Appendix

Algorithm Compendium

This appendix compiles many algorithms described in this book, most of which have been developed in the Remote Sensing Signal and Image Processing Laboratory (RSSIPL) at the University of Maryland, Baltimore County. In order to help readers implement these important algorithms, their MATLAB codes are also included for reference so that readers can write their own program codes without relying on software packages such as ENVI, ERDAS, etc. Each algorithm is described according to its functionality and its categorization and is then followed by its MATLAB codes.

A.1 Estimation of Virtual Dimensionality

The concept of virtual dimensionality (VD) was first coined in the book by Chang (2003a) and later published by Chang and Du (2004). It is defined as the number of spectrally distinct signatures present in the data and has received considerable interest in unsupervised hyperspectral data exploitation since it was introduced in 2003. Many approaches have been developed for estimating the value of VD. Nevertheless, the most popular algorithm to be used for this purpose is the one developed by Harsanyi et al. (1994a), whose idea was the origin of VD. All the details of VD along with techniques developed to estimate VD can be found in Chapter 5.

A.1.1 Harsanyi–Farrand–Chang Method

- **Algorithm name:** Harsanyi–Farrand–Chang (HFC) method
- **Authors:** J. C. Harsanyi and Chein-I Chang
- **Category:** preprocessing
- **Designed criteria:** eigenvlaues of sample correlation/covariance matrix
- **Designed method:** Neyman–Pearson detection theory
- **Typical use (LOI's addressed):** estimation of number of spectral distinct signatures
- **Inputs:** reflectance or radiance cube
- **Outputs:** a positive integer and false alarm probability
- **Assumptions:** no prior knowledge required
- **Sensitivity to LOI (target knowledge):** moderate
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational

Hyperspectral Data Processing: Algorithm Design and Analysis, First Edition. Chein-I Chang.

- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The HFC method was first envisioned in Harsanyi et al. (1994a) to detect spectral signatures present in AVIRIS data. It was then used to find the now-popular terminology, VD (Chapter 17 in Chang (2003a) and Chapter 5 in this book), which is defined as the number of spectral distinct signatures and later published in Chang and Du (2004). It calculates the difference between eigenvlaues in sample correlation matrix and sample covariance matrix and makes use of Neyman–Pearson detector to determine the value of VD.

MATLAB Codes of HFC Method

```
function number=HFC(HIM,t);
%
% HFC gives the VD number estimated by given false alarm property using HFC
% method.
%
% There are two parameters,HFC(HIM,t) where the HIM is the
% Hyperspectral image cube, which is a 3-D data matrix
% [XX,YY,bnd] = size(HIM), XX YY are the image size,
% bnd is the band number of the image cube.
% t is the false alarm probability.
%
% HFC uses the HFC algorithm developed by Dr. Chein-I Chang,
% see http://www.umbc.edu/rssipl/. The Matlab code was
% programmed by Jing Wang in Remote Sensing Signal and
% Image Processing Lab.
%

[XX,YY,bnd] = size(HIM);
pxl_no = XX*YY;
r = (reshape(HIM,pxl_no,bnd))';

R = (r*r')/pxl_no;
u = mean(r,2);
K = R-u*u';

%======HFC=====
D1=sort(eig(R));
D2=sort(eig(K));
sita=sqrt((D1.^2+D2.^2)*2/pxl_no);
P_fa=t;
Threshold=(sqrt(2))*sita*erfinv(1-2*P_fa);

Result=0;
for m=1:bnd
  if ((D1(m,1)-D2(m,1))>Threshold(m,1))
    Result(m,1)=1;
```

```
  else
    Result(m,1)=0;
  end
end
fprintf('The VD number estimated is');
disp(sum(Result));
number=sum(Result);
```

A.1.2 Noise-Whitened Harsany–Farrand–Chang (NWHFC) Method

- **Algorithm name:** noise-whitened Harsanyi–Farrand–Chang (HFC) method.
- **Authors:** Chein-I Chang
- **Category:** preprocessing
- **Designed criteria:** eigenvlaues of sample correlation/covariance matrix
- **Designed method:** Neyman–Pearson detection theory
- **Typical use (LOI's addressed):** estimation of number of spectral distinct signatures
- **Inputs:** reflectance or radiance cube
- **Outputs:** a positive integer and false alarm probability
- **Assumptions:** no prior knowledge required
- **Sensitivity to LOI (target knowledge):** moderate
- **Sensitivity to noise:** low
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The NWHFC method is a noise-whitened version of the HFC method and was developed in Chang (2003a) and Chang and Du (2004). It requires a reliable method to estimate the noise covariance matrix.

MATLAB Codes of the NWHFC Method

```
function number=NWHFC(HIM,t)
%
% NWHFC gives the VD number estimated by given false alarm property using
% NWHFC method.
%
% There are two parameters, NWHFC(HIM,t) where the HIM is the
% Hyperspectral image cube, which is a 3-D data matrix
% [XX,YY,bnd] = size(HIM), XX YY are the image size,
% bnd is the band number of the image cube.
% t is the false alarm probability.
%
% HFC uses the NWHFC algorithm developed by Dr. Chein-I Chang,
% see http://www.umbc.edu/rssipl/. The Matlab code was
% programmed by Jing Wang in Remote Sensing Signal and
% Image Processing Lab.
%
```

```
[XX,YY,bnd] = size(HIM);
pxl_no = XX*YY;
r = (reshape(HIM,pxl_no,bnd))';

R = (r*r')/pxl_no;
u = mean(r,2);
K = R-u*u';

%======Noise estimation=====
K_Inverse=inv(K);
tuta=diag(K_Inverse);
K_noise=1./tuta;
K_noise=diag(K_noise);

%=====Noise whitening===
y=inv(sqrtm(K_noise))*r;
y=reshape(y',XX,YY,bnd);
%=====Call HFC to estimate===
number=HFC(y,t);
```

A.2 Data Sphering

The purpose of data sphering is to allow users to analyze data structure characterized by high-order statistics. Before doing so, the data samples characterized by the first two orders of statistics, that is, mean and variances/covariances must be removed. The data sphering is designed to accomplish this task. It first removes the data sample mean by setting data set centered at the origin and then de-correlates data samples by zeroing all covariances via diagonalization of data sample covariance matrix. Finally, it normalizes data sample variances to 1 by placing all de-correlated data samples on the unit sphere. So, by means of matrix diagonalization and variance normalization, all data samples characterized by high-order statistics are either inside the sphere characterized by sub-Gaussian or outside the sphere characterized by super-Gaussian. Technically speaking, a whitening processing is a part of data sphering that only de-correlates data samples without normalization. However, in statistical signal processing and communications community as well as in many application whitening is indeed data sphering. In this book, we particularly make a distinction between them. In other words, whitening only de-correlates data samples by making co-variances zero but does not normalize variances to 1. It is only a part of data sphering. Details of data sphering can be found in Chapter 6.

- **Algorithm name:** data sphering
- **Category:** preprocessing
- **Designed criteria:** eigenvalues
- **Designed method:** sample covariance matrix
- **Typical use (LOI's addressed):** preprocessing
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale images
- **Assumptions:** no prior knowledge required
- **Sensitivity to LOI (target knowledge):** high to high-order statistics

- **Sensitivity to noise:** low
- **Operating Bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  Data sphering, also known as a whitening processing in statistical signal processing and communication, is a commonly used method to remove data sample vectors characterized by the first two-order statistics. It is a required preprocessing step prior to ICA (Hyvarinen and Oja, 2001).

MATLAB Codes of Data Sphering

```
function sphered_data=data_sphering(HIM)

% Initial variables
[row column band]=size(HIM);
% Center the data
HIM_row=reshape(HIM,row*column,band);
HIM_zeromean=HIM_row-repmat(mean(HIM_row),row*column,1);
cov=HIM_zeromean'*HIM_zeromean/(row*column);

% Eigen decomposition
[V D]=eig(cov);

% Transform the data set
for i=1:band,
  sphered_data(i,:)=(V(:,i)'*HIM_zeromean')./(D(i,i)^.5*row);
end

% Transform the data back
sphered_data=reshape(sphered_data',row,column,band);
```

A.3 Dimensionality Reduction by Transform

Four transformations are used to perform data dimensionality reduction (DR). Two are of second-order statistics-based transformations, data variance-based principal components analysis (PCA), and signal-to-noise ratio (SNR)-based maximum noise fraction (MNF). The other two are of high-order statistics (HOS)-based transformations, independent component analysis (ICA), and high-order statistical moment-based transforms. Details of these four DR transformations can be found in Chapter 6.

A.3.1 PCA

- **Algorithm name:** principal components analysis (PCA)
- **Category:** component analysis-based transform
- **Designed criteria:** eigenvalues
- **Designed method:** sample covariance matrix

- **Typical use (LOI's addressed):** data representation by eigenvectors
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale images
- **Assumptions:** no prior target knowledge required
- **Sensitivity to LOI (target knowledge):** targets of high-order statistics
- **Sensitivity to Noise:** low
- **Operating Bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  PCA (Gonzalez and Woods, 2002) is probably the most widely used component analysis transform that allows users to data in terms of eigenvalues/eigenvectors via eigen-decomposition where data are preserved and retained according to data variances in the descending order.

MATLAB Codes of PCA

```
function [PCs]=PCA(HIM,M);

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

x=reshape(HIM,xx*yy,bnd);
x=x';
L=size(x,1);
K=size(x,2);

u=mean(x,2); %dimension of u is 1*L
x_hat=x-u*ones(1,K);
m=mean(x,2);
C=(x*x')/size(x,2)-m*m';
%===========
[V,D]=eig(C);

d=(diag(D))';
[oo,Index]=sort(d);

for m=1:L
  D_sort(1,m)=d(1,Index(1,Lm));
  V_sort(:,m)=V(:,Index(1,Lm));
end

D=diag(D_sort);
V=V_sort;

D=D(1:M,1:M);
V=V(:,1:M);
```

```
%====for the matrix with full column rank, so the
A=V';
x_whitened=A*(x_hat);
PCs=x_whitened;
```

A.3.2 MNF

- **Algorithm name:** maximum noise fraction (MNF)
- **Authors:** A.A. Green, M. Berman, P. Switzer, and M.D. Craig
- **Category:** component analysis-based transform
- **Designed criteria:** signal-to-noise ratio
- **Designed method:** sample covariance matrix
- **Typical use (LOI's addressed):** data representation by SNR
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale images
- **Assumptions:** no prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high-order statistics
- **Sensitivity to Noise:** high
- **Operating Bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  MNF was developed by Green et al. (1988) and further modified by Lee et al. (1990) and referred to as Noise Adjusted Principal Component (NAPC) transform. It represents data in terms of SNR rather than data variances as PCA does.

MATLAB Codes of MNF

```
function [ImageCub_MNF,Matrix_of_Vector]=MNF(ImageCub);
Last_volumn=-10000000000.0000001;
[height,width,NumberOfSpectrum]=size(ImageCub);
ImageCub_MNF=zeros(height,width,NumberOfSpectrum);
% ——————————— begin to compute Matrix_of_Vector ——————————— %
meanSpect=zeros(NumberOfSpectrum,1);
for II=1:height
  for JJ=1:width
    meanSpect=meanSpect+squeeze(ImageCub(II,JJ,:))/height/width;
  end
end

TotalCovariance=zeros(NumberOfSpectrum,NumberOfSpectrum);

for II=1:height
  for JJ=1:width
    TotalCovariance=TotalCovariance+(squeeze(ImageCub(II,JJ,:))-
meanSpect)*(squeeze(ImageCub(II,JJ,:))-meanSpect)'/height/width;
  end
```

```
end
Matrix_F=zeros(NumberOfSpectrum,NumberOfSpectrum);
Cov_inv=inv(TotalCovariance);
for II=1:NumberOfSpectrum
  Matrix_F(II,II)=sqrt(Cov_inv(II,II));
end
adjusted_Cov=Matrix_F'*TotalCovariance*Matrix_F;

[V,D]=eig(adjusted_Cov);
eig_value=zeros(NumberOfSpectrum,2);
for II=1:NumberOfSpectrum
  eig_value(II,1)=D(II,II);
  eig_value(II,2)=II;
end
%disp(eig_value);
V_sort_min_to_max=sortrows(eig_value,1);
Matrix_of_Vector_before=zeros(NumberOfSpectrum,NumberOfSpectrum);
for II=1:NumberOfSpectrum
   Matrix_of_Vector_before(:,II)=squeeze(V(:,V_sort_min_to_max(NumberOf-
Spectrum-II+1,2)));
end
Matrix_of_Vector=Matrix_F*Matrix_of_Vector_before;
% ———————————— end of computing Matrix_of_Vector ———————————— %
for II=1:height
  for JJ=1:width
    r=squeeze(ImageCub(II,JJ,:))-meanSpect;
    ImageCub_MNF(II,JJ,:)=Matrix_of_Vector'*r;
  end
end
```

A.3.3 ICA

- **Algorithm name:** ICA
- **Category:** component analysis-based transform
- **Designed criteria:** infinite-order statistics
- **Designed method:** blind source separation via a linear mixture model
- **Typical use (LOI's addressed):** no prior knowledge is required
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale images
- **Assumptions:** no prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** low
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**

ICA (Hyvarinen et al., 2001) is widely used in blind source separation via a linear mixture model, which assumes that there exists at most one Gaussian source. A fast algorithm developed by Hyvarinen and Oja (1997), called FastICA, is the most used algorithm to implement ICA.

When ICA is implemented to perform DR, it suffers from a serious issue that the results are not repeatable and inconsistent due to its use of random initial conditions. As a result, the ICA-generated independent components (ICs) generally appear in a random order. In this case, the IC appearing earlier does not necessarily imply that it is more important or significant than that generated later. In order to resolve this issue, three versions of ICA, referred to as ICA-DR1, ICA-DR2, and ICA-DR3, have been proposed by Wang and Chang (2006a, 2006b) and re-named as statistics prioritized ICA-DR (SPICA-DR), random ICA (RICA-DR), and initialization driven ICA-DR (IDICA-DR) in Sections 6.4.1–6.4.3, respectively. In all the three algorithms, an algorithm developed by Hyvarinen and Oja (1997), called FastICA, is modified and implemented in MATLAB codes to realize ICA. To distinguish it from the original FastICA, it is named "My FastICA" and is provided as follows.

MATLAB Codes of My FastICA

```
function [ICs]=My_fastica_v5(HIM,M);

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

x=reshape(HIM,xx*yy,bnd);
x=x';
L=size(x,1);
K=size(x,2);
%====Data sphering =====
u=mean(x,2); %dimension of u is 1*L
x_hat=x-u*ones(1,K);
m=mean(x,2);
C=(x*x')/size(x,2)-m*m';
%===========
[V,D]=eig(C);
A=inv(sqrtm(D))*V';  % A is the whitening matrix....
x_whitened=A*(x_hat);
%=======
clear x;
clear x_hat;
%====rank the eigenvalues, which is used for using the eigenvector as
%initialization
%
% d=(diag(D))';
% [oo,Index]=sort(d);
%
%
% for m=1:L
```

```
%  D_sort(1,m)=d(1,Index(1,L+1-m));
%  V_sort(:,m)=V(:,Index(1,L+1-m));
% end
%
% D=diag(D_sort);
% V=V_sort;
%=====

%====Sphering finished

threshold = 0.0001;
B=zeros(L);
for round=1:M
  fprintf('IC %d', round);
  %===initial condition ===
 w=rand(L,1)-0.5;
% w=V(:,round);   Eigenvectors initialization
  %===
  w=w-B*B'*w;
  w=w/norm(w);
  wOld=zeros(size(w));
  wOld2=zeros(size(w));
  i=1;
  while i<=1000
    w=w-B*B'*w;
    w=w/norm(w);
    fprintf('.');
    if norm(w-wOld)<threshold | norm(w+wOld)<threshold
      fprintf('Convergence after %d steps\n', i);
      B(:,round)=w;
      W(round,:)=w';
      break;
    end
    wOld2=wOld;
    wOld=w;
    w=(x_whitened*((x_whitened'*w).^3))/K-3*w;
    w=w/norm(w);
    i=i+1;
  end
  if (i>1000)
     fprintf('Warning! can not converge after 1000 steps \n, no more
components');
     break;
  end
  round=round+1;
end

ICs=W*x_whitened;
```

```
% figure;
% for m=1:M
% s=reshape(abs(ICs(m,:)),xx,yy); subplot(6,8,m);
% imagesc(s); axis off;colormap(gray);
% end
```

MATLAB Codes of SPICA-DR (ICA-DR1)

```
function [IC_sorted]=sort_IC_DR1(HIM,M);
clear J;

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);
[ICs]=My_fastica_v5(HIM,M);
ICs=abs(ICs);

% %====Show the ICs in the original order===
% figure;
% for m=1:size(ICs,1)
%   s=reshape(abs(ICs(m,:)),xx,yy);
%   s=255*(s-min(min(s))*ones(size(s,1),size(s,2)))/(max(max(s))
-min(min(s)));
%   temp=mean(reshape(s,xx*yy,1));
%   subplot(6,8,m); imshow(uint8(s));
% %   title(m);
% %
% end

%======Calculate the contrast function
for m=1:size(ICs,1);
  s=ICs(m,:);
  var1=var(s);
  mean1=mean(s);
  sita=sqrt(var1);
  skew_temp=sum((s-mean1).^3)/(xx*yy-1);
  kurt_temp=sum((s-mean1).^4)/(xx*yy-1);

  J(1,m)=(skew_temp.^2)/12+((kurt_temp-3).^2)/48;

end

%======IC sorting ========
[a,b]=sort(J);
b=flipud(b');
IC_sorted=ICs(b',:);

%=========Show the ICS after sorting...
```

```
figure;

for m=1:size(ICs,1)
  s=reshape(abs(IC_sorted(m,:)),xx,yy);
  s=255*(s-min(min(s))*ones(size(s,1),size(s,2)))/(max(max(s))
-min(min(s)));
  temp=mean(reshape(s,xx*yy,1));
  subplot(6,8,m); imshow(uint8(s));
 % title(m);
%
end
```

MATLAB Codes of RICA-DR (ICA-DR2)

```
function [IC_selected]=sort_IC_DR2(HIM,M,run_times);

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

fprintf('first ICA run \n');
[ICs]=My_fastica_v5(HIM,M*2);
set1=abs(ICs);
set_com=set1;

size1=ones(1);
for round=1:run_times
  fprintf('ICA run, order is %d \n',round+1);
  [ICs]=My_fastica_v5(HIM,2*M);
  set2=abs(ICs);
  set_com_new=[];
  distance=0;
  for m=1:size(set_com,1)
    for n=1:size(set2,1)
      temp1=sqrt(sum(set_com(m,:).^2)); % SAM
      temp2=sqrt(sum(set2(n,:).^2));
      distance(m,n)=acos(sum(set_com(m,:).*set2(n,:))/(temp1*temp2));
    end
    t=distance(m,:)<=0.5;
    if (sum(t)>=1)
       set_com_new=cat(1,set_com_new,set_com(m,:));
    end
  end
  set_com=set_com_new;
  fprintf('the size of set_Com is');
  size(set_com_new)
  size1(round,1)=size(set_com_new,1);
  if (size(set_com_new,1)<=M)
```

```
    break;
  end

end
IC_selected=set_com;
```

MATLAB Codes of IDICA-DR (ICA-DR3)

```
function [Loc,Sig]=My_ATGP(HIM,M);

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

r=reshape(HIM,xx*yy,bnd);
r=r';
%=====Find the first point

temp=sum(r.*r);
[a,b]=max(temp);

if (rem(b,xx)==0)
  Loc(1,1)=b/xx;
  Loc(1,2)=xx;
elseif (floor(b/xx)==0)
  Loc(1,1)=1;
  Loc(1,2)=b;
else
  Loc(1,1)=floor(b/xx)+1; % y
  Loc(1,2)=b-xx*floor(b/xx); % x
end

Sig(:,1)=r(:,b);
fprintf('1\n');
%==========
for m=2:M

  U=Sig;
  P_U_perl=eye(bnd)-U*inv(U'*U)*U';
  y=P_U_perl*r;
  temp=sum(y.*y);
  [a,b]=max(temp);
  if (rem(b,xx)==0)
    Loc(m,1)=b/xx;
    Loc(m,2)=xx;
  elseif (floor(b/xx)==0)
    Loc(m,1)=1;
    Loc(m,2)=b;
```

```
  else
    Loc(m,1)=floor(b/xx)+1; % y
    Loc(m,2)=b-xx*floor(b/xx); % x
  end
  Sig(:,m)=r(:,b);
  disp(m)
end
%
%figure; imagesc(HIM(:,:,30)); colormap(gray); hold on
%axis off
%axis equal
%for m=1:size(Loc,1)
%  plot(Loc(m,1),Loc(m,2),'o','color','g');
%  text(Loc(m,1)+2,Loc(m,2),num2str(m),'color','y','FontSize',12);
%end
%
```

Since IDICA-DR requires an initialization algorithm to generate a specific set of initial condition, the following My FastICA implements the FastICA using ATGP-generated data sample vectors (program is called My_ATGP) as its initial condition

MATLAB Codes of My FastICA for IDICA-DR (ICA-DR3)

```
function [ICs]=My_fastica_DR3(HIM,M);

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

x=reshape(HIM,xx*yy,bnd);
x=x';

L=size(x,1);
K=size(x,2);

%====Sphering =====
u=mean(x,2); %dimension of u is 1*L
x_hat=x-u*ones(1,K);
m=mean(x,2);
C=(x*x')/size(x,2)-m*m';
%===========
[V,D]=eig(C);
A=inv(sqrtm(D))*V';  % A is the whitening matrix....
x_whitened=A*(x_hat);
%====for cuprite data===
clear x;
clear x_hat;
%=========for initialization
```

```
[Loc,Sig]=My_ATGP(reshape(x_whitened',xx,yy,L),M);

W_initial=Sig;

threshold = 0.0001;
B=zeros(L);
%===============find the first point=========

for round=1:M
  fprintf('IC %d', round);
  %===Initial condition
  w=W_initial(:,round);
  %===
  w=w-B*B'*w;
  w=w/norm(w);
  wOld=zeros(size(w));
  wOld2=zeros(size(w));
  i=1;
  while i<=1000
    w=w-B*B'*w;
    w=w/norm(w);
    fprintf('.');
    if norm(w-wOld)<threshold | norm(w+wOld)<threshold
      fprintf('Convergence after %d steps\n', i);
      B(:,round)=w;
      W(round,:)=w';
      break;
    end
    wOld2=wOld;
    wOld=w;
    w=(x_whitened*((x_whitened'*w).^3))/K-3*w;
    w=w/norm(w);
    i=i+1;
  end
  if (i>1000)
        fprintf('Warning! cannot converge after 1000 steps \n, no more
components');
        break;
  end
  round=round+1;
end

ICs=W*x_whitened;

figure
for m=1:M
  s=reshape(abs(ICs(m,:)),xx,yy);
  s=255*(s-min(min(s))*ones(size(s,1),size(s,2)))/(max(max(s))
```

```
-min(min(s)));
  temp=mean(reshape(s,xx*yy,1));
  subplot(5,6,m); imshow(uint8(s));
%
end

A=W;

% x_re_ICA=V*sqrtm(D)*(A*ICs)+u*ones(1,xx*yy);
```

A.3.4 HOS-DR

Variance-based PCA and SNR-based MNF represent second-order statistics-based component analysis transformations to perform DR. At the other extreme, ICA is an infinite-order statistics-based component analysis that makes use of mutual information to measure statistical independency among all ICs. However, due to complexity of implementing mutual information, the commonly used FastICA is actually developed by combining the third and fourth moments as a criterion to measure the dependency among its generated components. So, technically speaking, the FastICA-generated components are not really statistically independent. Instead, they can be only considered as high-order statistically dependent. HOS-DR is developed to extend DR transformation with order of statistics higher than 2 (Ren et al., 2006). In this context, the FastICA (Hyvarinen and Ojha, 1997) can be considered as a special case of HOS-DR transformation.

- **Algorithm name:** high-order statistics DR (HOS-DR)
- **Authors:** H. Ren and Chein-I Chang
- **Category:** component analysis-based transform
- **Designed criteria:** statistical moments higher than 2
- **Designed method:** moment projection
- **Typical use (LOI's addressed):** no prior knowledge is required
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale images
- **Assumptions:** no prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** low
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  ICA is widely used in blind source separation via a linear mixture model, which assumes that there exists at most one Gaussian source. A fast algorithm developed by Hyvarinen and Oja (1997), called FastICA, is the most used algorithm to implement ICA.

MATLAB Codes of HOS-DR

```
function [ICs]=high_order(HIM,M,k,Initial);
% HIM is the image cube.
```

```
% M is the number of components to generated using high order
% k is the order of statistics. For example, k=3, is the skewness, k=4, is
% the kurtosis, k=5 is the 5th moment, and so on...
% Initial condition preference, 0 is the random initial, 1 is the eigen
% initial, 2 is the unity initial. default is 0
if nargin < 3
  fprintf('Please identify the order of statistics!');
  ICs=[];
else

  if nargin < 4
    Initial=0;
  end

  bnd=size(HIM,3);
  xx=size(HIM,1);
  yy=size(HIM,2);

  x=reshape(HIM,xx*yy,bnd);
  x=x';

  L=size(x,1);
  K=size(x,2);

  %===input x is a matrix with size=L*K;
  %====Sphering =====
  u=mean(x,2); %dimension of u is 1*L
  x_hat=x-u*ones(1,K);

  %===jing's code of cov
  m=mean(x,2);
  C=(x*x')/size(x,2)-m*m';
  %===========
  [V,D]=eig(C);
  A=inv(sqrtm(D))*V';  % A is the whitening matrix....
  x_whitened=A*(x_hat);

  clear x;
  clear x_hat;

  % Seperating , using high-order
  threshold = 0.01;
  B=zeros(L);
  y=x_whitened;
  W=ones(1,L);

  P_U_perl=eye(bnd);
  for round=1:M
```

```
  fprintf('IC %d', round);
  %===initial condition===
  switch (Initial)
    case 0
      w=rand(L,1);
    case 1
      w=V(:,round); % Final version
    case 2
      w=ones(L,1);
    otherwise
      w=rand(L,1);
  end

  i=1;
  while i<=100 % maximum times of trying..
    a=(y.*repmat((w'*y).^(k-2),bnd,1))*y'; %skewness
    a=a/K;  % get the sample mean as expectation
    [V,D]=eig(a);
    D=abs(D);
    [C,I]=max(diag(D));
    V1 = V(:,I);
    fprintf('.');
    distance(round,1,i)=norm(w-V1);
    distance(round,2,i)=norm(w+V1);

    if norm(w-V1)<threshold | norm(w+V1)<threshold
      fprintf('Convergence after %d steps\n', i);

      B(:,round)=w;
      W(round,:)=w';
      break;
    end
    w=V1;
    i=i+1;
  end
  %===if not converge. then use the results after 10 iterations
  B(:,round)=w;
  W(round,:)=w';
  %=======
  P_U_perl=eye(bnd)-W'*inv(W*W')*W;
  y=P_U_perl*y;
end

ICs=W*x_whitened;

figure;

for m=1:M
```

```
    s=reshape(abs(ICs(m,:)),xx,yy);
    s=255*(s-min(min(s))*ones(size(s,1),size(s,2)))/(max(max(s))
-min(min(s)));
    temp=mean(reshape(s,xx*yy,1));
    subplot(5,6,m); imshow(uint8(s));
    %
  end

end
```

A.4 Endmember Extraction Algorithms

The MATLAB codes of three major endmember extraction algorithms, pixel purity index (PPI) algorithm with fast iterative PPI (FIPPI), N-finder algorithm (N-FINDR), simplex growing algorithm (SGA) are provided in this section.

A.4.1 Pixel Purity Index

- **Algorithm name:** PPI
- **Authors:** J. Boardman
- **Category:** convex geometry
- **Designed criteria:** orthogonal projection
- **Designed method:** randomly generated vectors, called skewers
- **Typical use (LOI's addressed):** endmember extraction with number of endmembers to be known
- **Inputs:** reflectance or radiance cube
- **Outputs:** endmembers
- **Assumptions:** the number of skewers to be generated, K must be sufficiently large
- **Sensitivity to LOI (target knowledge):** high to K as well as skewers
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  PPI was first developed by Boardman to extract endmembers (Boardman, 1994) and is available in ENVI software. Unfortunately, its detailed steps in implementation are not available for users who would like to make modifications or changes at their discretion. Its MATLAB codes provided in the following serve this purpose. It is developed based on the concept of the convex geometry and the criterion of orthogonal projection. It first generates a set of K random unit vectors, called skewers, to cover all possible projection directors and then orthogonally projects all data sample vectors on these skewers to find the maximal and minimal orthogonal projections of each skewer. For each data sample vector, it counts the number of skewers on which its orthogonal projections yield either maximal or minimal projections. This count is referred to as the PPI count, which will be used to determine whether or not a particular data sample vector is an endmember. In doing so, a threshold needs to be specified in advance. In ENVI, this is done manually. In addition, since its skewers are generated randomly, the results are not repeatable. In other words, the same user running PPI in different times or different users running PPI at the same time will all have different results. This serious drawback can further be fixed by

initialization-driven PPI (ID-PPI), such as a fast iterative PPI (FIPPI) algorithm (Chang and Plaza, 2006), and random PPI (RPPI) (Chang et al., 2010).

MATLAB Codes of PPI

```
function [eeindex score duration]=PPI(imagecub,skewer_no)
% The Matlab PPI algorithm
% ------ Input variables ----------
% 'imagecub' - The hyperspectral image cube
% 'skewer_no' - The number of skewers
%
% ------ Output variables ----------
% 'eeindex' - The locations of the final endmembers (x,y)
% 'score' - The PPI score of each pixel
% 'duration - The number of seconds used to run this program

% Initial Variables
[rows columns bands]=size(imagecub);
score=zeros(rows*columns,1);
switch_results=1;

% Record the start CPU time
start=cputime();

% Separate the total number of skewers into several sets and each set uses 500
skewers
skewer_sets=floor(skewer_no/500)+1;
last_skewer_no=mod(skewer_no,500);

for i=1:skewer_sets

   if (skewer_sets-i) == 0,
      skewer_no=last_skewer_no;
   else
      skewer_no=500;
   end

   % Generate skewers
   rand('state',sum(100*clock));
   skewers=rand(bands,skewer_no)-0.5;

   % Normalize skewers
   for i=1:skewer_no,
      skewers(:,i)=skewers(:,i)/norm(skewers(:,i));
   end

   % project every sample vector to the skewers
```

```
   projcub=reshape(imagecub, rows*columns, bands);
   proj_result=projcub*skewers;


   % Find the extrema set for each skewer and add 1 to their score
   for i=1:skewer_no,
      max_pos=find(proj_result(:,i)==max(proj_result(:,i)));
      min_pos=find(proj_result(:,i)==min(proj_result(:,i)));
      score(max_pos)=score(max_pos)+1;
      score(min_pos)=score(min_pos)+1;
   end
end

% Find the pixel which has score larger than 0
result=find(score>0);

% Find the position of the p highest scores
%result=[];
%for i=1:skewer_no,
%       result=[result find(max(score)==score,1)];
%       score(find(max(score)==score,1))=0;
%end

% Convert one dimension to two dimension index
if(switch_results),
  eeindex=translate_index(result,rows,columns,1);
else
   if(mod(result,rows)==0)
      eeindex(2,:)=floor(result./rows);
   else
      eeindex(2,:)=floor(result./rows)+1;
   end
   eeindex(1,:)=mod(result-1,rows)+1;
end

duration=cputime()-start;
```

MATLAB Codes of Fast Iterative Pixel Purity Index

```
function [FinalPositions running_time]=FIPPIoptimized(Image,
InitialSkewers)

% Fast Iterative Pixel Purity Index Algorithm
%
% Input parameters:
% ————————————————
%  Image: Hyperspectral image data after MNF dimensionality reduction
%  InitialSkewers: Positions of ATGP-generated pixels
```

```
%
% Output parameter:
% ————————————————
%  FinalPositions: Positions of FIPPI-generated endmember pixels
%  running_time - The total running time used by this run
%
% Authors: Chein-I Chang and Antonio Plaza
% Minor Modified by Chao-Cheng Wu

% Check CPU time at the beginning
start=cputime;

% Code initialization for data and visualization
[ns,nl,nb]=size(Image);
[VD,kk]=size(InitialSkewers);
Extrema=zeros(ns,nl);
ProjectionScores=zeros(ns,nl);
subplot(2,1,1);
imagesc(Image(:,:,1)); colormap(gray);
title('Pixels extracted by FPPI:');
set(gca,'DefaultTextColor','black','xtick',[],'ytick',[],'data-
aspectratio',[1 1 1]);
po1 = get(gca,'position');

% Use ATGP-generated pixels as the initial skewers
NewSkewers=InitialSkewers;

% Begin iterative process
Other = 1;
SkewersUsed = [];
while (Other >= 1)

  [ne,np]=size(NewSkewers);
  disp(['Iteration: ' int2str(Other)]);
  disp(['Skewers: ' int2str(ne)]);
  for k = 1:ne

    [ne_old,kk]=size(SkewersUsed);
    skewer=squeeze(Image(NewSkewers(k,1),NewSkewers(k,2),:));
    skewer=skewer/norm(skewer);
    SkewersUsed = union(SkewersUsed,skewer);
    [ne_new,kk]=size(SkewersUsed);

    subplot(2,1,2);
    drawnow;
    plot(skewer);
    title(['Current skewer: ' int2str(k)]);
```

```
    if (ne_new~=ne_old)
    % Project all the sample data vectors onto this particular skewer
      for i=1:ns
        for j=1:nl
          pixel = squeeze(Image(i,j,:));
          ProjectionScores(i,j) = dot(skewer,pixel);
        end
      end

      % Obtain the extrema set for each skewer (maximum and minimum
      % projection)
      [vals,mpos] = max(ProjectionScores(:));
          [vals,pos] = min(ProjectionScores(:));
      mposx = floor((mpos-1)/ns)+1; mposy = mod(mpos-1,ns)+1;
              posx = floor((pos-1)/ns)+1; posy = mod(pos-1,ns)+1;

      % Display the pixel positions of the pixels in the extrema set
      drawnow;
      subplot(2,1,1);
text(mposx,mposy,'o','Margin',1,'HorizontalAlignment','center','
FontSize',22,'FontWeight','light','FontName','Garamond','Color',
'yellow');

         drawnow;
      subplot(2,1,1);

text(posx,posy,'o','Margin',1,'HorizontalAlignment','center','
FontSize',22,'FontWeight','light','FontName','Garamond','Color',
'yellow');

      % Increase PPI count of extrema pixels
      Extrema(posy,posx)=Extrema(posy,posx)+1;
        Extrema(mposy,mposx)=Extrema(mposy,mposx)+1;

      % Incorporate sample vectors with PPI count greater than zero to
      % the skewer set
      vnew = [ mposx mposy ; posx posy ];
      NewSkewers = union(NewSkewers,vnew,'rows');
    end
  end

  % Check stopping rule
  [ne2,np]=size(NewSkewers);
  if (ne2==ne)
    Other = 0;
  else
    Other = Other+1;
```

```
      % Extrema=zeros(ns,nl);
  end
end


% Produce the positions of the final endmember set
Binary = Extrema>0;
ne = sum(Binary(:));
disp(['Extracted endmembers: ' int2str(ne)]);
FinalPositions = zeros(ne,2);
Current = 1;
for i=1:ns
  for j=1:nl
    if Binary(i,j)>0
      FinalPositions(Current,1)=i;
      FinalPositions(Current,2)=j;
      Current = Current+1;
    end
  end
end

% Check CPU time at the end
stop=cputime;
running_time=stop-start;
```

A.4.2 N-finder Algorithm

- **Algorithm name:** N-finder algorithm (N-FINDR)
- **Authors:** M.E. Winter
- **Category:** convex geometry
- **Designed criteria:** maximum simplex volume
- **Designed method:** finding a simplex with maximum volume
- **Typical use (LOI's addressed):** endmember extraction with number of endmembers to be known
- **Inputs:** reflectance or radiance cube
- **Outputs:** endmembers
- **Assumptions:** All the vertices of a simplex with maximal volume should be specified by endmembers
- **Sensitivity to LOI (target knowledge):** high to value of p
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The N-FINDR (Winter, 1999a, 1999b, 2004) is a popular endmember extraction algorithm (EEA) other than PPI. It is quite different from PPI in terms of its design criterion. The N-FINDR makes use of maximum simplex volume as a criterion as opposed to orthogonal projection used by PPI. So, its design criteria make PPI an unconstrained EEA and N-FINDR a fully

constrained EEA. Like PPI, which requires prior knowledge of the number of skewers, *K*, N-FINDR also needs to know the number of endmembers, *p*, *a priori*. Similar to PPI, N-FINDR also suffers the same issue encountered in PPI, which is its use of random initial conditions that result in unrepeatable and inconsistent endmember results. This issue is also addressed by Chang et al. (Plaza and Chang, 2006; Chang et al., 2011b). Another serious issue arising in N-FINDR implementation that is not encountered in PPI is its very high computational complexity. Many research efforts have been reported to address this issue. Details can be found in Xiong et al. (2011) and Chang (2013). Also, real-time implementation of N-FINDR has also been proposed in Wu et al. (2010) with details in Chang (2013).

MATLAB Codes of N-FINDR

```
function [endmemberindex duration]=NFINDR(imagecube,p)
% The N-FINDR algorithm
% ——— Input variables ———————
% 'imagecube' - The data transformed components [row column band]
% 'p' - The number of endmembers to be generated
%
% if band > p, then the program will automatically use Singular Value Decomposi-
tion to calculate the volume
% ——— Output variables ————————
% 'endmemberindex - The locations of the final endmembers (x,y)
% 'duration - The number of seconds used to run this program

% Set initial condition
endmemberindex=[];
newvolume = 0;
prevolume = -1;
[row, column, band]=size(imagecube);
switch_results=1;

% Determine to use SVD to calculate the volume or not
if(band > p),
   use_svd=1;
else
   use_svd=0;
end
% Start to count the CPU computing time
start=cputime();

% Randomly select p initial endmembers
rand('state',sum(100*clock));
for i=1:p
   while(1)
      temp1=round(row*rand);
      temp2=round(column*rand);
      if(temp1>0 & temp2>0)
```

```
        break;
      end
   end
   endmemberindex=[endmemberindex;[temp1 temp2]];
end
endmemberindex=endmemberindex';

% Generate endmember vector from reduced cub
display(endmemberindex);
endmember=[];
for i=1:p
   if(use_svd)
      endmember=[endmember squeeze(imagecube(endmemberindex(1,i),
endmemberindex(2,i),:))];
   else
      endmember=[endmember squeeze(imagecube(endmemberindex(1,i),
endmemberindex(2,i),1:p-1))];
   end
end

% calculate the endmember's volume
if(use_svd)
   s=svd(endmember);
   endmembervolume=1;
   for i=1:p,
      endmembervolume=endmembervolume*s(i);
   end
else
   jointmatrix=[ones(1,p) ; endmember];
   endmembervolume=abs(det(jointmatrix))/factorial(p-1);
end

% The main algorithm
while newvolume > prevolume, % if the new generated endmember volume is larger
than the old one, continue the algorithm

   % Use each sample vector to replace the original one, and calculate new volume
   for i=1:row,
      for j=1:column,
         for k=1:p,
            caculate=endmember;
            if(use_svd),
               caculate(:,k)=squeeze(imagecube(i, j, :));
               s=svd(caculate);
               volume=1;
               for z=1:p,
                  volume=volume*s(z);
               end
```

```
          else
             caculate(:,k)=squeeze(imagecube(i, j, 1:p-1));
             jointmatrix=[ones(1,p);calculate];
             volume=abs(det(jointmatrix))/factorial(p-1); % The formula of
Simplex volume
          end
          if volume > endmembervolume,
             endmemberindex(:,k)=[i;j];
             endmember=calculate;
             endmembervolume=volume;
          end
        end
      end
   end
   prevolume=newvolume;
   newvolume=endmembervolume;
end

stop=cputime();
duration=stop-start;
% Switch results for the standard
if(switch_results)
   endmemberindex(3,:)=endmemberindex(1,:);
   endmemberindex(1,:)=[];
   endmemberindex=endmemberindex';
end
```

A.4.3 Simplex Growing Algorithm

- **Algorithm name:** simplex growing algorithm (SGA)
- **Authors:** C.-I Chang
- **Category:** convex geometry
- **Designed criteria:** maximum simplex volume
- **Designed method:** growing maximum-volume simplexes
- **Typical use (LOI's addressed):** endmember extraction with number of endmembers to be known
- **Inputs:** reflectance or radiance cube
- **Outputs:** endmembers
- **Assumptions:** All the vertices of simplexes generated by SGA must be endmembers
- **Sensitivity to LOI (target knowledge):** high to value of p
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The SGA was developed in Chang et al. (2006) to address several serious implementation issues in N-FINDR. It produces one endmember at a time by growing simplexes vertex by vertex to

resolve the computational issue. In addition, the issue caused by random initial conditions can also be addressed by implementing SGA in real time (Chang et al., 2010).

MATLAB Codes of SGA

```
function [endmemberindex duration]=SGA(imagecube,p)
% Simplex Growing Algorithm
% - - - - - - - Input variables - - - - - - - - - - - -
% 'imagecube' - The data transformed components [row column band]
% 'p' - The number of endmembers to be generated
%
% if band > p, then the program will automatically use Singular Value Decomposi-
tion to calculate the volume
% - - - - - - - Output variables - - - - - - - - - - -
% 'endmemberindex - The locations of the final endmembers (x,y)
% 'duration - The number of seconds used to run this program

% Set initial condition
n=1;
initial=0;
[row, column, band]=size(imagecube);

% Determine to use SVD to calculate the volume or not
if(band > p),
  use_svd=1;
else
  use_svd=0;
end

% Start to count the CPU computing time
start_time=cputime();
% Randomly Select a point as the initial point
endmemberindex=[ceil(row*rand);ceil(column*rand)];

% The main algorithm
while n<p, % if get enough endmember group, it stops

       % Generate endmember vector from reduced cub
       endmember=[];
       for i=1:n
    if(use_svd)
      endmember=[endmember squeeze(imagecube(endmemberindex(1,i),
endmemberindex(2,i),:))];
    else
      endmember=[endmember squeeze(imagecube(endmemberindex(1,i),
endmemberindex(2,i),1:n))];
    end
       end
```

```
       % Use each sample vector to calculate new volume
       newendmemberindex=[];
       maxvolume=0;
       for i=1:row,
             for j=1:column,
       if(use_svd)
         jointpoint=[endmember squeeze(imagecube(i,j,:))];
         s=svd(jointpoint);
         volume=1;
         for z=1:n+1,
            volume=volume*s(z);
         end
      else
        jointpoint=[endmember squeeze(imagecube(i,j,1:n))];
        jointmatrix=[ones(1,n+1);jointpoint];
        volume=abs(det(jointmatrix))/factorial(n); % The formula of a simplex
volume
      end
          if volume > maxvolume,
             maxvolume=volume;
             newendmemberindex=[i;j];
           end
         end
     end
     endmemberindex=[endmemberindex newendmemberindex]; % Add this pixel into
the endmember group
      %nfinder_plot(endmemberindex);
     n=n+1;
     if initial==0, % Use new pixel as the initial pixel
          n=1;
          endmemberindex(:,1)=[];
          initial=initial+1;

      end
end

duration=cputime()-start_time;

% Switch the results back to X and Y
endmemberindex(3,:)=endmemberindex(1,:);
endmemberindex(1,:)=[];
endmemberindex=endmemberindex';
```

A.5 Supervised LSMA and KLSMA

Linear spectral mixture analysis (LSMA) is a mathematical theory that models a data samples as linear mixtures of a finite number of basic spectral constituents with appropriate weights from which data samples can be solved by finding these weights via a linear inverse problem. Linear

spectral unmixing is one of its applications. It assumes that there are p basic material substances $\{\mathbf{m}_j\}_{j=1}^{p}$ that can be used to represent data sample vectors in linear forms with their corresponding abundance fractions $\{\alpha_j\}_{j=1}^{p}$ that are unknown parameters. The spectral unmixing is then performed by finding best estimates of $\{\alpha_j\}_{j=1}^{p}$, denoted by $\{\hat{\alpha}_j\}_{j=1}^{p}$, and referred to as unmixed abundance fractions. In real applications, two physical constraints must be imposed on the used linear mixing model, which are abundance sum-to-one constraint (ASC), $\sum_{j=1}^{p}\alpha_j = 1$, and abundance nonnegativity constraint (ANC), $\alpha_j \geq 0$ for all $1 \leq j \leq p$. Three LSMA-based techniques developed in Chang (2003a) have been widely used for spectral unmixing. These are unconstrained orthogonal subspace projection (OSP)/least squares OSP, partially ANC-constrained method, Least Squares nonnegativity-constrained least squares (NCLS) and a fully abundance-constrained method, Constrained Least Squares (FCLS). Since OSP/LSOSP, NCLS, and FCLS are linear techniques, they may have difficulty solving linear nonseparable problems. To address this issue, these three techniques are further extended to their kernel-based counterparts, called Kernel OSP/LSOSP (KOSP/KLSOSP), Kernel NCKS (KNCLS), and Kernel FCLS (KFCLS).

A.5.1 OSP, LSOSP, KOSP, and KLSOSP

- **Algorithm name:** orthogonal subspace projection (OSP)
- **Authors:** J.C. Harsanyi and Chein-I Chang
- **Category:** spectrally matched filter
- **Designed Criteria:** SNR ratio
- **Designed Method:** *A priori*, supervised and unconstrained least squares-based linear spectral mixture analysis
- **Typical use (LOI's addressed):** detection, classification, discrimination, identification
- **Inputs:** reflectance or radiance cube, complete target knowledge
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** complete prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, real Time
- **Brief description:**
  The OSP approach was first developed in Harsanyi's Ph.D. dissertation in 1993 (Harsanyi, 1993) and was later published in *IEEE Transaction on Geoscinece and Remote Sensing*, July 1994 (Harsanyi and Chang, 1994). It is a linear unmixng method that takes advantage of a linear mixture model to detect, classify, and identify targets of interest. The idea is to separate target sources into desired and undesired targets and then use an orthogonal project to reject the desired targets before a matched filtration takes place. So, it operates two functions in sequence: an undesired target rejecter followed by a spectral matched filtration. Since OSP was originally designed as a detector and cannot accurately estimate signature abundance fractions, a least squares version of OSP, referred to as least squares OSP (LSOSP) was further developed in Chang et al. (1998b) for abundance fraction estimation. The only difference between OSP and LSOSP is that LSOSP includes a normalization constant to account for estimation error incurred in the OSP-derived detector (Chang, 2009). So, the MATLAB codes provided in the following is a more general version of OSP, LSOSP.

MATLAB Codes of LSOSP

```
% Least Square Orthogonal Subspace Projection
% input: image = image cube
%      d = desire signature vector
%      U = undesired signature matrix
% output: temp = resulting image cube

function temp=LSOSP(image,d,U)

[x y z]=size(image);
temp = zeros(x,y);

%Find the projectors that is orthogonal complement of U

[l,J] = size(U);
I = eye(l,l);
Pu=I-U*inv(U'*U)*U';
lsosp = (d'*Pu)/(d'*Pu*d);
% perform least-squares-based estimator on all image vectors
for i = 1:x
  for j = 1:y
    for k = 1:z
      r(k) = image(i,j,k);
    end;
    temp(i,j)=lsosp*r';
  end;
end;
```

MATLAB codes of KLSOSP

```
%% Kernel based LSOSP function
% Input:
%    image = image cube input
%    d   = desired signature, example: [2;3;4]
%    U   = undesired signature matrix
%    sig  = parameter that control RBF kernel function
% output:
%    temp  = resulting map

function temp=KOSP(image,d,U,sig)

[x y z]=size(image);
temp = zeros(x,y);

% perform least squares-based estimator on all image vectors

KdU = kernelized(d,U,sig,0);%disp(KdU),
```

```
KUU = kernelized(U,U,sig,0);%disp(KUU),
Kdd = kernelized(d,d,sig,0);
KUd = kernelized(U,d,sig,0);%disp(KUd),
for i = 1:x
  for j = 1:y
    for k = 1:z
      r(k,1) = image(i,j,k);
    end;
    Kdr = kernelized(d,r,sig,0);%disp(Kdr),
    KUr = kernelized(U,r,sig,0);%disp(KUr),
    temp(i,j)=(Kdr-KdU*inv(KUU)*KUr);%/(Kdd-KdU*inv(KUU)*KUd);
  end;
end;

%% kernelization function
function results = kernelized(x,y,d,chk)
x_l = size(x,2);
y_l = size(y,2);
results = zeros(x_l,y_l);
for i = 1:x_l
  for j = 1:y_l
    results(i,j)= exp((-1/2)*(norm(x(:,i)-y(:,j))^2)/(d^2));
%RBF kernel (can be changed)
  end
end
if chk == 1
  results = results-(sum(sum(results))/(x_l*y_l))*ones(x_l,y_l);
elseif chk == 2
  N = (1/(x_l*y_l))*ones(x_l,y_l);
  results = results-N*results-results*N+N*results*N;
end
```

A.5.2 NCLS and KNCLS

- **Algorithm name:** nonnegativity least squares (NCLS)
- **Authors:** Chein-I Chang and Daniel Heinz
- **Category:** least squares error-based spectral filter
- **Designed criteria:** least squares error
- **Designed method:** *A priori*, supervised and ANC-constrained LSMA
- **Typical use (LOI's addressed):** detection, classification, discrimination, identification
- **Inputs:** reflectance or radiance cube, complete target knowledge
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** complete prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational

- **Effectiveness:** high
- **Implementation:** simple, easy to use, real time
- **Brief description:**
  Nonnegativity constrained least squares (NCLS) was developed to improve OSP in signal detection (Chang and Heinz, 2000b). It imposes ANC on the linear mixing model to make sure that the unmixed abundance fractions are nonnegative. Despite not being fully constrained, on many occasions NCLS can perform abundance estimation as well as a fully abundance-constrained method in. However, in signal detection, NCLS generally performs better than unconstrained and fully constrained methods.

MATLAB Codes of NCLS

```
function [abundance,error_vector]=NCLS(MatrixZ,r1)
% input MatrixZ is the signatures of endmembers. It is of size [ bands p].
% input x is the signature whose abundance is to be estimated.
% output abundance is the abundance of each material in r1. It is of size [p 1].
% output error_vector is the error vector of size [bands 1].
% This function is written according to Dr. Chang's first book , P 47

x=r1; %rename r1 as x;
M=size(MatrixZ,2);
count_R=0;
count_P=M;
R=zeros(M,1);
P=ones(M,1);
%tolerance=0.000001;
d=zeros(M,1);
Alpha_ls=inv(MatrixZ'*MatrixZ)*MatrixZ'*x;
Alpha_ncls=Alpha_ls;
min_Alpha_ncls=min(Alpha_ncls);
M_t_r=MatrixZ'*x;
invMtM=inv(MatrixZ'*MatrixZ);
while(min_Alpha_ncls<-0.000000001)
  for II=1:M
    if((Alpha_ncls(II)<0)&(P(II)==1))
      R(II)=1;
      P(II)=0;
    end %%% end of if (Alpha_ncls(II)<0)
  end % end of for II=1:M
  S=R;

  goto_step6=1;
  while(1)

    sum_R=sum(R);
    Alpha_R=zeros(sum_R,1);
    count_for_Alpha_R=0;
    for II=1:M
```

```
  if (R(II)==1)
    count_for_Alpha_R=count_for_Alpha_R+1;
    Alpha_R(count_for_Alpha_R)=Alpha_ls(II);
    index_for_Lamda(count_for_Alpha_R)=II;
  end
end

count_1_for_P=0;
Sai_column=[];
for II=1:M
  if (P(II)~=1)
    Sai_column=[Sai_column squeeze(invMtM(:,II)) ];

  end
end
Sai=[];
for II=1:M
  if (P(II)~=1)
    Sai=[Sai
      squeeze(Sai_column(II,:)) ];
  end
end

Lamda=inv(Sai)*Alpha_R;
if(max(Lamda)<0)
  break;
end
[max_Lamda,index_Max_Lamda]=max(Lamda);
P(index_for_Lamda(index_Max_Lamda))=1;
R(index_for_Lamda(index_Max_Lamda))=0;

sum_R=sum(R);
Alpha_R=zeros(sum_R,1);
count_for_Alpha_R=0;
for II=1:M
  if (R(II)==1)
    count_for_Alpha_R=count_for_Alpha_R+1;
    Alpha_R(count_for_Alpha_R)=Alpha_ls(II);
    index_for_Lamda(count_for_Alpha_R)=II;
  end
end

Sai_column=[];

for II=1:M
  if (P(II)~=1)
    Sai_column=[Sai_column squeeze(invMtM(:,II)) ];

  end
```

```
  end

  Sai=[];
  for II=1:M
    if (P(II)~=1)
      Sai=[Sai
        squeeze(Sai_column(II,:)) ];
    end
  end

  Lamda=inv(Sai)*Alpha_R;

  Phai_column=[];
  for II=1:M
    if (P(II)~=1)
      Phai_column=[Phai_column squeeze(invMtM(:,II)) ];
    end
  end
  if (size(Phai_column,2)~=0)

    Alpha_s=Alpha_ls-Phai_column*Lamda;
  else
    Alpha_s=Alpha_ls;
  end

  goto_step6=0;
  find_smallest_in_S=zeros(M,2);
  find_smallest_in_S(:,1)=Alpha_s;
  find_smallest_in_S(:,2)=[1:M]';
  sort_find=sortrows(find_smallest_in_S,1);

  for II=1:M
    if ((S(II)==1)&(Alpha_s(II)<0))
      P(II)=0;
      R(II)=1;

      goto_step6=1;
    end
  end

end % end of while (gotostep6==1)

Phai_column=[];
for II=1:M
  if (P(II)~=1)
    Phai_column=[Phai_column squeeze(invMtM(:,II)) ];

  end
end
```

```
  if (size(Phai_column,2)~=0)

    Alpha_ncls=Alpha_ls-Phai_column*Lamda;

  else
    Alpha_ncls=Alpha_ls;

  end
  min_Alpha_ncls=min(Alpha_ncls);
end % end of while


abundance=zeros(M,1);
for II=1:M
  if (Alpha_ncls(II)>0)
    abundance(II)=Alpha_ncls(II);
  end
end
error_vector=MatrixZ*abundance-x;
```

MATLAB codes of KNCLS

```
function ab = KNCLS(r,M,d)
% Non-negative constrain Least-square abundance estimator
% input M = signature matrix to be estimated.
% input r = image pixel vector.
% output abundance = abundance vector correspondence to the signature
% matrix.

% initial stage k = 0
k = 0;
[x,y] = size(M);
P = 1:y;
R = [];

%least square estimate of abundance
inv_MTM = inv(kernelized(M,M,d,0));
a_ls = inv_MTM*kernelized(M,r,d,0);
%initially set abundance to least square abundance
ab = a_ls;
%iterate until all ab is positive
while any(ab<-0.5) && k<50
%  disp(k)
%  clc,
  k = k+1;
  % find negative index and move them from P to R
  neg_ind = find(ab<0);
```

```
  [P,R] = move_index(P,R,neg_ind);
  S = R;

  %initialize lum so the for loop can run at least once
  lum = 1;
  % iterate until all lum is smaller or equal to zero
  j=0;
  while any(lum>=0) && j<50
%  disp(j)
    j = j+1;
    a_R = a_ls(R);
    % define the steer matrix by removing row and columns defined by P
    a_steer_mat = inv_MTM;
    a_steer_mat(:,P) = [];
    a_steer_mat(P,:) = [];

    % calculate Lagrange multiplier lum
    lum = ((a_steer_mat)^(-1))*a_R;
    % if lum contains positive value
    if any(lum>0) && j<1000
      % find the maximum lum move its index from R to P create a new
      % lum by removing the max lum
      j = j+1;
      lum_max_ind = R(find(lum == max(lum)));
      [R,P]=move_index(R,P,lum_max_ind);
      a_steer_mat = inv_MTM;
      a_steer_mat(:,P) = [];
      a_steer_mat(P,:) = [];
      a_R = a_ls(R);
      lum_new = ((a_steer_mat)^(-1))*a_R;

      % form another lum steering matrix
      lum_steer_mat = inv_MTM;
      lum_steer_mat(:,P) = [];
      if length(lum_steer_mat) == 0
        lum_steer_mat = 0;
      end;
      % calcuate the abundance base on R and P and lum_new. If any
      % value specify by index S is negative move it from P to R.
      a_S = a_ls - lum_steer_mat*lum_new;
      s_neg_ind = S(find(a_S(S)<0));
      [P,R] = move_index(P,R,s_neg_ind);
    end;
  end;

  % calculate the new abundance base on the new lum found
  lum_steer_mat = inv_MTM;
  lum_steer_mat(:,P) = [];
```

```
  ab = a_ls - lum_steer_mat*lum;

end;
return;

% Move index function
function [P_new,R_new] = move_index(del_ind, get_ind, move_ind)

n = length(move_ind);
if n>0
  for i = 1:n
    a = find(del_ind == move_ind(i));
    del_ind(a)=[];
    b = get_ind - move_ind(i);
    if any(b == 0)
      get_ind = get_ind;
    else
      get_ind = [get_ind,move_ind(i)];
    end;
  end;
end;

P_new = sort(del_ind);
R_new = sort(get_ind);
return

function results = kernelized(x,y,d,chk)
x_l = size(x,2);
y_l = size(y,2);
results = zeros(x_l,y_l);
for i = 1:x_l
  for j = 1:y_l
    results(i,j)= exp((-1/2)*(norm(x(:,i)-y(:,j))^2)/(d^2));
  end
end
if chk == 1
  results = results-(sum(sum(results))/(x_l*y_l))*ones(x_l,y_l);
elseif chk == 2
  N = (1/(x_l*y_l))*ones(x_l,y_l);
  results = results-N*results-results*N+N*results*N;
end
return,
```

A.5.3 FCLS and KFCLS

- **Algorithm name:** fully constrained least squares (FCLS)
- **Authors:** Daniel Heinz and Chein-I Chang

- **Category:** least squares error-based spectral filter
- **Designed criteria:** least squares error (LSE)
- **Designed Method:** *a priori*, supervised and fully constrained LSMA
- **Typical use (LOI's addressed):** detection, classification, discrimination, identification
- **Inputs:** reflectance or radiance cube, complete target knowledge
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** complete prior target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, real time
- **Brief description:**
  Fully constrained least squares (FCLS) was developed in Heinz and Chang (2001) as an abundance estimator to accurately estimate abundance fractions. Therefore, as far as unmixing is concerned, FCLS is one of best designed linear estimator. But it does not imply that FCLS is also best for other applications such as detection, discrimination, classification, etc. As a matter of fact, NCLS generally outperforms FCLS in these applications, where NCLS does not comply with the constraint of ASC, specifically, in signal detection, where signals to be detected are corrupted by noise and ASC is certainly violated.

MATLAB Codes of FCLS

```
%-----------------------------------------------------------
% - Fully Constrain Least-Squares (FCLS) abundances estimate -
%
% function: results = FCLS(image,M,tol)
%
% input:  image  = image of size [X,Y,Z]
%     M    = endmember matrix of size [Z,p]
%     tol   = NCLS tolerance, e.g. -1e-6
%
% output:  results = resulting abundance map of size [X,Y,p]
%-----------------------------------------------------------
function results = FCLS_v2(image,M,tol)
[x,y,z] = size(image);
num_p = size(M,2);
results = zeros(x,y,num_p);
for i = 1:x
  for j = 1:y
    r = reshape(image(i,j,:),z,1);
    delta = 1/(10*max(max(M)));
    s = [delta.*r;1];
    N = [delta.*M;ones(1,num_p)];

    [ab] = NCLS(s, N, tol);
```

```
    ab = reshape(ab,[1,1,num_p]);
    results(i,j,:) = ab;
  end;
end;
return;
function [abundance]=NCLS(x, MatrixZ, tol)
% input MatrixZ is the signatures of endmembers. It is of size [bands p].
% input x is the signature whose abundance is to be estimated.
% output abundance is the abundance of each material in r1. It is of size [p 1].
% This function is written according to Dr. Chang's first book , P 47

M=size(MatrixZ,2);
R=zeros(M,1);
P=ones(M,1);
invMtM=(MatrixZ'*MatrixZ)^(-1);
Alpha_ls=invMtM*MatrixZ'*x;
Alpha_ncls=Alpha_ls;
min_Alpha_ncls=min(Alpha_ncls);
j=0;
while(min_Alpha_ncls<-tol && j<500)
  j = j+1;
  for II=1:M
    if((Alpha_ncls(II)<0)&&(P(II)==1))
      R(II)=1;
      P(II)=0;
    end %%% end of if (Alpha_ncls(II)<0)
  end % end of for II=1:M
  S = R;

  goto_step6=1;
  counter = 0;
  while(goto_step6==1)
    index_for_Lamda = find(R==1);
    Alpha_R = Alpha_ls(index_for_Lamda);
    Sai = invMtM(index_for_Lamda,index_for_Lamda);

    inv_Sai = (Sai)^(-1);     % remember inversion of Sai
    Lamda=inv_Sai*Alpha_R;

    [max_Lamda,index_Max_Lamda]=max(Lamda);
    counter = counter+1;
    if ( max_Lamda<=0 || counter == 200 )
      break;
    end

    temp_i = inv_Sai;      % simplify the inversion of matrix
    temp_i(1,:) = inv_Sai(index_Max_Lamda,:);
    if index_Max_Lamda>1
```

```
      temp_i(2:index_Max_Lamda,:) = inv_Sai(1:index_Max_Lamda-1,:);
    end
    inv_Sai_ex = temp_i;
    inv_Sai_ex(:,1) = temp_i(:,index_Max_Lamda);
    if index_Max_Lamda>1

      inv_Sai_ex(:,2:index_Max_Lamda) = temp_i(:,1:index_Max_Lamda-1);
    end
     inv_Sai_next = inv_Sai_ex(2:end,2:end) - inv_Sai_ex(2:end,1)*inv_Sai_ex
(1,2:end)/inv_Sai_ex(1,1);

    P(index_for_Lamda(index_Max_Lamda))=1;
    R(index_for_Lamda(index_Max_Lamda))=0;
    index_for_Lamda(index_Max_Lamda) = [];

    Alpha_R = Alpha_ls(index_for_Lamda);
    Lamda=inv_Sai_next*Alpha_R;

    Phai_column = invMtM(:,index_for_Lamda);

    if (size(Phai_column,2)~=0)
      Alpha_s=Alpha_ls-Phai_column*Lamda;
    else
      Alpha_s=Alpha_ls;
    end

    goto_step6=0;

    for II=1:M
      if ((S(II)==1)&&(Alpha_s(II)<0))
        P(II)=0;
        R(II)=1;
        goto_step6=1;
      end
    end
  end % end of while (gotostep6==1)

  index_for_Phai = find(R==1);
  Phai_column = invMtM(:,index_for_Phai);

  if (size(Phai_column,2)~=0)
    Alpha_ncls=Alpha_ls-Phai_column*Lamda;
  else
    Alpha_ncls=Alpha_ls;
  end

  min_Alpha_ncls=min(Alpha_ncls);
end % end of while
```

```
abundance=zeros(M,1);
for II=1:M
  if (Alpha_ncls(II)>0)
    abundance(II)=Alpha_ncls(II);
  end
end
return;
```

MATLAB Codes of KFCLS

```
function [abundance,error_vector]= KFCLS(M,r1,delta,d)
% input M is the signatures of endmembers. It is of size [ bands p].
% input r1 is the signature whose abundance is to be estimated.
% input delta is the parameter to control ASC: see explanation on Dr. Chang's
first book , P 183
% usually, delta is 1/(10*max(max(A));
% output abundance is the abundance of each material in r1. It is of size [p 1].
% output error_vector is the error vector of size [bands 1].
tic
%Dan's: optimal
A=M;
numloop=size(A,2);
e=delta;
eA=e*A;
E=[ones(1,numloop);eA];
EtE=kernelized(E,E,d,0);
[m,n] = size(EtE);
One=ones(m,1);
iEtE=inv(EtE);
iEtEOne=iEtE*One;
sumiEtEOne=sum(iEtEOne);
weights=diag(iEtE);

c=0;
sample=r1;
er=e*sample;
f=[1;er];
Etf=kernelized(E,f,d,0);

tol=1e-7;
%fcls1a

%%%% THIS IS lamdiv2
ls=iEtE*Etf;
lamdiv2=-(1-(ls'*One))/sumiEtEOne;
x2=ls-lamdiv2*iEtEOne;
x2old=x2;
if (any(x2<-tol))
```

```
  Z=zeros(m,1);
  iter=0;
  while(any(x2<-tol) && iter>(m))
    Z(x2<-tol)=1;
    zz=find(Z);
    x2=x2old;      % Reset x2
    L=iEtE(zz,zz);
    ab=size(zz);
    lastrow=ab(1)+1;
    lastcol=lastrow;
    L(lastrow,1:ab(1))=(iEtE(:,zz)'*One)';
    L(1:ab(1),lastcol)=iEtEOne(zz);
    L(lastrow,lastcol)=sumiEtEOne;
    xerow=x2(zz);
    xerow(lastrow,1)=0;
    lagra=L\xerow;
    while (any(lagra(1:ab(1))>0))  % Reset Lagrange multipliers
      maxneg=weights(zz).*lagra(1:ab(1));
      [yz,iz]=max(maxneg);  % Remove the most positive
      Z(zz(iz))=0;
      zz=find(Z);     % Will always be at least one (prove)
      L=iEtE(zz,zz);
      ab=size(zz);
      lastrow=ab(1)+1;
      lastcol=lastrow;
      L(lastrow,1:ab(1))=(iEtE(:,zz)'*One)';
      L(1:ab(1),lastcol)=iEtEOne(zz);
      L(lastrow,lastcol)=sumiEtEOne;
      xerow=x2(zz);
      xerow(lastrow,1)=0;
      lagra=L\xerow;
    end
    %problem with lamscls zz may be null
    if ~isempty(zz)
      x2=x2-iEtE(:,zz)*lagra(1:ab(1))-lagra(lastrow)*iEtEOne;
    end
    iter=iter+1;
  end
end
abundance=x2;
error_vector=A*abundance-r1;

function results = kernelized(x,y,d,chk)
x_l = size(x,2);
y_l = size(y,2);
results = zeros(x_l,y_l);
for i = 1:x_l
  for j = 1:y_l
```

```
    results(i,j)=exp((-1/2)*(norm(x(:,i)-y(:,j))^2)/(d^2));
  end
end
if chk == 1
  results = results-(sum(sum(results))/(x_l*y_l))*ones(x_l,y_l);
elseif chk == 2
  N = (1/(x_l*y_l))*ones(x_l,y_l);
  results = results-N*results-results*N+N*results*N;
end
```

A.6 Unsupervised Hyperspectral Target Detection

One of major strengths resulting from hyperspectral imaging is the ability to find subtle targets of interest such as subpixel targets, anomalies that cannot be resolved by multispectral imaging. However, it also comes with an issue of how to find them, since such targets are generally not visualized by inspection and must be found in an unsupervised manner without prior knowledge. This part extends the three supervised LSMA-based techniques, OSP/LSOSP, NCLS, and FCLS, to their unsupervised counterparts, automatic target generation process (ATGP), unsupervised NCLS (UNCLS), and unsupervised FCLS (UFCLS).

A.6.1 ATGP

- **Algorithm name:** automatic target generation process (ATGP)
- **Authors:** H. Ren and Chein-I Chang
- **Category:** unsupervised least squares-based filter
- **Designed criteria:** orthogonal projection (OP)
- **Designed method:** maximum OP
- **Typical use (LOI's addressed):** detection, classification, discrimination, identification
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** no target knowledge required
- **Sensitivity to LOI (target knowledge):** high
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, real time
- **Brief description:**
  The ATGP is derived from an algorithm, called automatic target detection, and classification (ATDCA) developed by Ren and Chang to find target pixels of interest for recognition without having prior target knowledge (Ren and Chang, 2003). It performs successive orthogonal projections to find a data sample vector, considered as a target of interest, which has the maximal projection after each orthogonal projection. The potential of ATGP has been shown to have a wide range of applications, such as VD determination, anomaly detection, endmember extraction, unsupervised LSMA, etc.

MATLAB Codes of ATGP

```
function [loc,Sig]=ATGP_new(HIM,num_targets);

% input HIM is the Hyperspectral data cube of size [height width bands].
% input M is the number of target p that you need to extract.
% output Sig is the signatures corresponding to the extracted targets. It is of
size [bands p].
% output loc is the positions of the targets. It is of size [p 2]
% with the first column being the vertical position, the second column being the
horizontal position.

bnd=size(HIM,3);
xx=size(HIM,1);
yy=size(HIM,2);

r=reshape(HIM,xx*yy,bnd);
r=r';
%=====Find the first point

temp=sum(r.*r);
[a,b]=max(temp);
if (rem(b,xx)==0)
  Loc(1,1)=b/xx;
  Loc(1,2)=xx;
elseif (floor(b/xx)==0)
  Loc(1,1)=1;
  Loc(1,2)=b;
else
  Loc(1,1)=floor(b/xx)+1; % y
  Loc(1,2)=b-xx*floor(b/xx); % x
end

Sig(:,1)=r(:,b);
%==========
for m=2:num_targets
  U=Sig;
  P_U_perl=eye(bnd)-U*inv(U'*U)*U';
  y=P_U_perl*r;
  temp=sum(y.*y);
  [a,b]=max(temp);
  if (rem(b,xx)==0)
    Loc(m,1)=b/xx;
    Loc(m,2)=xx;
  elseif (floor(b/xx)==0)
    Loc(m,1)=1;
    Loc(m,2)=b;
```

```
else
    Loc(m,1)=floor(b/xx)+1; % y
    Loc(m,2)=b-xx*floor(b/xx); % x
  end
  Sig(:,m)=r(:,b);
  %disp(m)
end
%
%figure; imagesc(HIM(:,:,30)); colormap(gray); hold on
%axis off
%axis equal
%for m=1:size(Loc,1)
%  plot(Loc(m,1),Loc(m,2),'o','color','g');
%  text(Loc(m,1)+2,Loc(m,2),num2str(m),'color','y','FontSize',12);
%end
%

loc(:,1)=Loc(:,2);
loc(:,2)=Loc(:,1);
```

A.6.2 UNCLS

- **Algorithm name:** unsupervised non-negativity constrained least squares (UNCLS)
- **Authors:** Chein-I Chang and Daniel Heinz
- **Category:** unsupervised least-squares-based filter
- **Designed criteria:** least squares error
- **Designed method:** least squares constrained method
- **Typical use (LOI's addressed):** detection, spectral unmixing, classification, quantification
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** no target knowledge required
- **Sensitivity to LOI (target knowledge):** moderate
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The UNCLS is an unsupervised version of the NCLS developed by Chang and Heinz (2000b). Unlike NCLS, which requires complete *a priori* target knowledge to perform spectral unmixing, UNCLS performs unsupervised target detection by using the NCLS to generate a set of potential targets directly from the data to be processed. So, its main goal is to find targets of interest without any prior knowledge. Because of that, UNCLS is primarily used for target detection and endmember extraction as opposed to NCLS, whose main functionality is spectral unmixing. That is, the NCLS and the UNCLS have rather different applications.

MATLAB Codes of UNCLS

```
function [location_pair,spectrum_of_targets]=UNCLS(ImageCub,
NumberOfClass);
```

```
% input ImageCub is of size [height width bands].
% input NumberOfClass is the number of target p that you need to extract.
% output spectrum_of_targets is the signatures corresponding to the extracted
targets. It is of size [bands p].
% output location_pair is the positions of the targets. It is of size [p 2] with
the first column being the vertical position, the second column being the
horizontal position.

NumberOfClass=NumberOfClass-1;
[height, width, NumberOfSpectrum]=size(ImageCub);
[xx, yy, bnd]=size(ImageCub);

r=reshape(ImageCub,xx*yy,bnd);
r=r';
%height=size(ImageCub,1);
%width=size(ImageCub,2);
%MatrixH=zeros( NumberOfSpectrum, NumberOfClass+1);

location_pair=zeros(NumberOfClass,2);
Brightest=10^(-100);
count=0;
NumberOfExisted=1;

%=====Find the first point=====

temp=sum(r.*r);
[a,b]=max(temp);

if (rem(b,xx)==0)
  Loc(1,1)=b/xx;
  Loc(1,2)=xx;
elseif (floor(b/xx)==0)
  Loc(1,1)=1;
  Loc(1,2)=b;
else
  Loc(1,1)=floor(b/xx)+1; % y
  Loc(1,2)=b-xx*floor(b/xx); % x
end

%location_pair(1:NumberOfExisted,:)=squeeze(BrightForEachClass(1,:));
location_pair(:,1)=Loc(:,2);
location_pair(:,2)=Loc(:,1);
MatrixH=r(:,b);

for II=1:NumberOfClass
%  disp([' Class : ' int2str(II) ]);
%  disp(II+NumberOfExisted);
  minError=0.00000000000000000000000001;
```

```
  for b=1:xx*yy
     trypixel=r(:,b);
     max_value=max(trypixel);
     if (max_value>0)
       [abundance]= NCLS(MatrixH,trypixel);
       error_vector=MatrixH*abundance-trypixel;
       error=error_vector'*error_vector;
       if(error>minError)
         minError=error;
         if (rem(b,xx)==0)
           location_pair(II+NumberOfExisted,1)=xx;
           location_pair(II+NumberOfExisted,2)=b/xx;
         elseif (floor(b/xx)==0)
           location_pair(II+NumberOfExisted,1)=b;
           location_pair(II+NumberOfExisted,2)=1;
         else
           location_pair(II+NumberOfExisted,1)=b-xx*floor(b/xx);
           location_pair(II+NumberOfExisted,2)=floor(b/xx)+1;
         end
       end
     end
  end
  MatrixH=[MatrixH
squeeze(ImageCub(location_pair(II+NumberOfExisted,1),location_pair(II+
NumberOfExisted,2),:))];
end
spectrum_of_targets=MatrixH;
```

A.6.3 UFCLS

- **Algorithm name:** unsupervised fully constrained least squares (UFCLS)
- **Authors:** Daniel Heinz and Chein-I Chang
- **Category:** unsupervised least-squares-based filter
- **Designed criteria:** least squares error
- **Designed method:** least squares constrained method
- **Typical use (LOI's addressed):** detection, spectral unmixing, classification, quantification
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** no target knowledge required
- **Sensitivity to LOI (target knowledge):** moderate
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:** simple, easy to use, but not real time
- **Brief description:**
  The UFCLS is an unsupervised version of the FCLS developed by Heinz and Chang (2001). Unlike FCLS, which requires complete *a priori* target knowledge to perform spectral unmixing for abundance fraction estimation, UFCLS performs unsupervised target detection by using

FCLS to generate a set of potential targets directly from the data to be processed. It is identical to UNCLS with the only difference that UFCLS uses FCLS instead of NCLS to find targets of interest. So, its main goal is to find targets of interest without any prior knowledge. Because of that, UFCLS is primarily used for target detection and endmember extraction as opposed to FCLS, whose main functionality is to unmix data sample vectors in terms of abundance fractions. Therefore, FCLS and UFCLS indeed have different applications.

MATLAB Codes of UFCLS

```
function [location_pair,spectrum_of_targets]=UFCLS(ImageCub,
NumberOfClass);
% input ImageCub is of size [height width bands].
% input NumberOfClass is the number of target p that you need to extract.
% output spectrum_of_targets is the signatures corresponding to the extracted
targets. It is of size [bands p].
% output location_pair is the positions of the targets. It is of size [p 2] with
the first column being the vertical position, the second column being the
horizontal position.

NumberOfClass=NumberOfClass-1;
[height, width, NumberOfSpectrum]=size(ImageCub);
[xx, yy, bnd]=size(ImageCub);

r=reshape(ImageCub,xx*yy,bnd);
r=r';
%height=size(ImageCub,1);
%width=size(ImageCub,2);
%MatrixH=zeros( NumberOfSpectrum, NumberOfClass+1);

location_pair=zeros(NumberOfClass,2);
Brightest=10^(-100);
count=0;
NumberOfExisted=1;

%=====Find the first point

temp=sum(r.*r);
[a,b]=max(temp);

if (rem(b,xx)==0)
  Loc(1,1)=b/xx;
  Loc(1,2)=xx;
elseif (floor(b/xx)==0)
  Loc(1,1)=1;
  Loc(1,2)=b;
else
  Loc(1,1)=floor(b/xx)+1; % y
  Loc(1,2)=b-xx*floor(b/xx); % x
```

```
end

%location_pair(1:NumberOfExisted,:)=squeeze(BrightForEachClass(1,:));
location_pair(:,1)=Loc(:,2);
location_pair(:,2)=Loc(:,1);
MatrixH=r(:,b);
delta=1/10/max(max(MatrixH));
for II=1:NumberOfClass
%  disp([' Class :' int2str(II) ]);
%  disp(II+NumberOfExisted);
  minError=0.00000000000000000000000001;
  for b=1:xx*yy
     trypixel=r(:,b);
     max_value=max(trypixel);
     if (max_value>0)
       [abundance]= FCLS(MatrixH,trypixel,delta);
       error_vector=MatrixH*abundance-trypixel;
       error=error_vector'*error_vector;
       if(error>minError)
         minError=error;
         if (rem(b,xx)==0)
           location_pair(II+NumberOfExisted,1)=xx;
           location_pair(II+NumberOfExisted,2)=b/xx;
         elseif (floor(b/xx)==0)
           location_pair(II+NumberOfExisted,1)=b;
           location_pair(II+NumberOfExisted,2)=1;
         else
           location_pair(II+NumberOfExisted,1)=b-xx*floor(b/xx);
           location_pair(II+NumberOfExisted,2)=floor(b/xx)+1;
         end
       end
    end
  end
  MatrixH=[MatrixH
squeeze(ImageCub(location_pair(II+NumberOfExisted,1),location_pair(II+
NumberOfExisted,2),:))];
end
spectrum_of_targets=MatrixH;
```

A.7 Constrained Band Selection

DR and band selection (BS) have been widely used for data compression. MATLAB codes of several component analysis-based DR techniques have been provided in Section A.3. This section presents a new approach to BS, called constrained band selection (CBS), which is based on the constrained energy minimization (CEM) developed for target detection in Chapter 2.

- **Algorithm name:** constrained band selection (CBS)
- **Authors:** Chein-I Chang and Su Wang

- **Category:** spectral filter
- **Designed criteria:** least squares error
- **Designed method:** linearly least squares constrained method
- **Typical use (LOI's addressed):** BS
- **Inputs:** reflectance or radiance cube
- **Outputs:** gray-scale abundance fractional images
- **Assumptions:** prior knowledge of the number of bands to be selected
- **Sensitivity to LOI (target knowledge):** moderate
- **Sensitivity to noise:** moderate
- **Operating bands:** VNIR through LWIR
- **Maturity:** mature/operational
- **Effectiveness:** high
- **Implementation:**
- **Brief description:**
  CBS was introduced in Chang and Wang (2006) to explore the idea of CEM that can be used for BS. It interprets a band image as a desired target signature vector while considering other band images as unknown signature vectors. As a result, the proposed CBS linearly constrains a band image while also minimizing band correlation or dependence provided by other band images is referred to as CEM-CBS. Four different criteria, referred to as band correlation constraint (BCC), band correlation minimization (BCM), band dependence constraint (BDC), and band dependence minimization (BDM), are derived for CEM-CBS. Since dimensionality resulting from conversion of a band image to a vector may be huge, the CEM-CBS is further reinterpreted as linearly constrained minimum variance (LCMV)-based CBS by constraining a band image as a matrix, where the same four criteria BCM, BCC, BDC, and BDM can also be used for LCMV-CBS. In order to determine the number of bands required to select, p, a recently developed concept, called virtual dimensionality (VD) is used to estimate the p. Once the p is determined, a set of p desired bands can be selected by the CEM/LCMV-CBS. In what follows, only MATLAB codes of four versions of CEM-CBS are provided, but these codes can be easily modified for their counterparts of LCMV-CBS.

MATLAB Codes of CEM/BCC

```
function [ band_select, newcube ] = CEM_BCC(imagecube, num);

close all;
[ xx, yy, band_num ] = size(imagecube);
%%%% get band image correlation %%%%
test_image = reshape(imagecube, xx*yy, band_num);
R = test_image * test_image'/band_num;
tt = inv(R);

%%%% get prioritization score for each band %%%%
for i = 1: band_num
  endmember_matrix = reshape(squeeze(imagecube(:, :, i)), xx*yy, 1);
  W = tt * endmember_matrix * inv(endmember_matrix' * tt * endmember_matrix);
  for j = 1: band_num
    if (i ~= j)
```

```
      test = reshape(squeeze(imagecube(:, :, j)), xx*yy, 1);
      score(i, j) = test' * W;
    else
      score(i,j) = 1;
    end
  end
end
clear endmember_matrix;
clear W;
clear i,j;

%%%% select small subset of original bands %%%%
weight = zeros(1, band_num);
for i = 1: band_num
  test = score(i,:);
  scalar = sum(test) - score(i,i);
  weight(i) = scalar;
end
%weight = abs(weight);
original = 1:band_num;
coefficient_integer = weight * 100000;
band_select = zeros(1, num);
i = 1;
while (i <= num)
  max_coe = max(coefficient_integer(:));
  index = find(coefficient_integer == max_coe);
  if (length(index) == 1)
    band_select(i) = original(index);
    i = i + 1;
  else
    j = 1;
    while (j <= length(index))&(i <= num )
      band_select(i) = original(index(j));
      i = i + 1;
      j = j + 1;
    end
  end
  coefficient_integer(index) = -10^10;
end
band_sort = sort(band_select);
clear coefficient_integer;
clear max_coe
clear index;
clear i;
clear j;

%get new imagecube
newcube = zeros(xx,yy, num);
```

```
for i = 1: xx
  for j = 1: yy
    test = squeeze(imagecube(i,j,:));
    newcube(i,j,:) = test(band_sort);
  end
end
```

MATLAB Codes of CEM/BCM

```
function [ band_select, newcube ] = CEM_BCM(imagecube, num);

close all;
[ xx, yy, band_num ] = size(imagecube);

%%%% get band image correlation %%%%
test_image = reshape(imagecube, xx*yy, band_num);
R = test_image * test_image'/band_num;

%%%% get prioritization score for each band %%%%
for i = 1: band_num
  endmember_matrix = reshape(squeeze(imagecube(:, :, i)), xx*yy, 1);
  W = tt * endmember_matrix * inv(endmember_matrix' * tt * endmember_matrix);
  score(i) = W' * R * W;
end
clear endmember_matrix;
clear W;
clear i;

%%%% select small subset of original bands %%%%
%weight = abs(score);
original = 1:band_num;
coefficient_integer = weight * 100000;
band_select = zeros(1, num);
i = 1;
while (i <= num)
  max_coe = max(coefficient_integer(:));
  index = find(coefficient_integer == max_coe);
  if (length(index) == 1)
    band_select(i) = original(index);
    i = i + 1;
  else
    j = 1;
    while (j <= length(index)) & (i <= num )
      band_select(i) = original(index(j));
      i = i + 1;
      j = j + 1;
    end
```

```
  end
  coefficient_integer(index) = -10^10;
end
band_sort = sort(band_select);
clear coefficient_integer;
clear max_coe
clear index;
clear i;
clear j;

%get new imagecube
newcube = zeros(xx,yy, num);
for i = 1: xx
  for j = 1: yy
    test = squeeze(imagecube(i,j,:));
    newcube(i,j,:) = test(band_sort);
  end
end
```

MATLAB Codes of CEM/BDM

```
function [ band_select, newcube ] = CEM_BDM(imagecube, num);

close all;
[ xx, yy, band_num ] = size(imagecube);

%%%% get band image correlation %%%%
test_image = reshape(imagecube, xx*yy, band_num);
R = test_image * test_image'/band_num;

%%%% get prioritization score for each band %%%%
for i = 1: band_num
  endmember_matrix = reshape(squeeze(imagecube(:, :, i)), xx*yy, 1);
  R_new = R - endmember_matrix * endmember_matrix';
  R_new = R_new/(band_num -1);
  tt = inv(R_new);
  W = tt * endmember_matrix * inv(endmember_matrix' * tt * endmember_matrix);
  score(i) = W' * R * W;
end
clear endmember_matrix;
clear W;
clear i;

%%%% select small subset of original bands %%%%
%weight = abs(score);
original = 1:band_num;
coefficient_integer = weight * 100000;
```

```
band_select = zeros(1, num);
i = 1;
while (i <= num)
  max_coe = max(coefficient_integer(:));
  index = find(coefficient_integer == max_coe);
  if (length(index) == 1)
    band_select(i) = original(index);
    i = i + 1;
  else
    j = 1;
    while (j <= length(index)) & (i <= num )
      band_select(i) = original(index(j));
      i = i + 1;
      j = j + 1;
    end
  end
  coefficient_integer(index) = -10^10;
end
band_sort = sort(band_select);
clear coefficient_integer;
clear max_coe
clear index;
clear i;
clear j;
%get new imagecube
newcube = zeros(xx,yy, num);
for i = 1: xx
  for j = 1: yy
    test = squeeze(imagecube(i,j,:));
    newcube(i,j,:) = test(band_sort);
  end
end
```