Matlab codes

We implemented Algorithms 2.1, 2.2, NHZ1 and NHZ2 for constrained nonlinear monotone equations with the following Matlab codes:

(1) Code for Algorithm 2.1

```
function DLclustering(fnum,dimnum,xnum,xlrange,tol,maxit,
    maxfev)
% A modied Dai-Liao method for constrained systems and image
    de-blurring via clustering of eigenvalues
% Mohammed Yusuf Waziri, Kabiru Ahmed, Abubakar Sani Halilu,
    Salisu Murtala, Habibu Abdullahi,
% and Ya'u Balarabe Musa 2024
% Global convergence method
% call: dlcs(f,x0,tol,maxit)
% Input:  dimnum= dimension
%          fnum= function number
%         xnum= initial iterate number
%         tol= stoping tolerance
%         maxit= maximum number of iteration
tic;
%%%%% default maxit, fev and tol, constant input
    %%%%%%%%%%%%%%%%%%%%%%%
if nargin<7
     maxfev=2000;
end
if nargin<6
     maxit=1000; % default max. iter
end
if nargin<5
     tol=10^(-10); % default tolerance
end
%%%%%%%%%%%%% variable input
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin<4
     xlrange=[]; % excel range
end
if nargin<3
     xnum=1; % default initial point
end
if nargin<2
     dimnum=1; % default problem
end
if nargin<1
     fnum=1; % default dimension
end
```

```
%%%%%%%%%%%%%%%%%%% defining dimension
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch dimnum
    case 1
        dim=5000;
    case 2
        dim=10000;
    case 3
        dim=50000;
    otherwise
        dim=dimnum;    % for any other dimension
end
%%%%%%%%%%%%%%%%%% defining problems
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem=fnum;
switch problem
       case 1
      l=0; u=+inf;
        f='cp0';
        proj='Pj';
       case 2
      l=0; u=+inf;
        f='cp5';
        proj='Pj';
       case 3
      l=0; u=+inf;
        f='k26';
        proj='Pj';
       case 4
       l=0; u=+inf;
        f='kab2';
        proj='Pj';
      case 5
      l=0; u=+inf;
        f='kab4';
        proj='Pj';
      case 6
      l=0; u=+inf;
        f='kab5';
        proj='Pj';
    otherwise
        f='fnum'; %for any other problem
end
%%%%%%%%%%%%%%%%%% defining initial points
   %%%%%%%%%%%%%%%%%%%%%%%
guess=xnum;
```

```
switch guess
    case 1
        x0=(((1:dim)')/dim);
    case 2
        x0=1-(((1:dim)')/dim);
    case 3
        x0=2-(((1:dim)')/dim);
    case 4
        x0=((dim-(1:dim))/dim)';
    case 5
        x0=(-1*((-1).^(1:dim)-2)')/4;
    case 6
        x0=(-2*((-1).^(1:dim)-2)')/2;
    otherwise


        x0=xnum; %for any other initial point
end
%Step 0 Initialization

ITER=0; %iteration
FEV=0; % function evaluation
bck=0; % backtracking counter
% line search parameters
% Step 1 stopping rule
F0=feval(f,x0); % evaluating F(x0);
FEV=FEV+1;
norm_F0=sqrt(sum(F0.^2)); % norm of F(x0)
d0=-F0; % initial direction
%%%%%% Step 2 main loop%%%%%%%%%%%%%%%%%%%%%%%%
while(ITER<=maxit && norm_F0>tol)
    ro=0.49; m=0; sig=0.0001; %w=0.6;
F0=feval(f,x0);
    bita=1;
    % Step 3: line search
    while (-((feval(f,x0+bita*(ro)^m*d0))'*d0) <sig*bita*(ro)^
       m*(norm(d0))^2 && m<=10)
               m=m+1;
        FEV=FEV+1;
    end
    if FEV>=maxfev
            disp('maximum number of function evalution reached
               ')
            return;
    end
    % backtracking counter
```

```
    if m
        bck=bck+1;
    end
    alph=bita*(ro)^(m);
    z=x0+alph*d0;
    Fz= feval(f,z); % computing f(z)
    FEV=FEV+1;
    if (feval(proj,(z),l,u)==z & norm(Fz)<tol)
        x0=z;
        F0=Fz;
        norm_F0= norm(F0);
        disp('zk is in the convex set and its the solution at
           iteration number')
        disp((num2str(ITER)))
        break
    else
   zetak=Fz'*(x0-z)/(Fz'*Fz); % computing zetak
        P=feval(proj,(x0-1.8*zetak*Fz),l,u); % projection on
           convex set
    x=P;
    F1=feval(f,x);
    s=z-x0;
    y=Fz-F0;
    r=0.01;
    gam=4;
    wk=y+r*s;
    %tk=2*gam*(s'*wk)/(s'*s)-(wk'*wk)/(s'*wk);
    tk=(1/gam)*(wk'*wk)/(s'*wk)+(1/gam)*(s'*wk)/(s'*s);
    bk1=(F1'*wk)/(d0'*wk);
    bk2=(F1'*s)/(d0'*wk);
    d1= -(1/gam)*F1+(1/gam)*bk1*d0-tk*bk2*d0;
    end
    x0=x;
    F0=F1;
    d0=d1;
    norm_F0=sqrt(sum(F0.^2));
    ITER=ITER+1;
end
x0;
disp([num2str(ITER) ' / ' num2str(FEV)    ' / '  num2str(bck)
   ' / ' num2str(toc)  ' / ' num2str(norm_F0) ])
 disp((num2str(f)))
disp((num2str(dim)))
table1='ClusteringDK.xlsx';
 T={ITER,FEV,toc,norm_F0};
 sheet=fnum;
```

```
 xclRange=xlrange;
 xlswrite(table1,T,sheet,xclRange);
%  table1='dlcs.xlsx';
%  T={num2str(ITER),num2str(FEV),num2str(toc),num2str(norm_F0)
   };
%  sheet=fnum;
%  xlRange=xlrange;
%  xlswrite(table1,T,sheet,xlRange);
%winopen(table1)
toc;
```

(2) Code for Algorithm 2.2

```
function DKclustering(fnum,dimnum,xnum,xlrange,tol,maxit,
   maxfev)
% Image de-blurring with a Dai-Kou-type method via clustering
   of eigenvalues
% Mohammed Yusuf Waziri, \textbf{Kabiru Ahmed}, Abubakar Sani
   Halilu, Salisu Murtala, Habibu Abdullahi,
% and Ya'u Balarabe Musa 2024
% Global convergence method
% call: dlcs(f,x0,tol,maxit)
% Input:  dimnum= dimension
%          fnum= function number
%         xnum= initial iterate number
%         tol= stoping tolerance
%         maxit= maximum number of iteration
tic;
%%%% default maxit, fev and tol, constant input
   %%%%%%%%%%%%%%%%%%%%%%%
if nargin<7
    maxfev=2000;
end
if nargin<6
    maxit=1000; % default max. iter
end
if nargin<5
    tol=10^(-10); % default tolerance
end
%%%%%%%%%%%%% variable input
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin<4
    xlrange=[]; % excel range
end
if nargin<3
    xnum=1; % default initial point
end
```

```
if nargin<2
    dimnum=1; % default problem
end
if nargin<1
    fnum=1; % default dimension
end
%%%%%%%%%%%%%%%%%%% defining dimension
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch dimnum
    case 1
        dim=5000;
    case 2
        dim=10000;
    case 3
        dim=50000;
    otherwise
        dim=dimnum;    % for any other dimension
end
%%%%%%%%%%%%%%%%%% defining problems
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem=fnum;
switch problem
     case 1
      l=0; u=+inf;
        f='kab4';
        proj='Pj';
       case 2
      l=0; u=+inf;
        f='cp5';
        proj='Pj';
       case 3
      l=0; u=+inf;
        f='k26';
        proj='Pj';
       case 4
       l=0; u=+inf;
        f='kab2';
        proj='Pj';
      case 5
      l=0; u=+inf;
        f='kab4';
        proj='Pj';
      case 6
      l=0; u=+inf;
        f='kab5';
        proj='Pj';
```

```
    otherwise
        f='fnum'; %for any other problem
end
%%%%%%%%%%%%%%%%%% defining initial points
   %%%%%%%%%%%%%%%%%%%%%%%
guess=xnum;
switch guess
    case 1
        x0=(((1:dim)')/dim);
    case 2
        x0=1-(((1:dim)')/dim);
    case 3
        x0=2-(((1:dim)')/dim);
    case 4
        x0=((dim-(1:dim))/dim)';
    case 5
        x0=(-1*((-1).^(1:dim)-2)')/4;
    case 6
        x0=(-2*((-1).^(1:dim)-2)')/2;
    otherwise


        x0=xnum; %for any other initial point
end
%Step 0 Initialization

ITER=0; %iteration
FEV=0; % function evaluation
bck=0; % backtracking counter
% line search parameters
% Step 1 stopping rule
F0=feval(f,x0); % evaluating F(x0);
FEV=FEV+1;
norm_F0=sqrt(sum(F0.^2)); % norm of F(x0)
d0=-F0; % initial direction
%%%%%% Step 2 main loop%%%%%%%%%%%%%%%%%%%%%%%%
while(ITER<=maxit && norm_F0>tol)
    ro=0.48; m=0; sig=0.0001;
F0=feval(f,x0);
    bita=1;
%     tau=1;
%     e=1;
%     nm=(norm(((feval(f,x0+bita*(ro)^m*d0)))));
    % Step 3: line search
    while (-((feval(f,x0+bita*(ro)^m*d0))'*d0) <sig*bita*(ro)^
       m*(norm(d0))^2 && m<=10)
```

```
    m=m+1;
    FEV=FEV+1;
end
if FEV>=maxfev
        disp('maximum number of function evalution reached
            ')
        return;
end
% backtracking counter
if m
    bck=bck+1;
end
alph=bita*(ro)^(m);
z=x0+alph*d0;
Fz= feval(f,z); % computing f(z)
FEV=FEV+1;
if (feval(proj,(z),l,u)==z & norm(Fz)<tol)
    x0=z;
    F0=Fz;
    norm_F0= norm(F0);
    disp('zk is in the convex set and its the solution at
       iteration number')
    disp((num2str(ITER)))
    break
else
zetak=Fz'*(x0-z)/(Fz'*Fz); % computing zetak
    P=feval(proj,(x0-1.8*zetak*Fz),l,u); % projection on
       convex set
x=P;
F1=feval(f,x);
s=z-x0;
y=Fz-F0;
r=0.01;
gam=0.25;
wk=y+r*s;
%tk=2*gam*(s'*wk)/(s'*s)-(wk'*wk)/(s'*wk);
%tauk=2*gam*(s'*wk)/(s'*s);
bk2=(gam*(F1'*wk)/(d0'*wk)-(gam*(wk'*wk)/(s'*wk)+gam*(s'*
   wk)/(s'*s))*(F1'*s)/(d0'*wk));
d1= -gam*F1+bk2*d0;
end
x0=x;
F0=F1;
d0=d1;
norm_F0=sqrt(sum(F0.^2));
```

```
    ITER=ITER+1;
end
x0;
disp([num2str(ITER) ' / ' num2str(FEV)    ' / '  num2str(bck)
   ' / ' num2str(toc)  ' / ' num2str(norm_F0) ])
 disp((num2str(f)))
disp((num2str(dim)))
table1='ClusteringDK.xlsx';
 T={ITER,FEV,toc,norm_F0};
 sheet=fnum;
 xclRange=xlrange;
 xlswrite(table1,T,sheet,xclRange);
%  table1='dlcs.xlsx';
%  T={num2str(ITER),num2str(FEV),num2str(toc),num2str(norm_F0)
   };
%  sheet=fnum;
%  xlRange=xlrange;
%  xlswrite(table1,T,sheet,xlRange);
%winopen(table1)
toc;
```

(3) Code for NHZ1

```
function NHZM1(fnum,dimnum,xnum,xlrange,tol,maxit,maxfev)
% Sparse signal reconstruction via Hager-Zhang-type schemes
   for constrained system of nonlinear equations,
% Mohammed Yusuf Waziri, Kabiru Ahmed, Abubakar Sani Halilu,
   and Salisu Murtala.
% Optimization, Vol 73, Issue 6, pp. 1949 - 1980, 2023.
% call: dlcs(f,x0,tol,maxit)
% Input:  dimnum= dimension
%          fnum= function number
%         xnum= initial iterate number
%         tol= stoping tolerance
%         maxit= maximum number of iteration
tic;
%%%% default maxit, fev and tol, constant input
   %%%%%%%%%%%%%%%%%%%%%%%
if nargin<7
    maxfev=2000;
end
if nargin<6
    maxit=1000; % default max. iter
end
if nargin<5
    tol=10^(-10); % default tolerance
end
```

```
%%%%%%%%%%%%% variable input
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin<4
    xlrange=[]; % excel range
end
if nargin<3
    xnum=1; % default initial point
end
if nargin<2
    dimnum=1; % default problem
end
if nargin<1
    fnum=1; % default dimension
end
%%%%%%%%%%%%%%%%%%% defining dimension
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch dimnum
    case 1
        dim=5000;
    case 2
        dim=10000;
    case 3
        dim=50000;
    otherwise
        dim=dimnum;    % for any other dimension
end
%%%%%%%%%%%%%%%%%% defining problems
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem=fnum;
switch problem
      case 1
      l=0; u=+inf;
        f='cp0';
        proj='Pj';
       case 2
      l=0; u=+inf;
        f='cp5';
        proj='Pj';
       case 3
      l=0; u=+inf;
        f='k26';
        proj='Pj';
       case 4
       l=0; u=+inf;
        f='kab2';
        proj='Pj';
```

```
        case 5
        l=0; u=+inf;
          f='kab4';
          proj='Pj';
        case 6
        l=0; u=+inf;
          f='kab5';
          proj='Pj';
    otherwise
          f='fnum'; %for any other problem
end
%%%%%%%%%%%%%%%%%% defining initial points
   %%%%%%%%%%%%%%%%%%%%%%%
guess=xnum;
switch guess
    case 1
        x0=(((1:dim)')/dim);
    case 2
        x0=1-(((1:dim)')/dim);
    case 3
        x0=2-(((1:dim)')/dim);
    case 4
        x0=((dim-(1:dim))/dim)';
    case 5
        x0=(-1*((-1).^(1:dim)-2)')/4;
    case 6
        x0=(-2*((-1).^(1:dim)-2)')/2;
    otherwise

        x0=xnum; %for any other initial point
end
%Step 0 Initialization

ITER=0; %iteration
FEV=0; % function evaluation
bck=0; % backtracking counter
% line search parameters
% Step 1 stopping rule
F0=feval(f,x0); % evaluating F(x0);
FEV=FEV+1;
norm_F0=sqrt(sum(F0.^2)); % norm of F(x0)
d0=-F0; % initial direction
%%%%%% Step 2 main loop%%%%%%%%%%%%%%%%%%%%%%%%%
while(ITER<=maxit && norm_F0>tol)
    ro=0.51; sig=0.0001; m=0;
F0=feval(f,x0);
```

```
bita=1;
% Step 3: line search
while  (-((feval(f,x0+bita*(ro)^m*d0))'*d0) < sig*bita*(ro
   )^m*(norm(d0))^2 && m<=10)

    m=m+1;
    FEV=FEV+1;
end
if FEV>=maxfev
        disp('maximum number of function evalution reached
           ')
        return;
end
% backtracking counter
if m
    bck=bck+1;
end
alph=bita*(ro)^(m);
z=x0+alph*d0;
Fz= feval(f,z); % computing f(z)
FEV=FEV+1;
if (feval(proj,(z),l,u)==z & norm(Fz)<tol)
    x0=z;
    F0=Fz;
    norm_F0=norm(F0);
    disp('zk is in the convex set and its the solution at
       iteration number')
    disp((num2str(ITER)))
   break
else
zetak=Fz'*(x0-z)/(Fz'*Fz); % computing zetak
    P=feval(proj,(x0-1.8*zetak*Fz),l,u); % projection on
       convex se
x=P;
F1=feval(f,x);
s=z-x0;
y=Fz-F0;
c=0.01;
wk=y+c*s;
gam=4;
e1=(2)/(4*gam);
thet1=(1/gam)*sqrt((s'*wk)/(norm(s)*norm(wk)))^3;
thet2=max(thet1,e1);
% modified y using line search
betak=(F1'*wk)/(d0'*wk)-(gam*thet2*(wk'*wk)*(F1'*d0))/((d0
   '*wk)^2);
```

```
    d1= -F1+betak*d0; % spectral Dai-Liao direction
    end
    x0=x;
    F0=F1;
    d0=d1;
    norm_F0=sqrt(sum(F0.^2));
    ITER=ITER+1;
end
x0;
disp([num2str(ITER) ' / ' num2str(FEV)    ' / '  num2str(bck)
   ' / ' num2str(toc)  ' / ' num2str(norm_F0) ])
 disp((num2str(f)))
disp((num2str(dim)))
table1='MHZnew.xlsx';
 T={ITER,FEV,toc,norm_F0};
 sheet=fnum;
 xclRange=xlrange;
 xlswrite(table1,T,sheet,xclRange);
%  table1='dlcs.xlsx';
%  T={num2str(ITER),num2str(FEV),num2str(toc),num2str(norm_F0)
   };
%  sheet=fnum;
%  xlRange=xlrange;
%  xlswrite(table1,T,sheet,xlRange);
%winopen(table1)
toc;
```

(4) Code for NHZ2

```
function NHZM2(fnum,dimnum,xnum,xlrange,tol,maxit,maxfev)
% Sparse signal reconstruction via Hager-Zhang-type schemes
   for constrained system of nonlinear equations,
% Mohammed Yusuf Waziri, Kabiru Ahmed, Abubakar Sani Halilu,
   and Salisu Murtala.
% Optimization, Vol 73, Issue 6, pp. 1949 - 1980, 2023.
% call: dlcs(f,x0,tol,maxit)
% Input:  dimnum= dimension
%         fnum= function number
%        xnum= initial iterate number
%        tol= stoping tolerance
%        maxit= maximum number of iteration
tic;
%%%%% default maxit, fev and tol, constant input
   %%%%%%%%%%%%%%%%%%%%%%%
if nargin<7
    maxfev=2000;
end
```

```
if nargin<6
    maxit=1000; % default max. iter
end
if nargin<5
    tol=10^(-10); % default tolerance
end
%%%%%%%%%%%%% variable input
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if nargin<4
    xlrange=[]; % excel range
end
if nargin<3
    xnum=1; % default initial point
end
if nargin<2
    dimnum=1; % default problem
end
if nargin<1
    fnum=1; % default dimension
end
%%%%%%%%%%%%%%%%%%% defining dimension
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
switch dimnum
    case 1
        dim=5000;
    case 2
        dim=10000;
    case 3
        dim=50000;
    otherwise
        dim=dimnum;    % for any other dimension
end
%%%%%%%%%%%%%%%%%% defining problems
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem=fnum;
switch problem
      case 1
      l=0; u=+inf;
        f='cp0';
        proj='Pj';
       case 2
      l=0; u=+inf;
        f='cp5';
        proj='Pj';
       case 3
      l=0; u=+inf;
```

```
        f='k26';
        proj='Pj';
       case 4
       l=0; u=+inf;
        f='kab2';
        proj='Pj';
      case 5
      l=0; u=+inf;
        f='kab4';
        proj='Pj';
      case 6
      l=0; u=+inf;
        f='kab5';
        proj='Pj';
    otherwise
        f='fnum'; %for any other problem
end
%%%%%%%%%%%%%%%%%% defining initial points
   %%%%%%%%%%%%%%%%%%%%%%%
guess=xnum;
switch guess
    case 1
        x0=(((1:dim)')/dim);
    case 2
        x0=1-(((1:dim)')/dim);
    case 3
        x0=2-(((1:dim)')/dim);
    case 4
        x0=((dim-(1:dim))/dim)';
    case 5
        x0=(-1*((-1).^(1:dim)-2)')/4;
    case 6
        x0=(-2*((-1).^(1:dim)-2)')/2;
    otherwise

        x0=xnum; %for any other initial point
end
%Step 0 Initialization

ITER=0; %iteration
FEV=0; % function evaluation
bck=0; % backtracking counter
% line search parameters
% Step 1 stopping rule
F0=feval(f,x0); % evaluating F(x0);
FEV=FEV+1;
```

```
norm_F0=sqrt(sum(F0.^2)); % norm of F(x0)
d0=-F0; % initial direction
%%%%%% Step 2 main loop%%%%%%%%%%%%%%%%%%%%%%%%
while(ITER<=maxit && norm_F0>tol)
    ro=0.50; sig=0.01; m=0;
F0=feval(f,x0);
    bita=1;
    % Step 3: line search
    while  (-((feval(f,x0+bita*(ro)^m*d0))'*d0) < sig*bita*(ro
       )^m*(norm(d0))^2 && m<=10)

        m=m+1;
        FEV=FEV+1;
    end
    if FEV>=maxfev
            disp('maximum number of function evalution reached
               ')
            return;
    end
    % backtracking counter
    if m
        bck=bck+1;
    end
    alph=bita*(ro)^(m);
    z=x0+alph*d0;
    Fz= feval(f,z); % computing f(z)
    FEV=FEV+1;
    if (feval(proj,(z),l,u)==z & norm(Fz)<tol)
        x0=z;
        F0=Fz;
        norm_F0=norm(F0);
        disp('zk is in the convex set and its the solution at
           iteration number')
        disp((num2str(ITER)))
       break
    else
   zetak=Fz'*(x0-z)/(Fz'*Fz); % computing zetak
        P=feval(proj,(x0-1.8*zetak*Fz),l,u); % projection on
           convex set
    x=P;
    F1=feval(f,x);
    s=z-x0;
    y=Fz-F0;
    c=0.1;
    wk=y+c*s;
    gam=4;
```

```
    e1=(2)/(4*gam);
    thet1=(1/gam)*(((s'*wk)^2)/((s'*s)*(wk'*wk)));
    thet2=max(thet1,e1);
    betak=((F1'*wk)/(d0'*wk))-(gam*thet2*(wk'*wk)*(F1'*d0))/((
       d0'*wk)^2);
    d1= -F1+betak*d0;
    end
    x0=x;
    F0=F1;
    d0=d1;
    norm_F0=sqrt(sum(F0.^2));
    ITER=ITER+1;
end
x0;
disp([num2str(ITER) ' / ' num2str(FEV)    ' / '  num2str(bck)
   ' / ' num2str(toc)  ' / ' num2str(norm_F0) ])
 disp((num2str(f)))
disp((num2str(dim)))
table1='MHZnew.xlsx';
 T={ITER,FEV,toc,norm_F0};
 sheet=fnum;
 xclRange=xlrange;
 xlswrite(table1,T,sheet,xclRange);
toc;
```

We implemented Algorithms 2.1, 2.2, NHZ1 and NHZ2 for image de-blurring and signal recovery with the following Matlab codes:

(1) Code for Algorithm 2.1 Image de-blurring

```
function [x,x_debias,objective,times,debias_start,mses,taus]=
   ...
    DLcluster(y,A,tau,varargin)
%
% HTTCGP_CS 1.0, Nov. 29, 2019
%
% This function solves the convex problem
% arg min_x = 0.5*|| y - A x ||_2^2 + tau || x ||_1
% using the algorithm modified three-term conjugate gradient
   method, described in the following paper
%
% This code is to use the well-known code CG_DESCENT to solve
   \ell_1 norm
% regularization least square problems.
%
%
   -----------------------------------------------------------------
```

```
% Copyright (2019): Jianghua Yin
%
   --------------------------------------------------------------------

%
%
% The first version of this code by Jianghua Yin, Nov. 29,
   2019
% test for number of required parametres
if (nargin-length(varargin)) ~= 3
  error('Wrong number of required parameters');
end

% flag for initial x (can take any values except 0,1,2)
Initial_X_supplied = 3333;

% Set the defaults for the optional parameters
stopCriterion = 3;
tolA = 0.01;
tolD = 0.0001;
debias = 0;
maxiter = 10000;
maxiter_debias = 500;
miniter = 5;
miniter_debias = 5;
init = 0;
compute_mse = 0;
AT = 0;
verbose = 1;
continuation = 0;
cont_steps = -1;
firstTauFactorGiven = 0;

% Set the defaults for outputs that may not be computed
debias_start = 0;
x_debias = [];
mses = [];

% 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)
    switch upper(varargin{i})
     case 'STOPCRITERION'
```

```
  stopCriterion = varargin{i+1};
case 'TOLERANCEA'
  tolA = varargin{i+1};
case 'TOLERANCED'
  tolD = varargin{i+1};
case 'DEBIAS'
  debias = varargin{i+1};
case 'MAXITERA'
  maxiter = varargin{i+1};
case 'MAXITERD'
  maxiter_debias = varargin{i+1};
case 'MINITERA'
  miniter = varargin{i+1};
case 'MINITERD'
  miniter_debias = varargin{i+1};
case 'INITIALIZATION'
  if prod(size(varargin{i+1})) > 1   % initial x supplied
      as array
         init = Initial_X_supplied;      % flag to be
            used below
         x = varargin{i+1};
  else
         init = varargin{i+1};
  end
case 'MONOTONE'
  enforceMonotone = varargin{i+1};
case 'CONTINUATION'
  continuation = varargin{i+1};
case 'CONTINUATIONSTEPS'
  cont_steps = varargin{i+1};
case 'FIRSTTAUFACTOR'
  firstTauFactor = varargin{i+1};
  firstTauFactorGiven = 1;
case 'TRUE_X'
  compute_mse = 1;
  true = varargin{i+1};
case 'ALPHAMIN'
  alphamin = varargin{i+1};
case 'ALPHAMAX'
  alphamax = varargin{i+1};
case 'AT'
  AT = varargin{i+1};
case 'VERBOSE'
  verbose = varargin{i+1};
otherwise
 % Hmmm, something wrong with the parameter string
```

```
      error(['Unrecognized option: ''' varargin{i} '''']);
    end;
  end;
end
%%%%%%%%%%%%%%

if (sum(stopCriterion == [0 1 2 3 4 5])==0)
  error(['Unknown stopping criterion']);
end

% if A is a function handle, we have to check presence of AT,
if isa(A, 'function_handle') & ~isa(AT,'function_handle')
  error(['The function handle for transpose of A is missing'])
     ;
end

% if A is a matrix, we find out dimensions of y and x,
% and create function handles for multiplication by A and A',
% so that the code below doesn't have to distinguish between
% the handle/not-handle cases
if ~isa(A, 'function_handle')
  AT = @(x) (x'*A)'; %A'*x;
  A = @(x) A*x;
end
% from this point down, A and AT are always function handles.

% Precompute A'*y since it'll be used a lot
Aty = AT(y);

% Initialization
switch init
    case 0   % initialize at zero, using AT to find the size
       of x
       x = AT(zeros(size(y)));
    case 1   % initialize randomly, using AT to find the size
       of x
       x = randn(size(AT(zeros(size(y)))));
    case 2   % initialize x0 = A'*y
       x = Aty;
    case Initial_X_supplied % initial x was given by user
       % initial x was given as a function argument; just
          check size
       if size(A(x)) ~= size(y)
          error(['Size of initial x is not compatible with A'
             ]);
       end
```

```
    otherwise
       error(['Unknown ''Initialization'' option']);
end

% now check if tau is an array; if it is, it has to
% have the same size as x
if prod(size(tau)) > 1
   try,
      dummy = x.*tau;
   catch,
      error(['Parameter tau has wrong dimensions; it should be
          scalar or size(x)']),
   end
end


% if the true x was given, check its size
if compute_mse & (size(true) ~= size(x))
   error(['Initial x has incompatible size']);
end

% if tau is scalar, we check its value; if it's large enough,
% the optimal solution is the zero vector
if prod(size(tau)) == 1
   aux = AT(y);
   max_tau = max(abs(aux(:)));
   if tau >= max_tau      %
      x = zeros(size(aux));
      if debias
         x_debias = x;
      end
      objective(1) = 0.5*(y(:)'*y(:));
      times(1) = 0;
      if compute_mse
          mses(1) = sum(true(:).^2);
      end
      return
   end                     %
end

% initialize u and v
u =  x.*(x >= 0);
v = -x.*(x <  0);

% define the indicator vector or matrix of nonzeros in x
nz_x = (x ~= 0.0);
```

```
num_nz_x = sum(nz_x(:));

% start the clock
t0 = cputime;

% store given tau, because we're going to change it in the
% continuation procedure
final_tau = tau;

% store given stopping criterion and threshold, because we're
   going
% to change them in the continuation procedure
final_stopCriterion = stopCriterion;
final_tolA = tolA;

% set continuation factors
if continuation&&(cont_steps > 1)
   % If tau is scalar, first check top see if the first factor
       is
   % too large (i.e., large enough to make the first
   % solution all zeros). If so, make it a little smaller than
       that.
   % Also set to that value as default
   if prod(size(tau)) == 1
      if (firstTauFactorGiven == 0)|(firstTauFactor*tau >=
         max_tau)
         firstTauFactor = 0.5*max_tau / tau;
         if verbose
             fprintf(1,'\n setting parameter FirstTauFactor\n'
                )
         end
      end
   end
   cont_factors = 10.^[log10(firstTauFactor):...
                       log10(1/firstTauFactor)/(cont_steps-1):0];
end

if ~continuation
  cont_factors = 1;
  cont_steps = 1;
end

iter = 1;
if compute_mse
       mses(iter) = sum((x(:)-true(:)).^2);
end
```

```
keep_continuation = 1;
cont_loop = 1;
iter = 1;
taus = [];
sigma = 0.0001;

% loop for continuation
while keep_continuation
    % Compute and store initial value of the objective
       function
    resid =  y - A(x);
    if cont_steps == -1
        gradq = AT(resid);
        tau = max(final_tau,0.2*max(abs(gradq)));
        if tau == final_tau
            stopCriterion = final_stopCriterion;
            tolA = final_tolA;
            keep_continuation = 0;                  % stop
               continuation
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    else
        tau = final_tau * cont_factors(cont_loop);%
        if cont_loop == cont_steps
            stopCriterion = final_stopCriterion;
            tolA = final_tolA;
            keep_continuation = 0;                 %
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    end
    taus = [taus tau];

    if verbose
        fprintf(1,'\nSetting tau = %0.5g\n',tau)
    end
    % if in first continuation iteration, compute and store
    % initial value of the objective function
    if cont_loop == 1
        alpha = 1.0;
        f = 0.5*(resid(:)'*resid(:)) + ...
             sum(tau(:).*u(:)) + sum(tau(:).*v(:));
```

```
    objective(1) = f;
    if compute_mse
        mses(1) = (x(:)-true(:))'*(x(:)-true(:));
    end
    if verbose
        fprintf(1,'Initial obj=%10.6e, alpha=%6.2e,
           nonzeros=%7d\n',...
            f,alpha,num_nz_x);
    end
end
% Compute the initial gradient and the useful
% quantity resid_base
resid_base = y - resid;
% control variable for the outer loop and iteration
   counter
keep_going = 1;
if verbose
    fprintf(1,'\nInitial obj=%10.6e, nonzeros=%7d\n',f,
       num_nz_x);
end
temp = AT(resid_base);
term  =  temp - Aty;
gradu =  term + tau; % Hz+c w.r.t. u
gradv = -term + tau; % Hz+c w.r.t. v
Lu = min(u,gradu);   % F(z) = min(z,Hz+c) w.r.t. u, z = [u
    v]'
Lv = min(v,gradv);   % F w.r.t. v
du = - Lu;
dv = - Lv;
while keep_going
    % compute dx
    dx = du-dv;
    auv = A(dx);
    Bdu = AT(auv);
    Bdv = -Bdu;
    % initial steplength
    betas = 1;   %  betas = 10 for paper;
    old_Lu = Lu;
    old_Lv = Lv;
    %NormF = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
    %NormFs = sqrt(NormF);
    Lu = min(u+betas*du, gradu+betas*Bdu); % F(z+betas*d)
       w.r.t. u where d=[du;dv];
    Lv = min(v+betas*dv, gradv+betas*Bdv); % F(z+betas*d)
       w.r.t. v;
```

```
Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);  % F(z+betas*d)
   '*d
dudv = du(:)'*du(:)+dv(:)'*dv(:);       % ||d||^2
normFz = sqrt(Lu(:)'*Lu(:)+Lv(:)'*Lv(:));
% - Luvduv < sigma*betas*max(0.001,min(0.8,NormFz))*
   dudv
while - Luvduv < sigma*betas*normFz*dudv
    % -Luvduv < sigma*betas*dudv
    betas = 0.5*betas;    % betas = 0.5*betas;
    Lu = min(u+betas*du,gradu+betas*Bdu);
    Lv = min(v+betas*dv,gradv+betas*Bdv);
    normFz = sqrt(Lu(:)'*Lu(:) + Lv(:)'*Lv(:));
    Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);
end
lambda = -1.8*Luvduv*betas/normFz^2;  % lambda = -
   1.6*Luvduv*betas/(Lu(:)'*Lu(:)+Lv(:)'*Lv(:));
old_u = u;
old_v = v;
u = old_u - lambda * Lu;
v = old_v - lambda * Lv;
uvmin = 0;% min(u,v);
u = u - uvmin;
v = v - uvmin;
x = u - v;
% calculate nonzero pattern and number of nonzeros (do
    this *always*)
nz_x_prev = nz_x;
nz_x = (x~=0.0);
num_nz_x = sum(nz_x(:));
% update residual and function
ALuv = A(Lu-Lv);
resid = y - resid_base + lambda*ALuv;     % y-Ax
resid_base = resid_base - lambda*ALuv;    % Ax
prev_f = f;
f = 0.5*(resid(:)'*resid(:)) +  sum(tau(:).*u(:)) +
   ...
  sum(tau(:).*v(:));
% compute new alpha
dd  = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
% print out stuff
if verbose
    fprintf(1,'It=%4d, obj=%9.5e, alpha=%6.2e, nz=%8d
        ',...
     iter, f, alpha, num_nz_x);
end
% update iteration counts, store results and times
```

```
    iter = iter + 1;
    objective(iter) = f;
    times(iter) = cputime-t0;
    % compute the next direction
    temp = AT(resid_base);
    term  =  temp - Aty;
    gradu =  term + tau; % Hz+c w.r.t. u
    gradv = -term + tau; % Hz+c w.r.t. v
    %Lu = min(u,gradu);   % F(x)
    %Lv = min(v,gradv);
    sku = betas*du;
    skv = betas*dv;
    %
    %norms=sqrt(sksk);
    %
    r=0.01;
    yku = (Lu-old_Lu)+r*sku;
    ykv = (Lv-old_Lv)+r*skv;
    %c
    gam=4;
     skyk=sku(:)'*yku(:)+skv(:)'*ykv(:);
     sksk=sku(:)'*sku(:)+skv(:)'*skv(:);
     Fksk=old_Lu(:)'*sku(:)+old_Lv(:)'*skv(:);
     Fkyk=old_Lu(:)'*yku(:)+old_Lv(:)'*ykv(:);
     dkyk=du(:)'*yku(:)+dv(:)'*ykv(:);
     ykyk=yku(:)'*yku(:)+ykv(:)'*ykv(:);
     bk1=(Fkyk)/(dkyk);
     bk2=(Fksk)/(dkyk);
     bk3=(ykyk)/(gam*(skyk));
     bk4=(skyk)/(gam*(sksk));
     tk=bk4+bk3;
        if ((iter > 1))
        du = -1/gam*min(u,gradu)+1/gam*bk1*du-tk*bk2*du;
        dv = -1/gam*min(v,gradv)+1/gam*bk1*dv-tk*bk2*dv;
        end
    %end
%end



    if compute_mse
        err = true - x;
        mses(iter) = (err(:)'*err(:));
    end

    switch stopCriterion
```

```
case 0,
    % compute the stopping criterion based on the
       change
    % of the number of non-zero components of the
       estimate
    num_changes_active = (sum(nz_x(:)~=nz_x_prev
       (:)));
    if num_nz_x >= 1
        criterionActiveSet = num_changes_active;
    else
        criterionActiveSet = tolA / 2;
    end
    keep_going = (criterionActiveSet > tolA);
    if verbose
        fprintf(1,'Delta n-zeros = %d (target = %e
           )\n',...
          criterionActiveSet , tolA)
    end
case 1,
    % compute the stopping criterion based on the
       relative
    % variation of the objective function.
    criterionObjective = abs(f-prev_f)/(prev_f);
    keep_going =  (criterionObjective > tolA);
    if verbose
        fprintf(1,'Delta obj. = %e (target = %e)\n
           ',...
          criterionObjective , tolA)
    end
case 2,
    % stopping criterion based on relative norm of
        step taken
    delta_x_criterion = norm(Lu(:)-Lv(:))/norm(x
       (:));
    keep_going = (delta_x_criterion > tolA);
    if verbose
        fprintf(1,'Norm(delta x)/norm(x) = %e (
           target = %e)\n',...
          delta_x_criterion,tolA)
    end
case 3,
    % compute the "LCP" stopping criterion - again
        based on the previous
    % iterate. Make it "relative" to the norm of x
       .
```

```
            w = [ min(gradu(:), old_u(:)); min(gradv(:),
               old_v(:)) ];
            criterionLCP = norm(w(:), inf);
            criterionLCP = criterionLCP / ...
              max([1.0e-6, norm(old_u(:),inf), norm(old_v
                 (:),inf)]);
            keep_going = (criterionLCP > tolA);
            if verbose
                fprintf(1,'LCP = %e (target = %e)\n',
                   criterionLCP,tolA)
            end
        case 4,
            % continue if not yeat reached target value
               tolA
            keep_going = (f > tolA);
            if verbose
                fprintf(1,'Objective = %e (target = %e)\n'
                   ,f,tolA)
            end
        case 5,
            % stopping criterion based on relative norm of
                step taken
            delta_x_criterion = sqrt(dd)/sqrt(x(:)'*x(:));
            keep_going = (delta_x_criterion > tolA);
            if verbose
                fprintf(1,'Norm(delta x)/norm(x) = %e (
                   target = %e)\n',...
                  delta_x_criterion,tolA)
            end
        otherwise,
            error(['Unknown stopping criterion']);
    end % end of the stopping criteria switch

    % take no less than miniter...
    if iter<=miniter
        keep_going = 1;
    elseif iter > maxiter %and no more than maxiter
       iterations
            keep_going = 0;
    end

end % end of the main loop of keep_going

% increment continuation loop counter
cont_loop = cont_loop+1;
```

```
end % end of the continuation loop
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Print results
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if verbose
   fprintf(1,'\nFinished the main algorithm!\nResults:\n')
   fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid(:))
   fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
   fprintf(1,'Objective function = %10.3e\n',f);
   nz_x = (x~=0.0); num_nz_x = sum(nz_x(:));
   fprintf(1,'Number of non-zero components = %d\n',num_nz_x);
   fprintf(1,'CPU time so far = %10.3e\n', times(iter));
   fprintf(1,'\n');
end

% If the 'Debias' option is set to 1, we try to remove the
   bias from the l1
% penalty, by applying CG to the least-squares problem
   obtained by omitting
% the l1 term and fixing the zero coefficients at zero.

% do this only if the reduced linear least-squares problem is
% overdetermined, otherwise we are certainly applying CG to a
   problem with a
% singular Hessian

if (debias & (sum(x(:)~=0)~=0))

  if (num_nz_x > length(y(:)))
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'There are too many nonzeros in x\n\n');
      fprintf(1,'nonzeros in x: %8d, length of y: %8d\n',...
          num_nz_x, length(y(:)));
    end
  elseif (num_nz_x==0)
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'x has no nonzeros\n\n');
```

```
    end
  else
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Starting the debiasing phase...\n\n')
    end

    x_debias = x;
    zeroind = (x_debias~=0);
    cont_debias_cg = 1;
    debias_start = iter;

    % calculate initial residual
    resid = A(x_debias);
    resid = resid-y;
    resid_prev = eps*ones(size(resid));

    rvec = AT(resid);

    % mask out the zeros
    rvec = rvec .* zeroind;
    rTr_cg = rvec(:)'*rvec(:);

    % set convergence threshold for the residual || RW
       x_debias - y ||_2
    tol_debias = tolD * (rvec(:)'*rvec(:));

    % initialize pvec
    pvec = -rvec;

    % main loop
    while cont_debias_cg

      % calculate A*p = Wt * Rt * R * W * pvec
      RWpvec = A(pvec);
      Apvec = AT(RWpvec);

      % mask out the zero terms
      Apvec = Apvec .* zeroind;

      % calculate alpha for CG
      alpha_cg = rTr_cg / (pvec(:)'* Apvec(:));

      % take the step
      x_debias = x_debias + alpha_cg * pvec;
      resid = resid + alpha_cg * RWpvec;
```

```
    rvec  = rvec  + alpha_cg * Apvec;

    rTr_cg_plus = rvec(:)'*rvec(:);
    beta_cg = rTr_cg_plus / rTr_cg;
    pvec = -rvec + beta_cg * pvec;

    rTr_cg = rTr_cg_plus;

    iter = iter+1;

    objective(iter) = 0.5*(resid(:)'*resid(:)) + ...
        sum(tau(:).*abs(x_debias(:)));
    times(iter) = cputime - t0;

    if compute_mse
      err = true - x_debias;
      mses(iter) = (err(:)'*err(:));
    end

    % in the debiasing CG phase, always use convergence
       criterion
    % based on the residual (this is standard for CG)
    if verbose
      fprintf(1,' Iter = %5d, debias resid = %13.8e,
         convergence = %8.3e\n', ...
          iter, resid(:)'*resid(:), rTr_cg / tol_debias);
    end
    cont_debias_cg = ...
        (iter-debias_start <= miniter_debias )| ...
        ((rTr_cg > tol_debias) & ...
        (iter-debias_start <= maxiter_debias));

  end
  if verbose
    fprintf(1,'\nFinished the debiasing phase!\nResults:\n')
    fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid
       (:))
    fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
    fprintf(1,'Objective function = %10.3e\n',f);
    nz = (x_debias~=0.0);
    fprintf(1,'Number of non-zero components = %d\n',sum(nz
       (:)));
    fprintf(1,'CPU time so far = %10.3e\n', times(iter));
    fprintf(1,'\n');
  end
end
```

```
  if compute_mse
    mses = mses/length(true(:));
  end

end
```

(2) Code for Algorithm 2.2 Image de-blurring

```
function [x,x_debias,objective,times,debias_start,mses,taus]=
   ...
    DKcluster(y,A,tau,varargin)
%
% HTTCGP_CS 1.0, Nov. 29, 2019
%
% This function solves the convex problem
% arg min_x = 0.5*|| y - A x ||_2^2 + tau || x ||_1
% using the algorithm modified three-term conjugate gradient
   method, described in the following paper
%
% This code is to use the well-known code CG_DESCENT to solve
   \ell_1 norm
% regularization least square problems.
%
%
   ----------------------------------------------------------------------

% Copyright (2019): Jianghua Yin
%
   ----------------------------------------------------------------------

%
%
% The first version of this code by Jianghua Yin, Nov. 29,
   2019
% test for number of required parametres
if (nargin-length(varargin)) ~= 3
  error('Wrong number of required parameters');
end

% flag for initial x (can take any values except 0,1,2)
Initial_X_supplied = 3333;

% Set the defaults for the optional parameters
stopCriterion = 3;
tolA = 0.01;
tolD = 0.0001;
```

```
debias = 0;
maxiter = 10000;
maxiter_debias = 500;
miniter = 5;
miniter_debias = 5;
init = 0;
compute_mse = 0;
AT = 0;
verbose = 1;
continuation = 0;
cont_steps = -1;
firstTauFactorGiven = 0;

% Set the defaults for outputs that may not be computed
debias_start = 0;
x_debias = [];
mses = [];

% 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)
    switch upper(varargin{i})
     case 'STOPCRITERION'
       stopCriterion = varargin{i+1};
     case 'TOLERANCEA'
       tolA = varargin{i+1};
     case 'TOLERANCED'
       tolD = varargin{i+1};
     case 'DEBIAS'
       debias = varargin{i+1};
     case 'MAXITERA'
       maxiter = varargin{i+1};
     case 'MAXITERD'
       maxiter_debias = varargin{i+1};
     case 'MINITERA'
       miniter = varargin{i+1};
     case 'MINITERD'
       miniter_debias = varargin{i+1};
     case 'INITIALIZATION'
       if prod(size(varargin{i+1})) > 1   % initial x supplied
           as array
           init = Initial_X_supplied;      % flag to be
              used below
           x = varargin{i+1};
```

```
       else
              init = varargin{i+1};
       end
     case 'MONOTONE'
       enforceMonotone = varargin{i+1};
     case 'CONTINUATION'
       continuation = varargin{i+1};
     case 'CONTINUATIONSTEPS'
       cont_steps = varargin{i+1};
     case 'FIRSTTAUFACTOR'
       firstTauFactor = varargin{i+1};
       firstTauFactorGiven = 1;
     case 'TRUE_X'
       compute_mse = 1;
       true = varargin{i+1};
     case 'ALPHAMIN'
       alphamin = varargin{i+1};
     case 'ALPHAMAX'
       alphamax = varargin{i+1};
     case 'AT'
       AT = varargin{i+1};
     case 'VERBOSE'
       verbose = varargin{i+1};
     otherwise
      % Hmmm, something wrong with the parameter string
      error(['Unrecognized option: ''' varargin{i} '''']);
    end;
  end;
end
%%%%%%%%%%%%%%

if (sum(stopCriterion == [0 1 2 3 4 5])==0)
  error(['Unknown stopping criterion']);
end

% if A is a function handle, we have to check presence of AT,
if isa(A, 'function_handle') & ~isa(AT,'function_handle')
  error(['The function handle for transpose of A is missing'])
     ;
end

% if A is a matrix, we find out dimensions of y and x,
% and create function handles for multiplication by A and A',
% so that the code below doesn't have to distinguish between
% the handle/not-handle cases
if ~isa(A, 'function_handle')
```

```
  AT = @(x) (x'*A)'; %A'*x;
  A = @(x) A*x;
end
% from this point down, A and AT are always function handles.

% Precompute A'*y since it'll be used a lot
Aty = AT(y);

% Initialization
switch init
    case 0   % initialize at zero, using AT to find the size
       of x
       x = AT(zeros(size(y)));
    case 1   % initialize randomly, using AT to find the size
       of x
       x = randn(size(AT(zeros(size(y)))));
    case 2   % initialize x0 = A'*y
       x = Aty;
    case Initial_X_supplied % initial x was given by user
       % initial x was given as a function argument; just
          check size
       if size(A(x)) ~= size(y)
          error(['Size of initial x is not compatible with A'
             ]);
       end
    otherwise
       error(['Unknown ''Initialization'' option']);
end

% now check if tau is an array; if it is, it has to
% have the same size as x
if prod(size(tau)) > 1
   try,
      dummy = x.*tau;
   catch,
      error(['Parameter tau has wrong dimensions; it should be
          scalar or size(x)']),
   end
end


% if the true x was given, check its size
if compute_mse & (size(true) ~= size(x))
   error(['Initial x has incompatible size']);
end
```

```
% if tau is scalar, we check its value; if it's large enough,
% the optimal solution is the zero vector
if prod(size(tau)) == 1
   aux = AT(y);
   max_tau = max(abs(aux(:)));
   if tau >= max_tau      %
      x = zeros(size(aux));
      if debias
         x_debias = x;
      end
      objective(1) = 0.5*(y(:)'*y(:));
      times(1) = 0;
      if compute_mse
          mses(1) = sum(true(:).^2);
      end
      return
   end                    %
end

% initialize u and v
u =  x.*(x >= 0);
v = -x.*(x <  0);

% define the indicator vector or matrix of nonzeros in x
nz_x = (x ~= 0.0);
num_nz_x = sum(nz_x(:));

% start the clock
t0 = cputime;

% store given tau, because we're going to change it in the
% continuation procedure
final_tau = tau;

% store given stopping criterion and threshold, because we're
   going
% to change them in the continuation procedure
final_stopCriterion = stopCriterion;
final_tolA = tolA;

% set continuation factors
if continuation&&(cont_steps > 1)
   % If tau is scalar, first check top see if the first factor
       is
   % too large (i.e., large enough to make the first
```

```
    % solution all zeros). If so, make it a little smaller than
        that.
    % Also set to that value as default
    if prod(size(tau)) == 1
       if (firstTauFactorGiven == 0)|(firstTauFactor*tau >=
          max_tau)
          firstTauFactor = 0.5*max_tau / tau;
          if verbose
              fprintf(1,'\n setting parameter FirstTauFactor\n'
                 )
          end
       end
    end
    cont_factors = 10.^[log10(firstTauFactor):...
                        log10(1/firstTauFactor)/(cont_steps-1):0];
end

if ~continuation
  cont_factors = 1;
  cont_steps = 1;
end

iter = 1;
if compute_mse
       mses(iter) = sum((x(:)-true(:)).^2);
end

keep_continuation = 1;
cont_loop = 1;
iter = 1;
taus = [];
sigma = 0.001;

% loop for continuation
while keep_continuation
     % Compute and store initial value of the objective
        function
     resid =  y - A(x);
     if cont_steps == -1
         gradq = AT(resid);
         tau = max(final_tau,0.2*max(abs(gradq)));
         if tau == final_tau
             stopCriterion = final_stopCriterion;
             tolA = final_tolA;
             keep_continuation = 0;                  % stop
                continuation
```

```
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    else
        tau = final_tau * cont_factors(cont_loop);%
        if cont_loop == cont_steps
            stopCriterion = final_stopCriterion;
            tolA = final_tolA;
            keep_continuation = 0;                    %
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    end
    taus = [taus tau];

    if verbose
        fprintf(1,'\nSetting tau = %0.5g\n',tau)
    end
    % if in first continuation iteration, compute and store
    % initial value of the objective function
    if cont_loop == 1
        alpha = 1.0;
        f = 0.5*(resid(:)'*resid(:)) + ...
             sum(tau(:).*u(:)) + sum(tau(:).*v(:));
        objective(1) = f;
        if compute_mse
            mses(1) = (x(:)-true(:))'*(x(:)-true(:));
        end
        if verbose
            fprintf(1,'Initial obj=%10.6e, alpha=%6.2e,
               nonzeros=%7d\n',...
                f,alpha,num_nz_x);
        end
    end
    % Compute the initial gradient and the useful
    % quantity resid_base
    resid_base = y - resid;
    % control variable for the outer loop and iteration
       counter
    keep_going = 1;
    if verbose
        fprintf(1,'\nInitial obj=%10.6e, nonzeros=%7d\n',f,
           num_nz_x);
    end
```

```
temp = AT(resid_base);
term  =  temp - Aty;
gradu =  term + tau; % Hz+c w.r.t. u
gradv = -term + tau; % Hz+c w.r.t. v
Lu = min(u,gradu);   % F(z) = min(z,Hz+c) w.r.t. u, z = [u
    v]'
Lv = min(v,gradv);   % F w.r.t. v
du = - Lu;
dv = - Lv;
while keep_going
    % compute dx
    dx = du-dv;
    auv = A(dx);
    Bdu = AT(auv);
    Bdv = -Bdu;
    % initial steplength
    betas = 1;   %  betas = 10 for paper;
    old_Lu = Lu;
    old_Lv = Lv;
    %NormF = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
    %NormFs = sqrt(NormF);
    Lu = min(u+betas*du, gradu+betas*Bdu); % F(z+betas*d)
       w.r.t. u where d=[du;dv];
    Lv = min(v+betas*dv, gradv+betas*Bdv); % F(z+betas*d)
       w.r.t. v;
    Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);  % F(z+betas*d)
       '*d
    dudv = du(:)'*du(:)+dv(:)'*dv(:);      % ||d||^2
    normFz = sqrt(Lu(:)'*Lu(:)+Lv(:)'*Lv(:));
    % - Luvduv < sigma*betas*max(0.001,min(0.8,NormFz))*
       dudv
    while - Luvduv < sigma*betas*normFz*dudv
        % -Luvduv < sigma*betas*dudv
        betas = 0.9*betas;   % betas = 0.5*betas;
        Lu = min(u+betas*du,gradu+betas*Bdu);
        Lv = min(v+betas*dv,gradv+betas*Bdv);
        normFz = sqrt(Lu(:)'*Lu(:) + Lv(:)'*Lv(:));
        Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);
    end
    lambda = -1.8*Luvduv*betas/normFz^2;  % lambda = -
       1.6*Luvduv*betas/(Lu(:)'*Lu(:)+Lv(:)'*Lv(:));
    old_u = u;
    old_v = v;
    u = old_u - lambda * Lu;
    v = old_v - lambda * Lv;
    uvmin = 0;% min(u,v);
```

```
u = u - uvmin;
v = v - uvmin;
x = u - v;
% calculate nonzero pattern and number of nonzeros (do
    this *always*)
nz_x_prev = nz_x;
nz_x = (x~=0.0);
num_nz_x = sum(nz_x(:));
% update residual and function
ALuv = A(Lu-Lv);
resid = y - resid_base + lambda*ALuv;     % y-Ax
resid_base = resid_base - lambda*ALuv;    % Ax
prev_f = f;
f = 0.5*(resid(:)'*resid(:)) +  sum(tau(:).*u(:)) +
   ...
  sum(tau(:).*v(:));
% compute new alpha
dd  = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
% print out stuff
if verbose
    fprintf(1,'It=%4d, obj=%9.5e, alpha=%6.2e, nz=%8d
        ',...
     iter, f, alpha, num_nz_x);
end
% update iteration counts, store results and times
iter = iter + 1;
objective(iter) = f;
times(iter) = cputime-t0;
% compute the next direction
temp = AT(resid_base);
term  =  temp - Aty;
gradu =  term + tau; % Hz+c w.r.t. u
gradv = -term + tau; % Hz+c w.r.t. v
%Lu = min(u,gradu);   % F(x)
%Lv = min(v,gradv);
sku = betas*du;
skv = betas*dv;
%
r=0.01;
yku = (Lu-old_Lu)+r*sku;
ykv = (Lv-old_Lv)+r*skv;
%c
 %
 gam=0.25;
 skyk=sku(:)'*yku(:)+skv(:)'*ykv(:);
 sksk=sku(:)'*sku(:)+skv(:)'*skv(:);
```

```
         Fksk=old_Lu(:)'*sku(:)+old_Lv(:)'*skv(:);
         Fkyk=old_Lu(:)'*yku(:)+old_Lv(:)'*ykv(:);
         dkyk=du(:)'*yku(:)+dv(:)'*ykv(:);
         ykyk=yku(:)'*yku(:)+ykv(:)'*ykv(:);
         bk1=(Fkyk)/(dkyk);
         bk2=(Fksk)/(dkyk);
         bk3=gam*(ykyk)/((skyk));
         bk4=gam*(skyk)/((sksk));
         tauk=bk4+bk3;
            if ((iter > 1))
             du = -gam*min(u,gradu)+gam*bk1*du-tauk*(bk2)*du;
             dv = -gam*min(v,gradv)+gam*bk1*dv-tauk*(bk2)*dv;
            end
        %end
    %end



        if compute_mse
            err = true - x;
            mses(iter) = (err(:)'*err(:));
        end

        switch stopCriterion
            case 0,
                % compute the stopping criterion based on the
                   change
                % of the number of non-zero components of the
                   estimate
                num_changes_active = (sum(nz_x(:)~=nz_x_prev
                   (:)));
                if num_nz_x >= 1
                    criterionActiveSet = num_changes_active;
                else
                    criterionActiveSet = tolA / 2;
                end
                keep_going = (criterionActiveSet > tolA);
                if verbose
                    fprintf(1,'Delta n-zeros = %d (target = %e
                       )\n',...
                      criterionActiveSet , tolA)
                end
            case 1,
                % compute the stopping criterion based on the
                   relative
                % variation of the objective function.
```

```
        criterionObjective = abs(f-prev_f)/(prev_f);
        keep_going =  (criterionObjective > tolA);
        if verbose
            fprintf(1,'Delta obj. = %e (target = %e)\n
               ',...
              criterionObjective , tolA)
        end
    case 2,
        % stopping criterion based on relative norm of
            step taken
        delta_x_criterion = norm(Lu(:)-Lv(:))/norm(x
           (:));
        keep_going = (delta_x_criterion > tolA);
        if verbose
            fprintf(1,'Norm(delta x)/norm(x) = %e (
               target = %e)\n',...
              delta_x_criterion,tolA)
        end
    case 3,
        % compute the "LCP" stopping criterion - again
            based on the previous
        % iterate. Make it "relative" to the norm of x
           .
        w = [ min(gradu(:), old_u(:)); min(gradv(:),
           old_v(:)) ];
        criterionLCP = norm(w(:), inf);
        criterionLCP = criterionLCP / ...
          max([1.0e-6, norm(old_u(:),inf), norm(old_v
             (:),inf)]);
        keep_going = (criterionLCP > tolA);
        if verbose
            fprintf(1,'LCP = %e (target = %e)\n',
               criterionLCP,tolA)
        end
    case 4,
        % continue if not yeat reached target value
           tolA
        keep_going = (f > tolA);
        if verbose
            fprintf(1,'Objective = %e (target = %e)\n'
               ,f,tolA)
        end
    case 5,
        % stopping criterion based on relative norm of
            step taken
        delta_x_criterion = sqrt(dd)/sqrt(x(:)'*x(:));
```

```
                keep_going = (delta_x_criterion > tolA);
                if verbose
                    fprintf(1,'Norm(delta x)/norm(x) = %e (
                       target = %e)\n',...
                      delta_x_criterion,tolA)
                end
            otherwise,
                error(['Unknown stopping criterion']);
        end % end of the stopping criteria switch

        % take no less than miniter...
        if iter<=miniter
            keep_going = 1;
        elseif iter > maxiter %and no more than maxiter
           iterations
                keep_going = 0;
        end

    end % end of the main loop of keep_going

    % increment continuation loop counter
    cont_loop = cont_loop+1;

end % end of the continuation loop
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Print results
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if verbose
   fprintf(1,'\nFinished the main algorithm!\nResults:\n')
   fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid(:))
   fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
   fprintf(1,'Objective function = %10.3e\n',f);
   nz_x = (x~=0.0); num_nz_x = sum(nz_x(:));
   fprintf(1,'Number of non-zero components = %d\n',num_nz_x);
   fprintf(1,'CPU time so far = %10.3e\n', times(iter));
   fprintf(1,'\n');
end

% If the 'Debias' option is set to 1, we try to remove the
   bias from the l1
```

```
% penalty, by applying CG to the least-squares problem
   obtained by omitting
% the l1 term and fixing the zero coefficients at zero.

% do this only if the reduced linear least-squares problem is
% overdetermined, otherwise we are certainly applying CG to a
   problem with a
% singular Hessian

if (debias & (sum(x(:)~=0)~=0))

  if (num_nz_x > length(y(:)))
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'There are too many nonzeros in x\n\n');
      fprintf(1,'nonzeros in x: %8d, length of y: %8d\n',...
          num_nz_x, length(y(:)));
    end
  elseif (num_nz_x==0)
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'x has no nonzeros\n\n');
    end
  else
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Starting the debiasing phase...\n\n')
    end

    x_debias = x;
    zeroind = (x_debias~=0);
    cont_debias_cg = 1;
    debias_start = iter;

    % calculate initial residual
    resid = A(x_debias);
    resid = resid-y;
    resid_prev = eps*ones(size(resid));

    rvec = AT(resid);

    % mask out the zeros
    rvec = rvec .* zeroind;
    rTr_cg = rvec(:)'*rvec(:);
```

```
% set convergence threshold for the residual || RW
   x_debias - y ||_2
tol_debias = tolD * (rvec(:)'*rvec(:));

% initialize pvec
pvec = -rvec;

% main loop
while cont_debias_cg

  % calculate A*p = Wt * Rt * R * W * pvec
  RWpvec = A(pvec);
  Apvec = AT(RWpvec);

  % mask out the zero terms
  Apvec = Apvec .* zeroind;

  % calculate alpha for CG
  alpha_cg = rTr_cg / (pvec(:)'* Apvec(:));

  % take the step
  x_debias = x_debias + alpha_cg * pvec;
  resid = resid + alpha_cg * RWpvec;
  rvec  = rvec  + alpha_cg * Apvec;

  rTr_cg_plus = rvec(:)'*rvec(:);
  beta_cg = rTr_cg_plus / rTr_cg;
  pvec = -rvec + beta_cg * pvec;

  rTr_cg = rTr_cg_plus;

  iter = iter+1;

  objective(iter) = 0.5*(resid(:)'*resid(:)) + ...
      sum(tau(:).*abs(x_debias(:)));
  times(iter) = cputime - t0;

  if compute_mse
    err = true - x_debias;
    mses(iter) = (err(:)'*err(:));
  end

  % in the debiasing CG phase, always use convergence
     criterion
  % based on the residual (this is standard for CG)
```

```
      if verbose
        fprintf(1,' Iter = %5d, debias resid = %13.8e,
           convergence = %8.3e\n', ...
            iter, resid(:)'*resid(:), rTr_cg / tol_debias);
      end
      cont_debias_cg = ...
          (iter-debias_start <= miniter_debias )| ...
          ((rTr_cg > tol_debias) & ...
          (iter-debias_start <= maxiter_debias));

    end
    if verbose
      fprintf(1,'\nFinished the debiasing phase!\nResults:\n')
      fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid
         (:))
      fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
      fprintf(1,'Objective function = %10.3e\n',f);
      nz = (x_debias~=0.0);
      fprintf(1,'Number of non-zero components = %d\n',sum(nz
         (:)));
      fprintf(1,'CPU time so far = %10.3e\n', times(iter));
      fprintf(1,'\n');
    end
  end

  if compute_mse
    mses = mses/length(true(:));
  end

end
```

(3) Code for NHZ1 signal reconstruction

```
function [x,x_debias,objective,times,debias_start,mses,taus]=
   ...
     NHZ1(y,A,tau,varargin)
%
% CGD_CS version 1.0, December 3, 2009
%
% This function solves the convex problem
% arg min_x = 0.5*|| y - A x ||_2^2 + tau || x ||_1
% using the algorithm modified PRP conjugate gradient method,
   described in the following paper
%
% This code is to use the well-known code CG_DESCENT to solve
   \ell_1 norm
% regularization least square problems.
```

```
%
%
   ---------------------------------------------------------------------

% Copyright (2010): Yunhai Xiao and Hong Zhu
%
   --------------------------------------------------------------------

%
%
% The first version of this code by Yunhai Xiao, Oct. 15. 2010
% test for number of required parametres
if (nargin-length(varargin)) ~= 3
  error('Wrong number of required parameters');
end

% flag for initial x (can take any values except 0,1,2)
Initial_X_supplied = 3333;
%%%%%%%%%%%%the parameter r is in y_{k-1}

% Set the defaults for the optional parameters
stopCriterion = 3;
tolA = 0.01;
tolD = 0.0001;
debias = 0;
maxiter = 10000;
maxiter_debias = 500;
miniter = 5;
miniter_debias = 5;
init = 0;
compute_mse = 0;
AT = 0;
verbose = 1;
continuation = 0;
cont_steps = -1;
firstTauFactorGiven = 0;

% Set the defaults for outputs that may not be computed
debias_start = 0;
x_debias = [];
mses = [];

% 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)
  switch upper(varargin{i})
   case 'STOPCRITERION'
     stopCriterion = varargin{i+1};
   case 'TOLERANCEA'
     tolA = varargin{i+1};
   case 'TOLERANCED'
     tolD = varargin{i+1};
   case 'DEBIAS'
     debias = varargin{i+1};
   case 'MAXITERA'
     maxiter = varargin{i+1};
   case 'MAXITERD'
     maxiter_debias = varargin{i+1};
   case 'MINITERA'
     miniter = varargin{i+1};
   case 'MINITERD'
     miniter_debias = varargin{i+1};
   case 'INITIALIZATION'
     if prod(size(varargin{i+1})) > 1   % initial x supplied
         as array
            init = Initial_X_supplied;       % flag to be
               used below
            x = varargin{i+1};
     else
            init = varargin{i+1};
     end
   case 'MONOTONE'
     enforceMonotone = varargin{i+1};
   case 'CONTINUATION'
     continuation = varargin{i+1};
   case 'CONTINUATIONSTEPS'
     cont_steps = varargin{i+1};
   case 'FIRSTTAUFACTOR'
     firstTauFactor = varargin{i+1};
     firstTauFactorGiven = 1;
   case 'TRUE_X'
     compute_mse = 1;
     true = varargin{i+1};
   case 'ALPHAMIN'
     alphamin = varargin{i+1};
   case 'ALPHAMAX'
     alphamax = varargin{i+1};
   case 'AT'
     AT = varargin{i+1};
   case 'VERBOSE'
```

```
        verbose = varargin{i+1};
      otherwise
       % Hmmm, something wrong with the parameter string
       error(['Unrecognized option: ''' varargin{i} '''']);
     end;
   end;
end
%%%%%%%%%%%%%%

if (sum(stopCriterion == [0 1 2 3 4 5])==0)
   error(['Unknown stopping criterion']);
end

% if A is a function handle, we have to check presence of AT,
if isa(A, 'function_handle') & ~isa(AT,'function_handle')
   error(['The function handle for transpose of A is missing'])
      ;
end

% if A is a matrix, we find out dimensions of y and x,
% and create function handles for multiplication by A and A',
% so that the code below doesn't have to distinguish between
% the handle/not-handle cases
if ~isa(A, 'function_handle')
   AT = @(x) (x'*A)'; %A'*x;
   A = @(x) A*x;
end
% from this point down, A and AT are always function handles.

% Precompute A'*y since it'll be used a lot
Aty = AT(y);

% Initialization
switch init
    case 0   % initialize at zero, using AT to find the size
       of x
       x = AT(zeros(size(y)));
    case 1   % initialize randomly, using AT to find the size
       of x
       x = randn(size(AT(zeros(size(y)))));
    case 2   % initialize x0 = A'*y
       x = Aty;
    case Initial_X_supplied % initial x was given by user
       % initial x was given as a function argument; just
          check size
       if size(A(x)) ~= size(y)
```

```
            error(['Size of initial x is not compatible with A'
               ]);
         end
      otherwise
         error(['Unknown ''Initialization'' option']);
end

% now check if tau is an array; if it is, it has to
% have the same size as x
if prod(size(tau)) > 1
   try,
      dummy = x.*tau;
   catch,
      error(['Parameter tau has wrong dimensions; it should be
          scalar or size(x)']),
   end
end


% if the true x was given, check its size
if compute_mse & (size(true) ~= size(x))
   error(['Initial x has incompatible size']);
end

% if tau is scalar, we check its value; if it's large enough,
% the optimal solution is the zero vector
if prod(size(tau)) == 1
   aux = AT(y);
   max_tau = max(abs(aux(:)));
   if tau >= max_tau
      x = zeros(size(aux));
      if debias
         x_debias = x;
      end
      objective(1) = 0.5*(y(:)'*y(:));
      times(1) = 0;
      if compute_mse
          mses(1) = sum(true(:).^2);
      end
      return
   end
end

% initialize u and v
u =  x.*(x >= 0);
v = -x.*(x <  0);
```

```
% define the indicator vector or matrix of nonzeros in x
nz_x = (x ~= 0.0);
num_nz_x = sum(nz_x(:));

% start the clock
t0 = cputime;

% store given tau, because we're going to change it in the
% continuation procedure
final_tau = tau;

% store given stopping criterion and threshold, because we're
   going
% to change them in the continuation procedure
final_stopCriterion = stopCriterion;
final_tolA = tolA;

% set continuation factors
if continuation&&(cont_steps > 1)
   % If tau is scalar, first check top see if the first factor
       is
   % too large (i.e., large enough to make the first
   % solution all zeros). If so, make it a little smaller than
       that.
   % Also set to that value as default
   if prod(size(tau)) == 1
      if (firstTauFactorGiven == 0)|(firstTauFactor*tau >=
         max_tau)
         firstTauFactor = 0.5*max_tau / tau;
         if verbose
             fprintf(1,'\n setting parameter FirstTauFactor\n'
                )
         end
      end
   end
   cont_factors = 10.^[log10(firstTauFactor):...
                    log10(1/firstTauFactor)/(cont_steps-1):0];
end

if ~continuation
  cont_factors = 1;
  cont_steps = 1;
end

iter = 1;
```

```
if compute_mse
       mses(iter) = sum((x(:)-true(:)).^2);
end

keep_continuation = 1;
cont_loop = 1;
iter = 1;
taus = [];

% loop for continuation
while keep_continuation
    % Compute and store initial value of the objective
       function
    resid =  y - A(x);
    if cont_steps == -1
        gradq = AT(resid);
        tau = max(final_tau,0.2*max(abs(gradq)));
        if tau == final_tau
            stopCriterion = final_stopCriterion;
            tolA = final_tolA;
            keep_continuation = 0;                  % stop
               continuation
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    else
        tau = final_tau * cont_factors(cont_loop);%
        if cont_loop == cont_steps
            stopCriterion = final_stopCriterion;
            tolA = final_tolA;
            keep_continuation = 0;                 %
        else
            stopCriterion = 1;
            tolA = 1e-5;
        end
    end

    taus = [taus tau];

    if verbose
        fprintf(1,'\nSetting tau = %0.5g\n',tau)
    end

    % if in first continuation iteration, compute and store
    % initial value of the objective function
```

```
if cont_loop == 1
    alpha = 1.0;
    f = 0.5*(resid(:)'*resid(:)) + ...
         sum(tau(:).*u(:)) + sum(tau(:).*v(:));
    objective(1) = f;
    if compute_mse
        mses(1) = (x(:)-true(:))'*(x(:)-true(:));
    end
    if verbose
        fprintf(1,'Initial obj=%10.6e, alpha=%6.2e,
           nonzeros=%7d\n',...
            f,alpha,num_nz_x);
    end
end

% Compute the initial gradient and the useful
% quantity resid_base
resid_base = y - resid;

% control variable for the outer loop and iteration
   counter
keep_going = 1;

if verbose
  fprintf(1,'\nInitial obj=%10.6e, nonzeros=%7d\n',f,
     num_nz_x);
end
while keep_going

  % compute gradient
  temp = AT(resid_base);
  term  =  temp - Aty;
  gradu =  term + tau;
  gradv = -term + tau;
  %
  Lu = min(u,gradu); %1A
  Lv = min(v,gradv);%1A
  %NormF = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
  %NormF2 = sqrt(NormF);
  %

  if (iter > 1)
     %(esku,eskv)^T=x_k-x_{k-1}=s_{k-1}
     su = u-old_u;
     sv = v-old_v;
     %Fkdk=F_k^d_{k-1}
```

```
    Fkdk= Lu(:)'*old_du(:)+Lv(:)'*old_dv(:);
    %sksk= ||s_{k-1}||^2
    sksk = su(:)'*su(:)+sv(:)'*sv(:);
    Normsk=sqrt(sksk);
    m=0.01;
    %(rku,rkv)^T=F_k-F_{k-1}=y_{k-1}
    rku = min(u,gradu)-min(old_u,old_gradu);
    rkv = min(v,gradv)-min(old_v,old_gradv);
    wk1=rku+m*su;
    wk2=rkv+m*sv;
    %skyb=s_{k-1}^Tyb
    %skyb = su(:)'*wk1(:)+sv(:)'*wk2(:);
    %tk=1+max(0,-(skyb/sksk));
    %wk=yb+tk*sk
    %wk1=yb1+tk*su;
    %wk2=yb2+tk*sv;
    %skwk=s_{k-1}^Tw_{k-1}
    skwk=su(:)'*wk1(:)+sv(:)'*wk2(:);
    %dkwk=d_{k-1}^Tw_{k-1}
    dkwk=old_du(:)'*wk1(:)+old_dv(:)'*wk2(:);
    %Fkwk=F_{k-1}^Tw_{k-1}
    Fkwk=Lu(:)'*wk1(:)+Lv(:)'*wk2(:);
    %||wk||^2=(yb+tk*sk)^2
    wkwk=wk1(:)'*wk1(:)+wk2(:)'*wk2(:);
    %||wk||=sqrt(wkwk)
    Normwk=sqrt(wkwk);
    gamma=4;
    ts=(1/gamma)*(sqrt(skwk/(Normsk*Normwk)))^3;
    L=(2/(4*gamma));
    tsb=max(ts,L);
    %Betak
    Betak=(Fkwk)/(dkwk)-gamma*tsb*((wkwk)*(Fkdk)/(dkwk)
       ^2);

end
%
old_gradu = gradu;
old_gradv = gradv;
% computation of search direction vector
du = - min(u, gradu);
dv = - min(v, gradv);
if (iter > 1)
    du = -min(u, gradu) +Betak*old_du;
    dv = -min(v, gradv) +Betak*old_dv;
end
dx = du-dv;
```

```
old_u = u;
old_v = v;
old_du = du;
old_dv = dv;
%Old_NormF = NormF;
% calculate useful matrix-vector product involving dx
auv = A(dx);
Bdu = AT(auv);
Bdv = -Bdu;
% preparetion for line search
sigma = 0.0001;
betas = 1;
Lu = min(u+betas*du, gradu+betas*Bdu);
Lv = min(v+betas*dv, gradv+betas*Bdv);
Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);
dudv = du(:)'*du(:)+dv(:)'*dv(:);
% line search  process
while - Luvduv < sigma*betas*dudv
    betas = 0.51*betas;
    Lu = min(u+betas*du,gradu+betas*Bdu);
    Lv = min(v+betas*dv,gradv+betas*Bdv);
    Luvduv = Lu(:)'*du(:) + Lv(:)'*dv(:);
end
% compute the projection steplength
lambda = - 1.8*Luvduv*betas/(Lu(:)'*Lu(:)+Lv(:)'*Lv(:));
    % lamda_k=-(alpha*d_k^T*F(z_k))/(norm(z_k))^2
%
u = old_u - lambda * Lu;
v = old_v - lambda * Lv;
uvmin = 0;% min(u,v);
u = u - uvmin;
v = v - uvmin;
x = u - v;
% calculate nonzero pattern and number of nonzeros (do
   this *always*)
nz_x_prev = nz_x;
nz_x = (x~=0.0);
num_nz_x = sum(nz_x(:));
% update residual and function
ALuv = A(Lu-Lv);
resid = y - resid_base + lambda*ALuv;
prev_f = f;
f = 0.5*(resid(:)'*resid(:)) +  sum(tau(:).*u(:)) + ...
    sum(tau(:).*v(:));
% compute new alpha
dd  = Lu(:)'*Lu(:) + Lv(:)'*Lv(:);
```

```
%
resid_base = resid_base - lambda*ALuv;
% print out stuff
if verbose
   fprintf(1,'It=%4d, obj=%9.5e, alpha=%6.2e, nz=%8d  ',
      ...
       iter, f, alpha, num_nz_x);
end
% update iteration counts, store results and times
iter = iter + 1;
objective(iter) = f;
times(iter) = cputime-t0;

if compute_mse
  err = true - x;
  mses(iter) = (err(:)'*err(:));
end

switch stopCriterion
    case 0,
        % compute the stopping criterion based on the
           change
        % of the number of non-zero components of the
           estimate
        num_changes_active = (sum(nz_x(:)~=nz_x_prev(:))
           );
        if num_nz_x >= 1
            criterionActiveSet = num_changes_active;
        else
            criterionActiveSet = tolA / 2;
        end
        keep_going = (criterionActiveSet > tolA);
        if verbose
            fprintf(1,'Delta n-zeros = %d (target = %e)\
               n',...
                criterionActiveSet , tolA)
        end
    case 1,
        % compute the stopping criterion based on the
           relative
        % variation of the objective function.
        criterionObjective = abs(f-prev_f)/(prev_f);
        keep_going =  (criterionObjective > tolA);
        if verbose
            fprintf(1,'Delta obj. = %e (target = %e)\n',
               ...
```

```
            criterionObjective , tolA)
    end
case 2,
    % stopping criterion based on relative norm of
       step taken
    delta_x_criterion = norm(Lu(:)-Lv(:))/norm(x(:))
       ;
    keep_going = (delta_x_criterion > tolA);
    if verbose
        fprintf(1,'Norm(delta x)/norm(x) = %e (
           target = %e)\n',...
            delta_x_criterion,tolA)
    end
case 3,
    % compute the "LCP" stopping criterion - again
       based on the previous
    % iterate. Make it "relative" to the norm of x.
    w = [ min(gradu(:), old_u(:)); min(gradv(:),
       old_v(:)) ];
    criterionLCP = norm(w(:), inf);
    criterionLCP = criterionLCP / ...
        max([1.0e-6, norm(old_u(:),inf), norm(old_v
           (:),inf)]);
    keep_going = (criterionLCP > tolA);
    if verbose
        fprintf(1,'LCP = %e (target = %e)\n',
           criterionLCP,tolA)
    end
case 4,
    % continue if not yeat reached target value tolA
    keep_going = (f > tolA);
    if verbose
        fprintf(1,'Objective = %e (target = %e)\n',f
           ,tolA)
    end
case 5,
  % stopping criterion based on relative norm of
     step taken
  delta_x_criterion = sqrt(dd)/sqrt(x(:)'*x(:));
  keep_going = (delta_x_criterion > tolA);
  if verbose
      fprintf(1,'Norm(delta x)/norm(x) = %e (target
         = %e)\n',...
          delta_x_criterion,tolA)
  end
otherwise,
```

```
            error(['Unknown stopping criterion']);
        end % end of the stopping criteria switch

        % take no less than miniter...
        if iter<=miniter
            keep_going = 1;
        elseif iter > maxiter %and no more than maxiter
           iterations
            keep_going = 0;
        end

    end % end of the main loop of the BB-QP algorithm

     % increment continuation loop counter
     cont_loop = cont_loop+1;

end % end of the continuation loop
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Print results
%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if verbose
   fprintf(1,'\nFinished the main algorithm!\nResults:\n')
   fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid(:))
   fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
   fprintf(1,'Objective function = %10.3e\n',f);
   nz_x = (x~=0.0); num_nz_x = sum(nz_x(:));
   fprintf(1,'Number of non-zero components = %d\n',num_nz_x);
   fprintf(1,'CPU time so far = %10.3e\n', times(iter));
   fprintf(1,'\n');
end

% If the 'Debias' option is set to 1, we try to remove the
   bias from the l1
% penalty, by applying CG to the least-squares problem
   obtained by omitting
% the l1 term and fixing the zero coefficients at zero.

% do this only if the reduced linear least-squares problem is
% overdetermined, otherwise we are certainly applying CG to a
   problem with a
% singular Hessian
```

```
if (debias & (sum(x(:)~=0)~=0))

  if (num_nz_x > length(y(:)))
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'There are too many nonzeros in x\n\n');
      fprintf(1,'nonzeros in x: %8d, length of y: %8d\n',...
          num_nz_x, length(y(:)));
    end
  elseif (num_nz_x==0)
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Debiasing requested, but not performed\n');
      fprintf(1,'x has no nonzeros\n\n');
    end
  else
    if verbose
      fprintf(1,'\n')
      fprintf(1,'Starting the debiasing phase...\n\n')
    end

    x_debias = x;
    zeroind = (x_debias~=0);
    cont_debias_cg = 1;
    debias_start = iter;

    % calculate initial residual
    resid = A(x_debias);
    resid = resid-y;
    resid_prev = eps*ones(size(resid));

    rvec = AT(resid);

    % mask out the zeros
    rvec = rvec .* zeroind;
    rTr_cg = rvec(:)'*rvec(:);

    % set convergence threshold for the residual || RW
       x_debias - y ||_2
    tol_debias = tolD * (rvec(:)'*rvec(:));

    % initialize pvec
    pvec = -rvec;
```

```
% main loop
while cont_debias_cg

  % calculate A*p = Wt * Rt * R * W * pvec
  RWpvec = A(pvec);
  Apvec = AT(RWpvec);

  % mask out the zero terms
  Apvec = Apvec .* zeroind;

  % calculate alpha for CG
  alpha_cg = rTr_cg / (pvec(:)'* Apvec(:));

  % take the step
  x_debias = x_debias + alpha_cg * pvec;
  resid = resid + alpha_cg * RWpvec;
  rvec  = rvec  + alpha_cg * Apvec;

  rTr_cg_plus = rvec(:)'*rvec(:);
  beta_cg = rTr_cg_plus / rTr_cg;
  pvec = -rvec + beta_cg * pvec;

  rTr_cg = rTr_cg_plus;

  iter = iter+1;

  objective(iter) = 0.5*(resid(:)'*resid(:)) + ...
      sum(tau(:).*abs(x_debias(:)));
  times(iter) = cputime - t0;

  if compute_mse
    err = true - x_debias;
    mses(iter) = (err(:)'*err(:));
  end

  % in the debiasing CG phase, always use convergence
     criterion
  % based on the residual (this is standard for CG)
  if verbose
    fprintf(1,' Iter = %5d, debias resid = %13.8e,
       convergence = %8.3e\n', ...
        iter, resid(:)'*resid(:), rTr_cg / tol_debias);
  end
  cont_debias_cg = ...
      (iter-debias_start <= miniter_debias )| ...
      ((rTr_cg > tol_debias) & ...
```

```
          (iter-debias_start <= maxiter_debias));

    end
    if verbose
      fprintf(1,'\nFinished the debiasing phase!\nResults:\n')
      fprintf(1,'||A x - y ||_2^2 = %10.3e\n',resid(:)'*resid
         (:))
      fprintf(1,'||x||_1 = %10.3e\n',sum(abs(x(:))))
      fprintf(1,'Objective function = %10.3e\n',f);
      nz = (x_debias~=0.0);
      fprintf(1,'Number of non-zero components = %d\n',sum(nz
         (:)));
      fprintf(1,'CPU time so far = %10.3e\n', times(iter));
      fprintf(1,'\n');
    end
  end

  if compute_mse
    mses = mses/length(true(:));
  end

end
```