



UNIVERSITY OF CALIFORNIA, SAN DIEGO

SPARSITY AND COMPRESSED SENSING

WINTER 2015

HOMEWORK III

Author:

Alican NALCI
ID: A53051826

Instructor:

Prof. Bhaskar RAO

February 10, 2015

Matlab Exercise

This week we use the conduct the same experiment from the previous week including the L1 optimization sparse signal recovery method for comparison. We use the CVX library, a famous convex programming library designed for this task, accesible www.cvxr.com. We follow the exercises in the textbook by Elad and provide figures similar to 3.5 and 3.6. We also include a brief complexity analysis between the L1 approach and different pursuit algorithms for different data distributions. The Matlab statement for the L1 optimization problem is as follows:

Listing 1: CVX L1 Sparse Recovery

```
cvx_begin quiet
    variable X_L1(dim2)
    minimize(norm(X_L1,1));
    subject to
        A*X_L1 == b;
cvx_end
```

The code above tries to find an optimal vector X_{L1} such that the constraint function is satisfied and 1-norm of the vector is minimized in each iteration. Note that in comparison to the Greedy algorithms this minimization approach takes more time to find an optimal value, and doesn't use support based pursuit as in Greedy algorithms.

The test we perform includes initialization of the matrix A of size 30×50 where entries are drawn from a normal distribution, and each column is normalized to have unit L2 norm. We also generate sparse vectors x with iid random supports of cardinalities in the range $[1,10]$. The non-zero entries of this vector is selected using four different distributions: 1)Elad's Distribution, 2)Normal Distribution ,3) ± 1 Binary Random Distribution, 4) Cauchy Distribution(Student's t). We apply 1000 such tests and present the average of the results in terms of mean square error of $\hat{x} - x$, and distance between the actual support and estimated support $dist(\hat{S}, S)$.

Figure 1 shows the average and relative L2 Error of $\hat{x} - x$. The immediate observation is that L1 based approach seems to perform the best among the pursuit type algorithms. We take a closer look and zoom along the y axis and obtain figure 2. L1 optimization based recovery approach indeed works the best among other algorithms for the same data distribution. The average mean square error for all distributions are very close to zero until a cardinality number of 6. After that value for Elad distribution, mean square error increases upto 0.025. Similarly, after cardinality 7 mean square error of L1 Binary increases to 0.027, for other distributions mean square error for L1 recovery increases very slowly and is below 0.005 at cardinality 10.

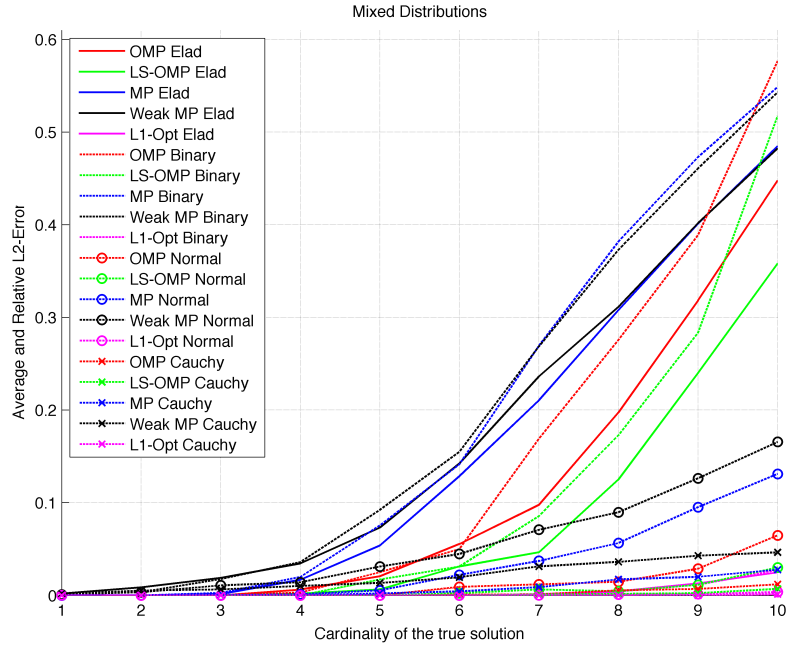


Figure 1: Average and Relative L2-Error for all Distributions

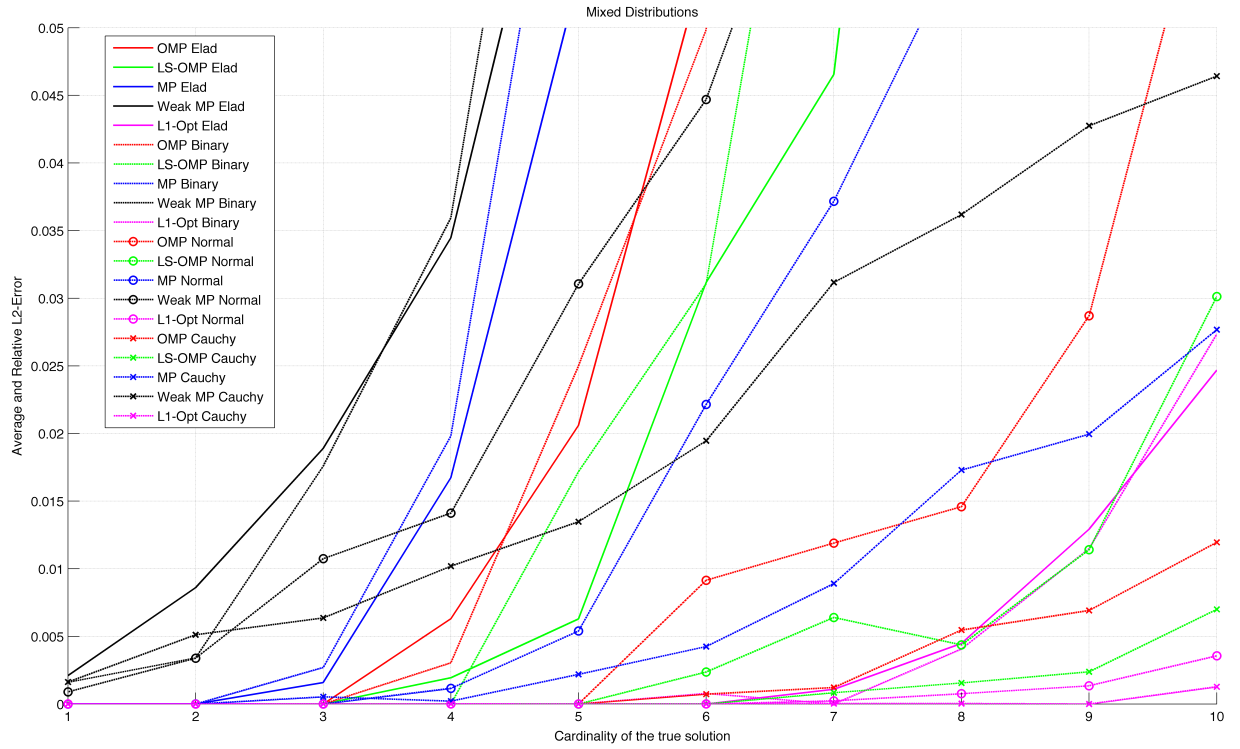


Figure 2: Average and Relative L2-Error for all Distributions Zoomed

Similarly Figure 3 and 4 show the probability of error in support, with figure 4 as the zoomed version. We see that regardless of data distribution the probability of error in support is very small, and follows the same increase behavior for different distributions. Largest L1 support error is observed with Normal data distribution, and for LS-OMP Normal and LS-OMP Cauchy the support error probability performance is very close to the L1 recovery method.

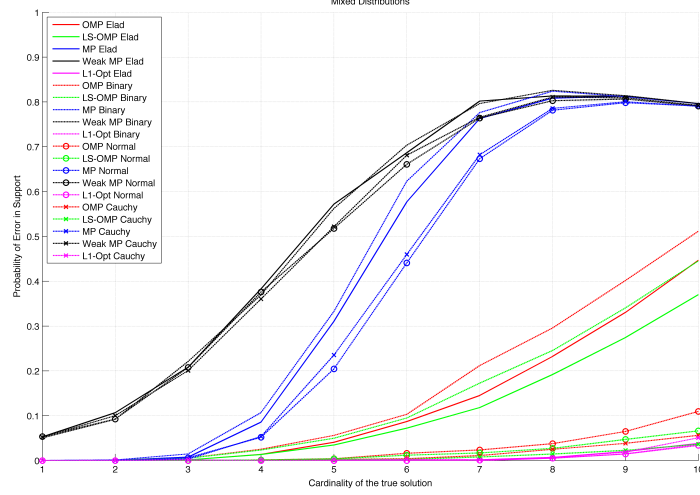


Figure 3: Probability of error in support for all distributions

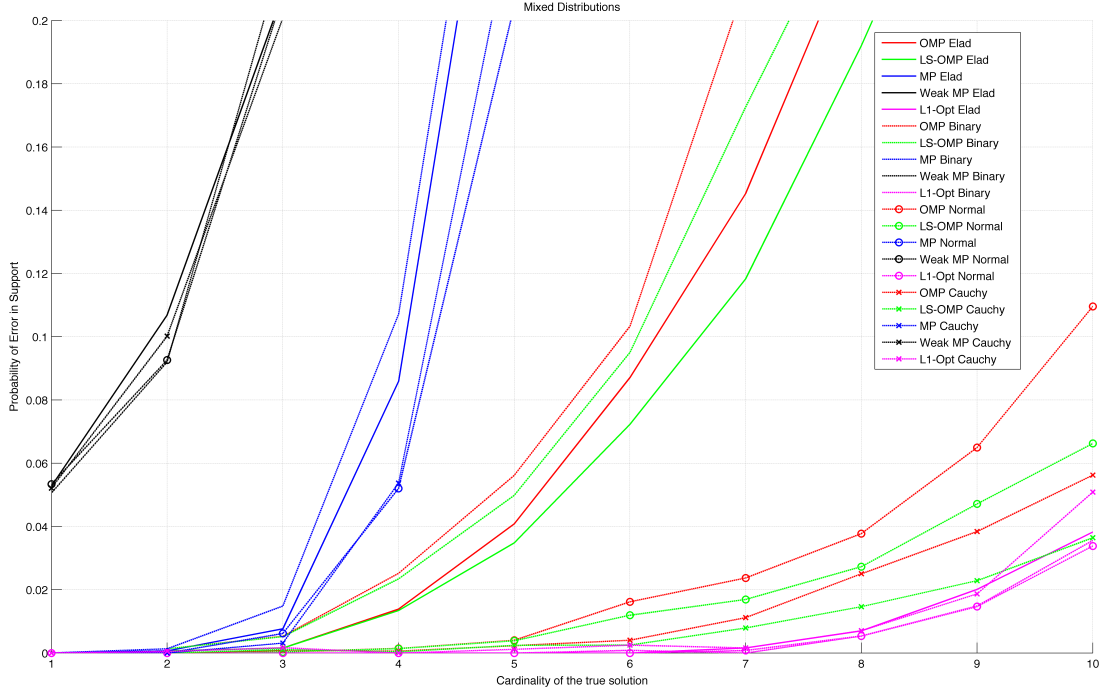


Figure 4: Probability of error in support for all distributions zoomed

In figures 1,2,3 and 4 we saw using the full L1 optimization based sparse recovery significantly improves the mean square error and probability of error in support. However, while doing the experiment we realized the greedy algorithms converge to a solution much faster than the L1 approach, and L1 approach takes significantly more time. We display the mean execution times in figure 5. We computed average running times of different pursuit algorithms for different cardinality and 1000 initializations, we see that complexity of L1 optimization approach has the largest complexity and generally seems independent of the number of cardinality. Surprisingly, for cardinality of 9 and 10 for Elad's distribution, we see that L1 approach takes less time to converge than LS-OMP. The reason that L1 takes more time is that, MP algorithms compute \hat{x} very fast using the greedy approach, not using all data, and thus their complexity is less than the L1 approach which uses all of the available data. We reproduce the theoretical complexity study as in previous homework:

The complexity of L1 approach is independent of the number of cardinality since we don't exploit the support vectors information, and the approach works full instead of working greedy. A basic complexity calculation for this approach yields we loop over possible X_{L1} values and in each step we have the multiplication AxX_{L1} which has $m \times n$ multiplications in each iteration. The comparison $AxX_{L1} == b$ takes linear time $O(n)$ and also taking the one norm is equivalent to summing elements of the current X_{L1} so it has complexity $O(n)$ finally looping over L different X_{L1} values. Therefore, overall complexity becomes $O(Lmn + 2n)$. Note that compared to the pursuit type greedy algorithms there is no dependency on the number of cardinality k_0 , and in the worst case we have complexity $O(n^3)$. Therefore, we can say that complexity is independent of k_0 and thus in figure 5 the average time elapsed to execute the L1 approach is constant with cardinality k_0 . Reproducing last weeks complexity calculations: There is an increasing trend in execution times as the Cardinality of true solution increases. This increase is *linear* for MP, OMP, and Weak-MP, and quadratic for LS-OMP. This could be theoretically verified by noting the complexity for MP and OMP are $O(k_0mn)$ where $m = 30$, $n = 50$ and k_0 is the cardinality. Therefore MP, OMP and Weak-MP(modified MP) are linear with cardinality as in figure 3. Using the full-least squares method we introduce an internal loop that loops over k_0 values hence complexity becomes $O(k_0^2mn)$ which is polynomial in k_0 .

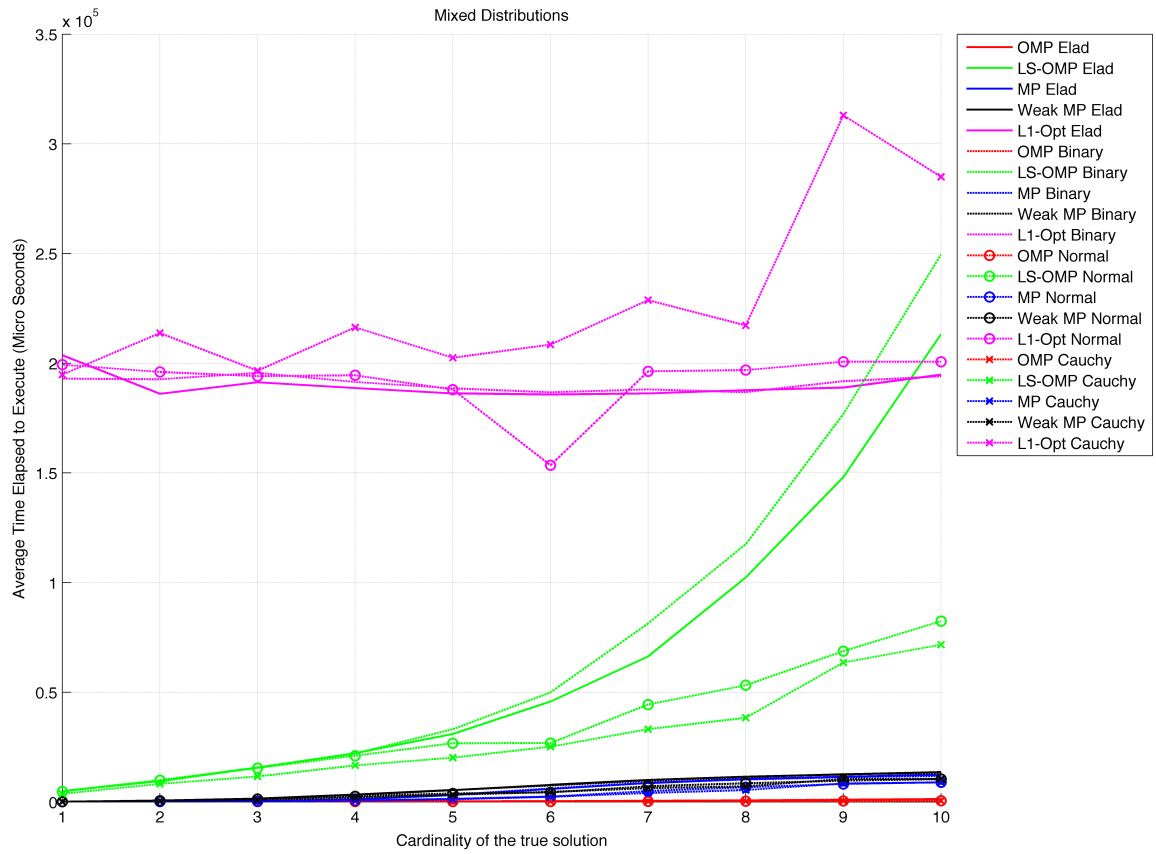


Figure 5: Average Execution Times(mean of 1000) for different cardinality and pursuit algorithms

Code

```

%% Simulation of the Matching Pursuit Algorithms
clear all;
close all;
clc;

%Generate Data
str      = 'Binary'; %Data Distribution
dim1     = 30;
dim2     = 50;
card     = 10;
consRuns = 1000;

errorOMP      = zeros(1,card);
errorLSOMP    = zeros(1,card);
errorMP       = zeros(1,card);
errorWeakMP   = zeros(1,card);
errorL1cvx    = zeros(1,card);

```

```

errorProbOMP = zeros(1,card);
errorProbLSOMP = zeros(1,card);
errorProbMP = zeros(1,card);
errorProbWeakMP = zeros(1,card);
errorProbL1cvx = zeros(1,card);

mutual_coherence = zeros(1,card*consRuns);
upper_bound = zeros(1,card*consRuns);

timeOMP = zeros(1,card);
timeLSOMP = zeros(1,card);
timeMP = zeros(1,card);
timeWeakMP = zeros(1,card);
timeL1cvx = zeros(1,card);
itercvx = zeros(1,card);
cputimecvx = zeros(1,card);

for s=1:card
    for k=1:1:consRuns
        A = randn(dim1,dim2); %Normally Distributed
        A = A/(diag(sqrt(diag(A'*A)))); %Normalize Cols Unit L2 Norm
        x = zeros(dim2,1);
        [x, shuffled] = genSparseVect(str,s,dim2);
        b=A*x;
        %OMP
        tic;
        [x_OMP, S_OMP] = OMP(A, b, 1e-4);
        timeOMP(s) = timeOMP(s)+toc;
        errorOMP(s) = errorOMP(s)+mean((x_OMP-x).^2)/mean(x.^2);
        errorProbOMP(s)= errorProbOMP(s)+(max(s,length(S_OMP))- ...
            max(size(intersect(S_OMP,shuffled(1:s)))))...
            /max(s,length(S_OMP));

        %LS-OMP
        tic;
        [x_LSOMP, S_LSOMP] = LSOMP(A, b, 1e-4);
        timeLSOMP(s) = timeLSOMP(s)+toc;
        errorLSOMP(s) = errorLSOMP(s)+mean((x_LSOMP-x).^2)/mean(x.^2);
        errorProbLSOMP(s)= errorProbLSOMP(s)+(max(s,length(S_LSOMP))- ...
            max(size(intersect(S_LSOMP,shuffled(1:s)))))...
            /max(s,length(S_LSOMP));

        %MP
        tic;
        [x_MP, S_MP] = MP( A, b, 1e-4);
        timeMP(s) = timeMP(s)+toc;
        errorMP(s) = errorMP(s)+mean((x_MP-x).^2)/mean(x.^2);
        errorProbMP(s) = errorProbMP(s)+(max(s,length(S_MP))- ...
            max(size(intersect(S_MP,shuffled(1:s)))))...
            /max(s,length(S_MP));
    end
end

```

```

%Weak MP
tic;
[x_WeakMP, S_WeakMP] = WeakMP( A, b, 1e-4, .5);
timeWeakMP(s) = timeWeakMP(s)+toc;
errorWeakMP(s) = errorWeakMP(s)+mean((x_WeakMP-x).^2)/mean(x.^2);
errorProbWeakMP(s) = errorProbWeakMP(s)+(max(s,length(S_WeakMP))-
    ...
        max(size(intersect(S_WeakMP,shuffled(1:s)))))...
        /max(s,length(S_WeakMP)));

%L1 Norm Optimization
tic;
cvx_begin quiet
    variable X_L1(dim2)
    minimize(norm(X_L1,1));
    subject to
        A*X_L1 == b;
cvx_end

timeL1cvx(s) = timeL1cvx(s)+toc;
errorL1cvx(s) = errorL1cvx(s)+mean((X_L1-x).^2)/mean(x.^2);
S_CVX = find(abs(X_L1)>1e-5);
errorProbL1cvx(s) = errorProbL1cvx(s)+(max(s,length(S_CVX))- ...
    max(size(intersect(S_CVX,shuffled(1:s)))))...
    /max(s,length(S_CVX));

itercvx(s) = itercvx(s)+cvx_slvitr;
cputimecvx(s) = cputimecvx(s)+cvx_cputime;

mutual_coherence((s-1)*consRuns+k) =
    mutual_coherence((s-1)*consRuns+k)+ ...
        mutualCoherence(A);

end
disp(['Cardinality:' num2str(s)])
end

timeOMPSingle = timeOMP/consRuns*1e6;
timeLSOMPSingle = timeLSOMP/consRuns*1e6;
timeMPSingle = timeMP/consRuns*1e6;
timeWeakMPSingle = timeWeakMP/consRuns*1e6;
timeL1cvx = timeL1cvx/consRuns*1e6;

upper_bound = (1/2)*(1+1./mutual_coherence);
errorOMP = errorOMP/consRuns;
errorProbOMP = errorProbOMP/consRuns;
errorLSOMP = errorLSOMP/consRuns;
errorProbLSOMP = errorProbLSOMP/consRuns;
errorMP = errorMP/consRuns;
errorProbMP = errorProbMP/consRuns;

```



```

errorWeakMP      = errorWeakMP/consRuns;
errorProbWeakMP  = errorProbWeakMP/consRuns;
errorL1cvx       = errorL1cvx/consRuns;
errorProbL1cvx   = errorProbL1cvx/consRuns;
itercvx          = itercvx/consRuns;
cputimecvx       = cputimecvx/consRuns;

save([str '_HW3_test.mat'])
%%
f = figure;
hold on;
plot(1:card,errorOMP,'r', 'LineWidth', 2);
plot(1:card,errorLSOMP,'g', 'LineWidth', 2);
plot(1:card,errorMP,'b', 'LineWidth', 2);
plot(1:card,errorWeakMP,'b', 'LineWidth', 2);
plot(1:card,errorL1cvx,'k', 'LineWidth', 2);
xlabel('Cardinality of the true solution');
ylabel('Average and Relative L2-Error');
title(['Non-Zero Entry Distribution:' str])
grid on;
legend('OMP','LS-OMP','MP', 'Weak MP','Location','NorthWest')
%print(f,'-dpng', '-r500', [str '_1.png'])
%%
f = figure;
hold on;
plot(1:card,errorProbOMP,'r', 'LineWidth', 2);
plot(1:card,errorProbLSOMP,'g', 'LineWidth', 2);
plot(1:card,errorProbMP,'b', 'LineWidth', 2);
plot(1:card,errorProbWeakMP,'b', 'LineWidth', 2);
plot(1:card,errorProbL1cvx,'k', 'LineWidth', 2);
xlabel('Cardinality of the true solution');
title(['Non-Zero Entry Distribution: ' str])
ylabel('Probability of Error in Support');
grid on;
legend('OMP','LS-OMP','MP', 'Weak MP','Location','NorthWest')
%print(f,'-dpng', '-r500',[str '_2.png'])
%%
f=figure;
hold on;
plot(1:card,timeOMPSingle,'r', 'LineWidth', 2);
plot(1:card,timeLSOMPSingle,'g', 'LineWidth', 2);
plot(1:card,timeMPSingle,'b', 'LineWidth', 2);
plot(1:card,timeWeakMPSingle,'k', 'LineWidth', 2);
xlabel('Cardinality of the true solution');
title(['Non-Zero Entry Distribution: ' str])
ylabel('Average Time Elapsed to Execute (Micro Seconds)');
grid on;
legend('OMP','LS-OMP','MP', 'Weak MP','Location','NorthWest')
%print(f,'-dpng', '-r500', [str '_3.png'])

```

```

f=figure;
hist(mutual_coherence)
xlabel('Bins');
title(['Histogram of Mutual Coherence of A, Distribution ' str])
ylabel('Frequency');
grid on;
%print(f,'-dpng', '-r500', [str '_4.png'])

```

```

f=figure;
hist(upper_bound)
xlabel('Bins');
title(['Histogram of Upper Bound of A, Distribution ' str])
ylabel('Frequency');
grid on;
%print(f,'-dpng', '-r500', [str '_5.png'])

```

```

function [xk_OMP, support] = OMP(A, b, epsilon)
    %Orthogonal Matching Pursuit as in Fig 3.1
    %epsilon -> Threshold
    xk_OMP = zeros(size(A,2),1);
    residual = b; % Residual
    support = []; % Initialize Sol. Support
    resNorm2 = norm(residual,2); %Init Residual Norm
    while resNorm2>epsilon
        sweepZj = abs(A'*residual);
        sweepPosj = find(sweepZj==max(sweepZj));
        support = [support,sweepPosj(1)];
        xk = A(:,support)\b;
        residual = b-A(:,support)*xk;
        resNorm2 = norm(residual,2);
    end
    xk_OMP(support)=xk;
end

```

```

function [xk_LSOMP, support] = LSOMP(A, b, epsilon)
    %Full Least Squares
    %Orthogonal Matching Pursuit as in
    %Extension to Eq. 3.4
    %epsilon -> Threshold

    xk_LSOMP = zeros(size(A,2),1);
    residual = b; % Residual
    support = []; % Initialize Sol. Support
    resNorm2 = norm(residual,2); %Init Residual Norm
    while resNorm2>epsilon
        sweepZj = zeros(size(A,2),1);
        for k = 1:size(A,2)

```

```

        suppKInt = [support k];
        xkInt     = pinv(A(:,suppKInt))*b;
        resKInt   = b-A(:,suppKInt)*xkInt;
        sweepZj(k) = resKInt'*resKInt;
    end
    sweepPosj = find(sweepZj==min(sweepZj),1);
    support = [support,sweepPosj(1)];
    xk       = A(:,support)\b;
    residual = b-A(:,support)*xk;
    resNorm2 = norm(residual,2);
end
xk_LSOMP(support)=xk;
end

```

```

function [ xk_MP, support] = MP(A, b, epsilon)
    %Matching Pursuit as in Fig 3.2
    %epsilon -> Threshold
    xk_MP     = zeros(size(A,2),1); %Init Soln
    residual = b;                    %Residual
    resNorm2 = norm(residual,2); %Init Residual Norm
    while resNorm2>epsilon
        sweepZj = A'*residual;%Norm aj=1 dont divide
        sweepPosZj = find(abs(sweepZj)==max(abs(sweepZj)),1);%Find
            Minimizer
        zj0      = A(:,sweepPosZj)'*residual;
        xk_MP(sweepPosZj) = xk_MP(sweepPosZj)+zj0;%Update Solution
        residual = residual-A(:,sweepPosZj)*zj0;
        resNorm2 = norm(residual,2);
    end
    support = find(abs(xk_MP)>1e-6);
end

```

```

function [xk_WeakMP, support] = WeakMP(A, b, epsilon, t)
    %Weak Matching Pursuit as in Fig 3.3
    %epsilon -> Threshold
    xk_WeakMP = zeros(size(A,2),1); %Init Soln
    residual = b;                    %Residual
    resNorm2 = norm(residual,2); %Init Residual Norm
    while resNorm2>epsilon
        sweepZj = abs(A'*residual); %Norm aj=1 dont divide
        sweepPosZj = find(sweepZj>=t*resNorm2,1);%Find Minimizer
        if isempty(sweepPosZj)
            sweepPosZj=find(sweepZj==max(sweepZj),1);
        end
        zj0      = A(:,sweepPosZj)'*residual;
        xk_WeakMP(sweepPosZj) = xk_WeakMP(sweepPosZj)+zj0;%Update Solution
        residual = residual-A(:,sweepPosZj)*zj0;
        resNorm2 = norm(residual,2);
    end

```

```

    end
    support =find(abs(xk_WeakMP)>1e-6);
end

```

```

function [ mu ] = mutualCoherence(A)
%MUTUALCOHERENCE Summary of this function goes here
% Detailed explanation goes here
    dim1 = size(A,2);
    cloak=eye(dim1)~=1;
    mu=max(max(abs(A'*A).*cloak));
end

```

```

function [ x, pos ] = genSparseVect( str, Card, dim2 )
%GENSPARSEVECT Summary of this function goes here
%Generate Sparse Vector of cardinality C
    if(strcmp(str,'Elad'))
        x = zeros(dim2,1);
        pos = randperm(dim2);
        x(pos(1:Card))=sign(randn(Card,1)).*(1+rand(Card,1));
    elseif(strcmp(str,'Normal'))
        x = zeros(dim2,1);
        pos = randperm(dim2);
        x(pos(1:Card))=randn(1,Card);
    elseif(strcmp(str,'Binary'))
        x = zeros(dim2,1);
        pos = randperm(dim2);
        temp = randi([0 1],1,Card);
        temp(temp==0)=-1;
        x(pos(1:Card))=temp;
    elseif(strcmp(str,'Cauchy'))
        x = zeros(dim2,1);
        pos = randperm(dim2);
        x(pos(1:Card))=trnd(1,Card,1);
    end
end

```
