FSMEM for MoG

Daniel Wagenaar, June 2000

FSMEM, or *free split/merge expectation maximization*, is a modification of the SMEM algorithm presented by Ueda et al in [1]. Unlike SMEM, FSMEM can modify the number of clusters dynamically. To compare likelihoods across solutions with different number of clusters, a minimal description length term is used.

I will be using the FSMEM algorithm exclusively for optimizing mixture of Gaussians, although the algorithm can be applied equally to other clustering problems. The following symbols will be used:

K refers to the number of clusters. k is a dummy index enumerating clusters.

N refers to the number of data point. n enumerates data points.

D refers to the dimensionality of the space.

Vectors in data space will be denoted $\vec{x} = (x^1, \ldots, x^D)$. Inner product notation will be used whenever convenient:

$$\vec{x} \cdot \Sigma \vec{x} \equiv \sum_{d_1, d_2} x^{d_1} \Sigma_{d_1 d_2} x^{d_2}.$$

1 Mixture of Gaussians and EM

Mixture of Gaussians models a probability density function of the form

$$p(\vec{x}) = \sum_k \pi_k p(\vec{x}; \Theta_k),$$

where π_k are non-negative and sum to one, and $\Theta_k = \{\vec{\mu}_k, \Sigma_k\}$ are the parameters for a Gaussian:

$$p(\vec{x}; \Theta_k) = \frac{1}{\sqrt{(2\pi)^D \det \Sigma}} e^{-\frac{1}{2}\vec{x} \cdot \Sigma^{-1} \vec{x}}$$

The standard EM learning algorithm for MoG consists of iterating the following steps:

1. Calculate responsibilities:
$$R_{kn} = \frac{\pi_k p(\vec{x}; \Theta_k)}{\sum_{k'} \pi_{k'} p(\vec{x}; \Theta_{k'})}$$
2. Update parameters:
$$\vec{\mu}_k' = \frac{\sum_n R_{kn} \vec{x}_n}{\sum_n R_{kn}}$$
$$\Sigma_k' = \frac{\sum_n R_{kn} R_{kn} (\vec{x}_n - \mu_k')(\vec{x}_n - \mu_k')^T}{\sum_n R_{kn}}$$
$$\pi_k = \sum_n R_{kn}.$$

These steps are to be iterated until the log likelihood $L = \sum_n p(\vec{x}_n)$ converges.

2 SMEM by Ueda et al

This is the original SMEM algorithm by Ueda et al:

Initialize parameters as for EM.
Run EM until convergence.
repeat
  Collect a list of merge candidates: pairs of clusters with their merge merits, and a list of split candidates: clusters with their split merits.
  Sort these lists, then
  for all promising triplets $\{k_{m_1}, k_{m_2}; k_s\}$ **do**
    Perform the split and merge operations.
    Reinitialize the affected clusters.
    Run partial EM on the affected clusters until convergence.
    Run full EM on all clusters until convergence.
    if the likelihood is better than before the merge/split **then**
      Ignore the other candidates, and go back to the candidate collection step.
    else
      Restore parameters.
until no candidate produced better result than old one

The split candidates are ordered by their *split merit*:

$$J_k^{(\text{split})} = \int d\vec{x} f_k(x) \log \frac{f_k(x)}{p(\vec{x}_n; \Theta_k)},$$

where

$$f_k(x) = \frac{\sum_n \delta(\vec{x} - \vec{x}_n) R_{kn}}{\sum_n R_{kn}}.$$

$J^{(\text{split})}$ is a measure of the KL distance between the local observed distribution $f_k(\vec{x})$ and the model distribution $p(\vec{x}; \Theta_k)$.

Similarly, merge candidates are ordered by their merge merit:

$$J_{kk'}^{(\text{merge})} = \sum_n R_{kn} R_{k'n}.$$

Note that split and merge merits are measured on different scales, so it is not possible to sort split and merge candidates together. Ueda et al state that it is only necessary to test about 5 promising candidates at each iteration, but they do not specify how they weigh split and merge merits together.

Initialization of parameters after a merge is as follows:

$$\begin{aligned}
\vec{\mu}' &= \frac{1}{2}(\vec{\mu}_1 + \vec{\mu}_2) \\
\Sigma' &= \frac{1}{2}(\Sigma_1 + \Sigma_2) \\
\pi' &= \pi_1 + \pi_2.
\end{aligned}$$

Note that no attempt is made to weight the parameters.

Initialization of parameters after a split is as follows:

$$\vec{\mu}'_{1,2} = \vec{\mu} + \vec{v}_{1,2},$$

where $\vec{v}_k$ is a small noise term. I choose to pick $\vec{v}$ from $\mathcal{G}(0, \Sigma)$.

$$\Sigma'_{1,2} = \sigma^2 \mathrm{I},$$

where σ^2 is the largest eigenvalue of Σ. Ueda et al use $\sigma^2 = (\det \Sigma)^{1/D}$, but I had problems with that approach in cases where Σ possessed some very small eigenvalues.

$$\pi_{1,2} = \frac{1}{2}\pi.$$

Partial EM is defined as EM acting on a subset K' of clusters. The equations are the same as for EM, except

$$R_{kn} = \frac{\pi_k p(\vec{x}_n; \Theta_k)}{\sum_{k' \in K'} \pi_{k'} p(\vec{x}_n; \Theta_{k'})} R^{(0)}_{kn},$$

where $R^{(0)}_{kn}$ are the responsibilities just after the split or merge. This modification ensures that $\sum_{k \in K'} R_{kn}$ remains constant during the partial EM, thus preserving normalization.

3 FSMEM

The proposed new FSMEM algorithm is subtly different:

Initialize parameters as for EM.
Run EM until convergence.
Let *state* be MERGE.
Let *fail* be zero.
repeat
  if *state* = MERGE **then**
    Collect a list of merge candidates.
    Sort this list.
    for all promising pairs $\{k_{m_1}, k_{m_2}\}$ **do**
      Perform the merge operation. {This reduces K by 1.}
      Reinitialize the affected cluster.
      Run partial EM on the affected cluster until convergence.
      Run full EM on all clusters until convergence.
      if the MDL likelihood is better than before the merge **then**
        Ignore the other candidates, let *fail* be zero again, and go back to the outer loop.
      else
        Restore parameters.
        Let *state* be SPLIT.
        Increment *fail* by one.
  else
    {So *state* = SPLIT}

Collect a list of split candidates.
Sort this list.
for all promising candidates k_s **do**
Perform the split operation. {This increments K by 1.}
Run partial EM on the affected clusters until convergence.
Run full EM on all clusters until convergence.
if the MDL likelihood is better than before the merge **then**
Ignore the other candidates, let *fail* be zero again, and go back to the outer loop.
else
Restore parameters.
Let *state* be MERGE.
Increment *fail* by one.
until *fail* equals two.

Thus split and merge operations can now be executed independently. The algorithm starts by looking for merge candidates, and continues to do that until it doesn't find any. Then it looks for split candidates, and continues to do that until there aren't any successful ones. Then it returns to scanning merge candidates, and so on until neither split nor merge finds an improvement in the MDL likelihood

$$L_{\mathrm{MDL}} = L - \frac{1}{2}\log(N)K(1 + D + \frac{1}{2}D(D+1).$$

The second term here expresses the entropy in the parameters, and corrects for the scaling of L with number of parameters.

4 Stability issues

4.1 Numerical

In several equations I found that numerical problems occurred when $p(\vec{x})$ was so small it was rounded to zero in the computer algorithms. This caused trouble when none of the clusters was near a particular point. I solved these problems by adding a vanishingly small constant $\varepsilon = 10^{-300}$ to some equations.

4.2 Fundamental

There is another, more fundamental stability issue identified by Ueda et al.: there are global optima in parameter space corresponding to the appearance of clusters with vanishing variance. To avoid being sucked into those, they suggest the following alternative for the update step for the variance matrix:

$$\Sigma_k' = \frac{\sum_n R_{kn}(\vec{x}_n - \mu_k')(\vec{x}_n - \mu_k')^T + \lambda I}{\sum_n R_{kn} + \lambda}.$$

(In their paper the λ in the denominator is replaced by 1, but I believe that to be a misprint.) I found good results for $\lambda \sim .01$.

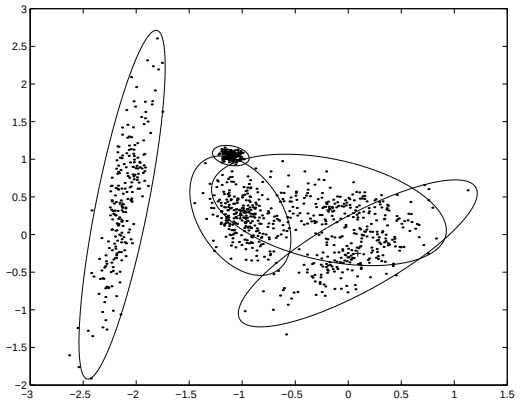
5 Results on artificial data

The SMEM algorithm was able to correctly find clusters in many cases where plain EM was trapped in a local optimum. It could not always correctly reconstruct clusters that were so much on top of each other that taking them apart didn't yield much of an improvement in log likelihood.

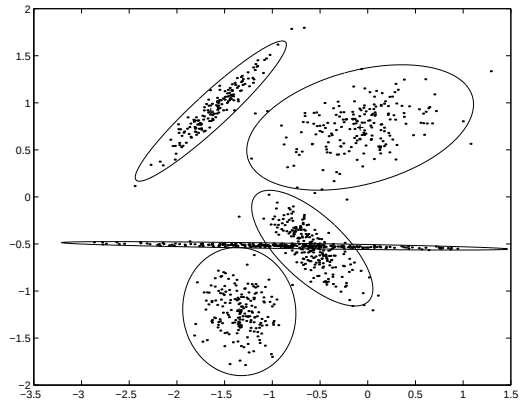
The FSMEM algorithm did just as well, with the added bonus that it could determine the correct number of clusters automatically. Predictably, this worked best if the target clusters were well-separated.

5.1 Two dimensional data

To gain some preliminary insight in the performance of the SMEM, I performed some runs on two dimensional data. The original data are shown below together with the log likelihoods[1] found by the various algorithms and the number of iterations they took. All results based on five runs per source. Reported are mean and spread, even though the distributions are far from Gaussian.

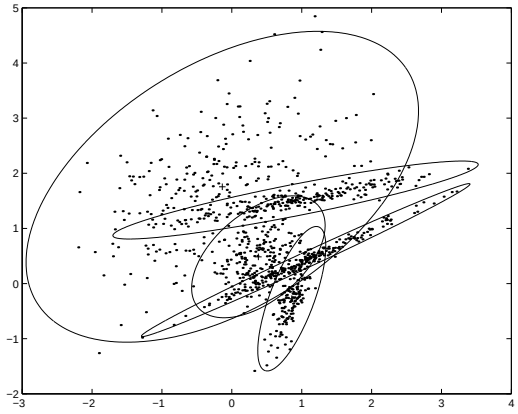


	EM	SMEM
logl:	-205 ± 32	-182 ± 0
iters:	40 ± 26	338 ± 37

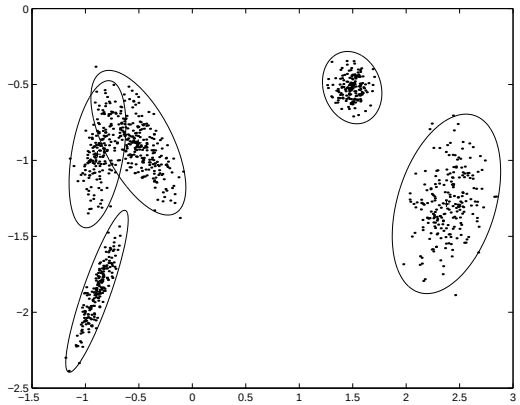


	EM	SMEM
logl:	-95 ± 154	16 ± 0
iters:	25 ± 12	379 ± 203

[1] Modified by the MDL term discussed above for easy comparison with FSMEM results.



	EM	SMEM
logl:	-1450 ± 154	-1278 ± 132
iters:	24 ± 17	269 ± 149



	EM	SMEM
logl:	526 ± 165	613 ± 58
iters:	21 ± 14	114 ± 31

It is seen that SMEM does better than simple EM in all cases, and moreover produces much more predictable results. However, there is a price to pay: the number of iterations is much higher. (Iteration counts include iterations made in discarded steps.) These data sets were generated using `mog_gendata(2,1000,5,.3)`, i.e. $D = 2$, $N = 1000$, $K = 5$ and typical cluster size .3 spread over a typical area of size 1.

5.2 Four dimensional data

After these tests, a more extended test series was done on four dimensional data, for EM, SMEM and FSMEM together. In all cases FSMEM was able to find the correct number of clusters, whether started from $K = 5$ (the correct number of clusters), $K = 1$ or $K = 10$. These were the log likelihoods found in 5 runs for each of 10 sources:

Run	EM	SMEM	FSMEM $K_0 = 1$	FSMEM $K_0 = 5$	FSMEM $K_0 = 10$
01	376.48 ± 55.70	417.24 ± 0.02	417.21 ± 0.09	417.21 ± 0.09	417.21 ± 0.09
02	782.42 ± 294.98	969.08 ± 280.43	1094.50 ± 0.00	1094.50 ± 0.00	1094.50 ± 0.00
03	29.59 ± 238.59	173.50 ± 0.00	173.50 ± 0.01	173.50 ± 0.01	173.50 ± 0.01
04	-197.61 ± 48.78	-161.98 ± 0.00	-162.00 ± 0.01	-162.00 ± 0.01	-162.00 ± 0.01
05	406.12 ± 150.98	570.17 ± 0.00	570.12 ± 0.04	570.12 ± 0.04	570.12 ± 0.04
06	136.20 ± 297.57	471.60 ± 0.00	471.58 ± 0.04	471.58 ± 0.04	471.58 ± 0.04
07	471.46 ± 252.24	708.11 ± 0.00	708.11 ± 0.00	708.11 ± 0.00	708.11 ± 0.00
08	90.66 ± 131.87	323.93 ± 0.00	323.93 ± 0.00	323.93 ± 0.00	323.93 ± 0.00
09	104.33 ± 98.83	175.02 ± 0.00	175.02 ± 0.00	175.02 ± 0.00	175.02 ± 0.00
10	225.83 ± 81.66	262.40 ± 0.03	262.20 ± 0.16	262.20 ± 0.16	262.20 ± 0.16

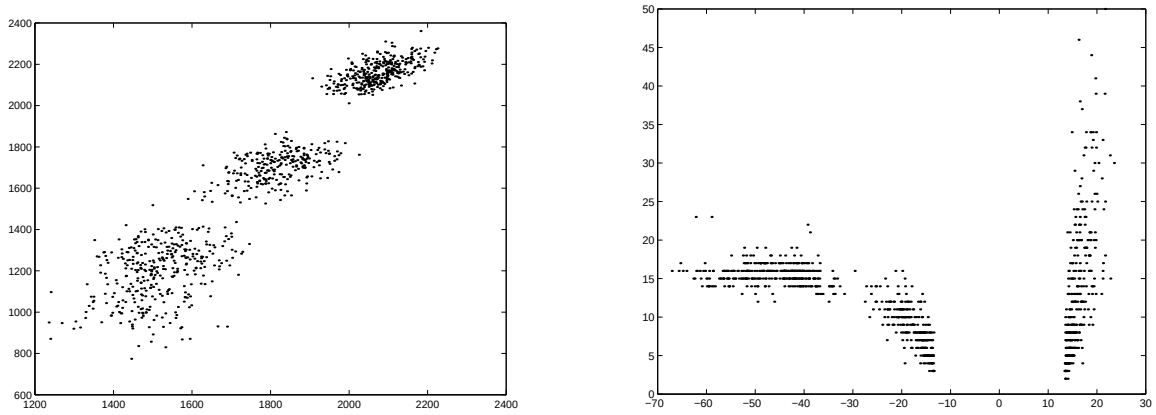
It is seen that FSMEM is at least as proficient than SMEM in avoiding local optima. Moreover, its run-time is much more predictable: these are the iteration count statistics:

Run	EM	SMEM	FSMEM $K_0 = 1$	FSMEM $K_0 = 5$	FSMEM $K_0 = 10$
01	376.48 ± 55.70	140.8 ± 29.7	289.0 ± 27.7	289.0 ± 27.7	289.0 ± 27.7
02	782.42 ± 294.98	179.4 ± 44.2	206.6 ± 24.6	206.6 ± 24.6	206.6 ± 24.6
03	29.59 ± 238.59	351.0 ± 149.5	263.4 ± 19.4	263.4 ± 19.4	263.4 ± 19.4
04	-197.61 ± 48.78	509.0 ± 104.5	275.4 ± 36.9	275.4 ± 36.9	275.4 ± 36.9
05	406.12 ± 150.98	381.0 ± 200.4	228.2 ± 44.8	228.2 ± 44.8	228.2 ± 44.8
06	136.20 ± 297.57	259.8 ± 57.5	225.6 ± 22.1	225.6 ± 22.1	225.6 ± 22.1
07	471.46 ± 252.24	141.8 ± 21.0	194.6 ± 21.5	194.6 ± 21.5	194.6 ± 21.5
08	90.66 ± 131.87	321.6 ± 105.4	229.6 ± 21.5	229.6 ± 21.5	229.6 ± 21.5
09	104.33 ± 98.83	184.8 ± 89.3	227.2 ± 17.3	227.2 ± 17.3	227.2 ± 17.3
10	225.83 ± 81.66	239.0 ± 148.2	233.0 ± 37.0	233.0 ± 37.0	233.0 ± 37.0

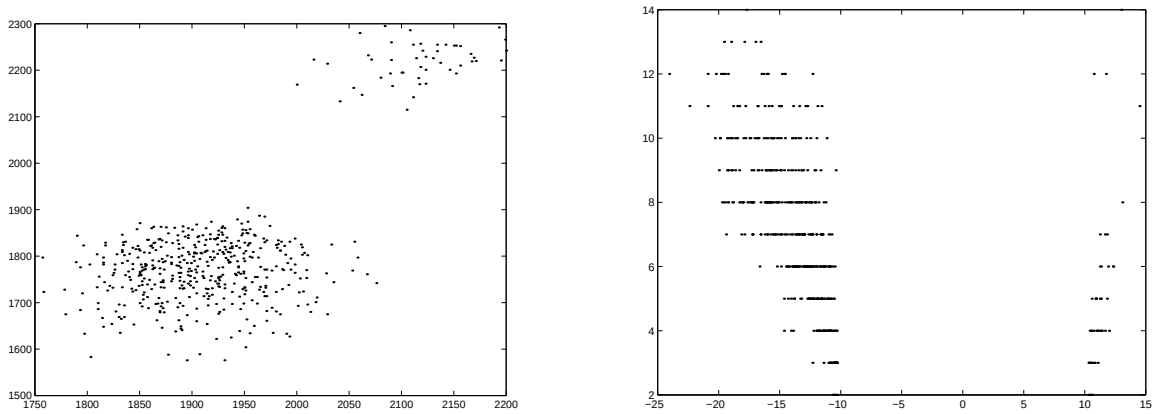
6 Results on real data

For a real test of the SMEM and FSMEM algorithms, I took a dataset of 15000 recorded action potentials from a multi-electrode array with 60 electrodes and several thousand neurons growing on them. These recordings consist of the electric potential measured over a period from 1 ms before to 2 ms after the peak of the action potential. Each electrode typically records from one to three neurons. MoG clustering can potentially be used to classify the recorded spikes according to responsible neuron.

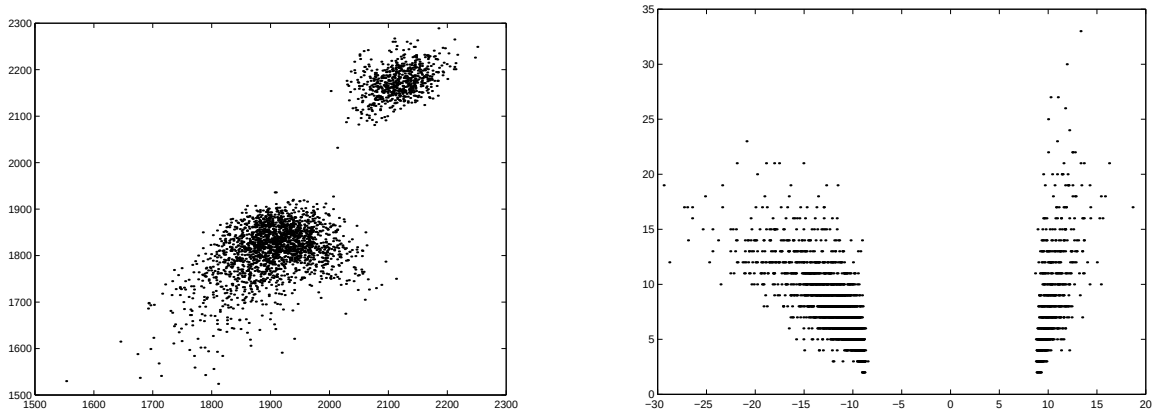
Here are a few examples of clustering results. For each, the clusters found are displayed in two forms: a plot of y_0 vs y_{-5}, i.e. the voltage right at the peak vs the voltage .2 ms earlier, and a plot of spike width versus spikes height. Results for EM and SMEM are not shown, because those algorithms are not able to judge the extract the number of clusters.



Log likelihood: -40204 ± 66. Iterations: 73 ± 22. Number of clusters: 3.



Log likelihood: -20170 ± 34. Iterations: 44 ± 11. Number of clusters: 2.



Log likelihood: -97930 ± 15. Iterations: 68 ± 12. Number of clusters: 3. (The blob on the left consists of two, partially overlapping, clusters.)

In a few cases the algorithm found different number of clusters on different runs (not shown).

Whether this way of clustering spike data is really useful will require some further study. For example, I suspect that in many cases the spikes with positive height (rightmost cluster in the righthand column) may not be physical, while some spikes detected wuth very low height are a noise artifact.

7 The code

This section presents all the matlab code written for this project. Electronic versions are available upon request.

The outer loop for SMEM:

```
function y = mog_smem(X,K,epsi,lambda)
% MOG_SMEM performs Ueda et al split/merge EM algorithm for mixture
% of gaussians.
% Input: X: DxN data
%        K: nr of clusters
%        epsi: relative change of log likelihood used for termination
```

```
% lambda: fudge parameter to prevent zero-variance attractor
% Output: structure with members
%   p:    1xK vector of mixing coefficients
%   mu:  DxK matrix of means
%   sig: K cell vector of DxD matrices of variances
%   likelies: successive local optima of log likelihood.
%   iters: succesive iteration counts of em runs: even numbers
%          refer to partial em runs. Discarded runs are included.

% Some constants:
max_iter = 100;
max_cands = 5;
 split_init_epsi   = .1;

par = mog_init(X,K);
 likelies  = zeros(0,1);
 iters  = zeros(0,1);

par = mog_fullem(X,par,epsi,lambda);
 iters  = cat(1, iters , par.iters );

FIG = 1;

fprintf(1,'First run converged after %i iterations\n',par.iters);
figure(FIG);
plot_mog(par,X);

for iter=1:max_iter;
  FIG = 3 - FIG;
  figure(FIG);
  fprintf(1,'mog_smem: major iteration %i\n',iter);
  merits = mog_merits(X,par);

   likelies  = cat(1, likelies , par.likely );

  for c=1:max_cands
    m1 = merits.Jmerge(c,2);
    m2 = merits.Jmerge(c,3);
    for d=1:K
      s = merits.Jsplit(d,2);
      if ((s ~= m1) & (s ~= m2))
        break; % found a candidate
      end
    end
    % So now m1,m2,s are a merge/split triplet
    disp(m1); disp(m2); disp(s);
    para = mog_mergeinit(par, m1, m2);
    para = mog_splitinit(para, s, m2, split_init_epsi );

    fprintf(1,'Trying candidate %i:\n',c);
    plot_mog(para,X);
```

```
58
59      para = mog_partialem(X,para,epsi,lambda,[m1,m2,s]);
60       iters = cat(1, iters , para.iters );
61
62      fprintf(1,'Partial EM converged after %i\n',para.iters);
63      plot_mog(para,X);
64
65      para = mog_fullem(X,para,epsi,lambda);
66       iters = cat(1, iters , para.iters );
67
68      fprintf(1,'Full EM converged after %i\n',para.iters);
69      plot_mog(para,X);
70
71      if (para.likely > par.likely)
72        fprintf(1,'Improvement found from candidate %i\n',c);
73        break; % out of candidate test loop
74      end
75    end % for candidates
76    if (para.likely <= par.likely)
77      fprintf('No more improvements available after %i major iters\n',iter);
78      break; % out of main loop: split/merge didn't produce better results
79    end
80    par = para;
81  end
82
83  fprintf(1,'Final result:\n');
84  plot_mog(par,X);
85
86  y=par;
87  y.likelies = likelies;
88  y.iters = iters;
```

The outer loop for FSMEM:

```
1  function y = mog_fsmem(X,K,epsi,lambda)
2  % MOG_FSMEMDW performs free split/merge EM: DWs modification of
3  % Ueda et al split/merge EM algorithm for mixture of gaussians.
4  % Input: X: DxN data
5  %        K: nr of clusters
6  %        epsi: relative change of log likelihood used for termination
7  %        lambda: fudge parameter to prevent zero-variance attractor
8  % Output: structure with members
9  %   p:    1xK vector of mixing coefficients
10 %   mu:   DxK matrix of means
11 %   sig:  K cell vector of DxD matrices of variances
12 %   likelies: successive local optima of log likelihood.
13 %   iters: succesive iteration counts of em runs: even numbers
14 %          refer to partial em runs. Discarded runs are included.
15
16 % Some constants:
17 max_iter = 100;
18 max_cands = 5;
```

```
 split_init_epsi    = 1;

par = mog_init(X,K);
 likelies   = zeros(0,1);
 iters  = zeros(0,1);

par = mog_fullem(X,par,epsi,lambda);
 iters  = cat(1, iters , par.iters );
 likelies   = cat(1, likelies  , par.likely );

FIG = 1;

fprintf(1,'First run converged after %i iterations\n',par.iters);
figure(FIG);

next  = 1;
 fail   = .5;

for  iter=1:max_iter;
  plot_mog(par,X);
  FIG = 3 - FIG;
  figure(FIG);
  fprintf(1,'mog_smem: major iteration %i\n',iter);

  merits = mog_merits(X,par);

  if (next > 0)
    % try to merge
    merge_cands = size(merits.Jmerge,1);
    if (merge_cands > max_cands)
      merge_cands = max_cands;
    end
    para.likely =-1e9;
    for c=1:merge_cands
      m1 = merits.Jmerge(c,2);
      m2 = merits.Jmerge(c,3);

      para = mog_mergeinit(par, m1, m2);
      para = mog_truncate(para, m2);

      fprintf(1,'Trying merge candidate %i [%i %i]:\n',c,m1,m2);
      plot_mog(para,X);

      para = mog_partialem(X,para,epsi,lambda,[m1]);
       iters  = cat(1, iters , para.iters );

      fprintf(1,'Partial EM converged after %i\n',para.iters);
      plot_mog(para,X);

      para = mog_fullem(X,para,epsi,lambda);
       iters  = cat(1, iters , para.iters );
```

```
70
71        fprintf(1,'Full EM converged after %i\n',para.iters);
72        plot_mog(para,X);
73
74        if (para.likely > par.likely)
75          fprintf(1,'Improvement found from merge candidate %i. New K is %i\n',c,length(para.p));
76          break; % out of candidate test loop
77        end
78      end % for merge candidates
79      if (para.likely > par.likely)
80        par = para;
81         likelies = cat(1, likelies , par.likely);
82        fprintf(1,'Setting par to para from merge. New K is %i\n',length(par.p));
83         fail = .5;
84      else
85        fail = fail + 1;
86        next = -1;
87      end
88    else
89      % try to split
90       split_cands = size(merits.Jsplit,1);
91      if ( split_cands > max_cands)
92         split_cands = max_cands;
93      end
94      for c=1:split_cands
95        s = merits.Jsplit(c,2);
96        para = mog_extend(par);
97        s2 = length(para.p);
98        para = mog_splitinit(para, s, s2, split_init_epsi);
99        fprintf(1,'Trying split candidate %i [%i]:\n',c,s);
100       plot_mog(para,X);
101
102       para = mog_partialem(X,para,epsi,lambda,[s,s2]);
103        iters = cat(1, iters , para.iters);
104
105       fprintf(1,'Partial EM converged after %i\n',para.iters);
106       plot_mog(para,X);
107
108       para = mog_fullem(X,para,epsi,lambda);
109        iters = cat(1, iters , para.iters);
110
111       fprintf(1,'Full EM converged after %i\n',para.iters);
112       plot_mog(para,X);
113
114       if (para.likely > par.likely)
115         fprintf(1,'Improvement found from split candidate %i. new K is %i\n',c,s2);
116         break; % out of candidate test loop
117       end
118     end % for split candidates
119     if (para.likely > par.likely)
120       par = para;
```

```
121          likelies = cat(1, likelies , par.likely );
122          fprintf(1,'Setting par to para from split. New K is %i\n',length(par.p));
123          fail = .5;
124        else
125          fail = fail +1;
126          next = 1;
127        end
128      end
129
130      if ( fail > 2)
131        fprintf('No more improvements available after %i major iters\n',iter);
132        break; % out of main loop: split nor merge produced better results
133      end
134    end
135
136    fprintf(1,'Final result:\n');
137    plot_mog(par,X);
138
139    y=par;
140    y.likelies = likelies ;
141    y.iters = iters ;
```

These use the following code for full EM:

```
1  function y = mog_fullem(X,par,epsi,lambda)
2  % MOG_FULLEM implements the EM algorithm for mixture of Gaussians.
3  % Input: X: DxN data
4  %        par: structure with members:
5  %          p:  1xK vector of mixing coefficients
6  %          mu: DxK matrix of means
7  %          sig: K cell vector of DxD matrices of variances
8  %        epsi: relative change of log likelihood used for termination
9  %        lambda: fudge parameter to prevent zero-variance attractor
10 % Output: structure with members as par, plus
11 %  likely: log likelihood at end of run
12 % Algorithm: Max Welling, in class notes for CS156b
13 % Coding:   Daniel Wagenaar, April-May 2000
14
15 % some constants
16 max_iter = 100;
17
18 % initialize parameters
19 mu = par.mu;
20 p = par.p;
21 sig = par.sig;
22
23 N=size(X,2);
24 D=size(X,1);
25 K=size(p,2);
26
27 old_likely = -1e9;
28
```

```
for iter = 1:max_iter

  lastiter = iter;

  % E step: compute responsibilities
  norma = zeros(1,N);
  R = zeros(N,K); % R(n,k) will be the responsibility of cluster k for point n
  px = zeros(1,N);
  for k=1:K
  % This section is copied verbatim from mog_responsibility.m
    s = sig{k};
    siginv = inv(s);
    detsig = det(s);
    dx = X - repmat(mu(:,k),[1 N]);
    sdx = siginv * dx;
    expo = -.5 * sum(dx .* sdx,1);
    G = (1/sqrt(2*pi*detsig)) * exp(expo); % a 1xN vector of p(x|k)
    pG = p(k)*G;
    px = px + pG;
    norma = norma + pG;
    R(:,k) = pG';
  % End of copied section
  end
  likely = sum(log(px+1e-300));

  if (abs((likely - old_likely)/likely) < epsi)
    break;
  end
  old_likely = likely;

  for k=1:K
    R(:,k) = R(:,k)./(norma+1e-300)';
  end

  % M step: recompute mu, sig, p
  for k=1:K
    sumR = sum(R(:,k));
    mu(:,k) = (X*R(:,k))./sumR; % DxN * Nx1 = Dx1
    dx = X - repmat(mu(:,k),[1 N]);
    Rdx = repmat(R(:,k)',[D 1]) .* dx; % DxN
    sig{k} = (Rdx*dx' +lambda*eye(D))/(sumR+lambda);
    % sig{k} = (Rdx * dx')/sumR; % DxN * NxD = DxD
  end
  p = mean(R,1);

end

% return parameters
y.p = p;
y.mu = mu;
y.sig = sig;
```

```
y.likely = likely - .5*log(N)*K*(1+D+.5*D*(D+1)); % Subtract MDL term
y.iters = lastiter ;
return;
```

And this for partial EM:

```
function y = mog_partialem(X,par,epsi,lambda,idx)
% MOG_PARTIAL implements the partial update alg for MoG from Ueda et al
% Input: X: DxN data
%        par: structure with members:
%          p:   1xK vector of mixing coefficients
%          mu:  DxK matrix of means
%          sig: K cell vector of DxD matrices of variances
%        epsi: relative change of log likelihood used for termination
%        lambda: fudge parameter to prevent zero-variance attractor
%        idx: index vector specifying which clusters should be updated
% Output: structure with members as par
% Algorithm: Max Welling, in class notes for CS156b and Ueda et al
% Coding:    Daniel Wagenaar, April-May 2000

% some constants
max_iter = 100;

% initialize parameters
mu = par.mu;
p = par.p;
sig = par.sig;

N=size(X,2);
D=size(X,1);
K=size(p,2);

% Compute initial responsibilities
R = mog_responsibility(X,par);
primalR = sum(R(:,idx),2);  % Nx1 vector of primary total responsibilities

old_likely = -1e9;

for iter = 1:max_iter

  lastiter = iter;

  % E step: compute responsibilities
  norma = zeros(1,N);
  R = zeros(N,K); % R(n,k) will be the responsibility of cluster k for point n
  px = zeros(1,N);
  for k = idx
  % This section is copied verbatim from mog_responsibility.m
    s = sig{k};
    siginv = inv(s);
    detsig = det(s);
    dx = X - repmat(mu(:,k),[1 N]);
```

```
    sdx = siginv * dx;
    expo = -.5 * sum(dx .* sdx ,1);
    G = (1/sqrt(2*pi*detsig )) * exp(expo ); % a 1xN vector of p(x|k)
    pG = p(k)*G;
    px = px + pG;
    norma = norma + pG;
    R(:,k ) = pG';
  % End of copied section
  end
  likely = sum(log(px+1e-300)); % This is only the 'local' likelihood
                                % within the idx group

  if (abs(( likely - old_likely )/ likely ) < epsi )
    break;
  end
  old_likely = likely ;

  norma = primalR ./ (norma+1e-300)';
  for k = idx
    R(:,k ) = R(:,k ) .* norma;
  end

  % M step: recompute mu, sig, p
  for k = idx
    sumR = sum(R(:,k));
    mu(:,k ) = ( X*R(:,k )) ./ sumR; % DxN * Nx1 = Dx1
    dx = X - repmat(mu(:,k),[1 N]);
    Rdx = repmat(R(:,k) ',[ D 1]) .* dx ; % DxN
    sig{k } = (Rdx*dx' +lambda*eye(D))/(sumR+lambda);
    % sig{k} = (Rdx * dx')/sumR; % DxN * NxD = DxD
    p(k ) = mean(R(:,k));
  end

end

% return parameters
y.p = p;
y.mu = mu;
y.sig = sig ;
y. iters = lastiter ;
return;
```

Initialization is done using:

```
function y = mog_init(X,K)
% MOG_INIT provides initialization for mixture of Gaussians EM.
% Input: X: DxN data
%        K: number of clusters
% Output: Structure that can be passed to mog_fullem
% Coding: Daniel Wagenaar, April-May 2000


```

```
% initialize parameters
N=size(X,2);
D=size(X,1);
p = (1/K) * ones(1,K);
idx = floor(rand(1,K)*N+1);
mu = X(:,idx);
datvar = diag(var(X')); % a DxD diagonal matrix containing the data variance
for k=1:K
  sig{k} = datvar / 40; % This is arbitrary, and not very clever
end

% return parameters
y.p = p;
y.mu = mu;
y.sig = sig;
return;
```

This calculates the split criterion:

```
function J = mog_splitmerit(X,par,R)
% MOG_SPLITMERIT computes the split merit matrix for MoG SMEM
% Input: X, par: as for mog_fullem
%        R: NxK responsibility matrix as from mog_responsibility
% Output: Kx1 split merit vector
% Algorithm: Ueda et al
% Coding: DW

% Jsplit_k := int dx f_k(x) log (f_k(x) / p(x|theta_k)),
% where f_k(x) := (sum_n delta(x-x_n) R_kn) / (sum_n R_kn)
% This can be re-expressed as
% Jsplit_k := sum_n f_kn log (f_kn / p_nk), [1]
% where f_kn := R_kn / sum_m R_km and p_nk = p(x|theta_k).
% This is not quite true: int dx delta(x-y) log delta(x-y)*f(x) [2]
% isn't quite equal to log f(y), but I suspect Ueda et al haven't
% been careful about this either. Strictly, of course, the integral
% [2] diverges. I suspect that the divergent term originates from
% the fact that the KL distance between a continuous and a discreet
% pdf is necessarily divergent. I hope the above prescription [1] is
% the sensible one.

mu = par.mu;
p = par.p;
sig = par.sig;

N=size(X,2);
D=size(X,1);
K=size(p,2);

J=zeros(K,1);

for k=1:K
  % This section is copied verbatim from mog_responsibility.m
```

```
    s = sig{k};
    siginv = inv(s);
    detsig = det(s);
    dx = X - repmat(mu(:,k),[1 N]);
    sdx = siginv * dx;
    expo = -.5 * sum(dx .* sdx,1);
    G = (1/sqrt(2*pi*detsig)) * exp(expo); % a 1xN vector of p(x|k)
    % End of copied section

    f = R(:,k) ./ sum(R(:,k));
    G = G + .00001;
    idx = find(f>.00001);
    J(k) = sum(f(idx).*log(f(idx)./G(idx)'));
end
```

And this the merge criterion:

```
function J = mog_mergemerit(X,par,R)
% MOG_MERGEMERIT computes the merge merit matrix for MoG SMEM
% Input: X, par: as for mog_fullem
%        R: NxK responsibility matrix as from mog_responsibility
% Output: KxK symmetric matrix of merge merits
% Algorithm: Ueda et al
% Coding: DW

J = R'*R;
```

These are called through a small wrapper:

```
function y = mog_merits(X,par)
% MOG_MERITS returns sorted lists of split and merge candidates.
% Input: X, par as per mog_fullem
% Output: structure with members
%    Jsplit : Kx2 matrix of [merit; clusterno] rows
%    Jmerge: (K*(K-1)/2)x3 matrix of [merit, clusterno_1, clusterno_2] rows
% By definition clusterno_1 < clusterno_2 for all reported
% candidates.
% Note that split and merge merits cannot be usefully added
% directly: they are unfortunately defined on different scales.

K=size(par.p,2);

R = mog_responsibility(X,par);

Jm = mog_mergemerit(X,par,R);
Jsplit = mog_splitmerit(X,par,R);

Jmerge = zeros(K*(K-1)/2,3);
idx = 1;
for k=1:K
  for l=(k+1):K
    Jmerge(idx,:) = [ Jm(k,l), k, l ];
    idx = idx + 1;
  end
```

```
end

 Jsplit  = cat(2, Jsplit ,[1:K]');

[dummy, idx] = sort(-Jmerge (:,1));
y.Jmerge = Jmerge(idx ,:);
[dummy, idx] = sort(- Jsplit  (:,1));
y. Jsplit  =  Jsplit (idx ,:);
```

Initialization after a merge is done by:

```
function y = mog_mergeinit(par , k , l , epsi )
% MOG_MERGEINIT(par,k,l,epsi) initializes the parameters for clusters k
% and l as the merges of cluster k and l into one. The result is
% stored in cluster k, and p(l) is set to zero.
% Input: par: parameters as for all mog fns
%        k: source cluster 1, also dest. cluster
%        l : source cluster 2
% Output: structure of updated parameters
% Alg: Ueda
% Coding: DW

y=par;

y.p(k ) = par.p(k)+par.p(l );
y.mu(:,k ) = ( par.mu(:,k ) + par.mu(:,l  )) ./ 2;
y. sig{k } = (par. sig{k } + par. sig{l  }) ./ 2;
y.p(l ) = 0;
```

And after a split by:

```
function y =  mog_splitinit (par , k , l , epsi )
% MOG_SPLITINIT(par,k,l,epsi) initializes the parameters for clusters k
% and l as the split of cluster k into two. The original cluster l
% is destroyed. Normalization is lost unless par.p(l ) was zero
% before the call.
% Input: par: parameters as for all mog fns
%        k: source cluster , also dest. cluster 1
%        l : destination cluster 2
%        epsi: scale of noise to be added to mean
% Output: structure of updated parameters
% Alg: Ueda / DW
% Coding: DW

D=size(par.mu,1);

y=par;
y.p(k ) = par.p(k)/2;
y.p(l ) = y.p(k );

[ U DD V ] = svd(par.sig{k });
stddev = sqrt(diag(DD));

% Move centres away according to principal components:
```

```
y.mu(:,k) = par.mu(:,k) + epsi .*U*(stddev.*randn(D,1));
y.mu(:,l) = par.mu(:,k) + epsi .*U*(stddev.*randn(D,1));

% Set sigma to maximal component (really primitive, I admit):
y.sig{k} = DD(1,1) * eye(D);
y.sig{l} = y.sig{k};
```

Helper functions to change the number of clusters are

```
function y = mog_extend(par)
% y=MOG_EXTEND(par) returns par unchanged, except that a new cluster
% with p=0, mu=0, sig=1 is added.
% Input: par: as for mog_fullem
% Output: as input
% Coding: DW

K = length(par.p);
D = size(par.mu,1);
y.p = cat(2,par.p ,[0]);
y.mu = cat(2,par.mu,zeros(D,1));
for k=1:K
  y.sig{k} = par.sig{k};
end
y.sig{K+1} = eye(D);
```

and

```
function y = mog_truncate(par,k0)
% y=MOG_TRUNCATE(par,k0) returns par unchanged, except that all
% information pertaining to cluster k0 is removed.
% Input: par: as for mog_fullem
%        k0: cluster to be removed
% Output: as input
% Coding: DW
% Note: the final cluster is renumbered as k0.

K = length(par.p);
Knew = K-1;
y.p = par.p(:,1:Knew);
y.mu = par.mu(:,1:Knew);
for k = 1:Knew
  y.sig{k} = par.sig{k};
end
if (k0<K)
  y.p(:,k0) = par.p(:,K);
  y.mu(:,k0) = par.mu(:,K);
  y.sig{k0} = par.sig{K};
end
```

This calculates the responsibilities for use outside the inner loops:

```
function R = mog_responsibility(X,par)
% MOG_RESPONSIBILITY performs the e step of mog. Using this
% function within a loop will be slow. However, it is nice and easy
% to use for non-looping calcs e.g. at the start of partialem and
```

```
% when calculating split/merge merits.
% Input: X, par as for mog_fullem
% Output: R: NxK responsibility matrix
% Algorithm: Max Welling
% Coding: DW

mu = par.mu;
p = par.p;
sig = par.sig;

N=size(X,2);
D=size(X,1);
K=size(p,2);

norma = zeros(1,N);
R = zeros(N,K); % R(n,k) will be the responsibility of cluster k for point n
px = zeros(1,N);
for k=1:K
  s = sig{k};
  siginv = inv(s);
  detsig = det(s);
  dx = X - repmat(mu(:,k),[1 N]);
  sdx = siginv * dx;
  expo = -.5 * sum(dx .* sdx,1);
  G = (1/sqrt(2*pi*detsig)) * exp(expo); % a 1xN vector of p(x|k)
  pG = p(k)*G;
  px = px + pG;
  norma = norma + pG;
  R(:,k) = pG';
end

for k=1:K
  R(:,k) = R(:,k) ./ ( norma+1e-300)';
end
```

This makes a graphical representation of the current stage, and prints various parameters as well:

```
function plot_mog(y,X)
% PLOT_MOG(y,X) plots the MoG results in y. Use y=em_mog(X,K) to
% fill y. The plot is a projection onto the first two axes.

K = length(y.p);

displ(X,K,y.mu,y.sig); % see below
%return;
disp(y.p);
disp(y.mu);
sigs = zeros(size(y.mu,1),0);
for k=1:K
  sigs = cat(2, sigs , y.sig{k});
end
```

```
disp(sigs );
fprintf (1,'Press enter to continue...\n');
pause;
return;

function displ(X,K,mu,sig)
% DISPL(X,K,mu,sig) is a helper function for plot_mog to
% plot the distribution.
% X: DxN
% K: 1x1
% mu: DxK
% sig: K cells of DxD


plot(X(1,:), X(2,:), 'b.');
hold on;
for k=1:K
  s = sig{k};
  plotGauss(mu(1,k),mu(2,k),s(1,1), s(2,2), s(1,2));
end
hold off;
drawnow;
return;
```

And finally, this generates a test data set:

```
function y =mog_gendata(D,N,K,sig)
% y=MOG_GENDATA(D,N,K,sig) generates DxN data points from a mixture of
% K gaussians with typical variance set by sig.
% y.X is the DxN data
% y.par contains the parameters:
%   p: 1xK
%   mu: DxK
%   sig: {K} DxD


X=zeros(D,0);
N0=1;
Nr=N;
Kr=K;

MEAN=zeros(D,0);
P=zeros(1,0);

for k=1:K
  % Compute variance matrix
  Rot = eye(D);
  for d=1:D
    for e=(d+1):D
      theta = rand(1) * 2*pi;
      rot = eye(D);
      rot(d,d) = cos(theta );
```

```
        rot(e,e) = cos(theta);
        rot(d,e) = sin(theta);
        rot(e,d) = -sin(theta);
        Rot = rot*Rot;
      end
    end
    Rot = sig*Rot*diag(randn(D,1) + 1);
    SIG{k} = Rot*Rot';
    % So now SIG is a randomly rotated covariance matrix
    Mean = randn(D,1);
    MEAN = cat(2,MEAN,Mean);
    Nk = floor((rand(1)+1.5)*(.5*Nr/Kr));
    if ((Nk>Nr) | (Kr==1))
      Nk = Nr;
    end
    P = cat(2,P,Nk/N);
    X = cat(2,X,Rot*randn(D,Nk)+repmat(Mean,1,Nk));
    Nr = Nr - Nk;
    Kr = Kr - 1;
  end

  par.mu=MEAN;
  par.sig = SIG;
  par.p = P;
  figure(3);
  plot_mog(par,X);
  y.X = X
  y.par = par;
```

8 References

[1]