Chapter 10

Image understanding

This chapter provides tools for extracting high-level information from images, building models, and identifying their parameters.

RANSAC—Random sample consensus—(function `ransac`, p. 140) is a general and widely used technique for identifying the most plausible model from data containing a large proportion of outliers. It is typically used for geometry recovery in stereo and 3D reconstruction tasks.

Gaussian mixture models (function `gaussianmixture`, p. 144) are standard for both uni-dimensional and multidimensional data that come from several approximately Gaussian sources. This is often the case for an image histogram or for positional measurements of several moving objects.

Point distribution models describe shape families based on a set of landmark points on their contours, capturing their typical variations by a mean shape and a small number of eigenshapes. The model can be learned from a set of training contours (function `pointdistrmodel`, p. 148). Using a priori shape information from a point distribution model in segmentation (called active shape models, `asmfit`, p. 154) leads to very robust techniques.

10.1 Problems

10.1. Give an example of a real-world image understanding technique based on a bottom-up strategy.

10.2. Give an example of a real-world image understanding technique based on a top-down strategy.

10.3. What information is represented by a point distribution model?

10.4. Define principal components analysis.

10.5. Explain the process of determining the modes of variation in point distribution models. How is it possible that only a small number of modes is enough to cover most of the shape variations?

10.6. Consider Table 10.1. How many principal components must be used in the point distribution model to leave less than 5% of the variation unexplained?

10.7. What is an active appearance model? How does it differ from an active shape model?

10.8. What is the purpose and principle of discrete labeling?

10.9. Explain how each of the terms of the objective function [Equation 10.36] contributes to image interpretation.

10.10. Draw a simple image, label objects in it, and determine a region adjacency graph.

10.11. Draw a simple image and a corresponding region adjacency graph. Merge two neighboring regions and explain how the adjacency graph is updated.

10.12. Define an order-k Markov model.

10.13. Define an order-k hidden Markov model.

10.14. Define (i) evaluation, (ii) decoding, and (iii) learning in the context of hidden Markov models.

10.15. Explain the discrete relaxation strategy. Choose a small real world example and show a possible relaxation sequence.

10.16. In the recursive contextual classification approach [Algorithm 10.10], after how many recursive steps will image information at location (53, 145) influence the labeling at location (45, 130)?

10.17. Considering the 'ball on the lawn' example [Figure 10.36], sketch a specific region adjacency graph for the image segmentation and interpretation hypotheses represented by the following genetic strings: (i) LLBLB, (ii) LLBBL, (iii) BLLLB.

10.18. Describe a genetic image interpretation algorithm for the task of labeling a segmented scene.

10.19. Consider an alphabet $\{0, 1\}$ and a string `000010000100001010000100`. Construct a first-order Markov model and estimate the transition probabilities. How likely is it to observe the sequence `110111`?

10.20. Suppose an image consists of 90% background and 10% foreground. The background pixels have a normal distribution with mean 5 and standard deviation 100, the foreground pixels are also normally distributed with mean 150 and standard deviation 30. What is the probability distribution of a randomly chosen pixel? (Write an equation.)

10.21. Take an image and shift it by several pixels in an arbitrary direction. Detect feature points in both images using the Harris detector (function `harris`, p. 61). Use RANSAC (function `ransac`, p. 140) to determine the shift. How does the number of points influence the computational time? Study the effect of adding random perturbations to the point positions.

10.22. Write an algorithm for fitting a learned point distribution model (PDM) to a new shape based on iteratively minimizing the number of misclassified pixels (pixels marked as object by the PDM and background in the new shape, and vice versa). Compare its speed, accuracy and capture range with the ASM fit, function `asmfit` (p. 154). You can use the hand image from Figure 10.6.

10.23. Determine empirically from a chosen text the transition probabilities of the characters of English words. (Hint: consider only letters within words, ignore punctuation). Construct a first-order HMM, with hidden states corresponding to the true letters. The input features might, for example, be the bay and lakes count [Figure 2:2.15] for each letter. Take a set of words, convert each to a sequence of feature vectors, and use the Viterbi algorithm [Section 2:10.9] to estimate the original words. How much error do you make? Refine your feature set to improve the performance.

10.2 Random sample consensus: `ransac`

RANSAC [Section 2:10.2] is a stochastic parameter estimation technique which is especially useful for data containing a large number of outliers. The implementation described here extends the basic version [Algorithm 2:10.4] by automatic estimation of the number of iterations to perform. Whenever an iteration is successful, the number of points Q consistent with the currently best model (the number of inliers) is used to calculate an estimate of the inlier ratio $\xi = Q/N$, where N is the total number of data points. If we accept a failure probability ζ that RANSAC does not find a correct solution, the total number of iterations to perform is

$$K = \frac{\log \zeta}{\log(1 - \xi^M)} , \tag{10.1}$$

where M is the number of data points used to determine the model parameters.

The interface between the RANSAC core and the particular model to be determined consists of two user-provided functions: `get_model` calculates the model parameter from a small part (sample) of the data and `get_inliers` determines which data points are consistent with a given model.

function `[best_model,inliers] = ransac(x,m,get_model,get_inliers,zeta,maxit,xi,verbosity)`

input

`x`	[d×N]	Input data matrix. Each column is one d-dimensional data point.
`m`	[1]	The number M of points used to determine the model parameters.
`get_model`	function	A function called as `model=get_model(sample)`, where `sample` is a randomly selected subset of M columns from `x` and `model` contains the model parameters determined from `sample`.
`get_inliers`	function	A function called as `inl=get_inliers(x,model)` where `x` is the input data matrix and `model` describes the model to evaluate. The binary row vector `inl` should contain `1` for each column of `x` that is considered to be an inlier, i.e., consistent with the model.
zeta	$\{10^{-3}\}$	The probability ζ of RANSAC not finding the correct solution that we are ready to accept. It is used to determine the number of iterations to perform. Larger values (10^{-2}) make the algorithm run somewhat faster at the expense of an increased failure ratio.

maxit	$\{10^4\}$	The maximum number of iterations to perform unless K from equation (10.1) stops us first.
xi	$\{0\}$	An initial estimate of the inlier ratio $\xi = Q/N$. The value is not critical since ξ is re-estimated after each succesful iteration.
verbosity	$\{0\}$	If set to 2, progress is reported after each iteration. If set to 1, there is only one message at convergence. The corresponding code is omitted here.
output		
best_model		The best model parameters found, as returned by `get_model`.
inliers	$[1\times N]$	A binary row vector determining inliers (data points consistent with the model), as returned by `get_inliers`.

We initialize the iteration counter `iter` and the best-so-far model parameters (`best_model`, `best_support`, `inliers`). The number of iterations to perform is estimated using function `numiters` that implements Equation 10.1.

```
[d,n] = size(x);
iter  = 0;
best_model   = [];
best_support = 0;
inliers = [];
maxiter = min( maxit, numiters(xi,zeta,m) );
```

The main loop starts by randomly drawing a set `ind` containing M unique numbers from $1 \ldots N$. It is used to get the M-column subset `sample` of `x`.

```
while iter<maxiter
  ind = randsample( n, m );
  sample = x(:,ind);
```

If the function :`randsample` is not available (because it belongs to Matlab's Statistical toolbox) we can use `ind=randperm(n); ind=ind(1:m)` instead at the expense of some slowdown. We calculate the model parameters and the corresponding support (number of data points consistent with the model) by calling the user-provided functions `get_model` and `get_inliers`.

```
  model = get_model( sample );
  inl = get_inliers( x, model );
  support = sum(inl);
```

If the support is smaller than M, there is something wrong, so we alert the user.

```
  if support<m
    warning('ransac: Support of the generated model is smaller than M.')
  end
```

If the current model is better than the best model so far, we update the best model parameters and the number of iterations `maxiter`.

```
  if support>best_support
    best_support = support;  best_model = model;  inliers = inl;
    xi = support/n;
    maxiter = min( maxit, numiters(xi,zeta,m) );
  end
```

We increment the iteration counter and loop again.

```
  iter = iter+1;
end % while loop
```

```
function iters = numiters(xi, zeta, m)
```

Function `numiter` determines the total number of iterations to perform from equation (10.1). Parameter `xi` is the inlier ratio and `zeta` the acceptable failure probability.

```
if xi<eps
  iters = Inf;
else
  iters = max( 1, ceil( log(zeta)/log(1-xi^m) ) );
end
```

Example

We demonstrate RANSAC on the task of identifying a straight line from a cloud of points. The line will be described as $y = a_0 + a_1 x$ with a model parameter vector $\left[a_0, a_1\right]$. In particular we choose a line $a_0 = 10$, $a_1 = 0.3$. We generate $N = 1000$ points of which ξN are inliers, distributed regularly on the given straight line for $x \in \left[0; 100\right]$, with normal noise with a standard deviation $\sigma = 1$ (`sigma`) added to the y component. The remaining $(1 - \xi)N$ points are outliers, distributed uniformly over the rectangle $(x, y) \in \left[0; 100\right]^2$. Finally, the data points are randomly permuted.

```
n = 1000;
ninl  = ceil(xi*n);   % number of inliers
noutl = n-ninl;       % number of outliers
a1 = 0.3;  a0 = 10;   % straight line parameters
t = 100*(0:ninl-1)/(ninl-1);
xinl  = [t; a0+t*a1+sqrt(sigma)*randn(1,ninl)];
xoutl = 100 * rand( 2, noutl );
x = [xinl xoutl];
x = x(:,randperm(n));
```

RANSAC is called with the default parameters. Functions `find_line` and `close_to_line` are defined below. For comparison, we also calculate a least squares estimate of the line parameters using function :`regress`.

```
[model,inliers] = ...
  ransac( x, 2, @find_line, @(x,model)close_to_line(x,model,sigma) );
outliers = not(inliers);
lregr = regress( x(2,:)', [ones(n,1) x(1,:)'] );
```

Finally, we show the original samples, the true line (blue), the line found by linear regression (green) and the line found by RANSAC (red).

```
t1 = [0 100];
plot( x(1,inliers),x(2,inliers),'b.', t1,a0+a1*t1,'b-', 'LineWidth',6);
hold on
plot( x(1,outliers), x(2,outliers), 'c.', ...
      t1, model(1)+model(2)*t1, 'r-', ...
      t1, lregr(1)+lregr(2)*t1, 'g-', 'LineWidth',2 );
hold off
```

```
function model = find_line(x)
```

Function `find_line` identifies line parameters $\begin{bmatrix} a_0 & a_1 \end{bmatrix}$ from two points passed as columns of `x`. For simplicity, we temporarily ignore division by zero as such deficient models will be refused later.

```
[d,n] = size(x);
s = warning( 'off', 'MATLAB:divideByZero' );
a1 = ( x(2,1)-x(2,2)) / (x(1,1)-x(1,2) );
warning(s);
a0 = x(2,1) - a1*x(1,1);
model = [a0 a1];
```

```
function inl = close_to_line(x,model,sigma)
```

Function `close_to_line` determines which points of `x` are 'sufficiently close' to the line described by `model`. We know that the y coordinate has been corrupted by Gaussian noise with a known standard deviation σ. We choose the threshold to determine 'sufficient closeness' as $3\,\sigma$, which corresponds to a 95% confidence interval.

```
[d,n] = size(x);
y = model(1)*ones(1,n) + model(2)*x(1,:);
inl = abs(x(2,:)-y) < 3*sigma;
```

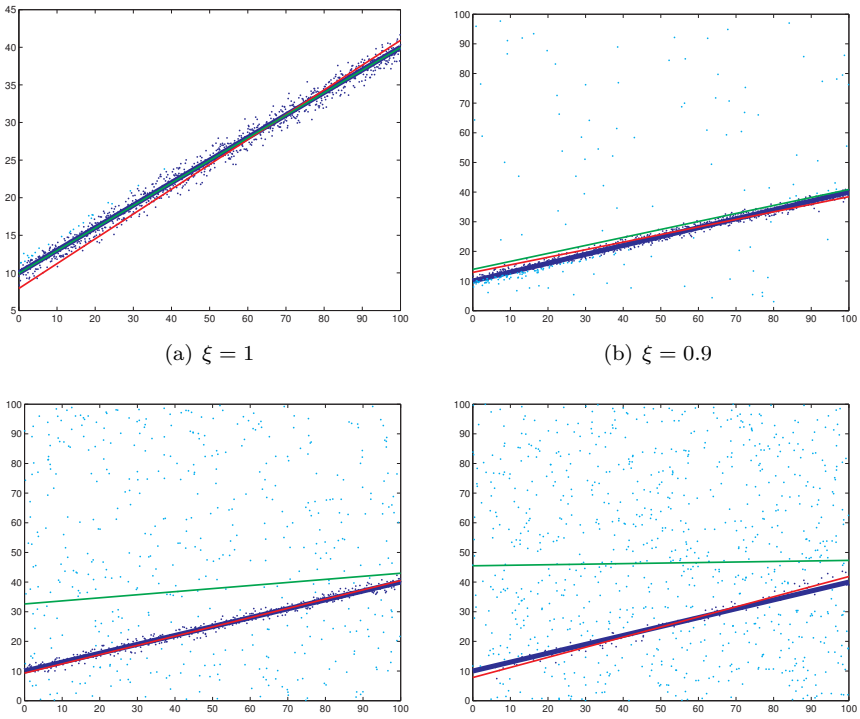
Figure 10.1 shows the results of the experiment above for different values of the inlier ratio ξ. For $\xi = 1$ (only inliers, top left), the least-squares approach is marginally better than RANSAC. With increasing ξ, the least-squares estimate deteriorates significantly and eventually breaks down completely, while RANSAC always identifies the correct line, even for ξ as low as 0.1. However, bear in mind that the number of RANSAC iterations increases significantly with decreasing ξ; from 5 for $\xi = 0.9$ to 281 for $\xi = 0.1$. This effect would be even more pronounced for higher M.

10.3 Gaussian mixture model estimation: `gaussianmixture`

A Gaussian mixture model [Section 10.10] is described by means μ_k, covariances Σ_k and weights w_k. The probability density is given as

$$p(\mathbf{x}) = \sum_{k=1}^{K} w_k \frac{1}{(2\pi)^{\frac{d}{2}} \, |\Sigma_k|^{\frac{1}{2}}} \, \exp\left(-\frac{1}{2} (\mathbf{x} - \mu_k)^T \, \Sigma_k^{-1} (\mathbf{x} - \mu_k) \right) ,$$

where d is the dimension. The parameters μ_k, Σ_k, and w_k can be estimated from N samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ using the expectation-maximization (EM) algorithm [Algorithm 10.17]. The algorithm performs well in practice even though global optimality of the result is not guaranteed.



(a) $\xi = 1$ (b) $\xi = 0.9$

(c) $\xi = 0.5$ (d) $\xi = 0.1$

Figure 10.1: Scatter plot of input data points for different values of the inlier ratio ξ. The true straight line to be identified is show in blue, its least squares estimate in green, and the RANSAC estimate in red. The points identified by RANSAC as inliers are shown in blue, outliers in light blue.

function `[mu,C,w] = gaussianmixture(x,k,verbosity)`

input

`x`	[d×N]	Input data matrix. Each column is one d-dimensional sample (point).
`k`	[1×1]	The number K of Gaussians to use.
`verbosity`	{0}	If set to 1, progress is reported. The corresponding code is omitted here.

output

`mu`	[d×K]	Centers μ_k.
`C`	$[d \times d \times K]$	Covariances Σ_k.
`w`	[K×1]	Weights w_k.

A suitable number of Gaussians K can be found either experimentally or by using more sophisticated approaches, such as the minimum description length principle [Section 10.10].

The initial centers μ_k are found using :kmeans clustering. The transpose (x', mu') serves to harmonize row/column conventions which are unfortunately not consistently used in Matlab.

```
[d,n] = size(x);
[idx,mu] = kmeans( x', k );
mu = mu';
```

The initial covariances C of each cluster (Gaussian) are calculated. Note that C is a 3D matrix (table). The weights w are initialized as uniform.

```
C = zeros( d, d, k );

for i = 1:k
  C(:,:,i) = cov( x(:,idx==i)' );
end

w = ones(n,1)/k;
```

The expectation-maximization (EM) core consists of a loop which is normally exited through the break statement (below) when convergence is detected. In the expectation step, we calculate for all data points $\mathbf{x}_i$ and all Gaussian components the probability that a particular point is generated by a particular component. (The function gausspdf is given below.) This probability is stored in p and normalized to sum to one for each point. The new weights w are the means of p for each Gaussian over all points.

```
for iter = 1:10000
  p = zeros( k, n );
  for i = 1:k
    p(i,:) = w(i) * gausspdf( x, mu(:,i), C(:,:,i) );
  end
  p = p ./ repmat(sum(p),k,1);
  w = mean( p, 2 );
```

In the maximization step, the new parameters μ_k and Σ_k are calculated using sample means and covariances weighted by the probabilities p [Equation :10.75]. sump serves to normalize p across components.

```
  oldmu = mu;  oldC = C;
  sump = sum( p, 2 );
  for i = 1:k
    mu(:,i) = x*p(i,:)'/sump(i);
    dif = x - repmat( mu(:,i), 1, n );
    C(:,:,i) = ( repmat(p(i,:),d,1).*dif ) * dif'/sump(i);
  end
```

Convergence is detected by comparing the maximum change of the Gaussian parameters μ_k and Σ_k with a threshold. This normally works well, however, for better flexibility, relative error could be tested too and both thresholds could be made user-selectable.

```
  e = max( [abs(mu(:)-oldmu(:))' abs(C(:)-oldC(:))'] );
  if e<1e-6, break; end
end % for iter
```

```
function prob = gausspdf(x,mean,sigma)
```

Given a mean `mean` (as a column vector) and covariance matrix `sigma` of a d-dimensional Gaussian distribution, evaluate the probability density function at all points `x`. Each column of `x` corresponds to one point.

```
[d,n] = size(x);
prb = zeros( n, 1 );
sigmainv = inv(sigma);
c = (2*pi)^(-0.5*d) * sqrt( det(sigmainv) );
for i = 1:n
  dif = x(:,i)-mean;
  prob(i) = c * exp( -0.5*dif'*sigmainv*dif );
end
```

Example

In a 1D example ($d = 1$), we randomly generate $N = 1000$ points using a mixture of $K = 3$ Gaussians with weights $w_1 = 0.2$, $w_2 = 0.5$, $w_3 = 0.3$, means $\mu_1 = 10$, $\mu_2 = 20$, $\mu_2 = 30$, and variances $\Sigma_1 = 4$, $\Sigma_2 = 25$, $\Sigma_3 = 1$. Note that the third parameter given to :`random` is a standard deviation (the square root of the variance).

```
n = 1000;
w1 = 0.2;  w2 = 0.5;  w3 = 0.3;
x = [random( 'norm', 10*ones(1,floor(n*w1)), 2*ones(1,floor(n*w1)) ) ...
     random( 'norm', 20*ones(1,floor(n*w2)), 5*ones(1,floor(n*w2)) ) ...
     random( 'norm', 30*ones(1,floor(n*w3)), 1*ones(1,floor(n*w3)) )];
```

Then the EM algorithm `gaussianmixture` (p. 144) is run. It takes about 230 iterations.

```
[mu,C,w] = gaussianmixture( x, 3 );
```

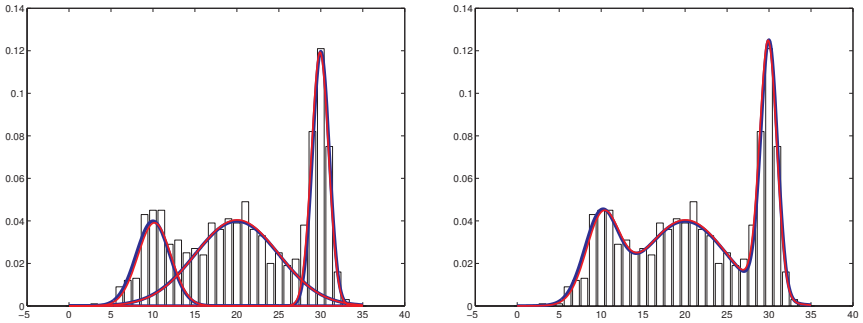
The estimated parameters are very close to the true ones, up to a permutation:

```
mu = 20.0356  10.1856  29.9062

C(:,:,1) = 24.5475
C(:,:,2) =  3.8672
C(:,:,3) =  1.0375

w' =  0.5024   0.1928   0.3048
```

Figure 10.2a shows the histogram of the generated samples calculated using function :`hist` with bin size 1, over which we superimpose the probability density functions of the three Gaussians and the total mixture density (Figure 10.2b). Observe that the estimate (in red) closely follows the true p.d.f. (in blue).

We continue with a 2D example ($d = 2$), with $N = 1000$ points generated with weights $w_1 = w_2 = 0.5$, means $\mu_1 = \begin{bmatrix}0 & 0\end{bmatrix}$, $\mu_2 = \begin{bmatrix}30 & 10\end{bmatrix}$ and covariances

$$\Sigma_1 = \begin{bmatrix}104 & 50 \\ 50 & 109\end{bmatrix}, \qquad \Sigma_2 = \begin{bmatrix}25 & 0 \\ 0 & 9\end{bmatrix}.$$



(a) (b)

Figure 10.2: Histogram of samples generated from a weighted mixture of three Gaussians with superimposed probability density functions of the Gaussians (a) and of the mixture (b). The blue curve represents the true underlying p.d.f., while the red curve was generated using the estimated parameters. Note that the curves overlap, indicating a very good fit.

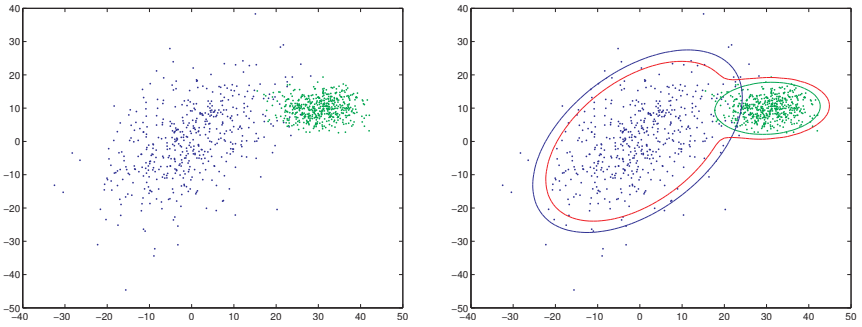
We generate samples from a random variable with identity covariance matrix. These samples are then linearly transformed to obtained the desired distribution. Note that the covariance matrices are the squares of the matrices used to multiply the original `xy` values.

```
xy = random( 'norm', zeros(2,n), ones(2,n) );
xy(:,1:n/2) = [2 10; 10 3] * xy(:,1:n/2);
xy(:,n/2+1:end) = [5 0; 0 3] * xy(:,n/2+1:end) + repmat([30 10]',1,n-n/2);
```

The EM algorithm `gaussianmixture` (p. 144) only needs about 30 iterations to converge and the results are reasonably good (again, up to a permutation).

```
[mu,C,w] = gaussianmixture( xy, 2 );

mu =        30.2204   -0.5529
             9.9383    0.0610

C(:,:,1) =  25.9354    1.3044
             1.3044   10.0829

C(:,:,2) = 102.9239   54.8432
            54.8432  125.7839

w' =         0.5053    0.4947
```

Figure 10.3a shows the generated point set. Points generated from the two Gaussian distributions are distinguished by color. Figure 10.3b adds contour levels for both Gaussian components and the mixture p.d.f. calculated from the parameters estimated by the EM algorithm. The contour level thresholds were chosen so that they delineate a 95% confidence region, i.e. 95% percent of the samples from the particular group are supposed to lie inside on the average. This turns out to correspond very well to reality in our case, there are 49 points outside the confidence region for the mixture p.d.f. (in red).



(a) (b)

Figure 10.3: (a) Scatter plot of samples generated from a weighted mixture of two Gaussians, with components distinguished by color. In (b) we also show 95% confidence regions calculated from the estimated parameters for both components (in blue and green) and the mixture (in red).

10.4 Point distribution models: `pointdistrmodel`

The point distribution model [Section 10.3] describes a family of shapes by their mean and a small number of eigenvectors. The shapes are represented by landmark coordinates on their contours. We show here how to automatically create the statistical description from a set of training examples [Algorithm 10.5].

function `[P,pmean,lambda] = pointdistrmodel(pts,alpha)`

input

`pts`	[2N×M]	A set of M training shapes. Each column corresponds to one shape and contains alternating x and y coordinates of N landmarks describing the shape: $\left[x_1, y_1, x_2, y_2, \ldots, x_N, y_N\right]$.
`alpha`	{0.95}	The constant $0 \leq \alpha \leq 1$ determines how much variation of the input data is captured by the reduced model [Section 10.3].

output

`P`	[2N×K]	The most important eigenvectors $\mathbf{P}$ of the model, corresponding to the K largest eigenvalues. The ordering of the eigenvectors in `P` corresponds to the ordering of the eigenvalues `lambda`.
`pmean`	[N×1]	The mean shape $\bar{\mathbf{p}}$.
`lambda`	[K×1]	K largest eigenvalues λ_i, sorted in decreasing order.

On return, the mean shape `pmean` is aligned with the first training shape `pts(:,1)`. Modified shapes can be obtained as $\bar{\mathbf{p}} + \mathbf{Pb}$ [Equation 10.5].
Start by aligning all other shapes with the first shape using function `pointalign` (p. 150). The mean `pmean` is calculated by averaging the transformed shapes.

```
[n,m] = size(pts);
for i = 2:m
  pts(:,i) = pointalign( pts(:,1), pts(:,i) );
end
pmean = mean( pts, 2 );
```

We iterate in the main `while`-loop until the change of the mean shape between iterations as measured by `r` (in pixels) becomes smaller than a predefined threshold. The convergence is fast, so a fixed threshold can be used.

```
r = inf;
while r<1e-6
```

In the main loop, we repeatedly align the mean shape `pmean` to the first shape `pts(:,1)`, align all other shapes to the mean shape `pmean`, and recalculate the mean.

```
  pmean = pointalign( pts(:,1), pmean );
  for i = 2:m
    pts(:,i) = pointalign( pmean, pts(:,i) );
  end

  oldmean = pmean;
  pmean = mean( pts, 2 );
  r = norm( (oldmean-pmean)/n );
end % while loop
```

The covariance matrix `S` is calculated from the differences from the mean shape `deltap`. We calculate its eigenvalues `lambda` and eigenvectors `P`.

```
deltap = pts - repmat(pmean,1,m);
S = cov(deltap');
[P,D] = eig(S);
lambda = diag(D);
```

Finally, we simplify the model by considering only the K largest eigenvalues, with the smallest K such that $\sum_{i=1}^{K} \lambda_i \geq \alpha \sum_{i=1}^{N} \lambda_i$. Note that the function :`eig` returns the eigenvalues in increasing order, so we need to consider the K last ones and reverse them.

```
limit = alpha*sum(lambda);
K = n;  partsum = lambda(K);
while K>1
  if partsum>limit, break; end
  K = K-1;  partsum = partsum+lambda(K);
end
lambda = lambda(end:-1:K);
P = P(:,end:-1:K);
```

Aligning the shapes

Suppose we have a moving shape and a reference shape described by landmark coordinates (x_i, y_i) and (x'_i, y'_i), respectively. We need to find a transformation consisting of rotation, translation, and scaling that transforms the moving shape onto the reference shape in the 'best' way [Section :10.3], defined as minimizing a sum of squared distances

$$E = \sum_{i=1}^{M} w_i \left\| s \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} - \begin{bmatrix} x'_i \\ y'_i \end{bmatrix} \right\|^2 \tag{10.2}$$

with parameters θ, s, t_x, t_y. We have included weights w_i into the formulation [Equation :10.2], however, this possibility is not used in our code.

We decompose the minimization of $E(\theta, s, t_x, t_y)$ to an outer minimization with respect to θ and inner minimization with respect to s, t_x, t_y. Minimization with respect to s, t_x, t_y is performed by setting the corresponding partial derivatives to zero

$$\frac{\partial E}{\partial t_x} = 0\,, \qquad \frac{\partial E}{\partial t_y} = 0\,, \qquad \frac{\partial E}{\partial s} = 0\,,$$

which leads to the following system of linear equations

$$\begin{aligned}
s\sum_{i=1}^{M} w_i\, q(\;\; y_i, -x_i, \theta) - N\, t_x &= -\sum_{i=1}^{M} w_i\, x'_i \\
s\sum_{i=1}^{M} w_i\, q(-x_i, -y_i, \theta) - N\, t_y &= -\sum_{i=1}^{M} w_i\, y'_i \\
s\sum_{i=1}^{M} w_i^2 \Big(q^2(y_i, -x_i, \theta) + q^2(x_i, y_i, \theta)\Big) - t_x \sum_{i=1}^{M} w_i\, q(y_i, -x_i, \theta) - t_y \sum_{i=1}^{M} w_i\, q(-x_i, -y_i, \theta) &= \\
-\sum_{i=1}^{M} w_i\, x'_i\, q(y_i, -x_i, \theta) + \sum_{i=1}^{M} w_i\, y'_i\, q(x_i, -y_i, \theta)\,,&
\end{aligned} \tag{10.3}$$

where $q(a, b, \theta) = a \sin\theta + b \cos\theta$. The dependency $E(\theta) = \min_{(s,t_x,t_y)} E(\theta, s, t_x, t_y)$ is non-linear, so the outer minimization with respect to θ is performed numerically. This normally only needs a few iterations, as the function is smooth and one dimensional.

function `[ptransf,theta,s,tx,ty] = pointalign(pref,p,theta0,w)`

input

`pref`	[2N×1]	The reference shape $[x'_1, y'_1, \ldots, x'_N, y'_N]$.
`p`	[2N×1]	The moving shape $[x_1, y_1, \ldots, x_N, y_N]$ to be aligned to the reference shape.
`theta0`	{0}	Initial guess of the angle θ, in radians.
`w`	[N×1]	Weights w_i, default to $w_i = 1$.

output

`ptransf`	[2N×1]	Transformed shape `p` aligned with the reference shape `pref` in the sense of criterion E (10.2).
`theta,s,tx,ty`	[1]	Parameters θ, s, t_x, and t_y of the optimal fit.

The minimization with respect to θ is performed by the function `fminunc`[1]. The criterion function $E(\theta)$ is evaluated by a function `crit` (below). Once the optimal θ is found, the transformed shape `ptransf` and the rest of the parameters are calculated using function `transf` (p. 151).

```
theta = fminunc( @(theta)(crit(p,pref,theta,w)), theta0, ...
                 optimset('Display','off','LargeScale','off'));
[E,ptransf,s,tx,ty] = transf( p, pref, theta, w );
```

[1] If this function is not available—it is part of the Matlab's Optimization Toolbox—:`fminsearch` will work equally well.

```
function E = crit(p,pref,theta,w)
```

Function `crit` is only a wrapper around `transf`.

```
[E,ptransf,s,tx,ty] = transf( p, pref, theta, w );
```

```
function [E,ptransf,s,tx,ty] = transf(p,pref,theta,w)
```

Function `transf` takes the moving and reference shapes `p` and `pref` and the parameter θ and weights w_i. It calculates optimal s, t_x, t_y from (10.3) and the transformed shape `ptransf`, and evaluates the criterion E (10.2).

We extract the x and y coordinates of the landmarks and precalculate $\sin\theta$, $\cos\theta$.

```
xy    = reshape( p,    2, [] );
xyref = reshape( pref, 2, [] );
n     = size( xy, 2 );
x    = xy(1,:);     y    = xy(2,:);
xref = xyref(1,:);  yref = xyref(2,:);
st   = sin(theta);  ct   = cos(theta);
```

Assemble and solve the linear system of equations (10.3) for unknowns s, t_x, t_y.

```
xw = x.*w';  yw = y.*w';
xrefw = xref.*w';  yrefw = yref.*w';
sx = sum(xw);  sy = sum(yw);
yst = st*sy;  xct = ct*sx;  xst = st*sx;  yct = ct*sy;
A = [yst-xct -n 0; -xst-yct 0 -n; ...
     sum((st.*yw-ct.*xw).^2+(st.*xw+ct.*yw).^2) -yst+xct xst+yct];
b = [-sum(xrefw) -sum(yrefw) dot(xrefw,-y*st+x*ct)+dot(yrefw,x*st+y*ct)]';
q = A\b;
s = q(1);  tx = q(2);  ty = q(3);
```

Transform the points using function `pointtransf` (below) and evaluate the criterion E.

```
ptransf = pointtransf( p, theta, s, tx, ty );
ptransf = reshape( ptransf, [], 1 );
E = sum( reshape(repmat(w',2,1),[],1).*(ptransf-pref).^2 );
```

```
function ptransf = pointtransf(p,theta,s,tx,ty)
```

Transform shape `p` according to parameters `theta`, `s`, `tx`, `ty` (see Equation 10.2).

```
xy = reshape( p, 2, [] );
n  = size( xy, 2 );
st = sin(theta);  ct = cos(theta);
ptransf = [s*ct (-s*st) tx; s*st s*ct ty] * [xy; ones(1,n)];
ptransf = reshape( ptransf, [], 1 );
```

Example

We use the hand point data made available by T. Cootes[2]. Figure 10.4a shows an example of one shape, drawn as follows:

```
xy = readpointfile( [dataDir 'hand.0.pts'] );
drawcontour(xy);
```

Functions `readpointfile` and `drawcontour` are given below.

We read all 18 datasets and store them into a matrix `pts`, each dataset to one column.

```
m = 18;
n = 2*size(xy,2);
pts = zeros(n,m);
for i = 1:m
  xy = readpointfile( [dataDir 'hand.' num2str(i-1) '.pts'] );
  pts(:,i) = xy(:);
end
```

Unaligned shapes 1 and 8 are shown in Figure 10.4b. Figure 10.4b shows the same shape after alignment using

```
ptransf = pointalign( pts(:,1), pts(:,8) );
```

Create the point distribution model. Setting $\alpha = 0.95$ (accounting for 95% of the variations) needs 5 eigenvectors.

```
alpha = 0.95;
[P,pmean,lambda] = pointdistrmodel( pts, alpha );
```

The mean shape and all aligned shapes are shown in Figure 10.5a. We can now show the principal modes of variation of the shape. Commands

```
p1 = pmean - 3*sqrt(lambda(i))*P(:,i);
p2 = pmean + 3*sqrt(lambda(i))*P(:,i);
```

calculate the extremal variations corresponding to $\pm 3\sigma$ for mode `i`. Figures 10.5bc illustrate these variations for the first principal modes corresponding to the two largest eigenvalues. Note that the first mode makes the fingers spread out, while the second mode makes them move right and left. Finally, we save the learned model for later analysis (Section 10.5).

```
save handpdm pmean P lambda
```

```
function xy = readpointfile(filename)
```

Read a file `filename` with point coordinates in the format used for T. Cootes' hand point data: the first line contains the number of points N, the remaining lines contain each two numbers corresponding to the x and y coordinates. It returns a $2 \times N$ array of point coordinates.

```
f  = fopen( filename );
n  = fscanf( f, '%d', 1 );
xy = fscanf( f, '%g', [2 n] );
```

[2] http://www.isbe.man.ac.uk/~bim/data/hand_data.html

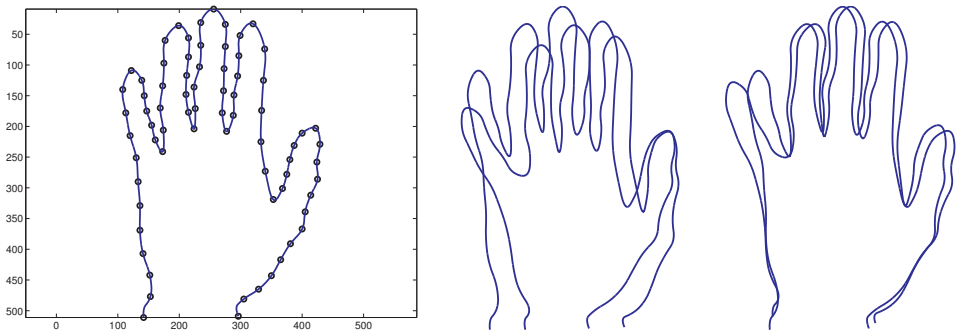


Figure 10.4: (a) Hand shape from dataset 1 with marked landmark points. Shapes 1 and (b) 8 before and (c) after alignment.

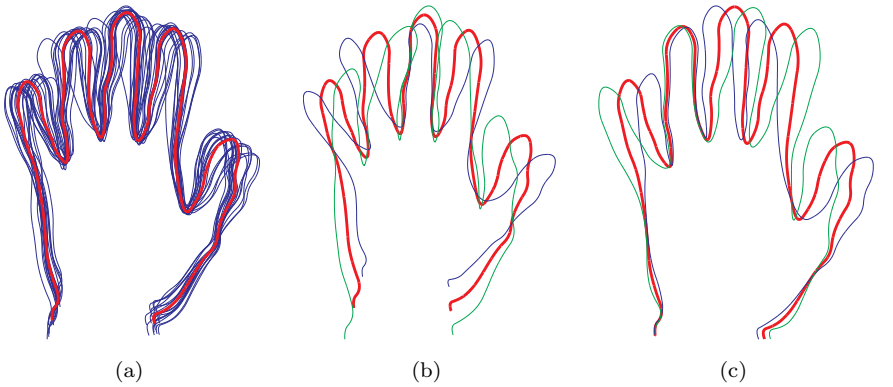


Figure 10.5: (a) The mean shape (in red) superimposed over all shapes after alignment. Changes corresponding to the (b) first and (c) second mode. The mean shape is in red, the shape corresponding to $-3\sqrt{\lambda}$ in blue and the shape corresponding to $+3\sqrt{\lambda}$ in green.

function `drawcontour(xy)`

Draws a smooth contour through given points. Matrix `xy` has size $2 \times N$ and each column determines one point.

This function uses B-spline interpolation `bsplineinterp` (p. 107). Note how the first and last points are duplicated to create neutral boundary conditions.

```
t = 2:0.01:size(xy,2) + 1;
degree = 2;
xt = bsplineinterp( [xy(1,1) xy(1,:) xy(1,end)], t, degree );
yt = bsplineinterp( [xy(2,1) xy(2,:) xy(2,end)], t, degree );
plot( xy(1,:),xy(2,:),'ko', xt,yt,'b-', 'LineWidth',2 );
```

10.5 Active shape model fit: `asmfit`

In Section 10.4, we have seen a function `pointdistrmodel` (p. 148) that creates a point distribution model (PDM) from a set of training shapes. Here we show how to fit the learned point distribution model to a given image [Algorithm :10.6]. It is a segmentation method similar to active contours such as snakes (Section 7.3).

The method is useful for images with pronounced edges that correspond well to the learned PDM. It is based on examining a narrow band around the current shape for improved landmark positions. The method is relatively fast but requires a good guess of the initial position.

The fitted shape is given by its pose parameters θ, s, t_x, t_y and shape parameters $\mathbf{b}$ [Equation, :10.1–10.5]

$$\mathbf{p} = s \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} (\mathbf{P}\mathbf{b} + \bar{\mathbf{p}}) + \begin{bmatrix} t_x \\ t_y \end{bmatrix} , \tag{10.4}$$

where the mean shape $\bar{\mathbf{p}}$ and principal eigenvectors $\mathbf{P}$ are provided by `pointdistrmodel`.

function `[p,theta,s,tx,ty,b] = asmfit(im,pmean,P,lambda,theta0,s0,tx0,ty0,width,rtol,display)`

input

`im`	[m×n]	Input image, normally an edge map. The higher the value, the larger the probability of a pixel being an edge. Zero value pixels are considered background.
`pmean`	[N×1]	The mean shape $\bar{\mathbf{p}}$, as returned by `pointdistrmodel`.
`P`	[2N×K]	Principal shape eigenvectors, as returned by `pointdistrmodel`.
`lambda`	[K×1]	Principal eigenvalues, as returned by `pointdistrmodel`.
`theta0`	[1]	Initial pose rotation parameter θ (10.4).
`s0`	[1]	Initial pose scaling parameter s (10.4).
`tx0,ty0`	[1]	Initial pose translation parameters t_x, t_y (10.4).
`width`	{10}	Width of the band around the current shape position to be searched in each iteration, in pixels.
`rtol`	{0.5}	Threshold on the maximum change of landmark coordinates to determine convergence.
`display`	{1}	Set to 1 for animation of the fitting procedure, 0 otherwise. The corresponding code is not included below.

output

`p`	[2N×1]	Final fitted shape $\mathbf{p} = [x_1, y_1, \ldots, x_N, y_N]$.
`theta`	[1]	Parameter θ of the final pose (10.4).
`s`	[1]	Parameter s of the final pose (10.4).
`tx,ty`	[1]	Parameters t_x, t_y of the final pose (10.4).
`b`	[K×1]	Final shape parameters $\mathbf{b}$ (10.4).

see also `pointdistrmodel` (p. 148).

Initialize pose parameters `theta`, `s`, `tx`, `ty` and shape parameters `b`. Variable `pold` is used in the stopping criterion and stores the previous value of `p`; `iter` is the iteration counter.

```
theta = theta0;  s=s0;  tx = tx0;  ty = ty0;
b = zeros( size(P,2), 1 );
[n,junk] = size(pmean);
pold = pmean;
iter = 1;
```

The main cycle is repeated until the change `r` of landmark positions between iterations decreases below the threshold `rtol`.

```
while true
  p = pointtransf( P*b+pmean, theta, s, tx, ty );
  r = max( abs(p-pold) );
  if (iter>1 && r<rtol) || iter>1000, break; end
  pold = p;
```

For each landmark we find a line (described by a vector `qx`, `qy`) normal to the shape contour at that point using function `perpendicular` (p. 156). We evaluate the edge map `im` for values `v` on the line `qx`, `qy` with step 1. The new landmark position `pnew` is the position of the maximum in `v`, unless the maximum is zero—in this case we are in a background region with no edges and the landmark is not moved. Note how the x, y coordinates need to be extracted from and stored into the 1D vectors `p`, `pnew`.

```
  t = -width:1:width;
  pnew = p;
  for i = 1:n/2
    x = p(2*i-1);  y = p(2*i);
    [qx,qy] = perpendicular( p, i );
    v = interp2( im, x+qx*t, y+qy*t, '*linear' );
    [maxv,j] = max(v);
    xn = x+qx*t(j);  yn = y+qy*t(j);
    if maxv>0
      pnew(2*i-1:2*i) = [xn yn];
    end
  end % for i
```

Once the new proposed landmark positions `pnew` are known, we call `pointalign` (p. 150) to find new pose parameters θ, s, t_x, t_y.

```
  [ptransf,theta,s,tx,ty] = pointalign( pnew, P*b+pmean, theta0 );
```

To find new shape parameters `b`, we first transform the landmark positions `pnew` into the original (canonical) coordinate space by applying an inverse transform using function `pointtransfinv` (p. 156). The difference from the mean shape `pmean` is then projected into the space spanned by the modes using the orthogonality of $\mathbf{P}$ [Algorithm 10.6].

```
  porig = pointtransfinv( pnew, theta, s, tx, ty );
  b = P'*(porig-pmean);
```

We repeat the whole loop until convergence.

```
  iter = iter+1;
end % while loop
```

```
function [qx,qy] = perpendicular(p,i)
```

Function `perpendicular` finds a unitary vector (`qx`, `qy`) perpendicular to the boundary described by points `p` at point number `i`. Indexes of neighbors used to calculate the normal direction—normally the left and right neighbors, except for the first and last points—are `im`, `ip`, with coordinates (`xm`, `ym`) and (`xp`, `yp`).

```
[n,junk] = size(p);
im = max( 1, i-1 );
ip = min( n/2, i+1 );
xm = p(2*im-1);  ym = p(2*im);
xp = p(2*ip-1);  yp = p(2*ip);
qx = yp-ym;
qy = xm-xp;
mag = sqrt( qx*qx + qy*qy );  % normalize length to 1
qx = qx/mag;  qy = qy/mag;
```

```
function ptransf = pointtransfinv(p,theta,s,tx,ty)
```

Inverse transformation to `pointtransf` (p. 151).

```
xy = reshape( p, 2, [] );
n = size(xy,2);
st = sin(-theta);  ct = cos(-theta);
rs = 1/s;
ptransf = [rs*ct (-rs*st); rs*st rs*ct] * ( xy-repmat([tx;ty],1,n) );
ptransf = reshape( ptransf, [], 1 );
```

Example

We read a previously learnt hand shape model (Section 10.4) and a hand image `im` (Figure 10.6a).

```
load handpdm
im = im2double( imread([ ImageDir 'hand.jpg']) );
```

The image is smoothed and a gradient magnitude image calculated in each color channel. The final edge map `g` is a maximum over the three color channels, thresholded to obtain a clean background.

```
h = fspecial( 'gaussian', 10, 1 );
g = zeros( size(im,1), size(im,2) );
for i = 1:3
  f = imfilter( im(:,:,i), h, 'symmetric' );
  [px,py] = gradient(f);
  g = max( g, sqrt(px.^2+py.^2) );
end
g = g .* (g>0.4*max(g(:)));
```

Figure 10.6b shows the shape model in the initial position superimposed over the inverted edge map g (black on white background). Initial pose parameters of the shape model were obtained manually.

```
s0 = 0.6;  theta0 = 0.0;  tx0 = 40;  ty0 = 50;
[p,theta,s,tx,ty,b] = asmfit( g, pmean, P, lambda, theta0, s0, tx0, ty0 );
```

Active shape model fitting takes 25 iterations. Figure 10.6c illustrates the first iteration, with normal search lines through each landmark and maxima (new proposed landmark positions) found. You can follow the fitting process in real-time by running asmfit (p. 154) with default parameters. The final position is shown in Figure 10.6d.

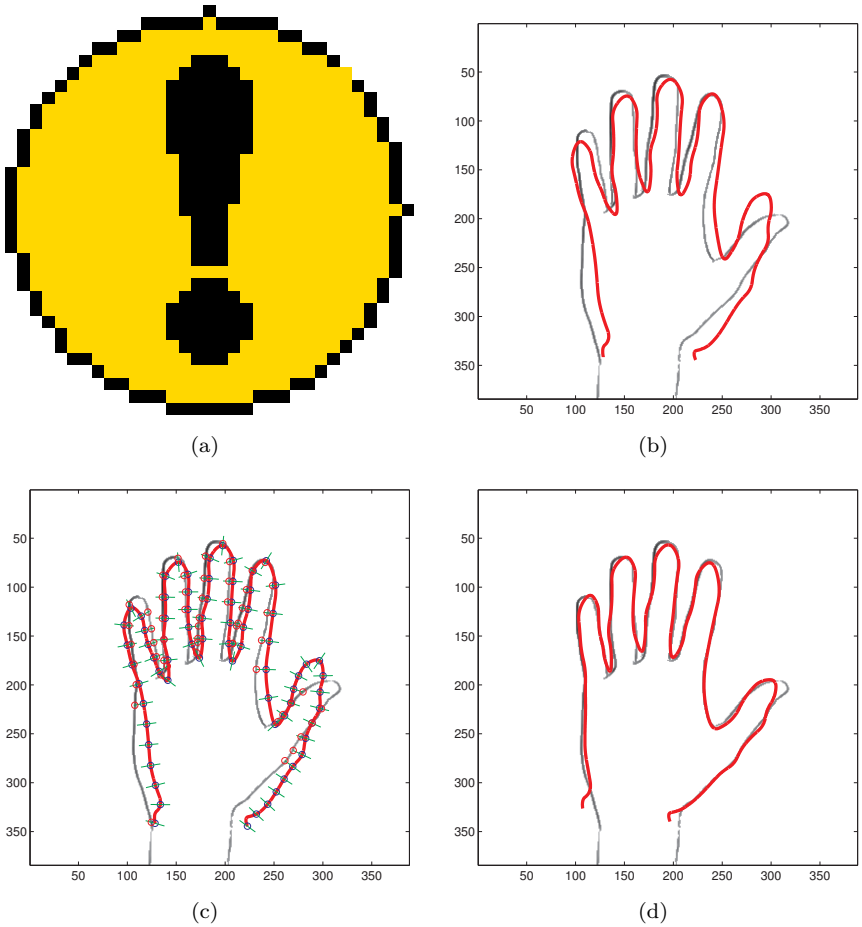


Figure 10.6: (a) Hand image. (b) The corresponding edge map with superimposed initial shape. (c) First iteration of the fitting process with the shape contour in red, current landmarks positions as blue circles, search lines in green, and new landmark positions as red circles. (d) Final fit.