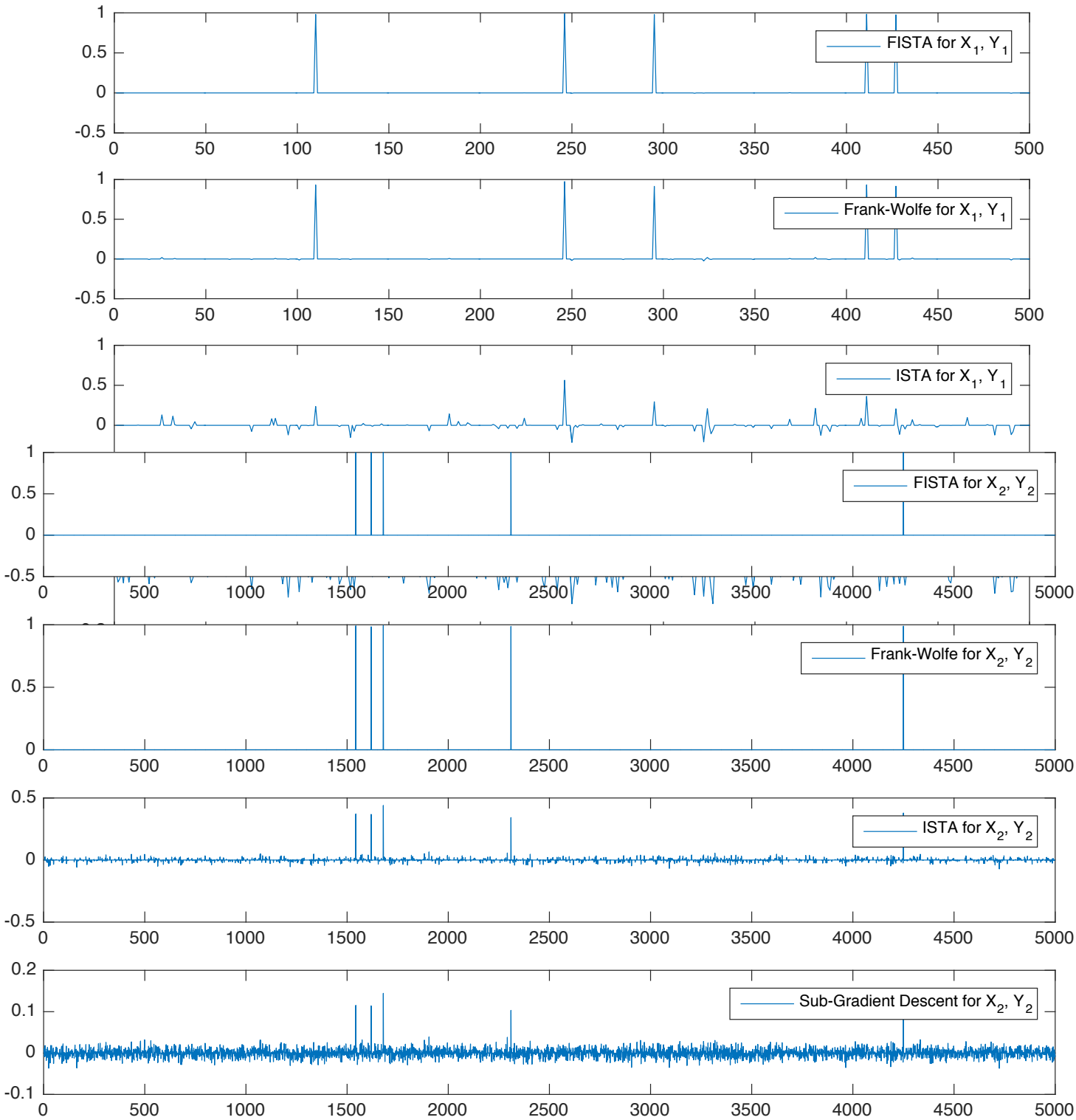
Abhishek Sinha

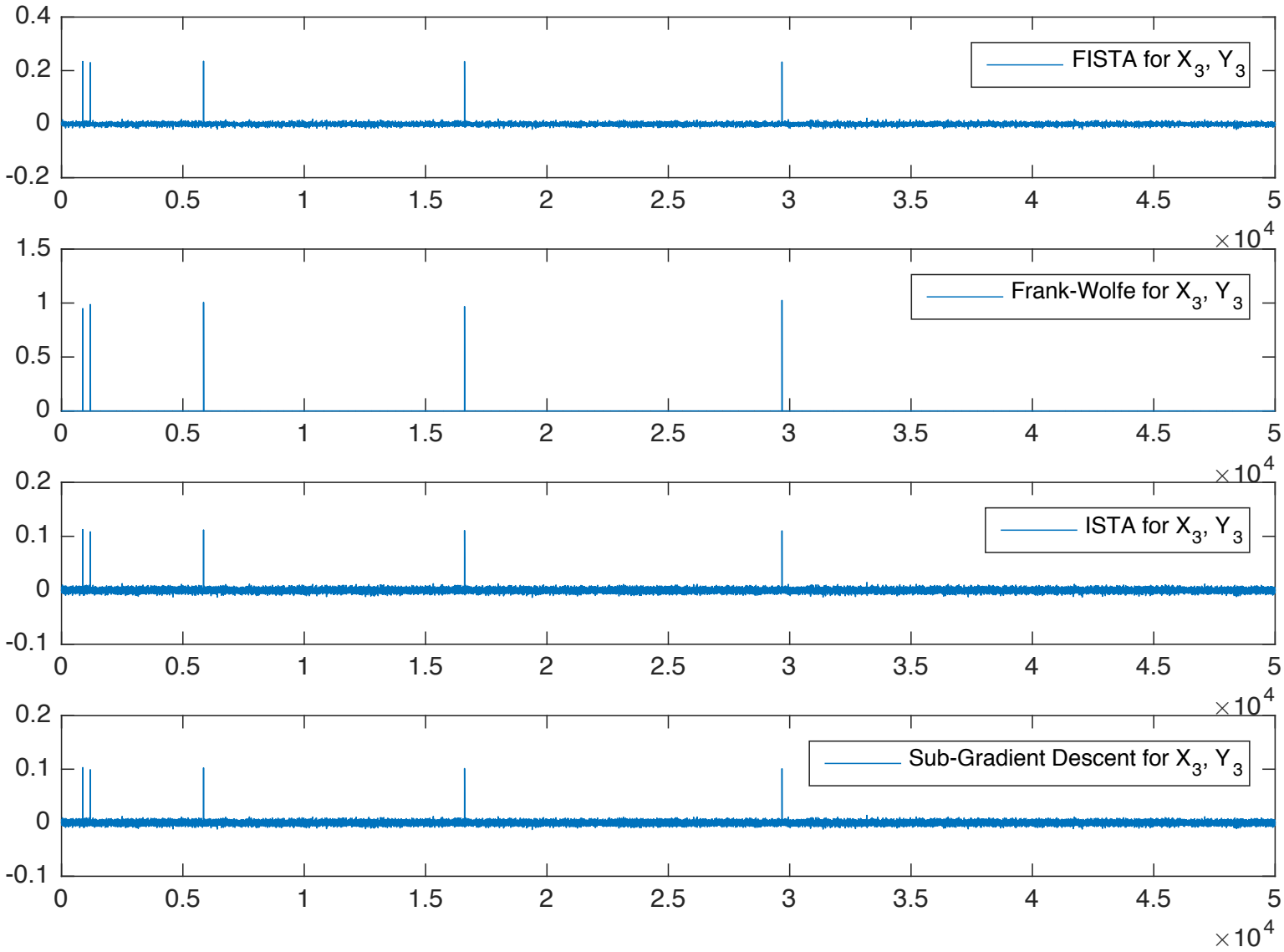
Problem Set 9
Computational Problems

Collaborators
Ashish Bora
Srilakshmi Pattabiraman

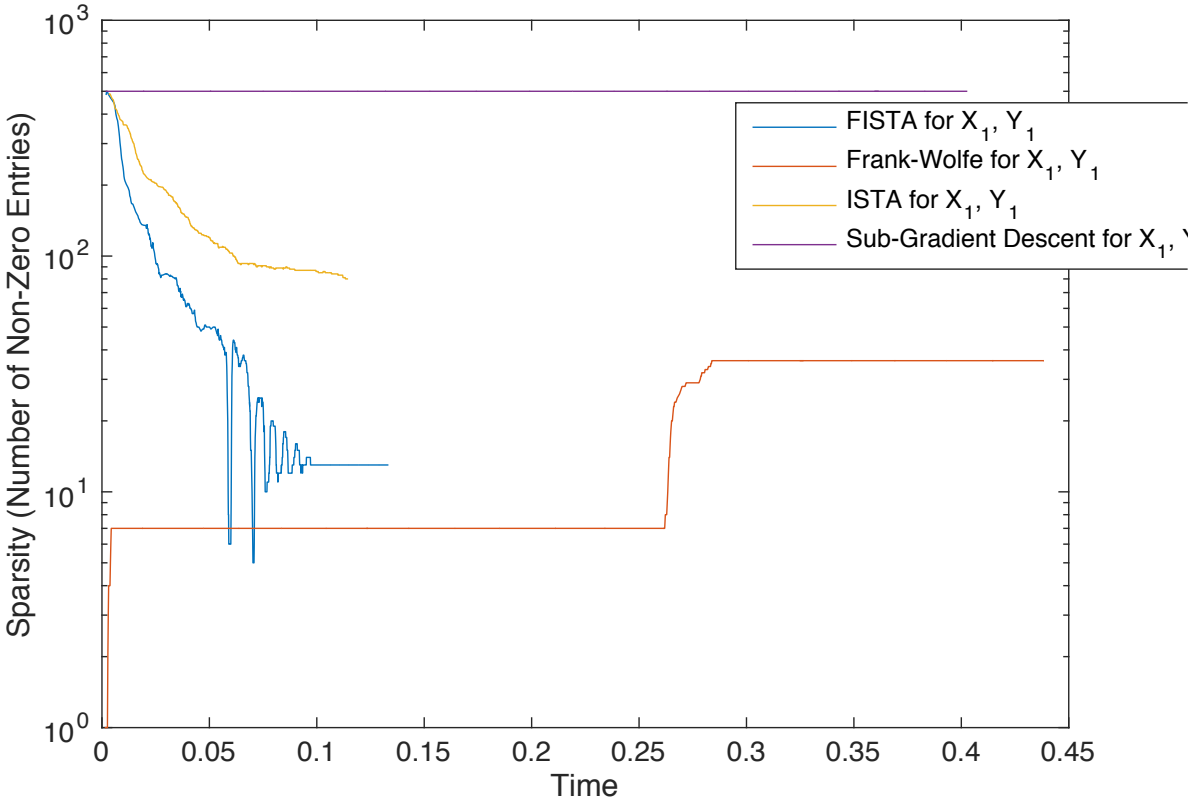
Question 1 & 2

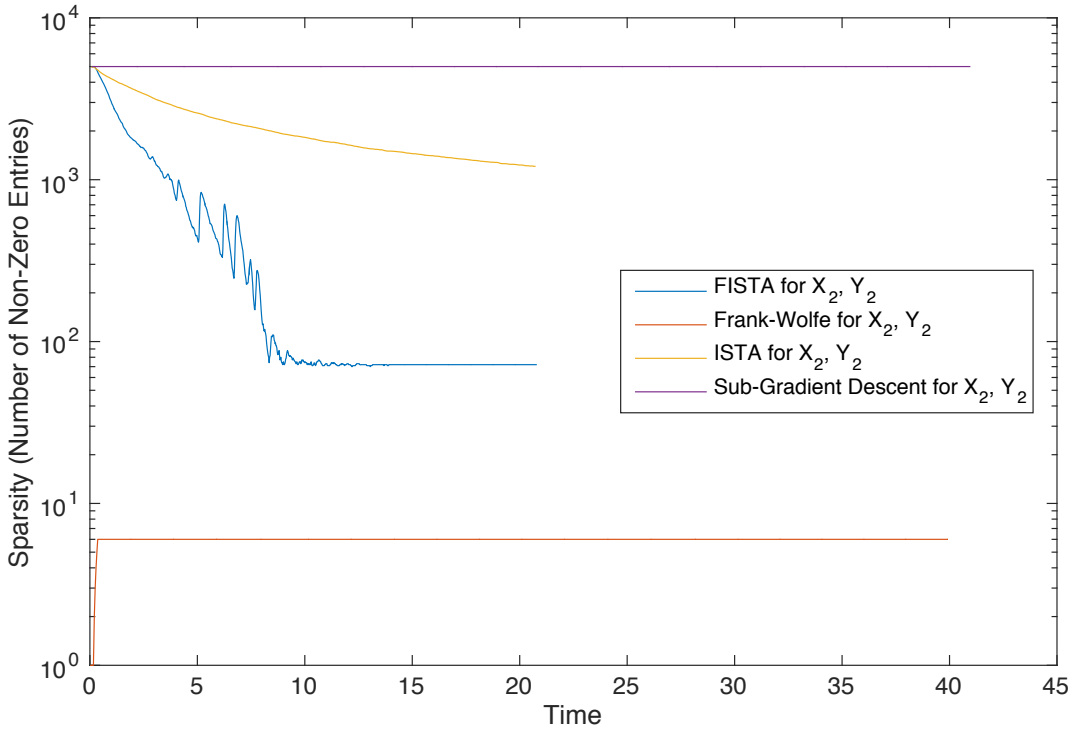
We show results for 4 algorithms , Sub-gradient descent, Proximal Gradient Descent (ISTA), Accelerated Proximal Gradient Descent (FISTA) and

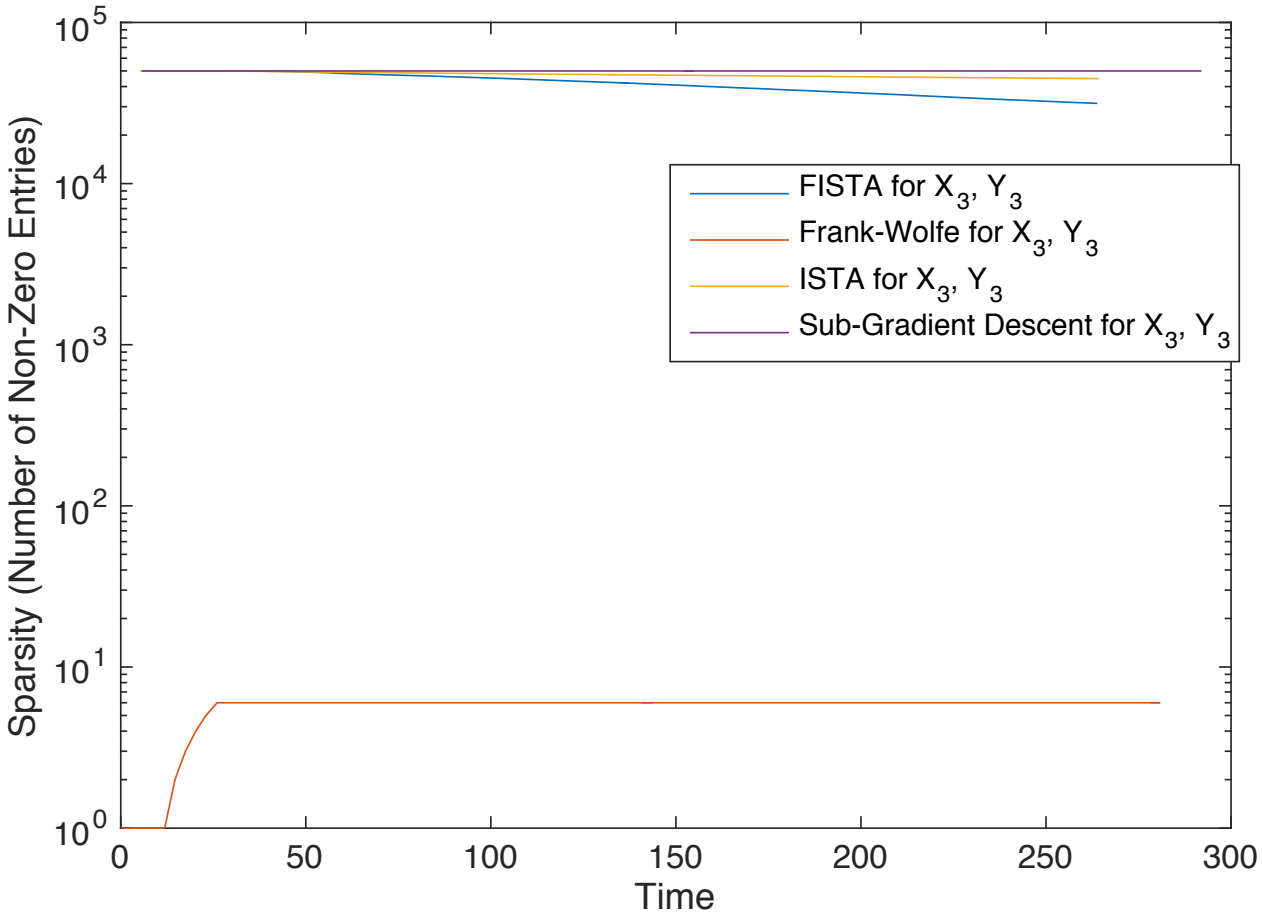




Sparsity as a Function of Time


Sparsity (Number of Non-Zero Entries)
Time
FISTA for X1, Y1
Frank-Wolfe for X1, Y1
ISTA for X1, Y1
Sub-Gradient Descent for X1, Y


Sparsity (Number of Non-Zero Entries)
Time
FISTA for X2, Y2
Frank-Wolfe for X2, Y2
ISTA for X2, Y2
Sub-Gradient Descent for X2, Y2

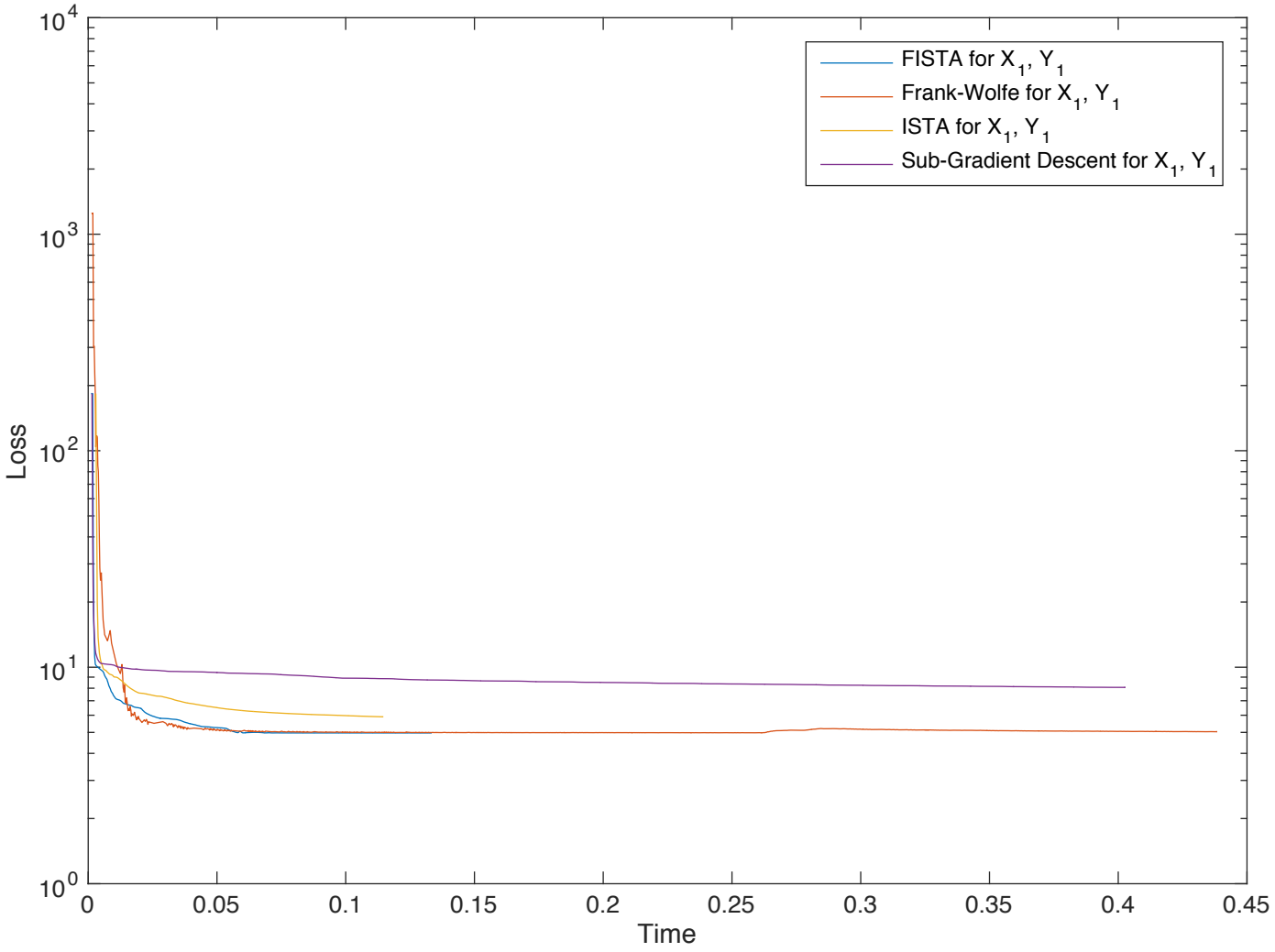


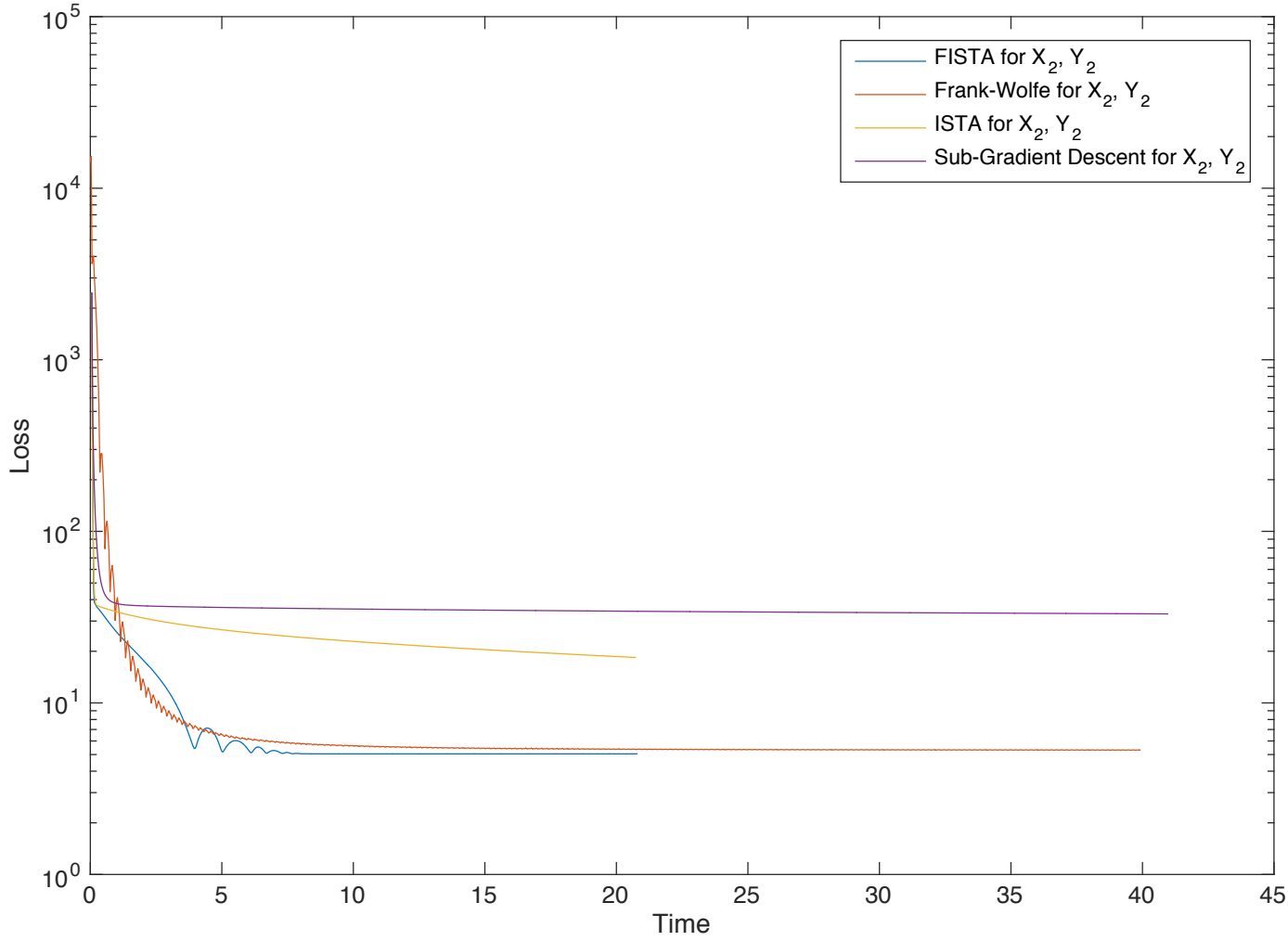
(b) Error as a function of time

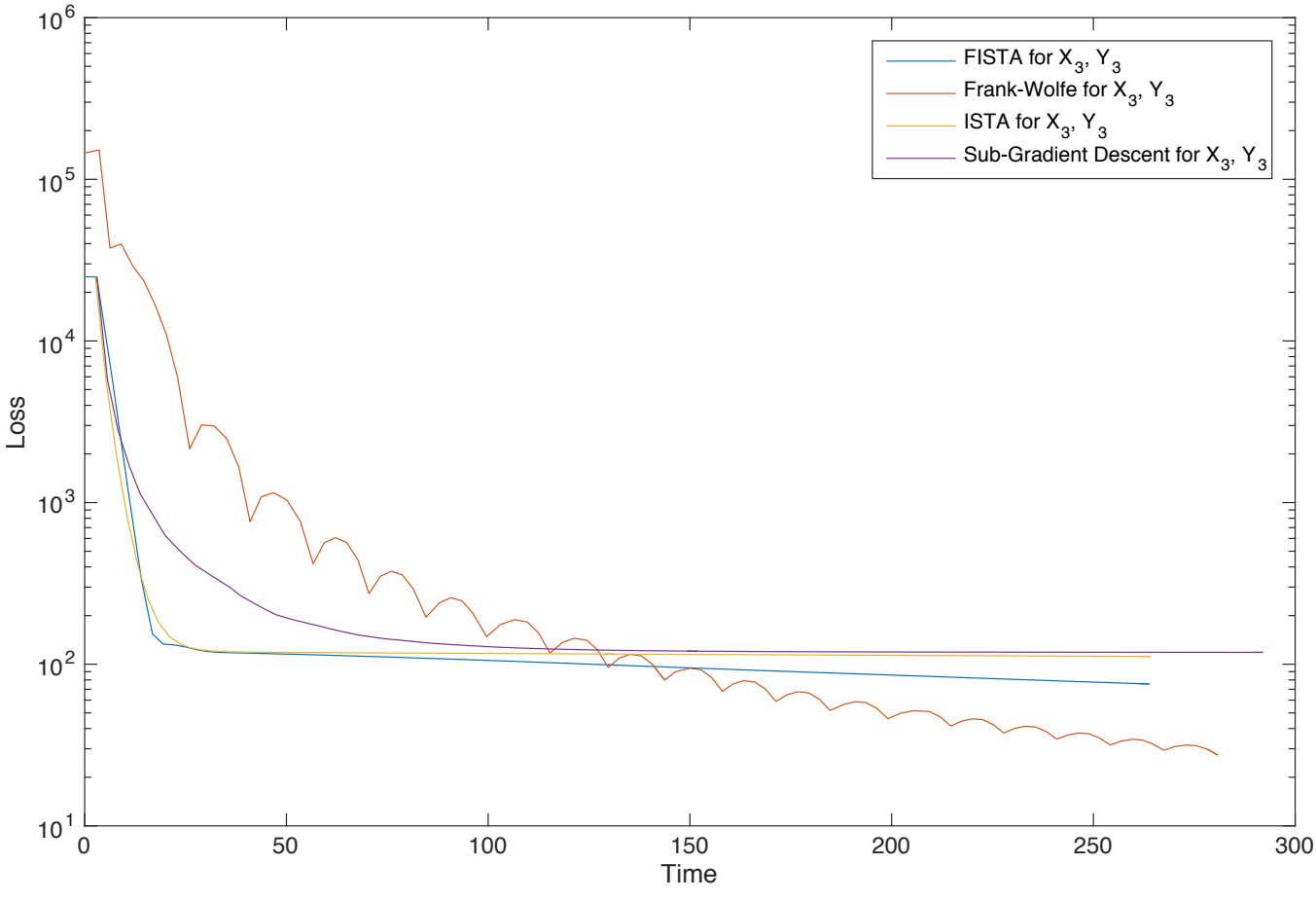
Again FISTA tends to show the smallest loss as well as test error at convergence. Initially Frank-Wolfe shows smaller error. But as the number of iterations increase FISTA algorithm picks up momentum and the error decreases at a much faster rate (as seen in the concave shape of the error plot in the initial part).

Again for the largest matrix, Frank-Wolfe shows the best results because we were able to run it only for a small number of iterations. For these smaller number of iterations, Frank-Wolfe produces a much sparser solution which explains the much lower test error for Frank-Wolfe in this scenario

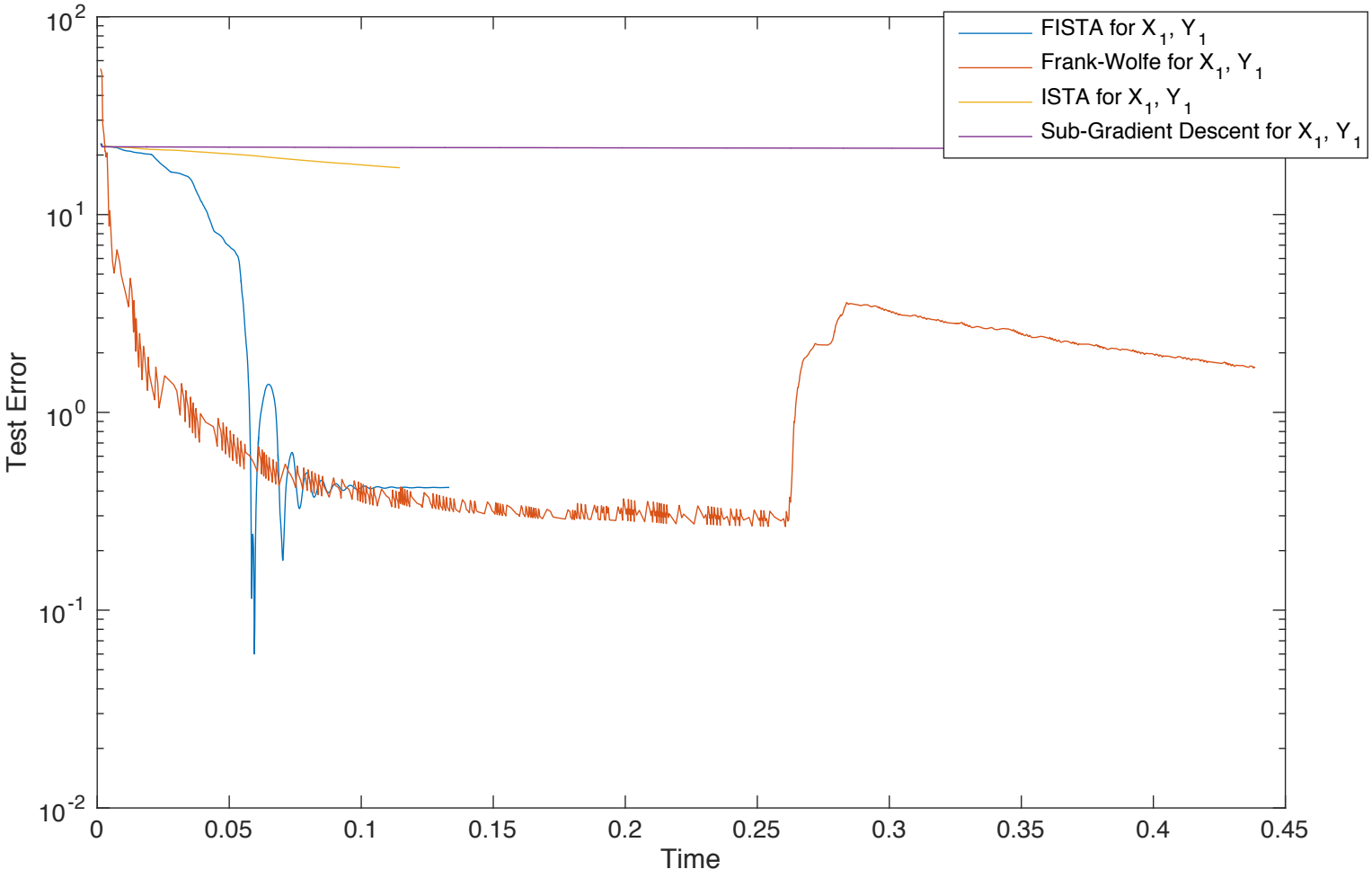
Loss as a Function of Time

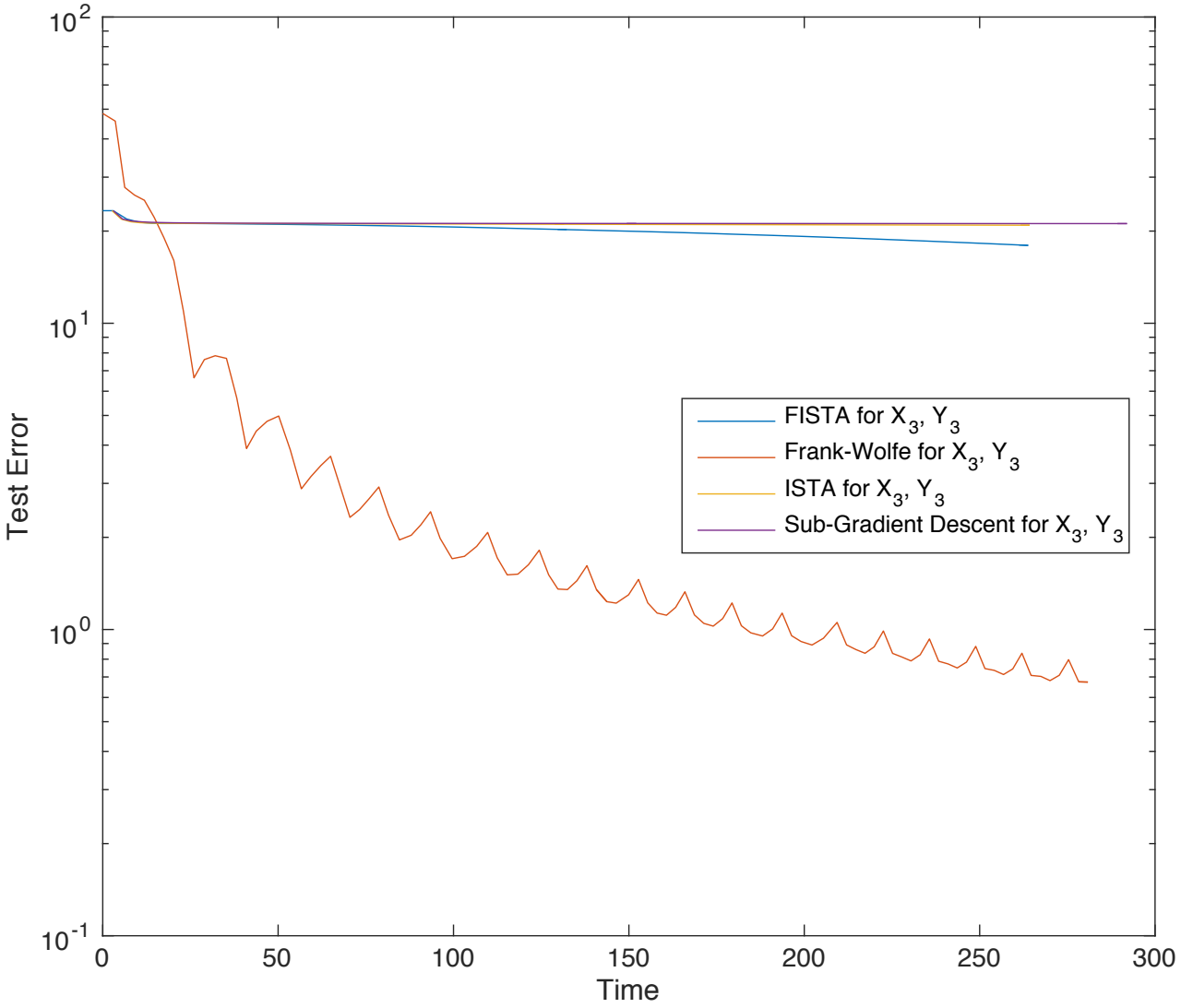


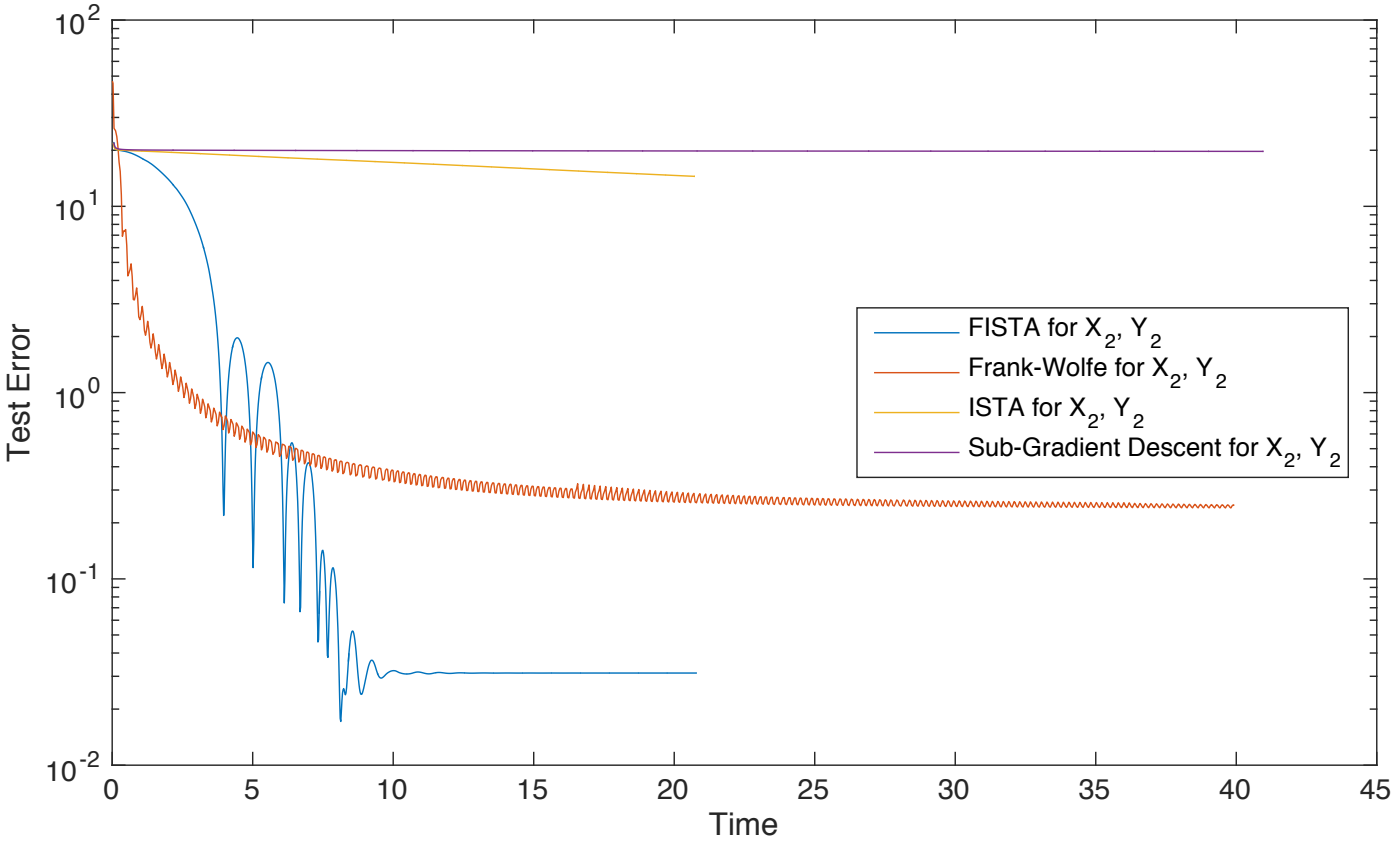

FISTA for X2, Y2
Frank-Wolfe for X2, Y2
ISTA for X2, Y2
Sub-Gradient Descent for X2, Y2
Loss
Time
10^5
10^4
10^3
10^2
10^1
10^0
0
5
10
15
20
25
30
35
40
45


FISTA for X3, Y3
Frank-Wolfe for X3, Y3
ISTA for X3, Y3
Sub-Gradient Descent for X3, Y3
Loss
Time
10^6
10^5
10^4
10^3
10^2
10^1
0
50
100
150
200
250
300

Test Error as a Function of Time




Test Error
Time
FISTA for X3, Y3
Frank-Wolfe for X3, Y3
ISTA for X3, Y3
Sub-Gradient Descent for X3, Y3


Test Error
Time
FISTA for X2, Y2
Frank-Wolfe for X2, Y2
ISTA for X2, Y2
Sub-Gradient Descent for X2, Y2

Question 3

(a) Projected Subgradient Descent

```
x = x - eta*subGradf(x)
x = proj(x)
```

where
subGradf(x) = X' * sign(X*beta - y)

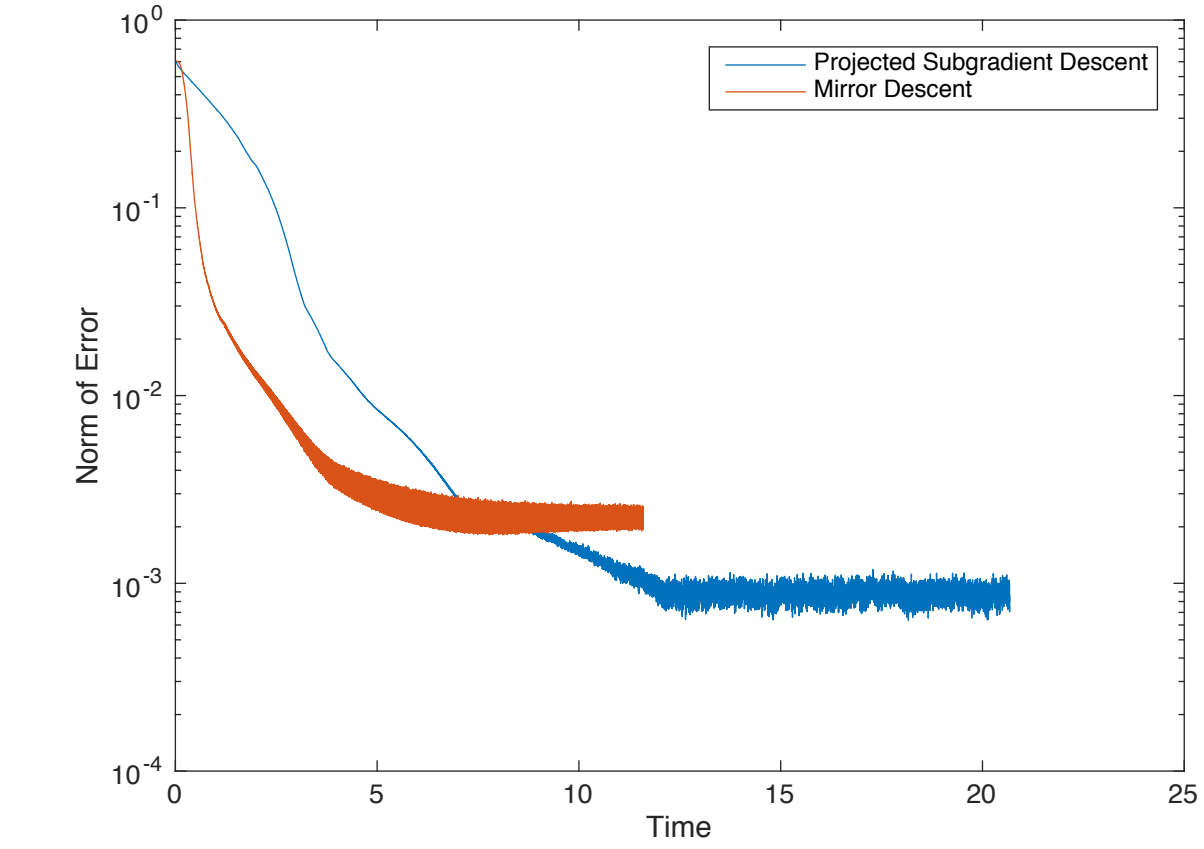
proj(x , z)
Projects the vector x on the simplex with magnitude z
— Let y be the vector composed of entries of x in sorted order. Then the projection is computed as follows
—

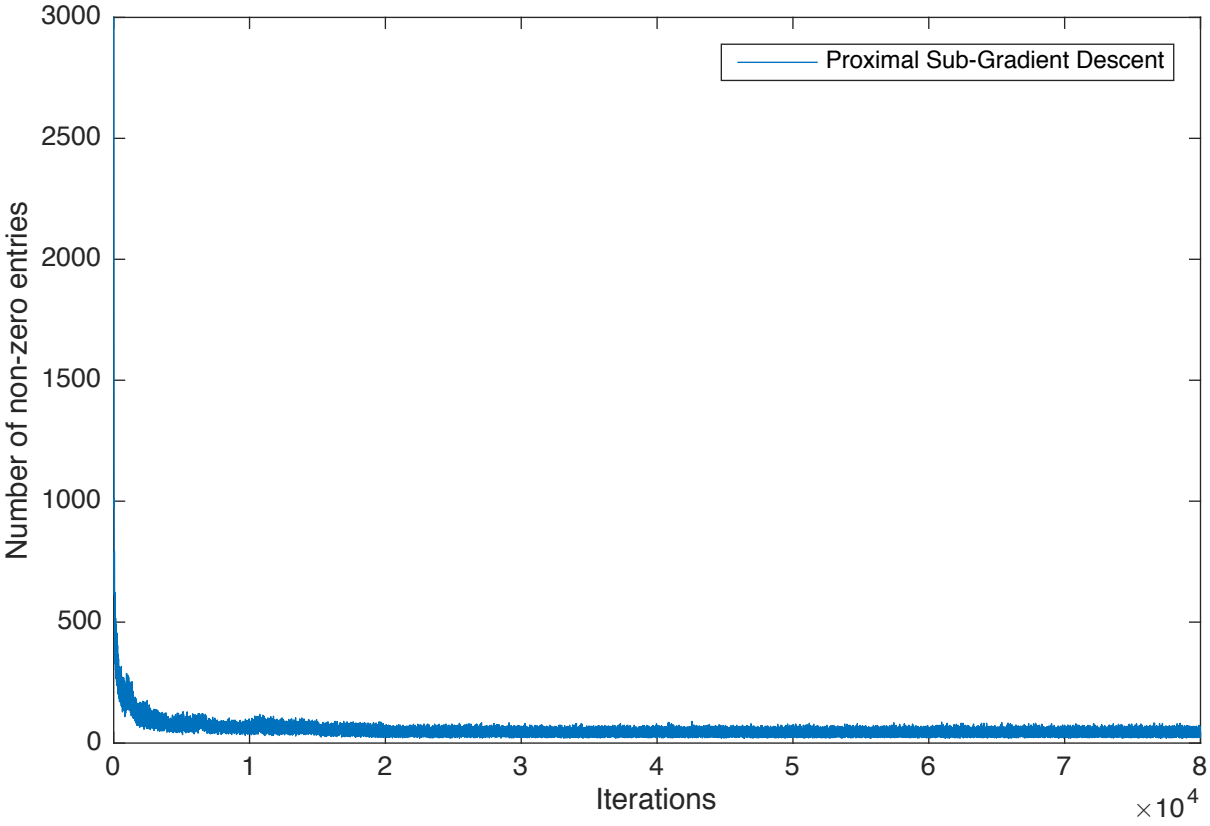
```
prefixSum = 0;
while(i<= n)
    prefixSum = prefixSum + y(i,1);
    if( (y(i,1) - (prefixSum - z)/i) <= 0)
        prefixSum = prefixSum - y(i,1);
        break;
    end;
    i = i+1;
end
theta = (prefixSum - z)/(i-1);
output = max( x - theta, 0);    // Done element wise
```

(b) Mirror Descent

```
betaNew(i) = beta(i) * exp( - eta * grad(i))
betaNew = betaNew / norm(betaNew, 1)
```

where grad(i) indicates the component i of the gradient evaluated at beta.


Projected Subgradient Descent
Mirror Descent
Norm of Error
Time
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
0
5
10
15
20
25

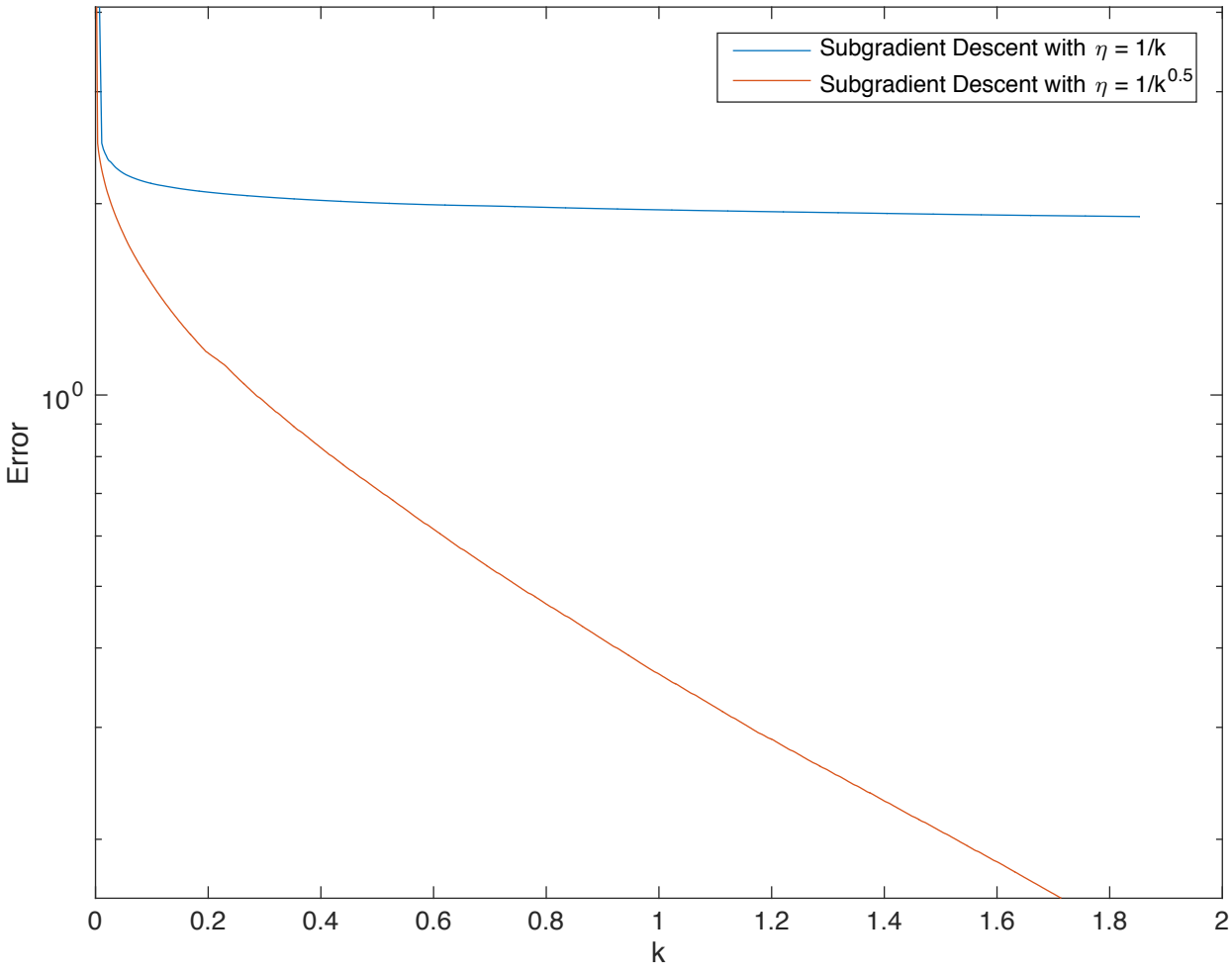

Proximal Sub-Gradient Descent
Number of non-zero entries
Iterations
$\times 10^{4}$
3000
2500
2000
1500
1000
500
0
0
1
2
3
4
5
6
7
8

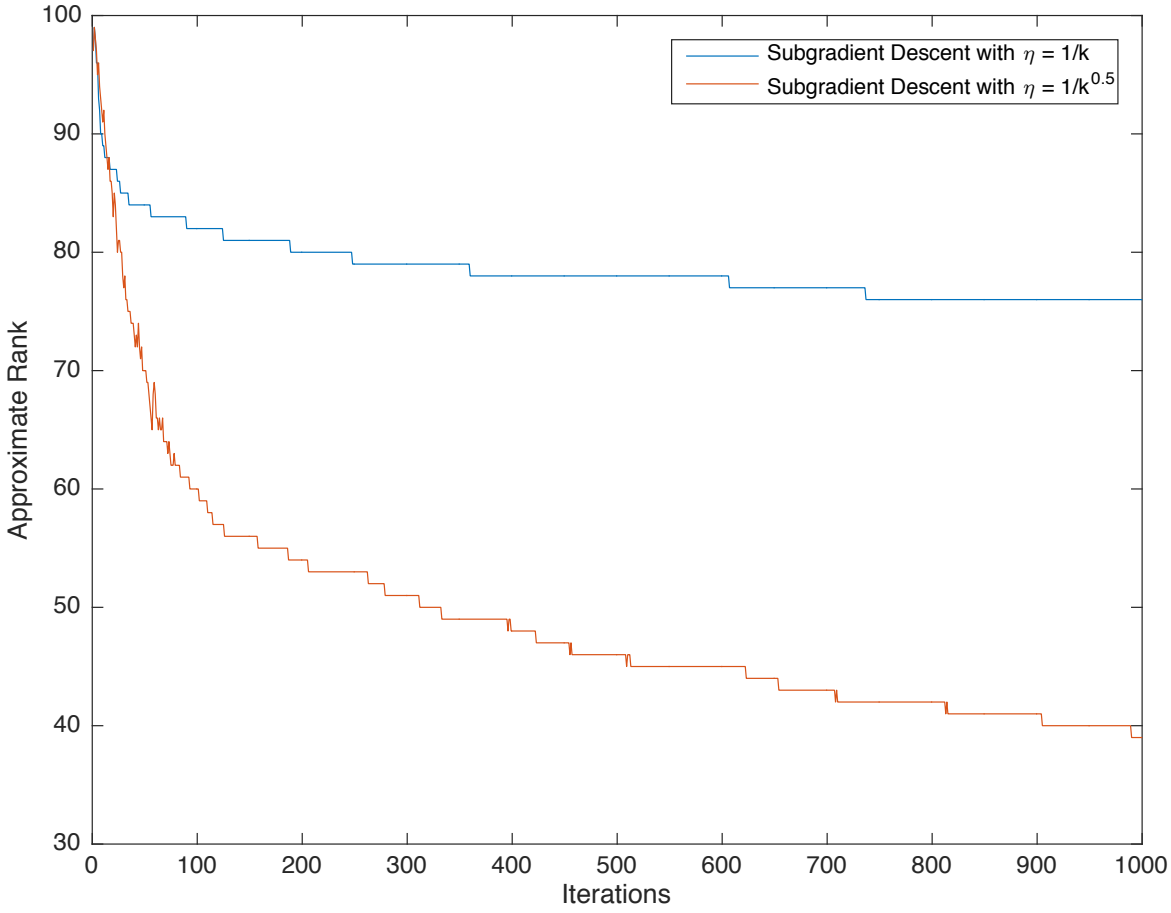
(c) The optimal solution was first computed using CVX. After that, we plotted the norm of the error as a function of time. While mirror descent was initially much faster than proximal sub gradient descent, in the end proximal sub-gradient descent converged to a lower error. The solution obtained via proximal sub-gradient descent was found to be much sparser with the final solution having

Question 4

(a) [U, S, V] = svd(A)

Z = UV'
W = 0 satisfies the necessary conditions



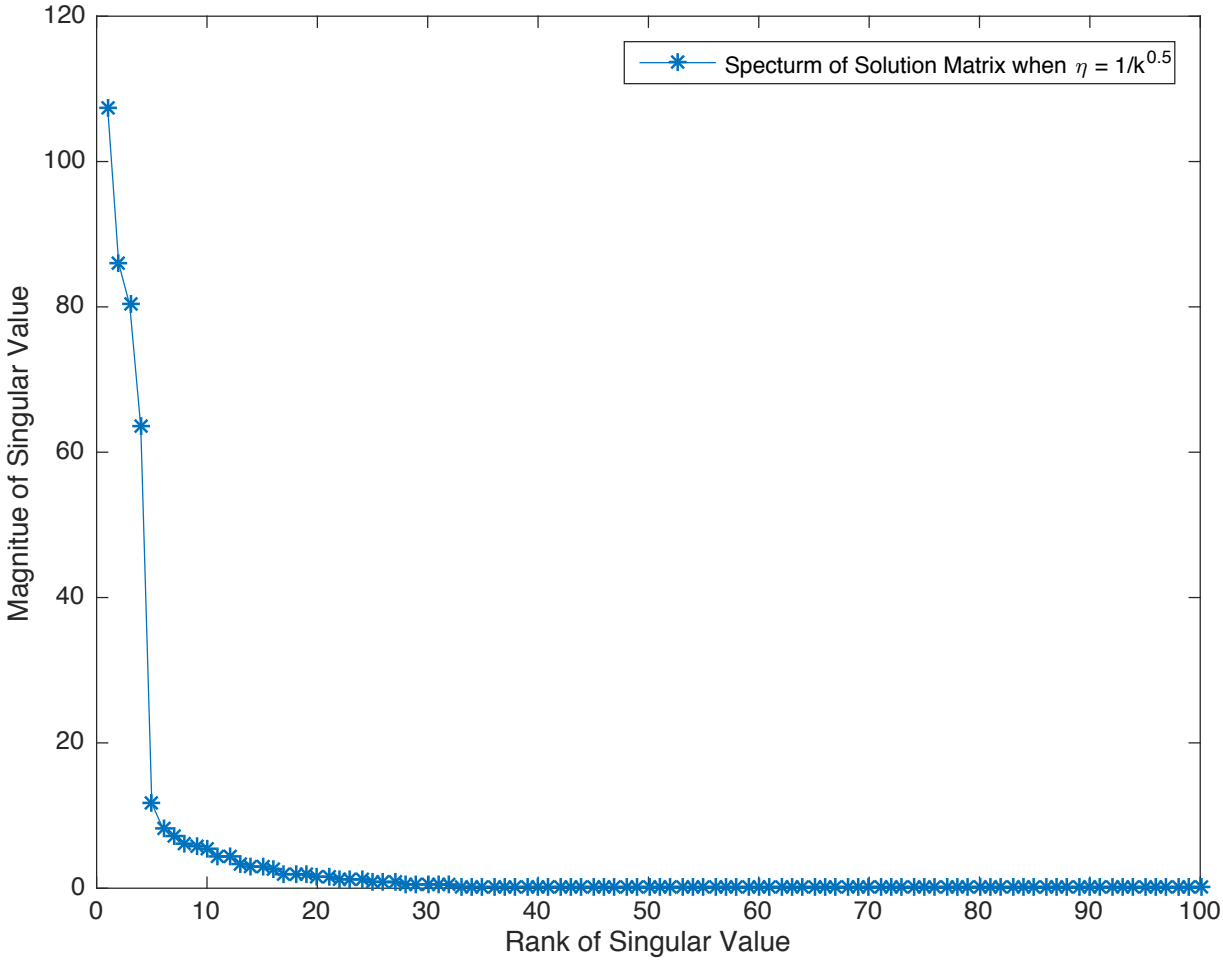


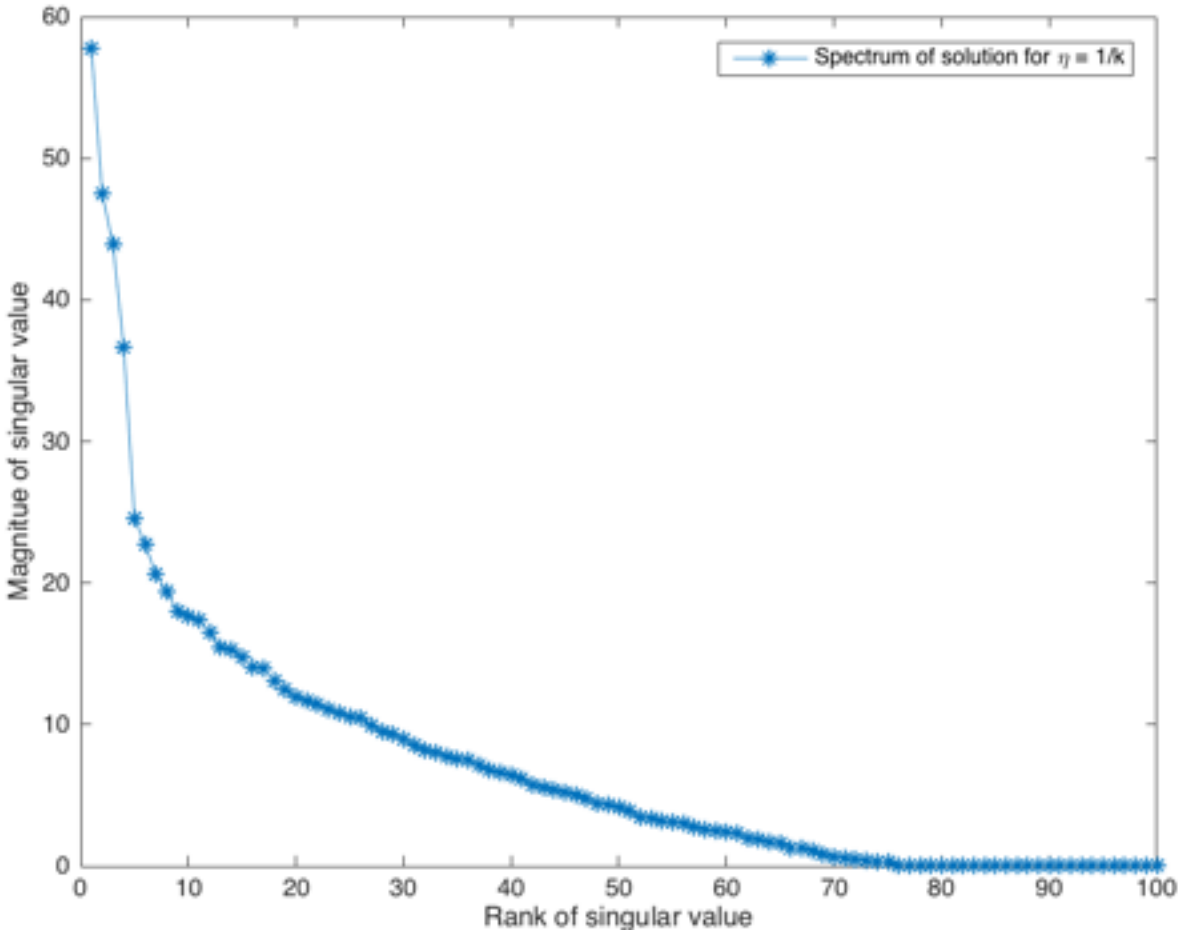
Also, we do not need to take the first 'r' entries since all the intermediate matrices are found to be full rank

(b) To project a matrix on the feasible set, we will set all positions belonging to omega to the corresponding value in M, and retain the remaining values as it is.

(d) The method with 1/k step size converges at a very slow rate to a high error, whereas the method with 1/sqrt(k) converges at a much faster rate.

(e) The exact rank of the intermediate vertices is found to be 100 (i.e. full rank). This is because even a small perturbation of any low rank matrix produces a full rank matrix with probability 1. However when we computed the approximate rank which is the number of singular values greater than a certain threshold (in this case 0.1), it was found to decrease in a regular fashion.


Specturm of Solution Matrix when η = 1/k$^{0.5}$
Magnitue of Singular Value
Rank of Singular Value
0
20
40
60
80
100
120
0
10
20
30
40
50
60
70
80
90
100


Spectrum of solution for η = 1/k
Magnitue of singular value
Rank of singular value
0
10
20
30
40
50
60
0
10
20
30
40
50
60
70
80
90
100

Also when we plotted the spectrum of the final solution matrices, then the following plots were obtained. In the case, where the step size was 1/ sqrt(k) , upon performing a k-means clustering of the final singular values into 2 clusters, the cluster with the higher singular value had size exactly 4 which is also the actual rank of the low-rank matrix M. Thus in this case, we can see that the final solution is a good approximation to M.

Appendix(Code)

Question 1 and 2

Accelerated Proximal Gradient Descent (FISTA)

```
function [ traj, k, times ] =
acceleratedProximalGradientDescent(n , prox, gradf,
eta, maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
xs = zeros(n, 1);
ys = xs;
lambdaS = 0;
k = 1;
tic
while( k <= maxIter)
    traj(k,:) = xs';
    times(k, : ) = toc;
    xs = xs - eta*gradf(xs);
    [ysp] = prox(xs);
    lambdaSP = (1 + (1 + 4*lambdaS^2)^0.5)/2;
    gammaS = (1-lambdaS)/lambdaSP;
    xs = (1-gammaS)*ysp + gammaS*ys;
    ys = ysp;
    lambdaS = lambdaSP;
    k = k+1;
end
```

```
k = k - 1;
end
```

Proximal Gradient Descent

```
function [ traj,  k, times ] =
proximalGradientDescent(n ,  prox, gradf,   eta,
maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter,1);
x = zeros(n,1);
k = 1;
tic;
while(  k <= maxIter)
    traj(k,:) = x';
    x = x - eta*gradf(x);
    [x] = prox(x);
    times(k,1) = toc;
    k = k+1;
end
k = k-1;
end
```

Sub-Gradient Descent

```
function [ traj,  k, times ] = subGradientDescent(n ,
x0, subGradf, eta0, ep, maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
x = x0;
n = norm(gradf(x));
k = 1;
eta = eta0;
tic;
while(  k <= maxIter)
    traj(k,:) = x';
    x = x - eta*subGradf(x);
    times(k,1) = toc;
```

```
    k = k+1
    eta = eta0/(k^0.5);
end
k = k-1;
end
```

Frank-Wolfe(Conditional Gradient Descent)

```
function [ traj,  times, k ] =
conditionalGradientDescent(n , linearMinimizer, gradf,
x0, maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
x = x0;
k = 1;
tic
while( k <= maxIter)
    size(traj)
    traj(k,:) = x';
    times(k, : ) = toc;
    gammaK = 2/(k+1);
    y = linearMinimizer(gradf(x));
    x = (1-gammaK)*x + gammaK*y;
    k = k+1;
end
k = k - 1;
end
```

Helper Functions

```
function [ y ] = minimizeOnSimplex( gradVector, r )
[ maxAbsVal, maxAbsValComp] = max(abs(gradVector));
y = zeros(size(gradVector));
y(maxAbsValComp, 1) = -r*sign(gradVector(maxAbsValComp,
1));
end
```

```
function [ output ] = lassoFunc( X,y,beta, lambda )
output = norm(X*beta - y)^2 + lambda*norm(beta, 1);
end


function [ output ] = gradientLeastSquares( X, XY, beta
)
output = 2*X'*(X*beta) - 2*XY;
end


function [ output ] = lassoSubGrad( X, XY, beta, lambda
)
output = gradientLeastSquares(X, XY, beta)
+lambda*sign(beta);
end


function [ y ] = proxAbsoluteValue( x, t, lambda )
t_net = t*lambda;
sz = size(x);
n = sz(1);
y = zeros(n,1);
for i = 1:n
    if(abs(x(i,1)) < t_net)
        y(i,1) = 0;
    else
        y(i,1) = x(i,1) - sign(x(i,1))*t_net;
    end
end
end
```

Wrapper Scripts

```
if(matrixInd == 1)
    X = X1;
```

```
    y = y1;
    Xtest = X1test;
    ytest = y1test;
else if(matrixInd == 2)
    X = X2;
    y = y2;
    Xtest = X2test;
    ytest = y2test;
else
    X = X3;
    y = y3;
    Xtest = X3test;
    ytest = y3test;
    end;
end;

lambda = [1];
[recoveredTimesSGD1, recoveredTrajSGD1, indexSGD,
recoveredTimesAPGD, recoveredTrajAPGD, indexAPGD,
recoveredTimesPGD1, recoveredTrajPGD1, indexPGD] =
lasso(X, y, Xtest, ytest, lambda);
szSGD = size(recoveredTimesSGD1);
szPGD = size(recoveredTimesPGD1);
szAPGD = size(recoveredTimesAPGD);
lossSGD = zeros(szSGD(1),1);
lossPGD = zeros(szPGD(1),1);
lossAPGD = zeros(szAPGD(1),1);

for i = 1:szSGD(1)
    lossSGD(i,1) =
lassoFunc(X,y,recoveredTrajSGD1(i,:)',
lambda(1,indexSGD));
end
for i = 1:szPGD(1)
    lossPGD(i,1) =
lassoFunc(X,y,recoveredTrajPGD1(i,:)',
lambda(1,indexPGD));
end
```

```
for i = 1:szAPGD(1)
    lossAPGD(i,1) =
lassoFunc(X,y,recoveredTrajAPGD(i,:)',
lambda(1,indexAPGD));
end
semilogy(recoveredTimesSGD1,    lossSGD);
hold on;
semilogy(recoveredTimesPGD1, lossPGD);
semilogy(recoveredTimesAPGD, lossAPGD);
xlabel('Time');
ylabel('Loss');
istr = num2str(i);
legend('Sub Gradient Descent for X3,Y3', 'Proximal
Gradient Descent for X3,Y3', 'Accelerated Proximal
Gradient Descent for X3,Y3');
```

Computing Sparsity

```
function [ output ] = computeSparsity( traj )
sz = size(traj);
n = sz(1);
output = zeros(n,1);
for i = 1:n
    output(i, 1) = nnz(traj(i,:));
end
end

sparsityCGD = computeSparsity(recoveredTrajCGD);
sparsityAPGD = computeSparsity(recoveredTrajAPGD);
sparsityPGD = computeSparsity(recoveredTrajPGD1);
sparsitySGD = computeSparsity(recoveredTrajSGD1);
```

Computing Test Error

```
function [ output ] = computeTestError( X, y, traj )
sz = size(traj);
n = sz(1);
```

```
output = zeros(n,1);
for i = 1:n
    output(i, 1) = norm(X*traj(i,:)' - y);
end
end

testSGD = computeTestError(Xtest, ytest,
recoveredTrajSGD1);
testCGD = computeTestError(Xtest, ytest,
recoveredTrajCGD);
testAPGD = computeTestError(Xtest, ytest,
recoveredTrajAPGD);
testPGD = computeTestError(Xtest, ytest,
recoveredTrajPGD1);
```

Plotting the Graphs

```
figure(1)
semilogy(recoveredTimesAPGD, lossAPGD);
hold on;
semilogy(recoveredTimesCGD, lossCGD);
semilogy(recoveredTimesPGD1, lossPGD);
semilogy(recoveredTimesSGD1, lossSGD);
xlabel('Time');
ylabel('Loss');
legend('FISTA for X_1, Y_1', 'Frank-Wolfe for X_1,
Y_1', 'ISTA for X_1, Y_1', 'Sub-Gradient Descent for
X_1, Y_1');

figure(2)
semilogy(recoveredTimesAPGD, testAPGD);
hold on;
semilogy(recoveredTimesCGD, testCGD);
semilogy(recoveredTimesPGD1, testPGD);
semilogy(recoveredTimesSGD1, testSGD);
xlabel('Time');
ylabel('Test Error');
```

```
legend('FISTA for X_1, Y_1', 'Frank-Wolfe for X_1,
Y_1', 'ISTA for X_1, Y_1', 'Sub-Gradient Descent for
X_1, Y_1', 'location', 'east');


figure(3)
semilogy(recoveredTimesAPGD, sparsityAPGD);
hold on;
semilogy(recoveredTimesCGD,  sparsityCGD);
semilogy(recoveredTimesPGD1, sparsityPGD);
semilogy(recoveredTimesSGD1, sparsitySGD);
xlabel('Time');
ylabel('Sparsity (Number of Non-Zero Entries)');
legend('FISTA for X_1, Y_1', 'Frank-Wolfe for X_1,
Y_1', 'ISTA for X_1, Y_1', 'Sub-Gradient Descent for
X_1, Y_1', 'location', 'east');


figure(4)
sz = size(recoveredTrajAPGD);
subplot(4,1, 1);
plot(recoveredTrajAPGD(sz(1), :));
legend('FISTA for X_1, Y_1');
subplot(4,1, 2);
plot(recoveredTrajCGD(sz(1), :));
legend('Frank-Wolfe for X_1, Y_1');
subplot(4,1, 3);
plot(recoveredTrajPGD1(sz(1), :));
legend('ISTA for X_1, Y_1');
subplot(4,1, 4);
plot(recoveredTrajSGD1(sz(1), :));
legend('Sub-Gradient Descent for X_1, Y_1');
```

Question 3

Mirror Descent

```
function [ traj,  k, times ] = mirrorDescent(n , x0,
proj, mirrorUpdate, eta0,  maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
x = x0;
n = norm(gradf(x));
k = 1;
eta = eta0;
tic;
while(  k <= maxIter)
    traj(k,:) = x';
    y = mirrorUpdate(x, eta);
    x = proj(y);
    times(k,1) = toc;
    k = k+1;
end
k = k-1;
end
```

Sub Gradient Descent

```
function [ traj,  k, times ] = subGradientDescent(n ,
x0, proj, subGradf, eta0,  maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
x = x0;
n = norm(gradf(x));
k = 1;
eta = eta0;
tic;
while(  k <= maxIter)
    traj(k,:) = x';
    x = x - eta*subGradf(x);
    x = proj(x);
    times(k,1) = toc;
    k = k+1;
end
k = k-1;
end
```

Helper Functions

```
function [ output ] = euclidenProjOnSimplex( x , z)
sz = size(x);
n = sz(1);
y = sort(x, 'descend');
prefixSum = 0;
i = 1;
while(i<= n)
    prefixSum = prefixSum + y(i,1);
    if( (y(i,1) - (prefixSum - z)/i) <= 0)
        prefixSum = prefixSum - y(i,1);
        break;
    end;
    i = i+1;
end
theta = (prefixSum - z)/(i-1);
output = max( x - theta, 0);
end
```

```
function [ output ] = bregmanProjectionOnSimplex( x )
output = x/norm(x,1);
end
```

```
function [ output ] = mirrorUpdateNegativeEntropy( X,
y, beta, eta )
output = beta.*exp(-eta*subGradL1Error(X, beta, y));
end
```

```
function [ output ] = subGradL1Error( X, beta, y )
output = X'*sign(X*beta - y);
end
```

Computing the Optimal Solution

```
function [ opt ] = getOptSolution( X, y , n)
cvx_begin
    variable bet(n);
    minimize (norm( X*bet - y, 1))
    subject to
        bet >= 0;
        sum_largest(bet, n) == 1;
cvx_end
opt = bet;
end
```

Wrapper

```
load robust_regression.mat;

sz = size(X);
n = sz(2);
betaOpt = getOptSolution(X, y, n);

%%%%%%%%%%%% Projected-Sub-Gradient
x0 = ones(n,1)/n;
proj = @(beta)  (euclidenProjOnSimplex(beta, 1));
subGradf = @(beta) (subGradL1Error(X, beta, y));
R = sqrt(2);
L = sqrt(sz(1))*norm(X);
maxIter = 80000;
eta0 = R/(L*sqrt(maxIter));
[ trajPSGD,   k, times ] = subGradientDescent(n , x0,
proj, subGradf, eta0,  maxIter);
normVals = zeros(maxIter, 1);
for i = 1:maxIter
     normVals(i, 1) = norm(trajPSGD(i,:) - betaOpt');
end
semilogy(times, normVals);
hold on;

%%%%%%%%%% Mirror Descent
```

```
R = sqrt(log(n));
maX = -1;
for i = 1:n
    maX = max(maX, norm(X(:,i),1));
end
kappa = 1.0;
eta0 = sqrt(2*kappa/maxIter)*R/maX
proj = @(beta) (bregmanProjectionOnSimplex(beta));
mirrorUpdate = @(beta, eta)
(mirrorUpdateNegativeEntropy(X, y, beta, eta));
[ trajMD,  k, times ] = mirrorDescent(n , x0, proj,
mirrorUpdate, eta0,  maxIter);
for i = 1:maxIter
    normVals(i, 1) = norm(trajMD(i,:) - betaOpt');
end
semilogy(times, normVals);

xlabel('Time');
ylabel('Norm of Error');
legend(  'Projected  Subgradient  Descent',  'Mirror
Descent’);
```

Question 4

Sub-Gradient Descent

```
function [ traj,  k, times ] = subGradientDescent(n ,
x0, proj, subGradf, getEta,  maxIter)
traj = zeros(maxIter, n);
times = zeros(maxIter, 1);
x = x0;
k = 1;
tic;
while(  k <= maxIter)
    eta = getEta(k);
    traj(k,:) = x';
    y = x - eta*subGradf(x);
    x = proj(y);
    times(k,1) = toc;
```

```
    k = k+1;
end
k = k-1;
end
```

Helper Functions

```
function [ output ] = subGradientMatrix( XV, m, n )
X = reshape(XV, m, n);
[U,S, V] = svd(X);
outputM = U*V';
output = outputM(:);
end
```

```
function [ output ] = projectOnSamples( XV, m, n, MO,
OC )
X = reshape(XV, m, n);
outputM = X.*(OC) + MO;
output = outputM(:);
end
```

```
function [ output ] = getEtaInv( eta0, k )
output = eta0/k;
end
```

```
function [ output] = getEtaSqInv( eta0, k )
output = eta0/sqrt(k);
end
```

Wrapper

```
load matrix_completion.mat;

MV = M(:);
xM0 = M.*O;
x0 = rand(10000, 1);
```

```
maxIter = 1000;
eta0 = 1.0;
[m,n] = size(M);
proj = @(X) (projectOnSamples(X, m, n, xM0, ones(m,n) -
O));
subGradf = @(X) (subGradientMatrix(X, m, n));

%%%%%%%%%%%%%% eta = 1/k case
getEta = @(i) (getEtaInv(eta0, i));
[n,m] = size(x0);
[ trajInv,   k, times ] = subGradientDescent(n , x0,
proj, subGradf, getEta,  maxIter);
error = zeros(maxIter, 1);
for j = 1:maxIter
    error(j,1) = (1e-4)*norm(MV - trajInv(j,:)')^2;
end
figure(1);
semilogy(times, error);
hold on;


%%%%%%%%%% eta = 1/sqrt(k) case
getEta = @(i) (getEtaSqInv(eta0, i));
[ trajSqInv,  k, times ] = subGradientDescent(n , x0,
proj, subGradf, getEta,  maxIter);
for j = 1:maxIter
    error(j,1) = (1e-4)*norm(MV - trajSqInv(j,:)')^2;
end
semilogy(times, error);
xlabel('k');
ylabel('Error');
legend('Subgradient Descent with \eta = 1/k',
'Subgradient Descent with \eta = 1/k^{0.5}');

approxRankInv = computeApproximateRank(trajInv, 1e-1);
approxRankSqInv   =   computeApproximateRank(trajSqInv,
1e-1);
figure(2);
```

```
plot(approxRankInv);
hold on;
plot(approxRankSqInv);
xlabel('Iterations');
ylabel('Approximate Rank');
legend('Subgradient Descent with \eta = 1/k',
'Subgradient Descent with \eta = 1/k^{0.5}');
```

Computing Approximate Rank

```
function [ output ] = computeApproximateRank( traj,
toler)
sz = size(traj);
n = sz(1);
m = sqrt(sz(2));
output = zeros(n,1);
for i = 1:n
    M = reshape(traj(i,:)', m, m);
    sv = svds(M, m);
    output(i, 1) = sum(sv > toler);
end
end
```