Chapter 15

Texture

Texture is hard to define, yet ubiquitous in images of natural and many man-made objects. Classical texture descriptors are based on co-occurrence matrices (function `haralick`, p. 209). As an example, we show how to use them for texture classification. Another group of texture descriptors is based on discrete wavelet frames (function `waveletdescr`, p. 213). They can be computed very efficiently and their classification performance is often better than for co-occurrence based features. As well as for texture classification, we show how to use wavelet descriptors for (supervised) texture segmentation (function `texturesegm`, p. 217), by combining them with a graph cut method (Section 7.6).

Finally, we present the famous Lindenmayer or L-system (function `lsystem`, p. 220) as an example of a syntactic grammar-based shape and texture description and generation tool.

15.1 Problems

15.1. What is a texel?

15.2. Explain the difference between weak and strong textures.

15.3. How does scale influence texture description and perception? How does texture appear if the scale is too small or too large?

15.4. Define statistical and syntactic texture description methods. Describe texture types for which each approach is expected to work well and when it is likely to fail and explain why.

15.5. Define fractal dimension and lacunarity, and explain how these measures can be used for texture description.

15.6. Which texture description features are well suited for characterization of directional textures? Give several examples.

15.7. Can deterministic grammars be used to describe real-world textures? What are the main limitations?

15.8. Define a stochastic grammar. How is it useful for texture description?

15.9. What is primitive grouping? Why is it done? To which textures is this approach applicable? Give examples. How can it be used for texture description?

15.10. Describe how texture descriptors may be used in region growing segmentation.

15.11. Name several application areas for texture description and recognition.

15.12. Given the following texture features, determine whether their value is higher for fine textures than for coarse textures: (i) Energy in a small-radius circle of the Fourier power spectrum [Figure :15.3a], (ii) Average edge frequency feature [Equation :15.10] for a small value of d, (iii) Short primitive emphasis [Equation :15.12], (iv) Long primitive emphasis, (v) Fractal dimension, (vi) Lacunarity.

15.13. Determine the co-occurrence matrices $P_{0^\circ,2}$, $P_{45^\circ,2}$, and $P_{90^\circ,3}$ for the image in [Figure :15.4].

15.14. For a 30×30 checkerboard image with 3×3 binary checkers (values of 0 and 1, 0-level checker in the upper left corner), determine the average edge frequency function [Equation :15.10] for $d \in \{1, 2, 3, 4, 5\}$. For this image and also for a 30×30 image with vertical binary 3-pixel-wide stripes (0-level stripe along the left image edge), determine the following texture descriptors: (i) Short primitive emphasis, (ii) Long primitive emphasis, (iii) Gray-level uniformity, (iv) Primitive length uniformity, (v) Primitive percentage.

15.15. Design a shape chain grammar that generates the texture shown in [Figure :12.4]. Show the first few steps of the generation process.

15.16. Implement Fourier transform based descriptors [Figure :15.3]. Compare their classification performance with descriptors given by functions `haralick` (p. 209) and `waveletdescr` (p. 213) using the same data and classifier as in the example for `haralick`.

15.17. Implement Law's texture measures [Section :15.1.5]. Compare their classification performance as in Problem 15.16.

15.18. Extend the texture segmentation code (`texturesegm`, p. 217) to accept many training images instead of one. Study experimentally whether and how the segmentation accuracy increases with training. You might want to use a bigger texture database for a meaningful comparison.

15.19. Extend the wavelet descriptor calculation (`waveletdescr`, p. 213) to other wavelets besides Haar, such as Battle-Lemarié, Daubechies, B-spline and D-spline [Unser, 1995]. Compare their computational efficiency and classification accuracy. You might want to use a bigger texture database for a meaningful comparison.

15.20. Following Unser [Unser, 1995], extend the texture segmentation code (`texturesegm`, p. 217) to unsupervised texture segmentation. Compare segmentation accuracy with the supervised approach.

15.21. Implement texture segmentation as in function `texturesegm` (p. 217) using region merging (function `regmerge`, p. 75) instead of graph cut. Compare the speed and segmentation accuracy of the two approaches.

15.22. Extend the L-system code (function `lsystem`, p. 220) to draw lines in green as well as black. Design (or find on the Internet) an L-system that produces a stylized branch with black stems and green leaves.

15.2 Haralick texture descriptors: haralick

Classical texture descriptors based on co-occurrence matrices were introduced by Haralick [Section 15.1.2] and are useful mainly for texture classification. For each angle $\phi \in \{0°, 45°, 90°, 135°\}$ and distance $d \in \{1 \dots d_{\max}\}$ defining a co-occurrence matrix [Algorithm 4.1], we shall calculate five texture descriptors: energy, entropy, contrast, inverse difference moment (homogeneity), and correlation. The descriptors are defined by Equations 15.4–15.9, with $\kappa = 1$, $\lambda = 1$. The distance $d = 0$ is treated specially: it only makes sense to calculate energy and entropy in this case since the co-occurrence matrix is diagonal. The descriptors for all (ϕ, d) are concatenated into a feature vector.

function `h = haralick(im,maxdist)`

input

`im`	[m×n]	Input image of type `uint8`. It should contain a sufficiently large patch of homogeneous texture to analyze; a typical size might be 100×100 pixels, depending on resolution. Images must be of the same size for feature vectors to be comparable.
`maxdist`	{10}	Maximum distance $d_{\max}$ between pixels to consider—to be chosen depending on the characteristic scale of the texture. Increasing $d_{\max}$ increases computational complexity and the number of features generated.

output

`h`	[k×1]	Feature vector of length $k = 20\,d_{\max} + 2$, characterizing the input texture `im`.

see also `waveletdescr` (p. 213), `cooc` (p. 37).

The restriction to `uint8` is necessary for the co-occurrence matrix calculation in `cooc` (p. 37).

Generate the offset vector for a function `cooc` (p. 37) which is used for calculating the co-occurrence matrices, each row corresponds to one offset. We go over distances $d \in \{1 \dots d_{\max}\}$ and angles $\phi \in \{0°, 45°, 90°, 135°\}$. Note that only angles smaller than 180° need to be considered thanks to the symmetry of the co-occurrence matrix formulation used [Equation 15.3].

```
t = (1:maxdist)';
z = 0*t;
offs = [t z; z t; t t; -t t];
n = size( offs, 1 );
```

First the descriptors for $d = 0$ are evaluated using function `hfeatures` and only energy and entropy are retained. Then, the descriptors for all other offsets are evaluated and stored into the output vector `h`. Note that the asymmetric co-occurrence matrix returned by `cooc` is made symmetric to correspond to the definition [Equation 15.3].

```
h0 = hfeatures( cooc(im,[0 0]) );
h = [h0(1:2); zeros(5*n,1)];

for i = 1:n
  c = cooc(im,offs(i,:));  c = c+c';
  h(5*i-2:5*i+2) = hfeatures(c);
end
```

```
function f = hfeatures(c)
```

Given a symmetric co-occurrence matrix `c`, this function evaluates the five texture descriptors: energy, entropy, contrast, inverse difference moment (homogeneity), and correlation [Equations 15.4–15.9].

```
[nc,mc] = size(c);  [x,y] = meshgrid( 1:nc, 1:mc );
f0 =  sum(sum( c.^2 ));              % energy
f1 = -sum(sum( c.*log(c+eps) ));     % entropy
f2 =  sum(sum( abs(x-y).*c ));       % contrast
f3 =  sum(sum( c./(1+abs(x-y)) ));   % inverse difference moment
mx =  sum(sum( x.*c ));  my = sum(sum( y.*c ));
sx =  sum(sum( (x-mx).^2.*c ));
sy =  sum(sum( (y-my).^2.*c ));
f4 =  sum(sum( (x-mx).*(y-my).*c )) / sqrt(sx*sy); % correlation
f  = [f0 f1 f2 f3 f4]';
```

Example

We consider ten texture samples from the Brodatz collection[1] [Brodatz, 1966], see Figure 15.1. Each texture sample is converted to a grayscale `uint8` image and cut to 36 non-overlapping patches 100×100 pixels. Each patch is assigned a label indicating its class $1 \ldots 10$.

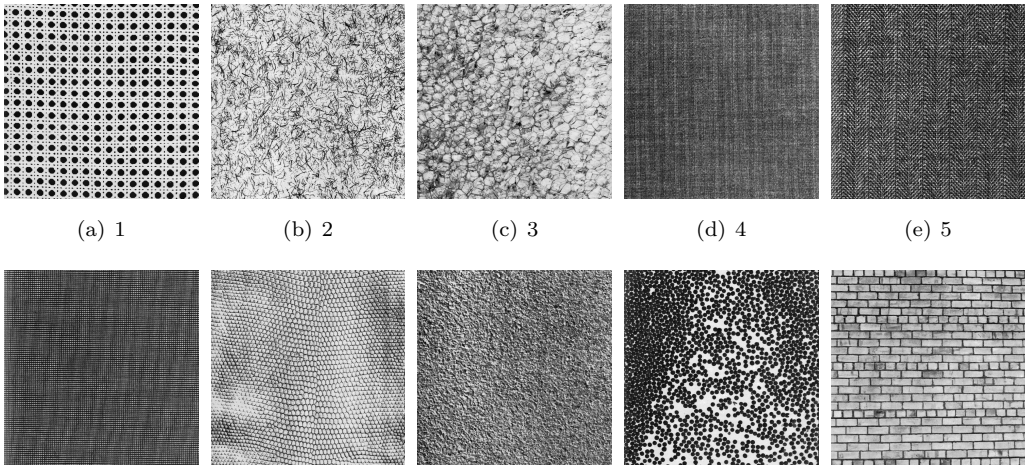
```
files = {'D101' 'D110' 'D112' 'D16' 'D17' 'D21' 'D3' 'D4' 'D67' 'D95'};
patchsize = 100;
textures = struct([]);
ind = 1;
for i = 1:size(files,2)
  fn = files{i};
  img = im2uint8( imread([ImageDir fn '.png']) );
  if size(img,3)>1, img = rgb2gray(img); end
  [ny,nx] = size(img);
  for ox = 1:patchsize-1:nx-patchsize+1
    for oy = 1:patchsize-1:ny-patchsize+1
      textures(ind).patch = img( oy:oy+patchsize-1, ox:ox+patchsize-1 );
      textures(ind).class = i;
      ind = ind+1;
    end
  end
end
```

The samples are randomly permuted and divided into training and testing data. Note that because the permutation is global, the number of training examples per class may not be the same for all classes. This might lead to slightly suboptimal classification results but corresponds to a realistic scenario.

```
n = ind-1;
textures = textures( randperm(n) );
```

[1] Available in digital form for example from http://sipi.usc.edu/database.

```
textures_train = textures( 1:n/2 );
textures_test = textures( n/2+1:end );
```



(a) 1 (b) 2 (c) 3 (d) 4 (e) 5

(f) 6 (g) 7 (h) 8 (i) 9 (j) 10

Figure 15.1: Ten selected textures with their class labels.

Haralick texture descriptors are calculated for each patch in the training and test sets and stored into a structure suitable for use by the STPRtool [Franc and Hlavac, 2004], function `mlcgmm`, later on.

```
ntrain = size( textures_train, 2 );
h0 = haralick( textures_train(1).patch );
n = size(h0,1);
features_train.X = zeros( n, ntrain );
features_train.y = zeros( 1, ntrain );
features_train.X(:,1) = h0;
features_train.y(1) = textures_train(1).class;
for i = 2:ntrain
  features_train.X(:,i) = haralick( textures_train(i).patch );
  features_train.y(i) = textures_train(i).class;
end

ntest = size(textures_test,2);
features_test.X = zeros(n,ntest);
features_test.y = zeros(1,ntest);

for i = 1:ntest
  features_test.X(:,i) = haralick( textures_test(i).patch );
  features_test.y(i) = textures_test(i).class;
end
```

As the descriptor calculation may take a few minutes, for convenience the precomputed descriptors can be saved and later restored.

```
save features features_test features_train ntrain ntest
load features
```

The training data is normalized so that all features have zero mean and unit standard deviation. This improves numerical stability of the subsequent steps.

```
m = mean( features_train.X, 2 );
X = features_train.X - repmat(m,1,ntrain);
v = sqrt( var(X') )';
features_train.X = X ./ repmat(v,1,ntrain);
```

There are 202 features, which is too many given that we only have 18 training patches (on the average) per class. We choose the 10 most relevant ones using function `goodfeatures` (p. 213).

```
ind = goodfeatures( features_train, 10 );
features_train.X = features_train.X(ind,:);
```

We use the maximum probability normal classifier, function `maxnormalclass` (p. 123) and determine the parameters of the Gaussian distributions for each class using STPRtool function `mlcgmm` [Franc and Hlavac, 2004]. Because of the relative scarcity of training data, we make the additional assumption of diagonality of the class covariance matrices. This completes the training phase.

```
model = mlcgmm( features_train, 'diag' );
```

To classify the test data, we normalize them using the parameters `m` and `v` determined from the training data. We also select the previously determined subset of 'good' features `ind`. The classification itself is performed by function `maxnormalclass` which provides a vector `ytest` with class labels for each test sample.

```
features_test.X = (features_test.X-repmat(m,1,ntest)) ./ repmat(v,1,ntest);
features_test.X = features_test.X(ind,:);
ytest = maxnormalclass( features_test.X, model );
```

The classification results are quite good, given the simple classifier and limited amount of training data. The overall classification accuracy (the ratio of correctly classified patches with respect to all test patches) is 94%. The results can also be presented as a confusion

	1	2	3	4	5	6	7	8	9	10
1	21	0	0	0	0	0	0	0	2	0
2	0	16	0	0	0	0	4	0	0	0
3	0	0	11	0	0	0	0	0	0	0
4	0	0	0	18	0	0	0	0	0	0
5	0	0	0	0	12	0	0	0	0	0
6	0	0	0	0	0	15	0	0	0	0
7	0	1	0	0	0	0	16	0	0	0
8	0	0	0	0	0	0	0	22	0	0
9	0	0	0	0	0	0	2	0	17	0
10	0	0	4	0	0	0	0	0	0	19

Table 15.1: Haralick texture classifier confusion matrix.

matrix, see Table 15.1. The number in row i and column j is the number of patches from class j classified as i. (Ideally, all off-diagonal elements should be zero.)

function `ind=goodfeatures(data,n)`

Function `goodfeatures` takes `data`, which is a structure with fields `X` (feature vectors) and `y` (labels), as expected by `mlcgmm` [Franc and Hlavac, 2004], and chooses n 'best' ones. The feature vectors are assumed normalized, i.e., each feature should have zero mean and unit variance.

Feature selection is a difficult problem. For simplicity, we use the ratio of intra-class and inter-class variance. Since we have normalized for inter-class (total) variance, we can calculate the total intra-class variance and choose n features with the smallest intra-class variance.

```
k = max( data.y );
sumv = zeros( size(data.X,1), 1 );
for y = 1:k
  ind = (data.y==y);
  sumv = sumv + var(data.X(:,ind)')';
end
[junk,ind] = sort( sumv );
ind = ind(1:n);
```

15.3 Wavelet texture descriptors: `waveletdescr`

Discrete wavelet frame texture descriptors [Unser, 1995], [Section 15.1.7] are an alternative to classical texture descriptors, based for example on co-occurrence matrices (Section 15.2). Discrete wavelet frames can be calculated fast using a filter-bank and the descriptors perform well for many applications.

For simplicity and computational efficiency, we shall use the Haar wavelet with a low-pass filter $H(z) = (1 + z)/2$ and a corresponding high-pass filter $G(z) = (z - 1)/2$.

function `v = waveletdescr(im,maxlevel)`

input

`im`	[m×n]	Input image. It should contain a sufficiently large patch of homogeneous texture to analyze. A typical size might be 100 × 100 pixels, depending on the resolution. It is recommended that images be of the same size and the same amplitude for the feature vectors to be comparable.
maxlevel	{3}	The number of multiresolution levels, chosen depending on the characteristic scale of the texture; the largest filters will have size $2^{\texttt{maxlevel}}$. Increasing `maxlevel` increases computational complexity and the number of features generated.

output

`v`	[k×1]	Feature vector of length $k = 3\,\texttt{maxlevel} + 1$, characterizing the input texture `im`.

see also `haralick` (p. 209).

Unlike for `haralick`, input images are not restricted to be of type `uint8`. Here the image is first converted to `double`, to avoid overflow problems. However, note that an optimized, all integer, implementation would be straightforward.

```
im = double(im);
[m,n] = size(im);
npix = m*n;
```

The main loop is repeated `maxlevel` times. At each level, we filter the input image `im` to provide four sub-bands by using the following filter combinations: H_xH_y, H_xG_y, G_xH_y, G_xG_y, where H_x is the low-pass filter applied along the x direction, G_y is the high-pass filter applied along the y direction etc. Note that thanks to separability, only six 1D filtering operations are required, implemented by functions `filterh` and `filterg`, below. The low-pass version (filtered by H_xH_y) is used as an input to the subsequent scale and the filter size `l` is doubled.

The features are the energies in the three high-pass (detail) sub-bands for each level. At the last level, the energy of the low-pass band is also added to the output feature vector `v`.

```
v = zeros( 3*maxlevel+1, 1 ); % the descriptors
for i = 1:maxlevel
  l = 2^i;
  % filtering in the y direction
  imhy = filterh( im, l );
  imgy = filterg( im, l );
  % filtering in the x direction
  vgg = sum(sum( filterg(imgy',l).^2 )) / npix;
  vhg = sum(sum( filterh(imgy',l).^2 )) / npix;
  vgh = sum(sum( filterg(imhy',l).^2 )) / npix;
  im = filterh( imhy', l )';
  v(3*i-2:3*i) = [vgg vhg vgh];
end

v(end) = sum(sum( im.^2 )) / npix; % low-pass band energy
```

Unlike a standard discrete wavelet transform, we are using a wavelet frame, so no subsampling of the filtered images is performed.

```
function imf = filterh(im,l)
```

Function `filterh` filters all columns of the image `im` by a low-pass Haar filter $H(z) = (1+z^l)/2$. Since the filter has only two non-zero elements, the filtering amounts to adding together two shifted copies of the image. Mirror boundary conditions are ensured by extending the image by `l` rows.

```
  imf = 0.5*[im; im(end-1:-1:end-l,:)];
  imf = imf(1:end-l,:) + imf(l+1:end,:);
```

```
function imf = filterg(im,l)
```

Function `filterg` works like `filterh` except the high-pass filter $G(z) = (z^l - 1)/2$ is used.

```
imf = 0.5*[im; im(end-1:-1:end-l,:)];
imf = imf(l+1:end,:) - imf(1:end-l,:);
```

Example

We compare the classification performance of wavelet descriptors `waveletdescr` (p. 213) with co-occurrence matrix based descriptors generated by `haralick` (p. 209). We use the same set of textures (Figure 15.1), the same set of training and testing patches, and the same classifier. The source code from Section 15.2 can be re-used, with all occurrences of `haralick` replaced by `waveletdescr`. The only difference is that for wavelet descriptors no feature selection is needed, since with the default settings (`maxlevel=3`) only 10 features are generated.

Notice that calculating wavelet descriptors is many times faster than calculating the descriptors generated by `haralick`. Also the performance is better, with an overall classification accuracy 97%, see Table 15.2.

	1	2	3	4	5	6	7	8	9	10
1	20	0	0	0	0	0	0	0	1	0
2	0	17	0	0	0	0	0	0	0	0
3	0	0	15	0	0	0	0	0	0	0
4	0	0	0	18	0	0	0	0	0	0
5	0	0	0	0	12	0	0	0	0	0
6	0	0	0	0	0	15	0	0	0	0
7	0	0	0	0	0	0	22	0	0	0
8	0	0	0	0	0	0	0	22	3	0
9	1	0	0	0	0	0	0	0	15	0
10	0	0	0	0	0	0	0	0	0	19

Table 15.2: Wavelet confusion matrix.

15.4 Texture based segmentation: `texturesegm`

Texture descriptors can also be used for segmentation of images consisting of several different textures. Here we follow Unser [Unser, 1995] and show how wavelet texture descriptors (Section 15.3) can be used for this purpose. We proceed in three steps: (i) We create a function `waveletsegdescr` which is derived from `waveletdescr` (p. 213) but instead of calculating the descriptors globally for the whole image, it calculates them for each pixel. (ii) Function `texturesegmtrain` (p. 217) takes a training image with a known segmentation and creates a model of the classes (textures) in the image. (iii) Finally, function `texturesegm` (p. 217) takes an unknown image and segments it

using the learnt model. Graph cut segmentation (Section 7.6) is used to obtain spatially coherent segmentation.

function `v = waveletsegdescr(im,maxlevel,sigma)`

input

`im`	[m×n]	Input image.
maxlevel	{3}	The number of multiresolution levels, see `waveletdescr`.
sigma	{10}	Standard deviation of the Gaussian filter used for descriptor averaging, in pixels. Large values result in more reliable classification at the expense of suppressing small details.

output

`v`	[k×m×n]	A matrix of feature vectors of length $k = 3\,$`maxlevel` $+ 1$ for each pixel.

see also `waveletdescr` (p. 213).

Since this function is so similar to `waveletdescr`, we will only comment on the differences here.

```
im = double(im);
[m,n] = size(im);
npix = m*n;

v = zeros( 3*maxlevel+1, m, n ); % an array to store the descriptors
```

Prepare the filter `h` for descriptor averaging. Note that `h` is unitary (has a unit gain). The energy in all bands is low-pass filtered by `h`, with symmetric boundary conditions.

```
h = fspecial( 'gaussian', ceil(3*sigma), sigma );

for i = 1:maxlevel
  l = 2^i;
  % filtering in the y direction
  imhy = filterh( im, l );
  imgy = filterg( im, l );
  % filtering in the x direction
  v(3*i-2,:,:) = imfilter( filterg(imgy',l)'.^2, h, 'symmetric' );
  v(3*i-1,:,:) = imfilter( filterh(imgy',l)'.^2, h, 'symmetric' );
  v(3*i,:,:) = imfilter( filterg(imhy',l)'.^2, h, 'symmetric' );
  im = filterh( imhy', l )';
end

v(end,:,:) = imfilter( im.^2, h, 'symmetric' ); % low-pass band energy
```

We have chosen the feature index to be the first index to `v` (instead of the last) even though it is less efficient here because it is more appropriate for subsequent processing in `texturesegmtrain` and `texturesegm`.

function `model = texturesegmtrain(im,mask,maxlevel,sigma)`

input

`im`	[m×n]	Input image.
`mask`	[m×n]	Segmentation for the image `im`. The numbers in `mask` denote the class for the corresponding pixel in `im` and should be from the range $1 \dots d$ where d is the number of classes.
maxlevel	{3}	The number of multiresolution levels, see `waveletdescr`.
sigma	{10}	Standard deviation of the Gaussian filter used for descriptor averaging, in pixels, see `waveletsegdescr`.

output

`model`	struct	Model of the texture classes to be used by `texturesegm`.

see also `texturesegm` (p. 217), `waveletdescr` (p. 213), `waveletsegdescr` (p. 216).

First, the texture descriptors `f` are calculated (for each pixel). The probability distribution of these descriptors for each class is assumed to be normal. Their parameters are estimated using STPRtool function `mlcgmm` [Franc and Hlavac, 2004], as in Section 9.2 or Section 15.2.

```
f = waveletsegdescr( im, maxlevel, sigma );
[k,m,n] = size(f);
features.X = reshape( f, k, m*n );
features.y = reshape( mask, 1, m*n );

model = mlcgmm( features, 'diag' );
model.maxlevel = maxlevel;
model.sigma = sigma;
```

function `texturesegm(im,model,regul)`

input

`im`	[m×n]	Input image to be segmented.
`model`	struct	Model of the texture classes as returned by `texturesegmtrain`.
regul	{200}	Regularization for the GraphCut segmentation algorithm, penalizing different class labels for neighborhood pixels. Increasing this parameter eliminates small regions but may decrease accuracy.

output

`l`	[m×n]	output labeling. Each pixel position contains an integer $1 \dots d$ corresponding to an assigned class; d is the number of classes.

see also `waveletdescr` (p. 213), `texturesegmtrain` (p. 217), `waveletsegdescr` (p. 216).

Texture descriptors `f` are calculated for each pixel of the image and the probability `p` of a pixel belonging to a particular class is evaluated using STPRtool function `pdfgauss` [Franc and Hlavac, 2004], see also Section 9.2.

```
f = waveletsegdescr( im, model.maxlevel, model.sigma );
[k,m,n] = size(f);
p = pdfgauss( reshape(f,k,m*n), model );
d = size(p,1);
```

The logarithm of `p` is used as the data term for the graph cut segmentation (Section 7.6).

```
Dc = -log( reshape(p',m,n,d)+eps );
Sc = regul*( ones(d)-eye(d) );
handle  = GraphCut( 'open', Dc, Sc );
[gch l] = GraphCut( 'expand', handle );
handle  = GraphCut( 'close', handle );
l = l+1; % as GraphCut classes start at 0
```

Example

We shall use four textures from Figure 15.1.

```
t1 = im2double( imread([ImageDir 'D112.png']) );
t2 = im2double( imread([ImageDir 'D17.png']) );
t3 = im2double( imread([ImageDir 'D4.png']) );
t4 = im2double( imread([ImageDir 'D95.png']) );
```

The training image consists of four vertical stripes, each containing one kind of texture. Figure 15.2a and 15.2d show the test image `imtrain` and the corresponding mask (`maskt`).

```
[m,n] = size(t1);
[x,y] = meshgrid( 1:n, 1:m );
maskt = 1 + floor( 4*(x-1)/m );
imtrain = t1.*(maskt==1) + t2.*(maskt==2) + t3.*(maskt==3) + t4.*(maskt==4);
```

The model is learnt from the training image.

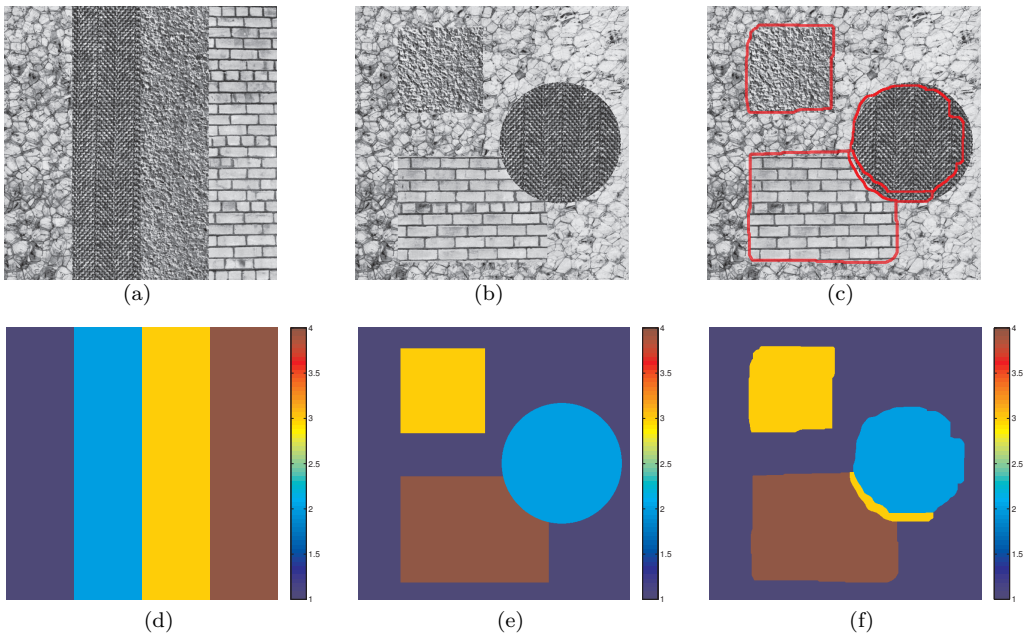


Figure 15.2: (a,d) Training image with the corresponding mask. (b,e) Test image with the corresponding mask. (c,f) Segmentation results as boundaries superimposed over the test image and as a mask.

```
model = texturesegmtrain( imtrain, maskt );
```

The test image **imtest** is constructed from the same textures but the mask (**mask**) is more complicated, Figure 15.2be.

```
mask1 = ( ((x-0.75*n).^2 + (y-0.5*m).^2) < 20000 );
mask2 = ( x>100 & x<300 & y>50 & y<250 );
mask3 = ( x>100 & x<450 & y>350 & y<600) & not(mask1);
mask  = 1 + mask1 + 2*mask2 + 3*mask3;
imtest = t1.*(mask==1) + t2.*(mask==2) + t3.*(mask==3) + t4.*(mask==4);
```

The texture segmentation algorithm is applied with default regularization.

```
l = texturesegm( imtest, model );
```

The final segmentation can be seen in Figure 15.2cf as boundaries superimposed over the test image and as a mask. Most of the texture regions were identified correctly.

15.5 L-system interpreter: `lsystem`

An L-system (Lindenmayer system) [Prusinkiewicz and Lindenmayer, 1990] is an example of a syntactic shape and texture description technique [Section 15.2]. It is mostly based on a recursive, context-free, deterministic grammar [Section 9.4.1] although context and stochastic versions also exist. The distinguishing feature of an L-system is that at each iteration: (i) all applicable rules are applied, and (ii) all rules are applied simultaneously, in parallel. For example, given a starting symbol `F` and a rule `F` $\longrightarrow$ `F+F-F+F`, the first two iterations are:

```
0:                          F
1:                      F+F-F+F
2:    F+F-F+F+F+F-F+F-F+F-F+F+F+F-F+F
...
```

The expansion is stopped after a predetermined number of iterations and the resulting string is interpreted, one character at a time. We implement symbols given in the following table—it is a simplified version of the interpretation used in [Prusinkiewicz and Lindenmayer, 1990] or by L-system interpreter programs such as Fractint[2].

Symbol	Action
`F`	Draw a line of a unit length in a current direction.
`f`	Move forward by a unit length in a current direction.
`+`	Turn left by angle $\Delta\phi$.
`-`	Turn right by angle $\Delta\phi$.
`[`	Remember the current state (position and direction).
`]`	Retrieve the remembered position.
other	No action (ignored).

[2] http://spanky.triumf.ca/www/fractint/fractint.html

The implementation given here draws directly into a current Matlab figure, which can be saved to a file using :`print`. The drawing starts at point $(0, 0)$ and the initial orientation is along the positive x-axis.

function `lsystem(s,rules,angle,n)`

input

`s`	string	The start symbol (axiom).
`rules`	struct	The grammar rules. A string `rules(i).left` contains the left side of a rule i, which must be a single letter symbol. A string `rules(i).right` contains the right side of a rule i. No two rules may have the same left side.
`angle`	[1]	Angle increment $\Delta\phi$ in radians.
`n`	[1]	Number of iterations. As most L-systems increase the string length exponentially, the number of iterations will rarely exceed 10.

output

There are no output parameters.

We iterate `n`-times over the grammar production rules, expanding the string `s`. In each iteration we go over the current string `s` from the left, character by character. For each character, all rules are considered. If a rule matches, its expansion is appended to the output string `os`. If no rule matches, the input character is copied to the output string unchanged.

```
for i = 1:n
  os = [];
  for j = 1:length(s)
    subst = false;              % a flag - has any rule matched?
    for k = 1:length(rules)
      if s(j)==rules(k).left    % rule matches
        os = [os rules(k).right];
        subst = true;  break;
      end
    end
    if not(subst)               % no rule matched so far
       os = [os s(j)];
    end
  end % for j loop
  s = os;
end % for i loop
```

The expanded string `s` is now interpreted. The current position and orientation is stored in `x`, `y`, and `d`, while `l` contains the length of a unit step. Operation '`[`' stores the current state into a `stack` and '`]`' retrieves it.

```
stackpos  = 1;                  % index to the stack
x = 0;
y = 0;
d = 0;
l = 1;
clf                             % start with a clean figure
```

```
for i = 1:length(s)
  cmd = s(i);
  switch( cmd )
    case 'F'
      x1 = l*cos(d) + x;
      y1 = l*sin(d) + y;
      line( [x x1], [y y1], 'Color','k', 'LineWidth',2 ); % draw
      x = x1;  y = y1;
    case 'f'
      x = l*cos(d) + x;
      y = l*sin(d) + y;
   case '+'
      d = d+angle;
   case '-'
      d = d-angle;
   case '['
      stack(stackpos).x = x;
      stack(stackpos).y = y;
      stack(stackpos).d = d;
      stackpos = stackpos+1;
   case ']'
      if stackpos<2, error('lsystem: Stack empty.'); end
      stackpos = stackpos-1;
      x = stack(stackpos).x;
      y = stack(stackpos).y;
      d = stack(stackpos).d;
  end
end
```

Example

Many different shapes and textures can be generated using L-systems. Here we present a few:[3]

Koch snowflake (Figure 15.3a):

```
rules(1).left = 'F';
rules(1).right = 'F+F--F+F';
lsystem( 'F--F--F', rules, pi/3, 3 );
```

Sierpinski triangle/gasket (Figure 15.3b):

```
rules(1).left = 'F';
rules(1).right = 'F+F-F-F+F';
lsystem( 'F', rules, 2/3*pi, 5 );
```

Rectangular grid (Figure 15.3c):

```
rules(1).left = 'F';
rules(1).right = 'F[+F][-F]F';
lsystem( 'F', rules, pi/2, 5 );
```

[3]Mostly taken from the Fractint tutorial http://spanky.triumf.ca/www/fractint/lsys/tutor.html.

Triangular grid with irregular borders (Figure 15.3d):

```
rules(1).left = 'X';
rules(1).right = 'FY[+FY][--FY]FY';
rules(2).left = 'Y';
rules(2).right = 'FX[++FX][-FX]FX';
rules(3).left = 'F';
rules(3).right = '';
lsystem( 'X', rules, pi/3, 4 );
```

Hexagonal grid (Figure 15.3e):

```
rules(1).left = 'F';
rules(1).right = '-F+F+[+F+F]-';
lsystem( 'F', rules, pi/3, 5 );
```

Hilbert space-filling curve (Figure 15.3f):

```
rules(1).left = 'L';
rules(1).right = '+RF-LFL-FR+';
rules(2).left = 'R';
rules(2).right = '-LF+RFR+FL-';
lsystem( 'L', rules, pi/2, 5 );
```

The most famous examples are *plant-like drawings*, such as branches (Figure 15.3g):

```
rules(1).left = 'F';
rules(1).right = 'FF';
rules(2).left = 'X';
rules(2).right = 'F[+X]F[-X]+X';
lsystem( 'X', rules, pi/9, 5 );
```

or *a bush* (Figure 15.3h):

```
rules(1).left = 'F';
rules(1).right = 'FF-[-F+F+F]+[+F-F-F]';
lsystem( '++++F', rules, pi/8, 4 );
```

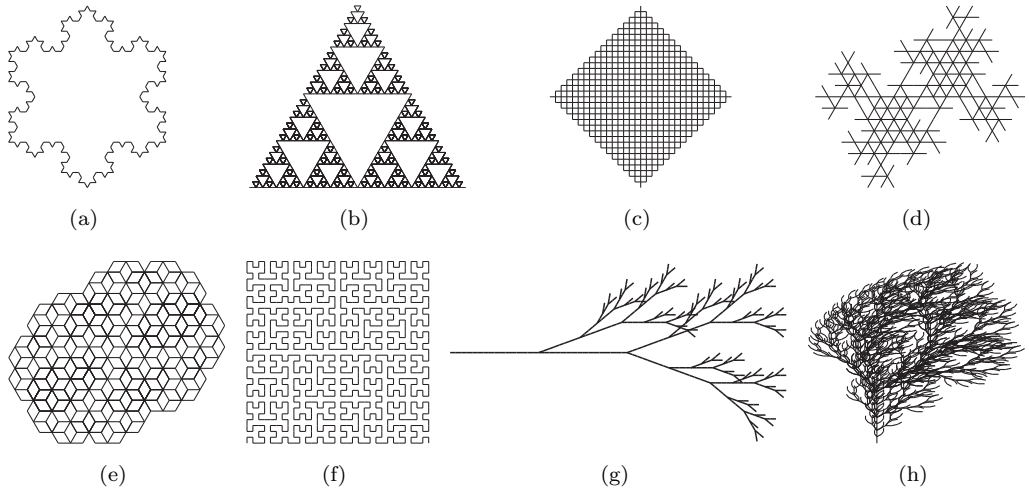


Figure 15.3: Drawings generated by L-systems: (a) Koch snowflake, (b) Sierpinski triangle, (c) rectangular grid, (d) triangular grid, (e) hexagonal grid, (f) Hilbert curve, (g) branch, and (h) bush.