```
function [theta,lambda,sigma] = deafun(x,y,varargin)
```

```
% DEAFUN data envelopment analysis function.
%   theta = DEAFUN(x,y) attempts to solve the data envelopment problem
%   for (n) DMUs whose (m) inputs and (s) outputs are denoted by the
%   vectors x(n,m) and y(n,s) respectively). DEAFUN returns theta, the
%   technical efficiency score. By default the returns-to-scale are set
%   to Constant (CRS), while the orientation is set to input.
%
%   theta = DEAFUN(x,y,options) the functions allows the user to set
%   three additional options: 'scale', which determines the
%   returns-to-scale (CRS, VRS or NIRS); 'orientation', which determines
%   the orientation (input or output) and 'roundoff', which determines
%   whether the results should be rounded or if the entire double should
%   be supplied
%
%   [theta,lambda] = DEAFUN(x,y,options) returns the relevant frontier
%   (lambda) for each individual DMU.
%
%   [theta,lambda,sigma] = DEAFUN(x,y,options) returns the inputs and
%   output slacks (sigma). Note that DEAFUN will return the slacks in
%   matrix form, with the output slacks before the input slacks
%   (i.e. [y x]);
%
%   Example 1:
%   >> x = [2 5;
%           2 4;
%           6 6;
%           3 2;
%           6 2];
%   y = [1; 2; 3; 1; 2];
%   [theta,lambda,sigma] = deafun(x,y,'scale','CRS')
%
%   theta =
%     0.5000
%     1.0000
%     0.8333
%     0.7143
%     1.0000
%   lambda =
%     0.0000    0.5000    0.0000    0.0000    0.0000
%     0.0000    1.0000    0.0000    0.0000    0.0000
%     0.0000    1.0000    0.0000    0.0000    0.5000
%     0.0000    0.2143    0.0000    0.0000    0.2857
%     0.0000    0.0000    0.0000    0.0000    1.0000
%   sigma =
%     -0.0000    0.0000    0.5000
%     -0.0000    0.0000    0.0000
%     -0.0000    0.0000    0.0000
%     -0.0000    0.0000    0.0000
%     -0.0000    0.0000    0.0000
%
```

```
%   Where, the minimized (frontier) output and inputs are:
%   >> lambda*[y x]+sigma
%   ans =
%     1.0000    1.0000    2.0050
%     2.0000    2.0000    4.0000
%     3.0000    5.0000    5.0000
%     1.0000    2.1429    1.4286
%     2.0000    6.0000    2.0000
%
%   And the efficiency scores (theta) are given by:
%   >> (lambda*[y x]+sigma) ./ [y x]
%   ans =
%     1.0000    0.5000    0.5000
%     1.0000    1.0000    1.0000
%     1.0000    0.8333    0.8333
%     1.0000    0.7143    0.7143
%     1.0000    1.0000    1.0000
%
%   Example 2:
%   x = [2; 4; 3; 5; 6];
%   y = [1; 2; 3; 4; 5];
%   theta = deafun(x,y,'scale','VRS')
%
%   theta =
%     1.0000
%     0.6250
%     1.0000
%     0.9000
%     1.0000
```

%% User defined variables;

```
epsilon = 1E-3;     % epsilon: non archimedian number for the DEA models;
rho = 1E3;          % rho: roundoff decimal;
```

%% Check arguments;

```
% check minimum number of arguments (2)
if nargin < 2
  error('insufficient number of arguments');
end
% check for empty input values;
if any(all(x <= 0,2))
  error('x contains rows with all elements equal to zero');
elseif any(any(x <= 0,2))
  warning('x contains elements equal to zero');
end
% check for empty output values;
if any(all(y <= 0,2))
  error('y contains rows with all elements equal to zero');
elseif any(any(y <= 0,2))
```

```
  warning('y contains elements equal to zero');
end

% default options
options = struct( ...
  'scale',        'CRS', ...
  'orientation',  'input', ...
  'roundoff',     false);
% get acceptable names
optionNames = fieldnames(options);
% check number of option arguments (multiple of 2)
nArgs = length(varargin);
if rem(nArgs,2)
  error('options require name/value pairs');
end
% set user defined options
for pair = reshape(varargin,2,[]) % pair is {propName;propValue}
  inpName = lower(pair{1}); % case insensitive property name
  if any(strcmp(inpName,optionNames))
    options.(inpName) = lower(pair{2}); % lower case property value
  else
    error('%s is not a recognized parameter name',inpName);
  end
end
% convert structure to variables for parfor loop
scale = options.scale;
orient = options.orientation;
roundoff = options.roundoff;
```

%% Prepare data;

```
% Shift dimension for DEAFUN to first non-singleton dimension or the
%   first corresponding dimension.
if numel(size(x)) > 2
  error('No. of dimensions greater than 2');
elseif all(size(x) == size(y)) % get first non-singleton dimension
  dim = find(size(x) ~= 1, 1)-1;
elseif any(size(x) == size(y)) % get first corresponding dimension
  dim = find(size(x) == size(y), 1)-1;
else
  error('No. of rows in x not equal to y');
end
% Transpose x and y (if necessary)
if dim
  x = x';
  y = y';
end

% Extracts the number of DMUs, inputs and outputs;
[n,m] = size(x);
[~,s] = size(y);
```

```
% Normalize input: set lowest x and y equal to 1/100 of min. val. [AD HOC];
if m < s
  div = min(x);
else
  div = min(y);
end
x = x ./ (div * 100);
y = y ./ (div * 100);
```

%% Compute results;

```
% The results from the selected model are in matrix Z;
Z = zeros(n,n+m+s+1);

lblambda = zeros(n,1);               % Lower bounds for (n) lambdas;
lboutput = zeros(s,1);               % Lower bounds for (s) output slacks;
lbinput  = zeros(m,1);               % Lower bounds for (m) input slacks;
switch orient;
  case ('output')
    % Objective function of the model: min(0*lambda -epsilon*(s+ + m-) - theta);
    f = [zeros(1,n) -epsilon*ones(1,s) -epsilon*ones(1,m) -1];
    % Lower bounds for lambdas (n), outputs (s+), inputs (m-) and theta;
    lb = [lblambda; lboutput; lbinput; 1];
  case ('input')
    % Objective function of the model: min(0*lambda -epsilon*(s+ + m-) + theta);
    f = [zeros(1,n) -epsilon*ones(1,s) -epsilon*ones(1,m) 1];
    % Lower bounds for lambdas (n), outputs (s+), inputs (m-) and theta;
    lb = [lblambda; lboutput; lbinput; 0];
end

parfor j=1:n % Loop DMUs;
  A = []; b = []; Aeq = []; beq = []; %#ok<NASGU> % clear temporary variables
  switch orient;
    case ('output')
      switch scale;
        case ('crs')     % Constant Returns-to-Scale (CRS);
          A = [];
          b = [];
          Aeq = [-y', eye(s,s), zeros(s,m), y(j);
                 x', zeros(m,s), eye(m,m), zeros(m,s)];
          beq = [0; x(j,:)'];
        case ('nirs')    % Non-Increasing Returns-to-Scale (NIRS);
          A = [ones(1,n), zeros(1,s+m), 0];
          b = 1;
          Aeq = [-y', eye(s,s), zeros(s,m), y(j);
                 x', zeros(m,s), eye(m,m), zeros(m,s)];
          beq = [0; x(j,:)'];
        case ('vrs')     % Variable Returns-to-Scale (VRS);
          A = [];
          b = [];
          Aeq = [-y', eye(s,s), zeros(s,m), y(j);
                 x', zeros(m,s), eye(m,m), zeros(m,s);
                 ones(n,1)', zeros(s+m+1,1)'];
```

```
            beq = [0; x(j,:)'; 1];
          otherwise
            error('scale variable unknown or empty; please set scale to CRS, VRS or NIRS...');
        end
      case ('input')
        switch scale;
          case ('crs')     % Constant Returns-to-Scale (CRS);
            A = [zeros(1,n), zeros(1,s+m), 1;
                zeros(1,n), zeros(1,s+m), -1];
            b = [1;0];
            Aeq = [y', -eye(s,s), zeros(s,m+1);
                  -x', zeros(m,s), -eye(m,m), x(j,:)'];
            beq = [y(j,:)'; zeros(m,1)];
          case ('nirs')    % Non-Increasing Returns-to-Scale (NIRS);
            A = [ones(1,n), zeros(1,s+m), 0];
            b = 1;
            Aeq = [y', -eye(s,s), zeros(s,m+1);
                  -x', zeros(m,s), -eye(m,m), x(j,:)'];
            beq = [y(j,:)'; zeros(m,1)];
          case ('vrs')     % Variable Returns-to-Scale (VRS);
            A = [];
            b = [];
            Aeq = [y', -eye(s,s), zeros(s,m+1);
                  -x', zeros(m,s), -eye(m,m), x(j,:)';
                  ones(n,1)', zeros(s+m+1,1)'];
            beq = [y(j,:)'; zeros(m,1); 1];
          otherwise
            error('scale variable unknown or empty; please set scale to CRS, VRS or NIRS...')
        end
      otherwise
        error('orientation unknown; please set orientation to input or output')
    end

    % LINPROG linear programming
    optim=optimset('Display', 'off');           % Mute linprog output;
    z = linprog(f,A,b,Aeq,beq,lb,[],[],optim);  % Run LINPROG for DMU (j);
    if strcmp(orient,'output');
      z(n+m+s+1) = z(n+m+s+1).^(-1)             % From gamma to theta;
    end
    Z(j,:) = z;                                 % Write results for DMU (j);
  end

  if roundoff % Roundoff results;
    Z = arrayfun(@round,Z*rho)/rho;
  end

  theta = Z(:,end);             % Technical efficiency (theta);
  lambda = Z(:,1:n);            % Relevant frontier (lambda);
  sigma = Z(:,n+1:n+s+m) .* ones(n,s+m) * 100; % Slacks (sigma) [AD HOC];

end
```