]);
>>    [p{i}, lambda{i}, err, ind, diverged(i)] = fit_model([], [], ...
                        im, shape_model, texture_model, 'color', precomp);
>> end
```

It may take several minutes to fit AAMs to all face images. In some rare cases, the inverse compositional algorithm may not converge, for instance, due to bad initialization. The AAM implementation detects such divergences, and sets the diverged-variable accordingly. Answer the following exercises while your computer is fitting AAMs.

**Question 1.** The first 4 numbers in each cell of the p-array now correspond to the loadings for the first 4 shape components. What information do the first 4 numbers encode? And what information

does the first number of the texture parameters encode? Is this information relevant to the identity or expressions of the faces?

**Question 2.** Although the AAM fitting on separate images is reasonably fast, it is not yet real-time. When AAMs are fitted on frames from a video sequence, such real-time can be obtained. Why is fitting AAMs to videos computationally cheap compared to fitting AAMs to separate images?

Once we fitted AAMs to all images, we will construct a dataset that we can use for classification experiments. We first remove the cases in which the AAM fitting was unsuccessful, and we remove the first 4 shape parameters and the first appearance parameter. Subsequently, we merge the shape and texture parameters into datasets that are labeled by expression. Type:

```
% Load labels, and remove diverged instances
>> load 'arf_labels.mat'
>> p(diverged) = [];
>> lambda(diverged) = [];
% Convert AAM fit parameters to a labeled dataset
>> p = cell2mat(p);
>> lambda = cell2mat(lambda);
>> X = dataset([p(:,5:end) lambda(:,2:end)], emotion_labels);
```

If somewhere along the way something went wrong, or you do not want to wait for the AAM fitting to complete, you can also load a pre-computed version of this dataset by typing:

```
>> load 'X.mat'
```

We can now use PRTools to train, e.g., a 1-nearest neighbor classifier to classify emotions and evaluate the performance of the classifier using 10-fold cross-validation. Type:

```
>> error = crossval(X, knnc([], 1), 10)
```

Also evaluate the error of some other classifiers you learned about (e.g., `ldc`, `loglc`, or `svc`). What classifier gives the best performance? For what percentage of the faces can you correctly predict the facial expression?

**Question 3.** In real life, we may perform slightly worse because we cheated a bit in the experimental setup described above. What did we do wrong?

# 5 Conditional and Markov Random Fields

**Exercise 5.1. Iterated conditional modes**

In this exercise, you will use the iterated conditional modes (ICM) algorithm to perform inference in a Markov Random Field that was designed for denoising binary images.

The following Matlab code loads an image `im`, converts it to a binary $\{-1, +1\}$ representation, and makes a copy of the image in the `true_im` variable:

```
>> im = imread('image.png');
>> im = rgb2gray(im);
>> [h, w] = size(im);
>> [val, ~, im] = unique(im);
>> im = reshape(im, [h w]);
>> im(im == 2) = -1;
>> true_im = im;
```

Inspect the resulting image that is stored in `true_im`.

Next, we will create a noisy copy of this image `noisy_im` by randomly permuting 10% of the binary pixel values:

```
>> mask = (rand(size(im)) > .9);
>> im(mask) = -im(mask);
>> noisy_im = im;
```

Inspect the resulting noise image `noisy_im`.

A good model to denoise this image is by defining a Markov Random Field (MRF) model over the graph $G = (V, E)$, where $V$ is the collection of all pixels and $E$ is the collection of all edges in the graph. Here, we will assume that $E$ forms a four-connected lattice: that is, each pixel is connected to its left, right, top, and bottom neighbor.

We define the following Markov Random Field over this graph:

$$p(\tilde{\mathbf{I}}|\mathbf{I}) = \frac{1}{Z} \exp\left(\eta \sum_{i \in V} \tilde{\mathbf{I}}_i \mathbf{I}_i\right) \exp\left(\alpha \sum_{i \in V} \tilde{\mathbf{I}}_i + \beta \sum_{(i,j) \in E} \tilde{\mathbf{I}}_i \tilde{\mathbf{I}}_j\right), \tag{9}$$

where $\mathbf{I}$ is the noisy image we obtained as input (*i.e.*, `noisy_im`) and $\tilde{\mathbf{I}}$ is the denoised image we wish to infer by maximizing the (log)-likelihood given in the above expression.

Recall that in the iterated conditional modes (ICM) algorithm, we pick a pixel $i \in V$, and maximize the log-likelihood $\log p(\tilde{\mathbf{I}}_i|\tilde{\mathbf{I}}_{/i}, \mathbf{I})$. Work out this expression for both possible values of $\tilde{\mathbf{I}}_i$, *viz.* $\tilde{\mathbf{I}}_i = -1$ and $\tilde{\mathbf{I}}_i = +1$. What do you notice about the two solutions you obtain? On how many pixels does the solution $\tilde{\mathbf{I}}_i^* = \text{argmax}_{\tilde{\mathbf{I}}_i} \log p(\tilde{\mathbf{I}}_i|\tilde{\mathbf{I}}_{/i}, \mathbf{I})$ depend?

Set the three parameters of the MRF as follows:

```
>> eta = .1;      % penalty for changing pixel value
>> alpha = -.05; % penalty for positive pixel value
>> beta = .2;     % penalty for dissimilar neighboring labels
```

Note that the new pixel value for $\tilde{\mathbf{I}}_i^*$ only depends on its four neighbors in the lattice $E$. As a result, we can update the values of a lot of pixels simultaneously. Below is an outline of the code for a single, complete ICM update:

```
>> for stride_x=2:3
>>   for stride_y=2:3
>>     im(stride_y:2:end - 1, stride_x:2:end - 1) = sign(...);
>>   end
>> end
```

Fill in the missing part of code by implementing the expression for $\tilde{\mathbf{I}}_i^*$ you derived above. Note that the neighbors of the pixels in `im(stride_y:2:end - 1, stride_x:2:end - 1)` can be obtained by `im(stride_y - 1:2:end - 2, stride_x:2:end - 1)`, `im(stride_y + 1:2:end, stride_x:2:end - 1)`, `im(stride_y:2:end - 1, stride_x - 1:2:end - 2)`, and `im(stride_y:2:end - 1, stride_x + 1:2:end)`, respectively. Also note that you will need to use the pixel values $\mathbf{I}_i$, which are stored in `noisy_im`, in the above bit of code.

Initialize your denoised image `im` with `noisy_im` and run a single ICM iteration. Plot the resulting image `im`. Have the results improved? (Note that you can compute the number of erroneous pixels by comparing the result with the "true" image that is stored in `true_im`.)

Run the ICM algorithm you just implemented for 25 iterations. What is the error in your final denoised image?

**Bonus question.** There is a small error in the above ICM implementation. Can you spot the error?

### Exercise 5.2. Graph cuts

To be able to run the graph-cut algorithm, add the `maxflow` folder to your Matlab path:

```
>> addpath('maxflow');
```

In this exercise, you will use the graph-cut algorithm to perform maximum a posterior (MAP) inference in the same Markov Random Field as in the previous exercise. Load the same image `true_im` as in the previous exercise, and corrupt it in the same way as before, storing the corrupted image in `noisy_im`. Also, set the parameters of the MRF in the same way as before.

The following code constructs a four-connected lattice of size `h` $\times$ `w`:

```
>> E = edges4connected(h, w);
```

What is the number of vertices in the graph we defined over the image? And the number of edges? What is the value that should be assigned to these edges?

We will now construct an adjacency matrix `A` to represent the edges between all vertices, by giving each edge $(i, j) \in E$ the value $\beta$ (*i*.e., the cost of assigning a different label to two connected pixels):

```
>> N = h * w;
>> V = repmat(beta, [size(E, 1) 1]);
>> A = sparse(E(:,1), E(:,2), V, N, N, 4 * N);
```

Why is the adjacency matrix `A` stored as a sparse matrix?

Next, we need to define a matrix `T` of size $N \times 2$ (with $N$ being the number of pixels in the image) that describes the unary factors associated with a pixel. The first column of this matrix represents the unary factor value associated with assigning pixels a value of $-1$. Likewise, the second column of `T` represents the unary factor value associated with assigning pixels a value of $+1$.

You can obtain a list of the pixel values in $\mathbf{I}$ via `noisy_im(:)`. This list will have exactly the same ordering as the list of pixels that was used to produce the adjacency matrix `A`. Construct matrix `T`.

In case matrix `T` is not sparse, make it sparse by typing: `T = sparse(T);`. Next, we run the graph-cut algorithm:

```
>> [~, im] = maxflow(A, T);
>> im(im == 0) = -1;
>> im = reshape(double(im), [h w]);
```

Plot the resulting, denoised image `im`. What percentage of the pixels has the same value as in `true_im`? Is the result you obtained with graph cuts better or worse than that obtained with ICM?

**Exercise 5.3. Fields of Experts**

Load a pretrained fields-of-experts model and inspect the filters:

```
>> patch_size = 5;
>> load(['foe_' num2str(patch_size) 'x' num2str(patch_size) '.mat']);
>> Wb = basis' * W;
>> for i=1:size(Wb, 2)
>>   subplot(5, 5, i);
>>   imagesc(reshape(Wb(:,i), [sqrt(size(Wb, 1)) sqrt(size(Wb, 1))]));
>>   colormap(gray); axis off;
>> end
```

Can you make any sense out of the filters?

Load an image that requires inpainting and the associated mask image:

```
>> im = imread('street.png');
>> mask = logical(imread('street_mask.png'));
```

Inspect both images. The field-of-experts model is essentially a prior distribution over images. To infer the most likely inpainted image, this prior needs to be combined with a likelihood. What would be an appropriate likelihood for inpainting?

Perform inpainting using the contrastive-divergence algorithm, using the pre-trained fields-of-experts model as an image prior:

```
>> painted_im = inpaint_foe(im, mask, basis, W, a);
```

The code shows an intermediate result every $10$ iterations, and also prints out the negative log-likelihood (which is referred to as the energy). Should the log-likelihood go up or down? And the energy?

The inference takes $2,500$ iterations to complete, so it may take up to $15$ minutes. There is no need to wait it out, but it may interesting to watch the inference procedure make progress. Which (parts of the) characters are disappearing first? And which ones last?

# 6 Object, scene, and shape recognition

**Exercise 6.1. Bag of visual words**

The `images` folder contains three subfolders with images of `airplanes`, `motorbikes`, and `schooners`.

Use the following code to obtain list of of images `image_list`, and a corresponding set of `labels`:

```
>> folders = dir('images/*');
>> label_count = 0;
>> labels = zeros(0);
>> image_list = cell(0);
>> for i=1:length(folders)
>>   if folders(i).name(1) ~= '.'
>>     label_count = label_count + 1;
>>     images = dir(['images/' folders(i).name '/*.jpg']);
>>     for j=1:length(images)
>>       image_list{end + 1} = ['images/' folders(i).name '/' ...
                                                    images(j).name];
>>       labels(end + 1) = label_count;
>>     end
>>   end
>> end
```

The function `extract_random_patches` extracts patches of size `patch_size` $\times$ `patch_size` from the images in the `image_list`. Gather a collection of 25,000 random image patches of size $9 \times 9$ pixels:

```
>> patches_per_image = 100;
>> patch_size = 9;
>> patches = extract_random_patches(image_list, ...
                          patches_per_image, patch_size);
```

To learn the codebook of prototypical image patches, we apply the $K$-means algorithm on the patches we have gathered. Perform k-means clustering with $K = 512$ clusters:

```
>> K = 512;
>> codebook = kmeans(patches, K);
```

The clustering may take up to two minutes to complete. How many prototypes does the codebook contain? Why can you obtain less than `K` prototypes? Update the value of `K` as follows:

```
>> K = size(codebook, 2);
```

Inspect the codebook you obtained using the `visualize_codebook` function:

```
>> visualize_codebook(codebook);
```

Explain what you are seeing the visualization.

You can use the function `im2col` to slide a window of the desired size over an image, and to

transform the contents of the sliding window to a column for each location of the window. Load an image from the `image_list`, convert it to a grayscale image, scale its pixel values between 0 and 1, and extract all $9 \times 9$ pixel patches from the image as follows:

```
>> im = imread(image_list{1});
>> if ~ismatrix(im)
>>   im = rgb2gray(im);
>> end
>> im = double(im) ./ 255;
>> im_patches = im2col(im, [patch_size patch_size], 'sliding');
```

Next, we need to assign each patch extracted from the image to its nearest neighbor in the codebook (based on the squared Euclidean distance). The squared Euclidean distance can be efficiently computed as follows:

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|^2 = \mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{y}^{\mathrm{T}}\mathbf{y} - 2\mathbf{x}^{\mathrm{T}}\mathbf{y}. \tag{8}$$

Note that the RHS expression can be used to efficiently compute the $N \times M$ squared Euclidean distance matrix between $N$ vectors $\mathbf{x}$ and $M$ vectors $\mathbf{y}$. (You will need to use the `bsxfun` or `repmat` function.)

Compute the $N \times K$ distance matrix `D` between the $N$ patches extracted from the image (`im_patches`) and the $K$ prototypes in the codebook. The following code will find the indices of the nearest prototype for each image patch:

```
>> [~, assignment] = min(D, [], 2);
```

To construct the final bag-of-visual-word feature vector for this image, construct a (normalized) prototype `assignment` histogram as follows:

```
>> feat = accumarray([ones(numel(assignment), 1) assignment], ...
                                ones(numel(assignment), 1), [1 K]);
>> feat = feat ./ sum(feat);
```

Perform the extraction of the bag-of-visual-word feature vectors (not the codebook construction!) for every image in the `image_list`, and store the results in data a matrix `X`. This may take up to five minutes, depending on the speed of your data connection and processor.

Make sure that the PRTools and minFunc toolboxes are added to the Matlab path, construct a labeled PRTools data set, and test the performance of a logistic regressor on the resulting data set using 10-fold cross-validation:

```
>> addpath('prtools'); addpath(genpath('minFunc'));
>> A = dataset(X', labels');
>> err = crossval(A, loglc2, 10)
```

What classification error do you obtain? Thinking about the feature representation we used to construct the codebook (and the assignments), can you think of a way to improve the quality of the bag-of-visual-word features?

Compute a (squared Euclidean) distance matrix `D2` between the bag-of-visual-word feature vectors in the matrix `X`. Use the matrix `D2` to look up the nearest neighbor for each of the images. Assuming the image with index `i` has as nearest neighbor the image with index `j`, you can plot the image and

its nearest neighbor as follows:

```
>> subplot(1, 2, 1); imshow(imread(im_list{j}));
>> subplot(1, 2, 2); imshow(imread(im_list{j}));
```

Inspect the nearest neighbors for some of the images. Are the results in line with what you would expect based on the error `err` you computer above?

**Bonus question.** Repeat the above classification experiment for different values of `K`. What appears to be a good value for `K`? Repeat the above experiment for different patch sizes, and inspect the error you obtain. What appears to be the optimal `patch_size`? Also, try concatenating bag-of-visual-word features obtained using different `patch_sizes` and classify the images based on the concatenated bag-of-visual-word features. Does this improve the results?

**Exercise 6.2. Dalal-Triggs detection**

In this exercise, you will train and evaluate a Dalal-Triggs detector for pedestrian detection. The folder `images` contains three folders: a folder with `positive` training examples, a folder with `negative` training examples, and a folder with `test` images.

Inspect some of the training images. Why do the positive images contain two versions of the same image? What type of invariance does this introduce in the detector? An why do you think the collection of negative images is so much larger than that of positive images?

Load one of the training images and extracts histogram-of-oriented gradient (HOG) features using a block size of 8 pixels:

```
>> block_size = 8;
>> im = im2double(imread('images/positive/crop_000010a.png'));
>> features = hog(im, block_size);
```

(You may need to do `mex -O hog.cc` first for this to work.) You can visualize the resulting HOG features using the following command:

```
>> imshow(visualizeHOG(features));
```

Run the code snippet above on a number of positive images. Do you recognize the pedestrians in the HOG features?

The function `load_hog_images` loads in all the training images, extracts HOG features from these images, and stores the resulting feature representations in a PRTools data set. It also returns the original size of the HOG feature matrices (before they were concatenated in a single feature vector). Build a HOG pedestrian-detection training set:

```
>> pos_folder = 'images/positive/';
>> neg_folder = 'images/negative/';
>> pos_files = dir([pos_folder '*.png']);
>> neg_files = dir([neg_folder '*.png']);
>> [A, hog_size] = load_hog_images(pos_folder, pos_files, ...
                                   neg_folder, neg_files, block_size);
```

This may take a few minutes to complete depending on the speed of your processor and network connection. If you encounter memory problems during this step, use a smaller, random subset of the negative examples in `neg_files` as input.

Next, train an L2-regularized linear logistic regressor on the pedestrian data set. Use a value of $0.1$ for the L2-regularization parameter $\lambda$:

```
>> lambda = 0.1;
>> W = loglc2(A, lambda);
```

Why is it important to use L2-regularization when training the pedestrian detector? What is the size of weight matrix learned by the logistic regressor? Why this size?

Evaluate the classification error of your logistic regressor via 5-fold cross-validation:

```
>> err = crossval(A, loglc2([], lambda), 5)
```

Do you think this error is sufficiently low to get a good pedestrian detector?

The weights learned by the logistic regressor can be visualized as a HOG image as follows:

```
>> imshow(visualizeHOG(reshape(W.data.E(:,2), hog_size)));
```

What has the logistic regressor learned?

The function `sliding_window_detector` applies the trained pedestrian classifier to a window that is slid over the image at multiple scales, and performs non-maxima suppression to make the final pedestrian detections. Why is it necessary to run the detector at multiple scales? Why do we need to perform non-maxima suppression?

Apply the pedestrian detector to the test images. The sliding window implementation is rather naive: it may take about a minute to perform the detection on a single image.

```
>> % Loop over test images
>> test_files = dir('images/test/*.png');
>> for j=1:length(test_files)
>>     % Load image
>>     im = im2double(imread(['images/test/' test_files(j).name]));
>>     [boxes, response] = sliding_window_detector(im, W, ...
                                        block_size, hog_size);
>>     % Plot results
>>     subplot(1, 2, 1); imagesc(response, [0 1]);
>>     colormap jet; axis equal tight off;
>>     subplot(1, 2, 2); imshow(im); hold on
>>     for i=1:size(boxes, 1)
>>         rectangle('Position', [boxes(i, 2)   boxes(i, 1) ...
>>                                boxes(i, 4) - boxes(i, 2) ...
>>                                boxes(i, 3) - boxes(i, 1)], ...
>>                           'LineWidth', 2, 'EdgeColor', [1 0 0]);
>>     end
>>     hold off; pause
>> end
```

The above code snippet shows the response surface in the left sub-image, and the detections in the right sub-image. What are false positives in the detections? And false negatives?

# 7 Tracking

**Exercise 7.1. Particle filter**

In this exercise, you will implement a simple particle filter based on sequential importance resampling. Specifically, you will implement the condensation algorithm that is presented in the lecture slides.

**Step 0.** Inspect the skeleton code you were given with this assignment. Where ever you encounter a `TODO` comment, you will fill in missing code to make the tracker work.

**Step 1.** Implement the appearance model in the `appearance_model` function. This function receives as input a color histogram of the object of interest (in the variable `colorModel`). It also receives as input the image `I` and a particle `s_t`. The function already extracts the corresponding image `patch`. Use the `color_histogram` function to extract the color histogram from this patch, and implement the following appearance likelihood:

$$L = p(\mathbf{I}_t|\mathbf{s}_t) \propto \exp(-\lambda d(\mathbf{h}(\mathbf{I}_0, \mathbf{s}_0), \mathbf{h}(\mathbf{I}_t, \mathbf{s}_t))), \tag{10}$$

where $\mathbf{h}(\mathbf{I}_0, \mathbf{s}_0)$ represents the color histogram of the object of interest and $\mathbf{h}(\mathbf{I}_t, \mathbf{s}_t)$ represents the color histogram of the current `patch`, and where the function $d(\cdot, \cdot)$ represents the Kullback-Leibler divergence between the two inputs. You can use the implementation of the Kullback-Leibler divergence provided in `kl_divergence`. Set $\lambda$ to 10.

**Step 2.** In the `motion_model` function, implement code that samples from a motion model $p(\mathbf{s}_t|\mathbf{s}_{t-1})$. The particles on which is conditioned are specified in `s_t1`, whereas the samples from the motion model should be returned in `s_t`. The $x, y$-location of the particles is represented in the first two columns of `s_t`. As motion model, use a Gaussian distribution that has mean `s_t1` and variance $\sigma^2 = 15$ (note that you can sample from a Gaussian distribution using the `randn` function).

**Step 3.** Around line 36-37 of the `tracker` script, implement code that initializes the `N` particles. Initialize the particles `s_t` to the bounding-box initialization specified by the user (`s_init`), and initialize the weights `w_t` to $\frac{1}{N}$.

**Step 4.** Around line 45 of the `tracker` script, implement code that resamples `N` particles from `s_t` according to their weights `w_t` (*i.e.* particles with higher weights have a higher probability of being selected). Store the resamples particles in `s_t` and `w_t`.

**Step 5.** Around line 49 of the `tracker` script, implement code that samples particles from the motion model given the resamples particles in `s_t`. Store the samples from the motion model in `s_t`.

**Step 6.** Around line 52 of the `tracker` script, implement code that computes the appearance likelihood of each of the newly generates particles. Store the resulting appearance likelihoods in an $N \times 1$ vector `L`.

**Step 7.** Around line 55 of the `tracker` script, implement code that updates the weights `w_t` based on the previous weights of the particles (`w_t`) and on the appearance likelihoods stored in `L`. Make sure that all conditions on the weights are satisfied.

**Step 8.** Based on the new particles in `s_t` and `w_t`, you need to estimate the expected location of the object. Around line 67 of the `tracker` script, implement this estimation code and store the resulting final estimate in `estimate_t`.

Run your tracker by typing `tracker`. Manually select the red-purple ball in the first frame. The tracker will run and will plot all particles along with the estimated object location.

Is the location predicted by the particle filter very stable? Can you think of a way to improve the stability of the particle-filter tracker at limited computational costs? Why would you sometimes want to use a particle filter instead of a sliding-window search?

Does the tracker resample often? Try commenting out the resampling code in your tracker, and re-run. What do you notice? Can you explain what happens when you do not use resampling?

Try increasing and decreasing the number of particles `N`. What do you notice? Which two characteristics of your tracker does the number of particles `N` trade off?

What would you change about your current tracker if you wanted to use to track pedestrians? (You may try to implement such a pedestrian tracker if you have time left.)