Image segmentation using $K$-means and EM

1. Introduction

The purpose of this tutorial is (a) to start work on the processing of digital images; (b) to progress as far as the classification of grey levels and colour; and (c) to see how successful this turns out to be for initiating segmentation. First, it is necessary to carry out basic imaging operations such as loading and saving images, converting images from colour to greyscale, and computing intensity histograms of greyscale images. As the chapter progresses, we will find how to perform $K$-means cluster analysis for grey levels and colour, and will show that this is valuable for image segmentation. Moving on to use of the EM algorithm for performing cluster analysis, we find that greyscale processing becomes more rigorous and more successful, though here we eschew the far more computationally intensive task of colour segmentation. Then we move on to the use of the EM algorithm for fitting Gaussians to 2-D scatter plots—a process that meets with a solid degree of success. Overall, we will have achieved the fitting of mixtures of Gaussians to both 1-D and 2-D datasets with the aid of the EM algorithm.

2. Some basic image processing routines

In this section we start by introducing a number of basic imaging functions and routines—in particular those for loading and saving images, converting them from colour to greyscale, and computing the intensity histograms of greyscale images. Particular functions to look out for are `imread`, `imwrite`, and `rgb2gray`. These are to be found in the following extract from a larger Matlab algorithm, together with a short piece of code designed to compute the intensity histogram of a greyscale image.

```matlab
frame=imread('road.bmp');
grey=rgb2gray(frame);
imwrite(grey,'roadgrey.bmp'); % save greyscale version of input image
seq=reshape(grey,1,320*240); % save greyscale version of input image
hh=zeros(1,256);
hh=uint16(hh);
intens=uint8(0);
for i=1:320*240
    intens=seq(i) + 1; % histogram index needs to be 1-256, not 0-255
    hh(intens)=hh(intens)+1;
end
... % indicates continuation to the next segment of the algorithm
```

First, the original and derived greyscale images are shown in Figure 1. Next, notice that the greyscale image is sequenced into a 1-D array called seq: this makes it straightforward to form an intensity histogram without having to use a double for loop to analyse the image. Notice also that whereas an intensity histogram would normally be expected to have index values ranging from 0 to 255, in Matlab it is necessary to use the index range 1–256. The histogram is computed by the standard procedure of accumulating pixel intensity values in the various histogram bins.
The next piece of code uses the intensity histogram already obtained to segment the grayscale image using the $K$-means algorithm. However, the initial intensity histogram is formulated using 16-bit unsigned integers ($hh$): here we proceed by converting it to double ($dhh$) to ensure that mean values can be computed with sufficient precision. It is also necessary to prime the algorithm using initial approximations for the $K$-means. In the case presented below, $K = 6$; 10 iterations are applied in order to achieve sufficient accuracy. But first, recall from Davies, Chapter 13 (Table 13.2), that the $K$-means algorithm has two main passes: in the first pass, we test each data point to see which cluster centre it is closest to, and use it to recalculate the position of that cluster centre; in the second pass, we assign each data point to the closest of the new cluster centre positions.

```matlab
% segmentation of a greyscale image using the K-means algorithm

dhh = double(hh);
K = 6;
mm = [10 20 60 160 220 250]; % starting values
imax = 10;

sq=zeros(1,K);
summ=zeros(1,K); num=zeros(1,K);
MM=zeros(1,imax);
for iter=1:imax % pass 1
    summ(:)=0; num(:)=0; sumsq=0;
    for j=1:256
        sq(:) = (j-mm(:)).^2;
        [mink,kk] = min(sq); % find minimum value and index
        summ(kk) = summ(kk)+dhh(j)*j;
        num(kk) = num(kk)+dhh(j);
        sumsq = sumsq+dhh(j)*mink;
    end
    mm = summ ./ num;
    MM(iter) = sqrt(sumsq); % save error information
end % pass 1

% assignment of classes to histogram bins
classk=zeros(1,256);
for j=1:256 % pass 2
    sq(:) = (j-mm(:)).^2;
    [mink,kk] = min(sq); % find index of minimum value
    classk(j)=kk; % assign class kk to bin j
end % pass 2
... % indicates continuation
```
Before moving on, notice that the `min` function was used not only for finding the minimum value of its argument, but also for finding the relevant index. Specifically, we used the following code:

```matlab
sq(:) = (j-mm(:)).^2;
[mink,kk] = min(sq); % find minimum value and index
summ(kk) = summ(kk)+dhh(j)*j;
```

The alternative would have been to use a fuller construct, such as:

```matlab
mink=300^2;
for k=1:K
    sq(k) = (j-mm(k)).^2;
    if sq(k)<mink
        mink = sq(k);
        kk=k;
    end
end
```
In fact, it is normally beneficial to use available Matlab functions such as \texttt{min}, as they tend to be significantly more general and to have more options; in addition, they are likely to be better tested and more highly optimised for speed and vectorisation.

It now remains to classify the individual pixels and convert the sequenced image into a normal 2-D image (Figure 2(a)) and also to display the convergence plot (Figure 2(b)). The latter confirms that convergence has been taken as far as necessary, while the vertical (cyan) lines on the histogram plot (Figure 2(c)) show that the \textit{K}-means algorithm has located the bumps in the distribution as well as can be expected for a such a rudimentary (i.e., non-model driven) procedure.

We now move on to the remainder of the algorithm, which gave rise to all three of the above figures. Note that the \texttt{saveas} function is used for saving plotted figures, and that saving eps versions of colour plots requires the 'epsc' descriptor.

```matlab
seq2=zeros(1,320*240);
seq2=uint8(seq2);
for i=1:320*240 % final assignment of intensities to image points
    j=seq(i) + 1;
    kk=classk(j);
    intens=mm(kk) - 1;
    seq2(i)=intens;
end
KMout=reshape(seq2,240,320); % re-form output image
figure; % show and save output image
imwrite(KMout);
figure;
plot(j-1,dhh/(320*240),'-r');
pbaspect([2 1 1]);
axis([0 255 0 0.025]);
saveas(gcf,'roadhist.tif')
saveas(gcf,'roadhist','epsc') % save eps version of color plot
figure;
plot(i,MM,'r');
pbaspect([2 1 1]);
axis([1 10 0 7000]);
saveas(gcf,'roadconv.tif')
saveas(gcf,'roadconv','epsc') % save eps version of color plot
```

```matlab
% plot histogram
figure;
set(gca,'fontsize',11);
box on; hold on;
for i=1:K
    line([mm(i)-1,mm(i)-1],[0,0.025],'color','c','linewidth',1.0)
end
j=1:256; % use j as a plot variable
plot(j-1,dhh/(320*240),'-r');
pbaspect([2 1 1]);
axis([0 255 0 0.025]);
saveas(gcf,'roadhist.tif')
saveas(gcf,'roadhist','epsc') % save eps version of color plot

% plot convergence
figure;
set(gca,'fontsize',11);
set(gca,'XTick',([ 1 2 3 4 5 6 7 8 9 10 ]))
set(gca,'YTick',([ 0 1000 2000 3000 4000 5000 6000 7000 ]))
grid on; box on; hold on
i=1:imax;
plot(i,MM,'r');
pbaspect([2 1 1]);
axis([1 10 0 7000]);
saveas(gcf,'roadconv.tif')
saveas(gcf,'roadconv','epsc') % save eps version of color plot
```
3. **K-means: progressing from a simple greyscale algorithm to colour**

It is instructive to compare the K-means algorithm presented above with a more direct version, which does not first compute the intensity histogram. As will be seen, the direct version is significantly shorter and easier to understand. However, its main loops are all executed \(320 \times 240\) times, which means that the algorithm runs about 300 times more slowly than the histogram-based algorithm. Actually, the histogram core of the histogram-based algorithm is K-means in its own right, and the most computationally expensive part of that algorithm is the subsequent conversion to the image representation.

```matlab
% K-means for road picture - greyscale
frame = imread('road.bmp');
grey = rgb2gray(frame);
imwrite(grey, 'roadgrey.bmp');

seq = reshape(grey, 1, 320*240);
tens = uint8(0);
seq2 = zeros(1, 320*240);
seq2 = uint8(seq2);
K = 6;
mm = [10 20 60 160 220 250]; % starting values
imax = 10;

sq = zeros(1, K);
summ = zeros(1, K);
um = zeros(1, K);
for iter = 1:imax % pass 1
    summ(:, 0) = 0;
    num(:, 0) = 0;
    for i = 1:320*240
        intens = seq(i);
        sq(:, i) = (double(intens) - mm(:, :)).^2;
        [mink, kk] = min(sq(:, i)); % find index of minimum value
        summ(kk) = summ(kk) + double(intens);
        num(kk) = num(kk) + 1;
    end
    mm = summ ./ num;
end % pass 1

for i = 1:320*240 % pass 2
    intens = seq(i);
    sq(:, i) = (double(intens) - mm(:, :)).^2;
    [mink, kk] = min(sq(:, i)); % find index of minimum value
    seq2(i) = mm(kk);
end % pass 2

KMout = reshape(seq2, 240, 320);
figure;
imshow(KMout);
imwrite(KMout, 'roadclasses.bmp');
```

We now move on to the use of K-means for vastly reducing the colour depth of colour images—namely from a full complement of \(256^3\), or even 256, to just a handful of colours. To achieve this we merely generalise the above algorithm to give the RGB version listed below.
% K-means for road picture - colour

frame=imread('road.bmp');
red=frame(:,:,1); green=frame(:,:,2); blue=frame(:,:,3);

seqr=reshape(red,1,320*240);
seqg=reshape(green,1,320*240);
seqb=reshape(blue,1,320*240);
intensr=uint8(0); intensg=uint8(0); intensb=uint8(0);
seq2r=zeros(1,320*240); seq2g=zeros(1,320*240); seq2b=zeros(1,320*240);
K = 8;
mnr=zeros(1,K); mnmg=zeros(1,K); mmmb=zeros(1,K);
% image sampling points (u,v)
u=[30 30 30 160 160 160 230 230];
v=[30 130 230 60 130 230 30 130];
for k=1:K
    mnr(k)=frame(u(k),v(k),1); % starting values
    mnmg(k)=frame(u(k),v(k),2);
    mmmb(k)=frame(u(k),v(k),3);
end
sq=zeros(1,K)
summr=zeros(1,K); summg=zeros(1,K); summb=zeros(1,K); num=zeros(1,K);
for i=1:320*240 % pass 1
    intensr = seqr(i); intensg = seqg(i); intensb = seqb(i);
    sq(i) = (double(intensr)-mnr(:)).^2 + ...
    (double(intensg)-mnmg(:)).^2 + (double(intensb)-mmmb(:)).^2;
    [mink,kk] = min(sq); % find index of minimum value
    summr(kk) = summr(kk) + double(intensr);
    summg(kk) = summg(kk) + double(intensg);
    summb(kk) = summb(kk) + double(intensb);
    num(kk) = num(kk) + 1;
end % pass 1
mmr = summr ./ num; mnmg = summg ./ num; mmmb = summb ./ num;
for i=1:320*240 % pass 2
    intensr = seqr(i); intensg = seqg(i); intensb = seqb(i);
    sq() = (double(intensr)-mmr(:)).^2 + ...
    (double(intensg)-mnmg(:)).^2 + (double(intensb)-mmmb(:)).^2;
    [mink,kk] = min(sq); % find index of minimum value
    seq2r(i) = mmr(kk); seq2g(i) = mnmg(kk); seq2b(i) = mmmb(kk);
end % pass 2
KMout(:,:,1)=reshape(seq2r,240,320); % red;
KMout(:,:,2)=reshape(seq2g,240,320); % green;
KMout(:,:,3)=reshape(seq2b,240,320); % blue;
figure;
imshow(KMout);
imwrite(KMout,'roadclasses.bmp');

The result of running this algorithm is seen in Figure 3.
Figure 3

4. EM algorithm for greyscale images

We now move on to the use of the EM algorithm, as described in Davies, Chapter 14. We prepare the ground exactly as for the histogram-based $K$-means algorithm, described earlier. Specifically, we convert the input colour image to greyscale and then compute its intensity histogram. To avoid divide-by-zero errors, it turns out to be easier to eliminate any empty bins at the low intensity end of the histogram: this is conveniently achieved by applying the `find` function:

```matlab
% EM algorithm for road picture - using greyscale only
frame=imread('road.bmp');
grey=rgb2gray(frame);
imwrite(grey,'roadgrey.bmp');

seq=reshape(grey,1,320*240);
hh=zeros(1,256);
hh=uint16(hh);
intens=uint8(0);
for i=1:320*240
    intens=seq(i) + 1;
    hh(intens)=hh(intens)+1;
end
x=1:256;
dhh_org=double(hh);
Pmin=1;
while hh(Pmin)==0, Pmin=Pmin+1; end
found = find(hh==0);
hh(found) = [];
x(found) = [];
P=length(hh); % use instead of 256
if P Pmin
    x=x';
end
...```

(Note that here and in what follows, '...' is used to show continuation of the algorithm—i.e., the next piece of code needs to follow without a break.)

Next, we provide initialisation data for the algorithm:
Before proceeding further, we show how the theory for the EM algorithm (Davies, Chapter 14, equations (14.22)–(14.24)) is modified for use in Matlab:

<table>
<thead>
<tr>
<th>theory</th>
<th>algorithm notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$-step</td>
<td>$\rho(z_a) = \frac{m_i \mathcal{N}(x, \mu_k, \Sigma_k)}{\sum_{i=1}^{N} m_i \mathcal{N}(x, \mu_k, \Sigma_k)}$</td>
</tr>
<tr>
<td></td>
<td>$\mu_k' = \frac{\sum_{i=1}^{N} \rho(z_a) x_i}{\sum_{i=1}^{N} \rho(z_a)}$</td>
</tr>
<tr>
<td>$M$-step</td>
<td>$\Sigma_k' = \frac{\sum_{i=1}^{N} \rho(z_a) (x_i - \mu_k) (x_i - \mu_k)^T}{\sum_{i=1}^{N} \rho(z_a)}$</td>
</tr>
<tr>
<td></td>
<td>$m_k' = \frac{1}{N} \sum_{i=1}^{N} \rho(z_a)$</td>
</tr>
<tr>
<td>log likelihood</td>
<td>$L' = \sum_{i=1}^{N} \ln \sum_{k=1}^{K} m_k \mathcal{N}(x, \mu_k', \Sigma_k')$</td>
</tr>
</tbody>
</table>

We should also recall that $\sum_{k=1}^{K} m_k' = 1$, which makes $N = \sum_{i=1}^{N} \sum_{k=1}^{K} \rho(z_{ik})$.

In the theory, the data points run over the range 1 to $N$, whereas use of the intensity histogram means that they now run over the range 1 to $P$. Therefore, all instances of $\sum_{i=1}^{N} \rho(z_{ik})$ need to be replaced by $\sum_{i=1}^{P} \text{dhh}(i)$, as $\text{dhh}(i)$ is itself a sum over all the pixels having intensity $i$. This results in a total of three instances of multiplication by $\text{dhh}$:
% EM algorithm - main loop

iter=1;
while iter<=imax

% E-step
w = zeros(P,K);
r = zeros(P,K); rx = zeros(P,K); rdd = zeros(P,K);
for k = 1:K
  w(:,k) = mixture(k) * normpdf(x(:),mu(k),sigma_k(k));
  w(:,k) = w(:,k) .* dhh(:);
end
% find responsibilities
k_sum_w = sum(w,2);
for i = 1:P
  r(i,:) = w(i,:) ./ k_sum_w(i);
end
% M-step
for i = 1:P
  r(i,:) = r(i,:) .* dhh(i);
end
for i = 1:P
  rx(i,:) = r(i,:) .* x(i);
end
i_sum_r = sum(r,1);
ik_sum_r = sum(sum(r,1),2);
i_sum_rx = sum(rx,1);
for k = 1:K

  % find new mixtures
  mixture(k) = i_sum_r(k) / ik_sum_r;
  % find new means
  mu(k) = i_sum_rx(k) ./ i_sum_r(k);
  % find new sigmas
  for i = 1:P
    dev = x(i) - mu(k);
    devsq = dev^2;
    rdd(i,:) = r(i,:) * devsq;
  end
  i_sum_rdd = sum(rdd,1);
  vari_k(k) = i_sum_rdd(k) ./ i_sum_r(k);
  sigma_k(k) = sqrt(vari_k(k));
end

% find log likelihood
for k = 1:K
  w(:,k) = mixture(k) * normpdf(x(:),mu(k),sigma_k(k));
end
k_sum_w = sum(w,2); % sum over k
W(:) = dhh(:) .* log(k_sum_w(:));
L = sum(W); % sum over i=1:P
LL(iter) = L - old_L; % change in L
old_L = L;

% find sum of absolute errors
diff=k_sum_w'-dhh/(320*240);
DD(iter)=sum(abs(diff));
iter=iter+1;
end % iteration loop
...

The rest of the code relates to (a) finding the borderlines between the component Gaussians (Figure 4(a)), so that the most appropriate intensities can be applied in the
final classified image; (b) plotting the intensity histogram with these and other markings; (c) presenting the convergence plots (Figure 4(b)); and (d) showing the output image with its classified intensities (Figure 4(c)).

```matlab
% find borderlines between Gaussians
border=zeros(1,K-1);
x=0;
for k=1:K-1
    y1=0; y2=-1;
    while y2<y1
        y1=mixture(k) * normpdf(x,mu(k)-1,sigma_k(k));
        kk=k+1;
        y2=mixture(kk) * normpdf(x,mu(kk)-1,sigma_k(kk));
        x=x+1;
    end
    border(k)=x-1;
end

% plot histogram
figure;
set(gca,'fontsize',11);
box on; hold on
j=1:256;
plot(j-1,dhh_org/(320*240),'g');
pbaspect([2 1 1]);
for k=1:K-1
    line([border(k)-1,border(k)-1],[0,0.025],color,'c',...
     'linewidth',0.5)
end
for k = 1:K
    y = mixture(k) * normpdf(j,mu(k)-1,sigma_k(k));
    plot(j-1,y,'r');
end
y = 0;
for k = 1:K
    y = y + mixture(k) * normpdf(j,mu(k)-1,sigma_k(k));
end
plot(j-1,y,'b');
axis([0 255 0 0.025]);
saveas(gcf,'roadhist.tif')
saveas(gcf,'roadhist','epsc')

% plot convergence
figure;
set(gca,'fontsize',11);
grid on; box on; hold on
ax=zeros(1,imax);
i=1:imax;
plot(i,ax,'b');
plot(i,LL,'r');
plot(i,(DD-DD(imax))*1800,'m'); % adjusted to be seen on the same scale
pbaspect([5 3 1]);
axis([4 imax -4 12]);
saveas(gcf,'roadconv.tif')
saveas(gcf,'roadconv','epsc')
```

It turns out that the convergence plot $LL$ representing the changes in $L$ goes down steadily but increasingly slowly, with no discernible minimum (red plot). For a definite optimum we consider whether the sum of absolute errors $DD$ might be a better indicator (magenta plot). In fact, $DD$ has no minimum right up to 100 iterations.
(note that in order to facilitate comparison between the two curves, what is displayed is not $DD$ itself but the function $[DD - DD(100)] \times 1800$: so $DD$ itself does not go down to zero after 100 iterations). Thus, in this case, the sum of absolute errors does not provide a better test of optimality than the log likelihood. Furthermore, the latter is particularly useful for checking when a certain level of precision of $L$ has been reached. Note that the iterations are controlled by a `while` loop rather than a `for` loop, so that termination can be decided by applying a suitable level of precision.

Figure 4. (a) Intensity histogram showing borderlines between the component Gaussians. (b) Convergence plots. The $LL$ (changes in $L$) plot is shown in red and closely resembles an exponential decay curve: the $DD$ (sum of absolute errors) plot is shown in magenta and does not provide a better test of optimality (see text). (c) Output image showing classified intensities.
5. EM algorithm for fitting 2-D Gaussian mixture data

In this section, we extend the use of the EM algorithm from 1-D to 2-D. The basic problems are not identical, as the 1-D work was aimed at fitting 1-D Gaussians to greyscale histograms, whereas here we aim to fit 2-D Gaussians to scatter plots obtained from three 2-D Gaussians. Though quite short, the Matlab code given below shows several important things: how random scatter plots can be derived from sets of 2-D Gaussians; how the resulting scatter plots can be fitted by the EM algorithm; how contour plots can be drawn using parameters obtained when fitting the data; how the subplot function can be used to present the gradually improving fit, at the same time using the position `get` and `set` functions to maintain acceptable spacings between the subplots; and how a convergence graph can be drawn for the EM algorithm. The resulting scatter and convergence plots (Figure 5) correspond roughly to Figures 14.3 and 14.4 in Davies (2017), Chapter 14—though the input data are significantly different and the fits were produced by different versions of the EM algorithm, each with its own programming options: here, `fitgmdist`, the standard Matlab provision, was employed.

In fact, the algorithm was made somewhat more complicated by iterating the EM routine in a single set of 60 passes and presenting the (input) scatter and (output) contour plots after every 10 of the 60 passes, which required 6 conditional statements.

To obtain the initial random scatter data, the Matlab `mvnrnd` function was employed, `mvnrnd` being fully multivariate in having general non-diagonal covariance matrices. Notice that if the convergence plot is to be computed coherently, the same seed has to be invoked for each iteration of the EM algorithm: this is achieved by applying the `rng` function with its 'default' setting. Next, the `meshgrid` function is used to pass the 2-D pattern of probability data to the contour plot via the `mvnpdf` function. (Again, this function is used as it is fully multivariate in having general covariance matrices.)

Interestingly, in this case the subplot option creates an overall figure that requires significantly more than the usual resolution. Unfortunately, the `saveas` operation cannot respond properly to this, so a `print` operation is needed in which the output resolution can be controlled—here, so as to give a full 300 dpi. The value of the print function is that it is able to use an appropriate rendering operation when generating the final image (Figure 5(a)).
% apply EM algorithm to scatter plots representing sets of 2-D Gaussians

figure;
axis([-6,6,-5,5]);
imax = 20;
LL=zeros(imax);
for i=1:imax
    iter(i)=i*2;
    % generate 2-D dataset
    mu1=[1 -2.5]; sigma1=[2 0; 0 .4];
    mu2=[2 1]; sigma2=[0.5 0; 0 1.5];
    mu3=[-2 1]; sigma3=[1 -0.5; -0.5 1];
    mu4=[-3 0]; sigma4=[0.09 0; 0 0.09];
    rng('default'); % ensures same random seed every time
    x = [mvnrnd(mu1,sigma1,200); mvnrnd(mu2,sigma2,200); ... 
         mvnrnd(mu3,sigma3,200); mvnrnd(mu4,sigma4,80)];
    options = statset('Display','off','MaxIter',iter(i));
    gm = fitgmdist(x,4,'Options',options); % fit 4 2D Gaussians
    mixture = gm.PComponents; mu = gm.mu; sigma = gm.Sigma;
    L = gm.NegativeLogLikelihood;
    LL(i)=L;
    % present the mixture of Gaussians using meshgrid
    [UU,VV] = meshgrid(-10:0.01:10,-10:0.01:10);
    u = UU(:); v = VV(:); s = size(UU,1);
    pdf = mvnpdf([u v],mu(1,:),sigma(:,:,1));
    gauss1 = reshape(pdf,s,s);
    pdf = mvnpdf([u v],mu(2,:),sigma(:,:,2));
    gauss2 = reshape(pdf,s,s);
    pdf = mvnpdf([u v],mu(3,:),sigma(:,:,3));
    gauss3 = reshape(pdf,s,s);
    pdf = mvnpdf([u v],mu(4,:),sigma(:,:,4));
    gauss4 = reshape(pdf,s,s);
    mix = mixture(1)*gauss1 + mixture(2)*gauss2 + mixture(3)*gauss3 ... 
         + mixture(4)*gauss4;
    % produce the contour plot
    if rem(iter(i),6)==0
        if iter(i)==6, rr=1; pp= 0.1; qq=0; elseif iter(i)==12, rr=2; pp=-0.1; qq=0; 
            elseif iter(i)==18, rr=3; pp= 0.1; qq=0.06; elseif iter(i)==24, rr=4; pp=-0.1; qq=0.06; 
            elseif iter(i)==30, rr=5; pp= 0.1; qq=0.12; elseif iter(i)==36, rr=6; pp=-0.1; qq=0.12; 
        end
gca=subplot(3,2,rr);
p=get(gca,'position');
p(1)=p(1)+pp; p(2)=p(2)+qq;
set(gca,'position',p);
scatter(gca,x(:,1),x(:,2),3,'s','filled','b'); hold on
contour(gca,UU,VV,mix,[0.005 0.01 0.02 0.03 0.04 ... 
                       0.05 0.07 0.1 0.15 0.25 0.40],','r');
axis([-6,6,-5,5]); pbaspect([4,3,1]); axis off
end
end % for

Finally, the convergence plot obtains data from the negative log likelihood output of the \texttt{fitgmdist} function, which is recorded in array \textit{LL}.
saveas(gcf,'EMquadruple.tif');
% saveas(gcf,'EMquadruple','epsc'); % gives too low a resolution
print('EMquadruple','-depsc2','-r300') % -r is followed by no. of dpi
% print -depsc2 -tiff -r300 -painters 'EMquadruple.eps' % alternative

% drawing a convergence plot for the EM algorithm
figure;
i=1:imax;
plot(iter(i),LL(i),'b'); grid on
saveas(gcf,'EMquadrupleconv.tif')
saveas(gcf,'EMquadrupleconv','epsc')

Figure 5. (a) 'Quadruple Gaussian' data with fitted contours. (b) convergence plot.
6. Initialisation of the data

In the interests of clarity, one essential factor was ignored in the algorithm descriptions given above. This was the extent to which learning depends on the order in which data is presented to the algorithm: this is especially important when the algorithm is initialised. In fact, poor data initialisation can result in significant bias in the results produced by a learning algorithm. However, total removal of bias cannot be guaranteed, and if a learning algorithm does not give the expected performance, it may well be better to re-apply it with a new randomisation of the data. (Needless to say, these remarks apply for both $K$-means and EM algorithms.)

One way to achieve this is to initialise the algorithm by selecting random sets of data points to provide initial approximations to class mean values $\mu_i$. In the case of the EM algorithm, mixture values are normally sufficiently randomised if they are distributed equally between the classes; and it is generally sufficient to initialise covariances as diagonal matrices with variances equal to those of the global data distribution.

To select suitable data points during initialisation, the `randperm` function is valuable. In fact, while `randperm(n)` returns a row vector containing a random permutation of the integers 1 to $n$, more valuable is the `randperm(n, K)` version of the function which returns a row vector containing $K$ unique integers selected randomly from 1 to $n$. In our case, the $K$ unique integers will be the specific data points that need to be assigned to the $K$ available classes during initialisation. Note that two randomisations are introduced by `randperm`: one is the random selection of $K$ data points, and the other is the random order in which they are applied.

Fortunately, `fitgmdist`, the Matlab EM algorithm function, includes this sort of randomisation, though it was only used in our scatterplot fitting algorithm. The three algorithms that used $K$-means and the algorithm that used the EM algorithm had ad hoc initialisations that were not rigorously randomised. Perhaps oddly, this did not seem to matter. In fact, in the histogram cases, selecting a row of suitable positions to use for initialisation was not especially exacting. This also applied for the relatively few image points used to obtain starting colour values. Nevertheless, it would be worthwhile to study the extent to which the final results converge on the same solutions if different initialisations are tried. We leave this as an exercise for the reader.

7. Index of useful Matlab functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>contour, fitgmdist, get, imread, imwrite, meshgrid, min, mvnpdf</td>
<td>12</td>
</tr>
<tr>
<td>mvnrnd, print, randperm, rgb2gray, rng, saveas, scatter, set</td>
<td>12</td>
</tr>
</tbody>
</table>