学习FastICA算法的代码实现

Alex / 2022-07-16 / free_learner@163.com / AlexBrain.cn

更新于2023-09-21，主要是文字排版上的更新，内容基本保持不变。

学习FastICA算法的MATLAB实现细节。

一、背景

独立成分分析和FastICA算法原理请参考我以前的博客或下面的文献：

Hyvärinen, A., & Oja, E. (2000). Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5), 411-430.

FastICA的MATLAB代码来自于这里。

我下面的描述的方式是，在原始代码上添加自己的注释，代码原本的注释以`%`开头，我添加的注释以`%%`开头。

二、整体结构

在`Contents.m`文件里可以看到每个函数的用途，和FastICA算法有关的只有5个函数文件：`fastica.m`作为主函数来解析参数和调用其他函数，`fpica.m`用于实现核心算法（approximation of negentropy & fixed-point iteration）、`whitenv.m`用于白化数据、`pcamat.m`用于主成分分析、`remmean.m`用于去除均值。个人学习感受是，`fpica.m`比较复杂，其他函数相对简单。

三、具体函数

1. fastica.m

```
function [Out1, Out2, Out3] = fastica(mixedsig, varargin)
%FASTICA - Fast Independent Component Analysis
%
% FastICA for Matlab 7.x and 6.x
% Version 2.5, October 19 2005
% Copyright (c) Hugo Gävert, Jarmo Hurri, Jaakko Särelä, and Aapo Hyvärinen.
%
% FASTICA(mixedsig) estimates the independent components from given
% multidimensional signals. Each row of matrix mixedsig is one
% observed signal.  FASTICA uses Hyvarinen's fixed-point algorithm,
% see http://www.cis.hut.fi/projects/ica/fastica/. Output from the
% function depends on the number output arguments:
%
% [icasig] = FASTICA (mixedsig); the rows of icasig contain the
% estimated independent components.
%
% [icasig, A, W] = FASTICA (mixedsig); outputs the estimated separating
% matrix W and the corresponding mixing matrix A.
%
% [A, W] = FASTICA (mixedsig); gives only the estimated mixing matrix
% A and the separating matrix W.
%
% Some optional arguments induce other output formats, see below.
%
% A graphical user interface for FASTICA can be launched by the
% command FASTICAG
%
% FASTICA can be called with numerous optional arguments. Optional
% arguments are given in parameter pairs, so that first argument is
% the name of the parameter and the next argument is the value for
% that parameter. Optional parameter pairs can be given in any order.
%
% OPTIONAL PARAMETERS:
%
% Parameter name        Values and description
%
%=========================================================================
% --Basic parameters in fixed-point algorithm:
%
% 'approach'            (string) The decorrelation approach used. Can be
%                       symmetric ('symm'), i.e. estimate all the
%                       independent component in parallel, or
%                       deflation ('defl'), i.e. estimate independent
%                       component one-by-one like in projection pursuit.
%                       Default is 'defl'.
%
% 'numOfIC'             (integer) Number of independent components to
```

```
%                       be estimated. Default equals the dimension of data.
%
%===================================================================
% --Choosing the nonlinearity:
%
% 'g'                   (string) Chooses the nonlinearity g used in
%                       the fixed-point algorithm. Possible values:
%
%                       Value of 'g':      Nonlinearity used:
%                       'pow3' (default)   g(u)=u^3
%                       'tanh'             g(u)=tanh(a1*u)
%                       'gauss             g(u)=u*exp(-a2*u^2/2)
%                       'skew'             g(u)=u^2
%
% 'finetune'        (string) Chooses the nonlinearity g used when
%                       fine-tuning. In addition to same values
%                       as for 'g', the possible value 'finetune' is:
%                       'off'              fine-tuning is disabled.
%
% 'a1'                  (number) Parameter a1 used when g='tanh'.
%                       Default is 1.
% 'a2'                  (number) Parameter a2 used when g='gaus'.
%                       Default is 1.
%
% 'mu'          (number) Step size. Default is 1.
%                       If the value of mu is other than 1, then the
%                       program will use the stabilized version of the
%                       algorithm (see also parameter 'stabilization').
%
%
% 'stabilization'       (string) Values 'on' or 'off'. Default 'off'.
%                       This parameter controls weather the program uses
%                       the stabilized version of the algorithm or
%                       not. If the stabilization is on, then the value
%                       of mu can momentarily be halved if the program
%                       senses that the algorithm is stuck between two
%                       points (this is called a stroke). Also if there
%                       is no convergence before half of the maximum
%                       number of iterations has been reached then mu
%                       will be halved for the rest of the rounds.
%
%===================================================================
% --Controlling convergence:
%
% 'epsilon'             (number) Stopping criterion. Default is 0.0001.
%
% 'maxNumIterations'    (integer) Maximum number of iterations.
%                       Default is 1000.
```

```
%
% 'maxFinetune'         (integer) Maximum number of iterations in
%                       fine-tuning. Default 100.
%
% 'sampleSize'          (number) [0 - 1] Percentage of samples used in
%                       one iteration. Samples are chosen in random.
%                       Default is 1 (all samples).
%
% 'initGuess'           (matrix) Initial guess for A. Default is random.
%                       You can now do a "one more" like this:
%                       [ica, A, W] = fastica(mix, 'numOfIC',3);
%                       [ica2, A2, W2] = fastica(mix, 'initGuess', A, 'numOfIC', 4);
%
%======================================================================
% --Graphics and text output:
%
% 'verbose'             (string) Either 'on' or 'off'. Default is
%                       'on': report progress of algorithm in text format.
%
% 'displayMode'         (string) Plot running estimates of independent
%                       components: 'signals', 'basis', 'filters' or
%                       'off'. Default is 'off'.
%
% 'displayInterval'     Number of iterations between plots.
%                       Default is 1 (plot after every iteration).
%
%======================================================================
% --Controlling reduction of dimension and whitening:
%
% Reduction of dimension is controlled by 'firstEig' and 'lastEig', or
% alternatively by 'interactivePCA'.
%
% 'firstEig'            (integer) This and 'lastEig' specify the range for
%                       eigenvalues that are retained, 'firstEig' is
%                       the index of largest eigenvalue to be
%                       retained. Default is 1.
%
% 'lastEig'             (integer) This is the index of the last (smallest)
%                       eigenvalue to be retained. Default equals the
%                       dimension of data.
%
% 'interactivePCA'      (string) Either 'on' or 'off'. When set 'on', the
%                       eigenvalues are shown to the user and the
%                       range can be specified interactively. Default
%                       is 'off'. Can also be set to 'gui'. Then the user
%                       can use the same GUI that's in FASTICAG.
%
% If you already know the eigenvalue decomposition of the covariance
```

```
% matrix, you can avoid computing it again by giving it with the
% following options:
%
% 'pcaE'                (matrix) Eigenvectors
% 'pcaD'                (matrix) Eigenvalues
%
% If you already know the whitened data, you can give it directly to
% the algorithm using the following options:
%
% 'whiteSig'            (matrix) Whitened signal
% 'whiteMat'            (matrix) Whitening matrix
% 'dewhiteMat'          (matrix) dewhitening matrix
%
% If values for all the 'whiteSig', 'whiteSig' and 'dewhiteMat' are
% supplied, they will be used in computing the ICA. PCA and whitening
% are not performed. Though 'mixedsig' is not used in the main
% algorithm it still must be entered - some values are still
% calculated from it.
%
% Performing preprocessing only is possible by the option:
%
% 'only'                (string) Compute only PCA i.e. reduction of
%                       dimension ('pca') or only PCA plus whitening
%                       ('white'). Default is 'all': do ICA estimation
%                       as well.  This option changes the output
%                       format accordingly. For example:
%
%                       [whitesig, WM, DWM] = FASTICA(mixedsig,
%                       'only', 'white')
%                       returns the whitened signals, the whitening matrix
%                       (WM) and the dewhitening matrix (DWM). (See also
%                       WHITENV.) In FastICA the whitening matrix performs
%                       whitening and the reduction of dimension. Dewhitening
%                       matrix is the pseudoinverse of whitening matrix.
%
%                       [E, D] = FASTICA(mixedsig, 'only', 'pca')
%                       returns the eigenvector (E) and diagonal
%                       eigenvalue (D) matrices  containing the
%                       selected subspaces.
%
%=========================================================================
% EXAMPLES
%
%       [icasig] = FASTICA (mixedsig, 'approach', 'symm', 'g', 'tanh');
%               Do ICA with tanh nonlinearity and in parallel (like
%               maximum likelihood estimation for supergaussian data).
%
%       [icasig] = FASTICA (mixedsig, 'lastEig', 10, 'numOfIC', 3);
```

```
%               Reduce dimension to 10, and estimate only 3
%               independent components.
%
%       [icasig] = FASTICA (mixedsig, 'verbose', 'off', 'displayMode', 'off');
%               Don't output convergence reports and don't plot
%               independent components.
%
%
% A graphical user interface for FASTICA can be launched by the
% command FASTICAG
%
%   See also FASTICAG

% @(#)$Id: fastica.m,v 1.14 2005/10/19 13:05:34 jarmo Exp $

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% check the number of inputs, the dimension of mixed signals, any NaNs and double precision or
not
% Check some basic requirements of the data
if nargin == 0,
  error ('You must supply the mixed data as input argument.');
end

if length (size (mixedsig)) > 2,
  error ('Input data can not have more than two dimensions.');
end

if any (any (isnan (mixedsig))),
  error ('Input data contains NaN''s.');
end

if ~isa (mixedsig, 'double')
  fprintf ('Warning: converting input data into regular (double) precision.\n');
  mixedsig = double (mixedsig);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Remove the mean and check the data
%% the first step is removing the row-wise mean

[mixedsig, mixedmean] = remmean(mixedsig);

[Dim, NumOfSampl] = size(mixedsig);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Default values for optional parameters
```

```
% All
verbose           = 'on';

% Default values for 'pcamat' parameters
firstEig          = 1;
lastEig           = Dim;
interactivePCA    = 'off';

% Default values for 'fpica' parameters
approach          = 'defl';
numOfIC           = Dim;
g                 = 'pow3';
finetune          = 'off';
a1                = 1;
a2                = 1;
myy               = 1;
stabilization     = 'off';
epsilon           = 0.0001;
maxNumIterations  = 1000;
maxFinetune       = 5;
initState         = 'rand';
guess             = 0;
sampleSize        = 1;
displayMode       = 'off';
displayInterval   = 1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parameters for fastICA - i.e. this file
%% jumpPCA=0, jumpWhitening=0 means that PCA and whitening would not be skipped
%% only=3 means doing preprocessing and ICA (instead of PCA or whitening only)
%% userNumOfIC=0 means the IC numbers would be automatically determined.

b_verbose = 1;
jumpPCA = 0;
jumpWhitening = 0;
only = 3;
userNumOfIC = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% pay attention to the way of parsing arguments and dealing with upper/lower cases
% Read the optional parameters

if (rem(length(varargin),2)==1)
  error('Optional parameters should always go by pairs');
else
  for i=1:2:(length(varargin)-1)
    if ~ischar (varargin{i}),
      error (['Unknown type of optional parameter name (parameter' ...
```

```
        ' names must be strings).']);
    end
    % change the value of parameter
    switch lower (varargin{i})
     case 'stabilization'
      stabilization = lower (varargin{i+1});
     case 'maxfinetune'
      maxFinetune = varargin{i+1};
     case 'samplesize'
      sampleSize = varargin{i+1};
     case 'verbose'
      verbose = lower (varargin{i+1});
      % silence this program also
      if strcmp (verbose, 'off'), b_verbose = 0; end
     case 'firsteig'
      firstEig = varargin{i+1};
     case 'lasteig'
      lastEig = varargin{i+1};
     case 'interactivepca'
      interactivePCA = lower (varargin{i+1});
     case 'approach'
      approach = lower (varargin{i+1});
     case 'numofic'
      numOfIC = varargin{i+1};
      % User has supplied new value for numOfIC.
      % We'll use this information later on...
      userNumOfIC = 1;
     case 'g'
      g = lower (varargin{i+1});
     case 'finetune'
      finetune = lower (varargin{i+1});
     case 'a1'
      a1 = varargin{i+1};
     case 'a2'
      a2 = varargin{i+1};
     case {'mu', 'myy'}
      myy = varargin{i+1};
     case 'epsilon'
      epsilon = varargin{i+1};
     case 'maxnumiterations'
      maxNumIterations = varargin{i+1};
     case 'initguess'
      % no use setting 'guess' if the 'initState' is not set
      initState = 'guess';
      guess = varargin{i+1};
     case 'displaymode'
      displayMode = lower (varargin{i+1});
     case 'displayinterval'
```

```
      displayInterval = varargin{i+1};
     case 'pcae'
      % calculate if there are enought parameters to skip PCA
      jumpPCA = jumpPCA + 1;
      E = varargin{i+1};
     case 'pcad'
      % calculate if there are enought parameters to skip PCA
      jumpPCA = jumpPCA + 1;
      D = varargin{i+1};
     case 'whitesig'
      % calculate if there are enought parameters to skip PCA and whitening
      jumpWhitening = jumpWhitening + 1;
      whitesig = varargin{i+1};
     case 'whitemat'
      % calculate if there are enought parameters to skip PCA and whitening
      jumpWhitening = jumpWhitening + 1;
      whiteningMatrix = varargin{i+1};
     case 'dewhitemat'
      % calculate if there are enought parameters to skip PCA and whitening
      jumpWhitening = jumpWhitening + 1;
      dewhiteningMatrix = varargin{i+1};
     case 'only'
      % if the user only wants to calculate PCA or...
      switch lower (varargin{i+1})
       case 'pca'
    only = 1;
       case 'white'
    only = 2;
       case 'all'
    only = 3;
      end

     otherwise
      % Hmmm, something wrong with the parameter string
      error(['Unrecognized parameter: ''' varargin{i} '''']);
    end;
  end;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% print information about data
if b_verbose
  fprintf('Number of signals: %d\n', Dim);
  fprintf('Number of samples: %d\n', NumOfSampl);
end

% Check if the data has been entered the wrong way,
% but warn only... it may be on purpose
```

```
if Dim > NumOfSampl
  if b_verbose
    fprintf('Warning: ');
    fprintf('The signal matrix may be oriented in the wrong way.\n');
    fprintf('In that case transpose the matrix.\n\n');
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculating PCA
%% the second step is doing PCA

% We need the results of PCA for whitening, but if we don't
% need to do whitening... then we dont need PCA...
%% to skip whitening (so PCA is also skipped), the whitened signals and whitening/dewhitening
matrices should be given
if jumpWhitening == 3
  if b_verbose,
    fprintf ('Whitened signal and corresponding matrises supplied.\n');
    fprintf ('PCA calculations not needed.\n');
  end;
else
  % OK, so first we need to calculate PCA
  % Check to see if we already have the PCA data
  %% if jumpPCA = 2, the eigenvalues and eigenvectors are given
  if jumpPCA == 2,
    if b_verbose,
      fprintf ('Values for PCA calculations supplied.\n');
      fprintf ('PCA calculations not needed.\n');
    end;
  else
    % display notice if the user entered one, but not both, of E and D.
    if (jumpPCA > 0) & (b_verbose),
      fprintf ('You must suply all of these in order to jump PCA:\n');
      fprintf ('''pcaE'', ''pcaD''.\n');
    end;

    % Calculate PCA
    [E, D]=pcamat(mixedsig, firstEig, lastEig, interactivePCA, verbose);
  end
end

% skip the rest if user only wanted PCA
if only > 1
%% the third step is whitening the data
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Whitening the data
```

```
  % Check to see if the whitening is needed...
  if jumpWhitening == 3,
    if b_verbose,
      fprintf ('Whitening not needed.\n');
    end;
  else

    % Whitening is needed
    % display notice if the user entered some of the whitening info, but not all.
    if (jumpWhitening > 0) & (b_verbose),
      fprintf ('You must suply all of these in order to jump whitening:\n');
      fprintf ('''whiteSig'', ''whiteMat'', ''dewhiteMat''.\n');
    end;

    % Calculate the whitening
    [whitesig, whiteningMatrix, dewhiteningMatrix] = whitenv ...
                             (mixedsig, E, D, verbose);
  end

end % if only > 1

% skip the rest if user only wanted PCA and whitening
if only > 2
%% the final step is fixed-point method, the core of FastICA
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Calculating the ICA

  % Check some parameters
  % The dimension of the data may have been reduced during PCA calculations.
  % The original dimension is calculated from the data by default, and the
  % number of IC is by default set to equal that dimension.

  Dim = size(whitesig, 1);

  % The number of IC's must be less or equal to the dimension of data
  if numOfIC > Dim
    numOfIC = Dim;
    % Show warning only if verbose = 'on' and user supplied a value for 'numOfIC'
    if (b_verbose & userNumOfIC)
      fprintf('Warning: estimating only %d independent components\n', numOfIC);
      fprintf('(Can''t estimate more independent components than dimension of data)\n');
    end
  end

  % Calculate the ICA with fixed point algorithm.
  [A, W] = fpica (whitesig,  whiteningMatrix, dewhiteningMatrix, approach, ...
          numOfIC, g, finetune, a1, a2, myy, stabilization, epsilon, ...
```

```
        maxNumIterations, maxFinetune, initState, guess, sampleSize, ...
        displayMode, displayInterval, verbose);

  % Check for valid return
  if ~isempty(W)
    % Add the mean back in.
    if b_verbose
      fprintf('Adding the mean back to the data.\n');
    end
    icasig = W * mixedsig + (W * mixedmean) * ones(1, NumOfSampl);
    %icasig = W * mixedsig;
    %% max(abs(W * mixedmean)) > 1e-9 means the mean is non-zero
    if b_verbose & ...
      (max(abs(W * mixedmean)) > 1e-9) & ...
      (strcmp(displayMode,'signals') | strcmp(displayMode,'on'))
      fprintf('Note that the plots don''t have the mean added.\n');
    end
  else
    icasig = [];
  end

end % if only > 2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The output depends on the number of output parameters
% and the 'only' parameter.

if only == 1    % only PCA
  Out1 = E;
  Out2 = D;
elseif only == 2  % only PCA & whitening
  if nargout == 2
    Out1 = whiteningMatrix;
    Out2 = dewhiteningMatrix;
  else
    Out1 = whitesig;
    Out2 = whiteningMatrix;
    Out3 = dewhiteningMatrix;
  end
else      % ICA
  if nargout == 2
    Out1 = A;
    Out2 = W;
  else
    Out1 = icasig;
    Out2 = A;
    Out3 = W;
```

```
  end
end
```

2. remmean.m

```
function [newVectors, meanValue] = remmean(vectors);
%REMMEAN - remove the mean from vectors
%
% [newVectors, meanValue] = remmean(vectors);
%
% Removes the mean of row vectors.
% Returns the new vectors and the mean.
%
% This function is needed by FASTICA and FASTICAG

% @(#)$Id: remmean.m,v 1.2 2003/04/05 14:23:58 jarmo Exp $
%% the vectors are mixed signals, where the row means variable, column means sample

newVectors = zeros (size (vectors));
meanValue = mean (vectors')';
newVectors = vectors - meanValue * ones (1,size (vectors, 2));
```

3. pcamat.m

```
function [E, D] = pcamat(vectors, firstEig, lastEig, s_interactive, ...
    s_verbose);
%PCAMAT - Calculates the pca for data
%
% [E, D] = pcamat(vectors, firstEig, lastEig, ...
%                 interactive, verbose);
%
% Calculates the PCA matrices for given data (row) vectors. Returns
% the eigenvector (E) and diagonal eigenvalue (D) matrices containing the
% selected subspaces. Dimensionality reduction is controlled with
% the parameters 'firstEig' and 'lastEig' - but it can also be done
% interactively by setting parameter 'interactive' to 'on' or 'gui'.
%
% ARGUMENTS
%
% vectors       Data in row vectors.
% firstEig      Index of the largest eigenvalue to keep.
%               Default is 1.
% lastEig       Index of the smallest eigenvalue to keep.
%               Default is equal to dimension of vectors.
% interactive   Specify eigenvalues to keep interactively. Note that if
%               you set 'interactive' to 'on' or 'gui' then the values
%               for 'firstEig' and 'lastEig' will be ignored, but they
%               still have to be entered. If the value is 'gui' then the
%               same graphical user interface as in FASTICAG will be
%               used. Default is 'off'.
% verbose       Default is 'on'.
%
%
% EXAMPLE
%       [E, D] = pcamat(vectors);
%
% Note
%       The eigenvalues and eigenvectors returned by PCAMAT are not sorted.
%
% This function is needed by FASTICA and FASTICAG

% For historical reasons this version does not sort the eigenvalues or
% the eigen vectors in any ways. Therefore neither does the FASTICA or
% FASTICAG. Generally it seams that the components returned from
% whitening is almost in reversed order. (That means, they usually are,
% but sometime they are not - depends on the EIG-command of matlab.)

% @(#)$Id: pcamat.m,v 1.5 2003/12/15 18:24:32 jarmo Exp $

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% the parameters are determined by positions instead of name-value pair in the main function
```

```
% Default values:
if nargin < 5, s_verbose = 'on'; end
if nargin < 4, s_interactive = 'off'; end
if nargin < 3, lastEig = size(vectors, 1); end
if nargin < 2, firstEig = 1; end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Check the optional parameters;
switch lower(s_verbose)
 case 'on'
  b_verbose = 1;
 case 'off'
  b_verbose = 0;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''verbose''\n', s_verbose));
end

switch lower(s_interactive)
 case 'on'
  b_interactive = 1;
 case 'off'
  b_interactive = 0;
 case 'gui'
  b_interactive = 2;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''interactive''\n', ...
        s_interactive));
end

oldDimension = size (vectors, 1);
%% if the interactive is off, check whether the firstEig & lastEig are valid
if ~(b_interactive)
  if lastEig < 1 | lastEig > oldDimension
    error(sprintf('Illegal value [ %d ] for parameter: ''lastEig''\n', lastEig));
  end
  if firstEig < 1 | firstEig > lastEig
    error(sprintf('Illegal value [ %d ] for parameter: ''firstEig''\n', firstEig));
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate PCA

% Calculate the covariance matrix.
%% cov(X, 1) means the covariance matrix is normalized by N instead of N-1
if b_verbose, fprintf ('Calculating covariance...\n'); end
covarianceMatrix = cov(vectors', 1);
```

```
% Calculate the eigenvalues and eigenvectors of covariance matrix.
[E, D] = eig (covarianceMatrix);

% The rank is determined from the eigenvalues - and not directly by
% using the function rank - because function rank uses svd, which
% in some cases gives a higher dimensionality than what can be used
% with eig later on (eig then gives negative eigenvalues).
rankTolerance = 1e-7;
maxLastEig = sum (diag (D) > rankTolerance);
if maxLastEig == 0,
  fprintf (['Eigenvalues of the covariance matrix are' ...
        ' all smaller than tolerance [ %g ].\n' ...
        'Please make sure that your data matrix contains' ...
        ' nonzero values.\nIf the values are very small,' ...
        ' try rescaling the data matrix.\n'], rankTolerance);
  error ('Unable to continue, aborting.');
end

% Sort the eigenvalues - decending.
eigenvalues = flipud(sort(diag(D)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Interactive part - command-line
if b_interactive == 1

  % Show the eigenvalues to the user
  hndl_win=figure;
  bar(eigenvalues);
  title('Eigenvalues');

  % ask the range from the user...
  % ... and keep on asking until the range is valid :-)
  areValuesOK=0;
  while areValuesOK == 0
    firstEig = input('The index of the largest eigenvalue to keep? (1) ');
    lastEig = input(['The index of the smallest eigenvalue to keep? (' ...
                    int2str(oldDimension) ') ']);
    % Check the new values...
    % if they are empty then use default values
    if isempty(firstEig), firstEig = 1;end
    if isempty(lastEig), lastEig = oldDimension;end
    % Check that the entered values are within the range
    areValuesOK = 1;
    if lastEig < 1 | lastEig > oldDimension
      fprintf('Illegal number for the last eigenvalue.\n');
      areValuesOK = 0;
    end
    if firstEig < 1 | firstEig > lastEig
```

```
      fprintf('Illegal number for the first eigenvalue.\n');
      areValuesOK = 0;
    end
  end
  % close the window
  close(hndl_win);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Interactive part - GUI
%% this part is removed because the GUI is of no interest at present

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% See if the user has reduced the dimension enough

if lastEig > maxLastEig
  lastEig = maxLastEig;
  if b_verbose
    fprintf('Dimension reduced to %d due to the singularity of covariance matrix\n',...
           lastEig-firstEig+1);
  end
else
  % Reduce the dimensionality of the problem.
  if b_verbose
    if oldDimension == (lastEig - firstEig + 1)
      fprintf ('Dimension not reduced.\n');
    else
      fprintf ('Reducing dimension...\n');
    end
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Drop the smaller eigenvalues
if lastEig < oldDimension
  lowerLimitValue = (eigenvalues(lastEig) + eigenvalues(lastEig + 1)) / 2;
else
  lowerLimitValue = eigenvalues(oldDimension) - 1;
end

lowerColumns = diag(D) > lowerLimitValue;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Drop the larger eigenvalues
if firstEig > 1
  higherLimitValue = (eigenvalues(firstEig - 1) + eigenvalues(firstEig)) / 2;
else
  higherLimitValue = eigenvalues(1) + 1;
```

```
end
higherColumns = diag(D) < higherLimitValue;

% Combine the results from above
selectedColumns = lowerColumns & higherColumns;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% print some info for the user
if b_verbose
  fprintf ('Selected [ %d ] dimensions.\n', sum (selectedColumns));
end
if sum (selectedColumns) ~= (lastEig - firstEig + 1),
  error ('Selected a wrong number of dimensions.');
end

if b_verbose
  fprintf ('Smallest remaining (non-zero) eigenvalue [ %g ]\n', eigenvalues(lastEig));
  fprintf ('Largest remaining (non-zero) eigenvalue [ %g ]\n', eigenvalues(firstEig));
  fprintf ('Sum of removed eigenvalues [ %g ]\n', sum(diag(D) .* ...
    (~selectedColumns)));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Select the colums which correspond to the desired range of eigenvalues.
E = selcol(E, selectedColumns);
D = selcol(selcol(D, selectedColumns)', selectedColumns);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some more information
if b_verbose
  sumAll=sum(eigenvalues);
  sumUsed=sum(diag(D));
  retained = (sumUsed / sumAll) * 100;
  fprintf('[ %g ] %% of (non-zero) eigenvalues retained.\n', retained);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function newMatrix = selcol(oldMatrix, maskVector);

% newMatrix = selcol(oldMatrix, maskVector);
%
% Selects the columns of the matrix that marked by one in the given vector.
% The maskVector is a column vector.

% 15.3.1998
%% based on my understanding, there is no need to write a function to extract specified
```

```
columns.
%% maybe the authors have other considerations

if size(maskVector, 1) ~= size(oldMatrix, 2),
  error ('The mask vector and matrix are of uncompatible size.');
end

numTaken = 0;

for i = 1 : size (maskVector, 1),
  if maskVector(i, 1) == 1,
    takingMask(1, numTaken + 1) = i;
    numTaken = numTaken + 1;
  end
end

newMatrix = oldMatrix(:, takingMask);
```

4. whitenv.m

```
function [newVectors, whiteningMatrix, dewhiteningMatrix] = whitenv ...
    (vectors, E, D, s_verbose);
%WHITENV - Whitenv vectors.
%
% [newVectors, whiteningMatrix, dewhiteningMatrix] = ...
%                              whitenv(vectors, E, D, verbose);
%
% Whitens the data (row vectors) and reduces dimension. Returns
% the whitened vectors (row vectors), whitening and dewhitening matrices.
%
% ARGUMENTS
%
% vectors       Data in row vectors.
% E             Eigenvector matrix from function 'pcamat'
% D             Diagonal eigenvalue matrix from function 'pcamat'
% verbose       Optional. Default is 'on'
%
% EXAMPLE
%       [E, D] = pcamat(vectors);
%       [nv, wm, dwm] = whitenv(vectors, E, D);
%
%
% This function is needed by FASTICA and FASTICAG
%
%   See also PCAMAT

% @(#)$Id: whitenv.m,v 1.3 2003/10/12 09:04:43 jarmo Exp $

% ============================================================
% Default value for 'verbose'
if nargin < 4, s_verbose = 'on'; end

% Check the optional parameter verbose;
switch lower(s_verbose)
 case 'on'
  b_verbose = 1;
 case 'off'
  b_verbose = 0;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''verbose''\n', s_verbose));
end

% ============================================================
% In some cases, rounding errors in Matlab cause negative
% eigenvalues (elements in the diagonal of D). Since it
% is difficult to know when this happens, it is difficult
% to correct it automatically. Therefore an error is
```

```
% signalled and the correction is left to the user.
if any (diag (D) < 0),
  error (sprintf (['[ %d ] negative eigenvalues computed from the' ...
           ' covariance matrix.\nThese are due to rounding' ...
           ' errors in Matlab (the correct eigenvalues are\n' ...
           'probably very small).\nTo correct the situation,' ...
           ' please reduce the number of dimensions in the' ...
           ' data\nby using the ''lastEig'' argument in' ...
           ' function FASTICA, or ''Reduce dim.'' button\nin' ...
           ' the graphical user interface.'], ...
          sum (diag (D) < 0)));
end

% ===========================================================
% Calculate the whitening and dewhitening matrices (these handle dimensionality
simultaneously).
%% note that the whitening matrix is different from the formula given in the Hyvärinen's paper,
but it is
%% easy to check that the whitened data has an identity covariance matrix
whiteningMatrix = inv (sqrt (D)) * E';
dewhiteningMatrix = E * sqrt (D);

% Project to the eigenvectors of the covariance matrix.
% Whiten the samples and reduce dimension simultaneously.
if b_verbose, fprintf ('Whitening...\n'); end
newVectors =  whiteningMatrix * vectors;

% ===========================================================
% Just some security...
if ~isreal(newVectors)
  error ('Whitened vectors have imaginary values.');
end

% Print some information to user
if b_verbose
  fprintf ('Check: covariance differs from identity by [ %g ].\n', ...
    max (max (abs (cov (newVectors', 1) - eye (size (newVectors, 1))))));
end
```

5. fpica.m

```
function [A, W] = fpica(X, whiteningMatrix, dewhiteningMatrix, approach, ...
            numOfIC, g, finetune, a1, a2, myy, stabilization, ...
            epsilon, maxNumIterations, maxFinetune, initState, ...
            guess, sampleSize, displayMode, displayInterval, ...
            s_verbose);
%FPICA - Fixed point ICA. Main algorithm of FASTICA.
%
% [A, W] = fpica(whitesig, whiteningMatrix, dewhiteningMatrix, approach,
%        numOfIC, g, finetune, a1, a2, mu, stabilization, epsilon,
%        maxNumIterations, maxFinetune, initState, guess, sampleSize,
%        displayMode, displayInterval, verbose);
%
% Perform independent component analysis using Hyvarinen's fixed point
% algorithm. Outputs an estimate of the mixing matrix A and its inverse W.
%
% whitesig                              :the whitened data as row vectors
% whiteningMatrix                       :can be obtained with function whitenv
% dewhiteningMatrix                     :can be obtained with function whitenv
% approach      [ 'symm' | 'defl' ]     :the approach used (deflation or symmetric)
% numOfIC       [ 0 - Dim of whitesig ] :number of independent components estimated
% g             [ 'pow3' | 'tanh' |     :the nonlinearity used
%                 'gaus' | 'skew' ]
% finetune      [same as g + 'off']     :the nonlinearity used in finetuning.
% a1                                    :parameter for tuning 'tanh'
% a2                                    :parameter for tuning 'gaus'
% mu                                    :step size in stabilized algorithm
% stabilization [ 'on' | 'off' ]        :if mu < 1 then automatically on
% epsilon                               :stopping criterion
% maxNumIterations                      :maximum number of iterations
% maxFinetune                           :maximum number of iteretions for finetuning
% initState     [ 'rand' | 'guess' ]    :initial guess or random initial state. See below
% guess                                 :initial guess for A. Ignored if initState = 'rand'
% sampleSize    [ 0 - 1 ]               :percentage of the samples used in one iteration
% displayMode   [ 'signals' | 'basis' | :plot running estimate
%                 'filters' | 'off' ]
% displayInterval                       :number of iterations we take between plots
% verbose       [ 'on' | 'off' ]        :report progress in text format
%
% EXAMPLE
%       [E, D] = pcamat(vectors);
%       [nv, wm, dwm] = whitenv(vectors, E, D);
%       [A, W] = fpica(nv, wm, dwm);
%
%
% This function is needed by FASTICA and FASTICAG
%
%   See also FASTICA, FASTICAG, WHITENV, PCAMAT
```

```
% @(#)$Id: fpica.m,v 1.7 2005/06/16 12:52:55 jarmo Exp $

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Global variable for stopping the ICA calculations from the GUI
global g_FastICA_interrupt;
if isempty(g_FastICA_interrupt)
  clear global g_FastICA_interrupt;
  interruptible = 0;
else
  interruptible = 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Default values

if nargin < 3, error('Not enough arguments!'); end
[vectorSize, numSamples] = size(X);
if nargin < 20, s_verbose = 'on'; end
if nargin < 19, displayInterval = 1; end
if nargin < 18, displayMode = 'on'; end
if nargin < 17, sampleSize = 1; end
if nargin < 16, guess = 1; end
if nargin < 15, initState = 'rand'; end
if nargin < 14, maxFinetune = 100; end
if nargin < 13, maxNumIterations = 1000; end
if nargin < 12, epsilon = 0.0001; end
if nargin < 11, stabilization = 'on'; end
if nargin < 10, myy = 1; end
if nargin < 9, a2 = 1; end
if nargin < 8, a1 = 1; end
if nargin < 7, finetune = 'off'; end
if nargin < 6, g = 'pow3'; end
if nargin < 5, numOfIC = vectorSize; end     % vectorSize = Dim
if nargin < 4, approach = 'defl'; end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the data

if ~isreal(X)
  error('Input has an imaginary part.');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for verbose

switch lower(s_verbose)
 case 'on'
```

```
  b_verbose = 1;
 case 'off'
  b_verbose = 0;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''verbose''\n', s_verbose));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for approach

switch lower(approach)
 case 'symm'
  approachMode = 1;
 case 'defl'
  approachMode = 2;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''approach''\n', approach));
end
if b_verbose, fprintf('Used approach [ %s ].\n', approach); end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for numOfIC

if vectorSize < numOfIC
  error('Must have numOfIC <= Dimension!');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the sampleSize
if sampleSize > 1
  sampleSize = 1;
  if b_verbose
    fprintf('Warning: Setting ''sampleSize'' to 1.\n');
  end
elseif sampleSize < 1
  %% if the final sample size < 1000, change the proportion so that the final sample size is
1000 or use the
  %% full sample size
  if (sampleSize * numSamples) < 1000
    sampleSize = min(1000/numSamples, 1);
    if b_verbose
      fprintf('Warning: Setting ''sampleSize'' to %0.3f (%d samples).\n', ...
          sampleSize, floor(sampleSize * numSamples));
    end
  end
end
if b_verbose
  if  b_verbose & (sampleSize < 1)
```

```
    fprintf('Using about %0.0f%% of the samples in random order in every
step.\n',sampleSize*100);
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for nonlinearity.

switch lower(g)
 case 'pow3'
  gOrig = 10;
 case 'tanh'
  gOrig = 20;
 case {'gaus', 'gauss'}
  gOrig = 30;
 case 'skew'
  gOrig = 40;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''g''\n', g));
end
%% if not use full sample size, the nonlinearity index + 2, e.g., 10 -> 12
if sampleSize ~= 1
  gOrig = gOrig + 2;
end
%% if use stabilization, the nonlinearity index + 1, e.g, 10 -> 11
if myy ~= 1
  gOrig = gOrig + 1;
end

if b_verbose,
  fprintf('Used nonlinearity [ %s ].\n', g);
end

finetuningEnabled = 1;
%% for fine-tuning, the stabilized version was used
switch lower(finetune)
 case 'pow3'
  gFine = 10 + 1;
 case 'tanh'
  gFine = 20 + 1;
 case {'gaus', 'gauss'}
  gFine = 30 + 1;
 case 'skew'
  gFine = 40 + 1;
 case 'off'
%% if myy ~=1, it means the nonlinearity index is already +1
  if myy ~= 1
    gFine = gOrig;
```

```
  else
    gFine = gOrig + 1;
  end
  finetuningEnabled = 0;
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''finetune''\n', ...
        finetune));
end

if b_verbose & finetuningEnabled
  fprintf('Finetuning enabled (nonlinearity: [ %s ]).\n', finetune);
end

switch lower(stabilization)
 case 'on'
  stabilizationEnabled = 1;
 case 'off'
  if myy ~= 1
    stabilizationEnabled = 1;
  else
    stabilizationEnabled = 0;
  end
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''stabilization''\n', ...
        stabilization));
end

if b_verbose & stabilizationEnabled
  fprintf('Using stabilized algorithm.\n');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some other parameters
myyOrig = myy;
% When we start fine-tuning we'll set myy = myyK * myy
myyK = 0.01;
% How many times do we try for convergence until we give up.
failureLimit = 5;
%% these codes seem redundant, because they are initialized again below
usedNlinearity = gOrig;
stroke = 0;
notFine = 1;
long = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for initial state.

switch lower(initState)
```

```
 case 'rand'
  initialStateMode = 0;
 case 'guess'
  if size(guess,1) ~= size(whiteningMatrix,2)
    initialStateMode = 0;
    if b_verbose
      fprintf('Warning: size of initial guess is incorrect. Using random initial guess.\n');
    end
  else
    initialStateMode = 1;
    if size(guess,2) < numOfIC
      if b_verbose
    fprintf('Warning: initial guess only for first %d components. Using random initial guess
for others.\n', size(guess,2));
      end
      guess(:, size(guess, 2) + 1:numOfIC) = ...
                          rand(vectorSize,numOfIC-size(guess,2))-.5;
    elseif size(guess,2)>numOfIC
      guess=guess(:,1:numOfIC);
      fprintf('Warning: Initial guess too large. The excess column are dropped.\n');
    end
    if b_verbose, fprintf('Using initial guess.\n'); end
  end
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''initState''\n', initState));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Checking the value for display mode.
%% mode 1/2/3 plot the sources, mixing matrix and unmixing matrix separately
switch lower(displayMode)
 case {'off', 'none'}
  usedDisplay = 0;
 case {'on', 'signals'}
  usedDisplay = 1;
  if (b_verbose & (numSamples > 10000))
    fprintf('Warning: Data vectors are very long. Plotting may take long time.\n');
  end
  if (b_verbose & (numOfIC > 25))
    fprintf('Warning: There are too many signals to plot. Plot may not look good.\n');
  end
 case 'basis'
  usedDisplay = 2;
  if (b_verbose & (numOfIC > 25))
    fprintf('Warning: There are too many signals to plot. Plot may not look good.\n');
  end
 case 'filters'
  usedDisplay = 3;
```

```
  if (b_verbose & (vectorSize > 25))
    fprintf('Warning: There are too many signals to plot. Plot may not look good.\n');
  end
 otherwise
  error(sprintf('Illegal value [ %s ] for parameter: ''displayMode''\n', displayMode));
end

% The displayInterval can't be less than 1...
if displayInterval < 1
  displayInterval = 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if b_verbose, fprintf('Starting ICA calculation...\n'); end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SYMMETRIC APPROACH
if approachMode == 1,

  % set some parameters more...
  usedNlinearity = gOrig;
  stroke = 0;
  notFine = 1;
  long = 0;
  %% vectorSize means the number of rows of whitened data X
  A = zeros(vectorSize, numOfIC);  % Dewhitened basis vectors.
  if initialStateMode == 0
    % Take random orthonormal initial vectors.
    B = orth (randn (vectorSize, numOfIC));
  elseif initialStateMode == 1
    % Use the given initial vector as the initial state
    %% M=AS, whiteningMatrix * M = whiteningMatrix * AS, let B=whiteningMatrix * A, M means
unwhitened data
    %% whiteningMatrix * M = BS, S=WM, let W=B-1 * whiteningMatrix, B-1=BT, as B is an
orthogonal matrix
    B = whiteningMatrix * guess;
  end
  %% BOld and BOld2 store the estimates at n-1th and n-2th iterations, which could be used to
detect stroke.
  BOld = zeros(size(B));
  BOld2 = zeros(size(B));

  % This is the actual fixed-point iteration loop.
  for round = 1:maxNumIterations + 1,
    %% if the maximum iteration is reached, exit the loop
    if round == maxNumIterations + 1,
      fprintf('No convergence after %d steps\n', maxNumIterations);
      fprintf('Note that the plots are probably wrong.\n');
```

```
      %% if the final estimate is not empty, use it as output
      if ~isempty(B)
    % Symmetric orthogonalization.
    B = B * real(inv(B' * B)^(1/2));

    W = B' * whiteningMatrix;
    A = dewhiteningMatrix * B;
      else
    W = [];
    A = [];
      end
      return;
    end

    if (interruptible & g_FastICA_interrupt)
      if b_verbose
        fprintf('\n\nCalculation interrupted by the user\n');
      end
      if ~isempty(B)
    W = B' * whiteningMatrix;
    A = dewhiteningMatrix * B;
      else
    W = [];
    A = [];
      end
      return;
    end

    % Symmetric orthogonalization.
    %% Formula (45) in Hyvarinen's paper
    B = B * real(inv(B' * B)^(1/2));

    % Test for termination condition. Note that we consider opposite
    % directions here as well.
    %% the inner product of two vectors with unit length is the cosine of the angle between the
two vectors
    %% the inner product is equal to 1/-1 when two vectors are parallel
    %% use the absolute minimum of vector pairs to represent the cost
    minAbsCos = min(abs(diag(B' * BOld)));
    minAbsCos2 = min(abs(diag(B' * BOld2)));

    if (1 - minAbsCos < epsilon)
      %% fine-tuning means using stabilized updating rule on the basis of the current estimate
      %% fine-tuning is only considered after initial convergence is reached
      %% if fine-tuning is enabled and fine-tuning is not started
      if finetuningEnabled & notFine
        if b_verbose, fprintf('Initial convergence, fine-tuning: \n'); end;
        notFine = 0;
```

```
        usedNlinearity = gFine;
        myy = myyK * myyOrig;
        BOld = zeros(size(B));
        BOld2 = zeros(size(B));

      else
        if b_verbose, fprintf('Convergence after %d steps\n', round); end

        % Calculate the de-whitened vectors.
        A = dewhiteningMatrix * B;
        break;
      end
    elseif stabilizationEnabled
      %% if convergence is not achieved, consider stabilization
      %% stroke is defined that the round-th and (round-2)-th estimates are parallel
      %% if stroke is detected, set the myy to the half momentarily (see below)
      if (~stroke) & (1 - minAbsCos2 < epsilon)
    if b_verbose, fprintf('Stroke!\n'); end;
    stroke = myy;
    myy = .5*myy;
    %% for stabilization, use stabilized updating rule (indexed by odd numbers)
    if mod(usedNlinearity,2) == 0
      usedNlinearity = usedNlinearity + 1;
    end
      elseif stroke
    %% if stroke is detected in the last iteration, restore the myy value (which is saved in
stroke)
    %% also reset the nonlinearity into the original one if myy == 1
    myy = stroke;
    stroke = 0;
    if (myy == 1) & (mod(usedNlinearity,2) ~= 0)
      usedNlinearity = usedNlinearity - 1;
    end
    %% if iterations > 500, set the myy into the half in the rest of iterations
    %% if no stroke, the stabilized version will only be enabled after the half of max
iterations
      elseif (~long) & (round>maxNumIterations/2)
    if b_verbose, fprintf('Taking long (reducing step size)\n'); end;
    long = 1;
    myy = .5*myy;
    if mod(usedNlinearity,2) == 0
      usedNlinearity = usedNlinearity + 1;
    end
      end
    end

    BOld2 = BOld;
    BOld = B;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Show the progress...
if b_verbose
  if round == 1
    fprintf('Step no. %d\n', round);
  else
    fprintf('Step no. %d, change in value of estimate: %.3g \n', round, 1 - minAbsCos);
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Also plot the current state...
switch usedDisplay
 case 1
  if rem(round, displayInterval) == 0,
% There was and may still be other displaymodes...
% 1D signals
%% plot the estimated components or sources
icaplot('dispsig',(X'*B)');
drawnow;
  end
 case 2
  if rem(round, displayInterval) == 0,
% ... and now there are :-)
% 1D basis
%% plot the mixing matrix
A = dewhiteningMatrix * B;
icaplot('dispsig',A');
drawnow;
  end
 case 3
  if rem(round, displayInterval) == 0,
% ... and now there are :-)
% 1D filters
%% plot the unmixing matrix
W = B' * whiteningMatrix;
icaplot('dispsig',W);
drawnow;
  end
 otherwise
end

switch usedNlinearity
  % pow3
  %% 10 means the original fixed iteration method;
  %% 11 means the stabilized version, Formula (47) in Hyvarinen's paper
  %% 12 means using part of the sample size in each iteration
```

```
 %% 13 combines 11 and 12
case 10
 B = (X * (( X' * B) .^ 3)) / numSamples - 3 * B;
case 11
 Y = X' * B;
 Gpow3 = Y .^ 3;
 Beta = sum(Y .* Gpow3);
 D = diag(1 ./ (Beta - 3 * numSamples));
 B = B + myy * B * (Y' * Gpow3 - diag(Beta)) * D;
case 12
 Xsub=X(:, getSamples(numSamples, sampleSize));
 B = (Xsub * (( Xsub' * B) .^ 3)) / size(Xsub,2) - 3 * B;
case 13
 Ysub=X(:, getSamples(numSamples, sampleSize))' * B;
 Gpow3 = Ysub .^ 3;
 Beta = sum(Ysub .* Gpow3);
 D = diag(1 ./ (Beta - 3 * size(Ysub', 2)));
 B = B + myy * B * (Ysub' * Gpow3 - diag(Beta)) * D;

 % tanh
case 20
 hypTan = tanh(a1 * X' * B);
 B = X * hypTan / numSamples - ...
 ones(size(B,1),1) * sum(1 - hypTan .^ 2) .* B / numSamples * ...
 a1;
case 21
 Y = X' * B;
 hypTan = tanh(a1 * Y);
 Beta = sum(Y .* hypTan);
 D = diag(1 ./ (Beta - a1 * sum(1 - hypTan .^ 2)));
 B = B + myy * B * (Y' * hypTan - diag(Beta)) * D;
case 22
 Xsub=X(:, getSamples(numSamples, sampleSize));
 hypTan = tanh(a1 * Xsub' * B);
 B = Xsub * hypTan / size(Xsub, 2) - ...
 ones(size(B,1),1) * sum(1 - hypTan .^ 2) .* B / size(Xsub, 2) * a1;
case 23
 Y = X(:, getSamples(numSamples, sampleSize))' * B;
 hypTan = tanh(a1 * Y);
 Beta = sum(Y .* hypTan);
 D = diag(1 ./ (Beta - a1 * sum(1 - hypTan .^ 2)));
 B = B + myy * B * (Y' * hypTan - diag(Beta)) * D;

 % gauss
case 30
 U = X' * B;
 Usquared=U .^ 2;
 ex = exp(-a2 * Usquared / 2);
```

```
 gauss =  U .* ex;
 dGauss = (1 - a2 * Usquared) .*ex;
 B = X * gauss / numSamples - ...
 ones(size(B,1),1) * sum(dGauss)...
 .* B / numSamples ;
case 31
 Y = X' * B;
 ex = exp(-a2 * (Y .^ 2) / 2);
 gauss = Y .* ex;
 Beta = sum(Y .* gauss);
 D = diag(1 ./ (Beta - sum((1 - a2 * (Y .^ 2)) .* ex)));
 B = B + myy * B * (Y' * gauss - diag(Beta)) * D;
case 32
 Xsub=X(:, getSamples(numSamples, sampleSize));
 U = Xsub' * B;
 Usquared=U .^ 2;
 ex = exp(-a2 * Usquared / 2);
 gauss =  U .* ex;
 dGauss = (1 - a2 * Usquared) .*ex;
 B = Xsub * gauss / size(Xsub,2) - ...
 ones(size(B,1),1) * sum(dGauss)...
 .* B / size(Xsub,2) ;
case 33
 Y = X(:, getSamples(numSamples, sampleSize))' * B;
 ex = exp(-a2 * (Y .^ 2) / 2);
 gauss = Y .* ex;
 Beta = sum(Y .* gauss);
 D = diag(1 ./ (Beta - sum((1 - a2 * (Y .^ 2)) .* ex)));
 B = B + myy * B * (Y' * gauss - diag(Beta)) * D;

 % skew
case 40
 B = (X * ((X' * B) .^ 2)) / numSamples;
case 41
 Y = X' * B;
 Gskew = Y .^ 2;
 Beta = sum(Y .* Gskew);
 D = diag(1 ./ (Beta));
 B = B + myy * B * (Y' * Gskew - diag(Beta)) * D;
case 42
 Xsub=X(:, getSamples(numSamples, sampleSize));
 B = (Xsub * ((Xsub' * B) .^ 2)) / size(Xsub,2);
case 43
 Y = X(:, getSamples(numSamples, sampleSize))' * B;
 Gskew = Y .^ 2;
 Beta = sum(Y .* Gskew);
 D = diag(1 ./ (Beta));
 B = B + myy * B * (Y' * Gskew - diag(Beta)) * D;
```

```
     otherwise
      error('Code for desired nonlinearity not found!');
    end
  end

  % Calculate ICA filters.
  W = B' * whiteningMatrix;

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Also plot the last one...
  switch usedDisplay
   case 1
    % There was and may still be other displaymodes...
    % 1D signals
    icaplot('dispsig',(X'*B)');
    drawnow;
   case 2
    % ... and now there are :-)
    % 1D basis
    icaplot('dispsig',A');
    drawnow;
   case 3
    % ... and now there are :-)
    % 1D filters
    icaplot('dispsig',W);
    drawnow;
   otherwise
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEFLATION APPROACH
if approachMode == 2
  %% vectorSize means the number of rows
  B = zeros(vectorSize);

  % The search for a basis vector is repeated numOfIC times.
  %% round means the round-th IC
  round = 1;

  numFailures = 0;

  while round <= numOfIC,
  %% initialize parameters when estimate one of ICs
  %% myy is used for stabilized version
  %% notFine means the fine-tuning is not started
  %% long means whether the iterations are over 500 times
```

```
  %% endFinetuning means the limit of iterations (see below)
    myy = myyOrig;
    usedNlinearity = gOrig;
    stroke = 0;
    notFine = 1;
    long = 0;
    endFinetuning = 0;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Show the progress...
    if b_verbose, fprintf('IC %d ', round); end

    % Take a random initial vector of lenght 1 and orthogonalize it
    % with respect to the other vectors.
    %% w is one column of B, don't mix it with W, W=B' * whiteningMatrix
    if initialStateMode == 0
      w = randn (vectorSize, 1);
    elseif initialStateMode == 1
      w=whiteningMatrix*guess(:,round);
    end
    %% orthogonalize by subtracting the projection of w into B. Formula (44) in Hyvarinen's
paper
    w = w - B * B' * w;
    w = w / norm(w);
    %% wOld2 was used to detect stroke
    wOld = zeros(size(w));
    wOld2 = zeros(size(w));

    % This is the actual fixed-point iteration loop.
    %    for i = 1 : maxNumIterations + 1
    i = 1;
    %% gabba is used to specify the number of fine-tuning iterations
    %% this parameter is not available in symmetric estimation
    gabba = 1;
    while i <= maxNumIterations + gabba
      if (usedDisplay > 0)
    drawnow;
      end
      if (interruptible & g_FastICA_interrupt)
        if b_verbose
          fprintf('\n\nCalculation interrupted by the user\n');
        end
        return;
      end

      % Project the vector into the space orthogonal to the space
      % spanned by the earlier found basis vectors. Note that we can do
      % the projection with matrix B, since the zero entries do not
```

```
      % contribute to the projection.
      w = w - B * B' * w;
      w = w / norm(w);

      %% if fine-tuning is not started
      %% fine-tuning would be only be invoked when initial convergence is reached
      if notFine
    if i == maxNumIterations + 1
      if b_verbose
        fprintf('\nComponent number %d did not converge in %d iterations.\n', round,
maxNumIterations);
      end
      %% if the round-th estimation not converged, repeat the estimation until the failureLimit
is reached
      %% failureLimit is not available in symmetric approach
      %% round - 1 means the round-th IC would be re-estimated after non-convergence
      round = round - 1;
      numFailures = numFailures + 1;
      if numFailures > failureLimit
        if b_verbose
          fprintf('Too many failures to converge (%d). Giving up.\n', numFailures);
        end
        if round == 0
          A=[];
          W=[];
        end
        return;
      end
      % numFailures > failurelimit
      break;
    end
    % i == maxNumIterations + 1
      else
    % if notFine
    %% if fine-tuning is invoked and the MaxFinetune is reached, stop fine-tuning
    %% endFinetuning is equal to (initial convergence iterations + MaxFinetune)
    if i >= endFinetuning
      wOld = w; % So the algorithm will stop on the next test...
    end
      end
      % if notFine

      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      % Show the progress...
      if b_verbose, fprintf('.'); end;

      % Test for termination condition. Note that the algorithm has
      % converged if the direction of w and wOld is the same, this
```

```
        % is why we test the two cases.
        if norm(w - wOld) < epsilon | norm(w + wOld) < epsilon
          if finetuningEnabled & notFine
            if b_verbose, fprintf('Initial convergence, fine-tuning: '); end;
            notFine = 0;
        gabba = maxFinetune;
            wOld = zeros(size(w));
            wOld2 = zeros(size(w));
            usedNlinearity = gFine;
            myy = myyK * myyOrig;

        endFinetuning = maxFinetune + i;

          else %% fine-tuning not enabled or fine-tuning has already been done
            %% if the round-th IC is converged, reset the number of Failures
            numFailures = 0;
            % Save the vector
            B(:, round) = w;

            % Calculate the de-whitened vector.
            A(:,round) = dewhiteningMatrix * w;
            % Calculate ICA filter.
            W(round,:) = w' * whiteningMatrix;

            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % Show the progress...
            if b_verbose, fprintf('computed ( %d steps ) \n', i); end

            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % Also plot the current state...
            switch usedDisplay
         case 1
          if rem(round, displayInterval) == 0,
            % There was and may still be other displaymodes...
            % 1D signals
            temp = X'*B;
            icaplot('dispsig',temp(:,1:numOfIC)');
            drawnow;
          end
         case 2
          if rem(round, displayInterval) == 0,
            % ... and now there are :-)
            % 1D basis
            icaplot('dispsig',A');
            drawnow;
          end
         case 3
          if rem(round, displayInterval) == 0,
```

```
      % ... and now there are :-)
      % 1D filters
      icaplot('dispsig',W);
      drawnow;
    end
      end
  % switch usedDisplay
  break; % IC ready - next...
    end
%if finetuningEnabled & notFine
  elseif stabilizationEnabled
if (~stroke) & (norm(w - wOld2) < epsilon | norm(w + wOld2) < ...
        epsilon)
  stroke = myy;
  if b_verbose, fprintf('Stroke!'); end;
  myy = .5*myy;
  if mod(usedNlinearity,2) == 0
    usedNlinearity = usedNlinearity + 1;
  end
elseif stroke
  myy = stroke;
  stroke = 0;
  if (myy == 1) & (mod(usedNlinearity,2) ~= 0)
    usedNlinearity = usedNlinearity - 1;
  end
elseif (notFine) & (~long) & (i > maxNumIterations / 2)
  if b_verbose, fprintf('Taking long (reducing step size) '); end;
  long = 1;
  myy = .5*myy;
  if mod(usedNlinearity,2) == 0
    usedNlinearity = usedNlinearity + 1;
  end
end
  end

  wOld2 = wOld;
  wOld = w;

  switch usedNlinearity
% pow3
   case 10
w = (X * ((X' * w) .^ 3)) / numSamples - 3 * w;
   case 11
EXGpow3 = (X * ((X' * w) .^ 3)) / numSamples;
Beta = w' * EXGpow3;
w = w - myy * (EXGpow3 - Beta * w) / (3 - Beta);
   case 12
Xsub=X(:,getSamples(numSamples, sampleSize));
```

```
    w = (Xsub * ((Xsub' * w) .^ 3)) / size(Xsub, 2) - 3 * w;
       case 13
    Xsub=X(:,getSamples(numSamples, sampleSize));
    EXGpow3 = (Xsub * ((Xsub' * w) .^ 3)) / size(Xsub, 2);
    Beta = w' * EXGpow3;
    w = w - myy * (EXGpow3 - Beta * w) / (3 - Beta);
    % tanh
       case 20
    hypTan = tanh(a1 * X' * w);
    w = (X * hypTan - a1 * sum(1 - hypTan .^ 2)' * w) / numSamples;
       case 21
    hypTan = tanh(a1 * X' * w);
    Beta = w' * X * hypTan;
    w = w - myy * ((X * hypTan - Beta * w) / ...
                (a1 * sum((1-hypTan .^2)') - Beta));
       case 22
    Xsub=X(:,getSamples(numSamples, sampleSize));
    hypTan = tanh(a1 * Xsub' * w);
    w = (Xsub * hypTan - a1 * sum(1 - hypTan .^ 2)' * w) / size(Xsub, 2);
       case 23
    Xsub=X(:,getSamples(numSamples, sampleSize));
    hypTan = tanh(a1 * Xsub' * w);
    Beta = w' * Xsub * hypTan;
    w = w - myy * ((Xsub * hypTan - Beta * w) / ...
                (a1 * sum((1-hypTan .^2)') - Beta));
    % gauss
       case 30
    % This has been split for performance reasons.
    u = X' * w;
    u2=u.^2;
    ex=exp(-a2 * u2/2);
    gauss =  u.*ex;
    dGauss = (1 - a2 * u2) .*ex;
    w = (X * gauss - sum(dGauss)' * w) / numSamples;
       case 31
    u = X' * w;
    u2=u.^2;
    ex=exp(-a2 * u2/2);
    gauss =  u.*ex;
    dGauss = (1 - a2 * u2) .*ex;
    Beta = w' * X * gauss;
    w = w - myy * ((X * gauss - Beta * w) / ...
                (sum(dGauss)' - Beta));
       case 32
    Xsub=X(:,getSamples(numSamples, sampleSize));
    u = Xsub' * w;
    u2=u.^2;
    ex=exp(-a2 * u2/2);
```

```
    gauss =  u.*ex;
    dGauss = (1 - a2 * u2) .*ex;
    w = (Xsub * gauss - sum(dGauss)' * w) / size(Xsub, 2);
       case 33
    Xsub=X(:,getSamples(numSamples, sampleSize));
    u = Xsub' * w;
    u2=u.^2;
    ex=exp(-a2 * u2/2);
    gauss =  u.*ex;
    dGauss = (1 - a2 * u2) .*ex;
    Beta = w' * Xsub * gauss;
    w = w - myy * ((Xsub * gauss - Beta * w) / ...
                (sum(dGauss)' - Beta));
    % skew
       case 40
    w = (X * ((X' * w) .^ 2)) / numSamples;
       case 41
    EXGskew = (X * ((X' * w) .^ 2)) / numSamples;
    Beta = w' * EXGskew;
    w = w - myy * (EXGskew - Beta*w)/(-Beta);
       case 42
    Xsub=X(:,getSamples(numSamples, sampleSize));
    w = (Xsub * ((Xsub' * w) .^ 2)) / size(Xsub, 2);
       case 43
    Xsub=X(:,getSamples(numSamples, sampleSize));
    EXGskew = (Xsub * ((Xsub' * w) .^ 2)) / size(Xsub, 2);
    Beta = w' * EXGskew;
    w = w - myy * (EXGskew - Beta*w)/(-Beta);

       otherwise
    error('Code for desired nonlinearity not found!');
      end

      % Normalize the new w.
      w = w / norm(w);
      i = i + 1;
    end
    round = round + 1;
end
if b_verbose, fprintf('Done.\n'); end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Also plot the ones that may not have been plotted.
if (usedDisplay > 0) & (rem(round-1, displayInterval) ~= 0)
  switch usedDisplay
   case 1
    % There was and may still be other displaymodes...
    % 1D signals
```

```
      temp = X'*B;
      icaplot('dispsig',temp(:,1:numOfIC)');
      drawnow;
     case 2
      % ... and now there are :-)
      % 1D basis
      icaplot('dispsig',A');
      drawnow;
     case 3
      % ... and now there are :-)
      % 1D filters
      icaplot('dispsig',W);
      drawnow;
     otherwise
    end
  end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In the end let's check the data for some security
if ~isreal(A)
  if b_verbose, fprintf('Warning: removing the imaginary part from the result.\n'); end
  A = real(A);
  W = real(W);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Subfunction
% Calculates tanh simplier and faster than Matlab tanh.
function y=tanh(x)
y = 1 - 2 ./ (exp(2 * x) + 1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% max means the total sample size, percentage means the proportion of sample size
%% rand generates `max` random numbers following uniform distribution at the interval of (0,1)
%% the function returns the position or index of selected samples
function Samples = getSamples(max, percentage)
Samples = find(rand(1, max) < percentage);
```