Project A Sparse solutions of linear equations

Table of Contents

Exercise 3.1

```
type sparsify.m
x = randn(20,1)
sparsify(x,7)
```

```
function x = sparsify(x,k)
%%SPARSIFY Returns a vector with only the largest k entries in
 absolute
%value unaltered; all others become 0.

    [xsort ind] = sort(abs(x), 'descend'); % Sort entries by absolute
 value, from largest to smallest
    x(ind(k+1:end),1) = 0; % Set smallest n-k entries in absolute
 value to 0
end
```

```
x =

    0.5377
    1.8339
   -2.2588
    0.8622
    0.3188
   -1.3077
   -0.4336
    0.3426
    3.5784
    2.7694
   -1.3499
```

```
    3.0349
    0.7254
   -0.0631
    0.7147
   -0.2050
   -0.1241
    1.4897
    1.4090
    1.4172


ans =

         0
    1.8339
   -2.2588
         0
         0
         0
         0
         0
    3.5784
    2.7694
         0
    3.0349
         0
         0
         0
         0
         0
    1.4897
         0
    1.4172
```

Exercise 3.2

The random vector is generated as follows. First, we draw all the coefficients from the uniform distribution on $[0, 1]$. Second, we take $n - k$ positive integers from 1 to n inclusive at random. Finally, we set the coefficients corresponding to these drawn integers to 0, and output the final vector.

```
type randomvector.m
type one_step.m
```

```
function xrandom = randomvector(n,k)
%%RANDOMVECTOR Generates a k-sparse vector with coefficients from
%the uniform distribution on [0,1] in uniformly random places.

xrandom = rand(n,1); % First draw all entries from the uniform
 distribution on [0,1]
y = randsample(n,n-k); % Draw n-k random positive integers from 1 to n
 inclusive
```

```
for i = 1:n-k
    xrandom(y(i)) = 0; % For the randomly chosen integers, set the
 entries corresponding to those indices to 0 instead
end

function errorone = one_step(k,m,n)
%%ONE_STEP Generates an instance of Problem 1 and runs Algorithm 1
%to give the error in approximating the initial sparse random
%vector.

    % Initial setup of Problem 1
    A = randn(m,n); % A matrix with i.i.d stand. Normal coeff.
    x_star = randomvector(n,k); % Calls function, obtains random
 vector
    b = A * x_star;
    B = pinv(A); % The Moore-Penrose pseudo-inverse

    x_hyphen = B * b; % Step 1 of algorithm
    x_tilde = sparsify(x_hyphen, k); % Step 2 of algorithm
    errorone = (norm(x_tilde-x_star))/(norm(x_star)); % Error
end
```

Exercise 3.3

```
one_step(30,200,400)


ans =

    0.5570
```

Proof of closed-form solution of the optimization

In this section we prove the result on page 11 from the project booklet, which states that the closed form for the Eucledian projection onto the set for which $Ax = b$ is given by $z^* = z + A^\dagger(b - Az)$. To show this, we first prove a useful lemma.

Lemma: If $Ax = b$, the minumum $min(x^T x)$ is achieved for $x^* = A^\dagger b$.

Proof: On page 9 it is noted that x^* is indeed a solution of $Ax = b$. Now note that for any other x for which $Ax = b$, we must have that $A(x - x^*) = 0$. For convenience of notation, define $y = x - x^*$ so that Ay = 0. Now consider $y^T x^*$, i.e. the dot product of y and x^*. They are perpendicular, since $y^T x^* = y^T A^T (AA^T)^{-1} b = (Ay)^T * (AA^T)^{-1} b = 0$. Thus we have: $x^T x = (y + x^*)^T (y + x^*) =$ $= y^T y + (x^*)^T x^* \geq (x^*)^T x^*$. This proves the lemma.

Back to the problem, we substitute $t = z^* - z$, and we hence need to minimize $t^T t$ subject to $At = A(z^* - z) = b - Az$. Hence, applying the lemma, the desired closed form for the solution is given by $t = A^\dagger(b - Az)$, or rearranging $z^* = z + A^\dagger(b - Az)$ as desired.

Exercise 3.4

```
type alt_proj.m
```

```
function erroralt = alt_proj(k,m,n)
%%ALT_PROJ Finds the error obtained by running the more sophisticated
%alternating projection algorithm as opposed to the one step.

    % Initial setup of Problem 1
    A = randn(m,n); % A matrix with i.i.d stand. Normal coeff.
    x_star = randomvector(n,k); % Calls function, obtains random
 vector
    b = A * x_star;
    B = pinv(A); % The Moore-Penrose pseudo-inverse

    % Conditions specific to the iterations/initial vector
    x = zeros(n,1); % An nx1 column vector of zeroes
    i = 0; % Initial count to keep track of while loop iterations
    I = 500; % Number of iterations

    while (i < I)
        x_hyphen = x + B * (b - A * x); % Step 1
        x = sparsify(x_hyphen,k); % Step 2
        i = i + 1; % Step 3
    end

    x_tilde = x; % Recovered vector
    erroralt = (norm(x_tilde-x_star))/(norm(x_star)); % Error
end
```

Exercise 3.5

```
alt_proj(30,200,400)
```

```
ans =

   9.0837e-17
```

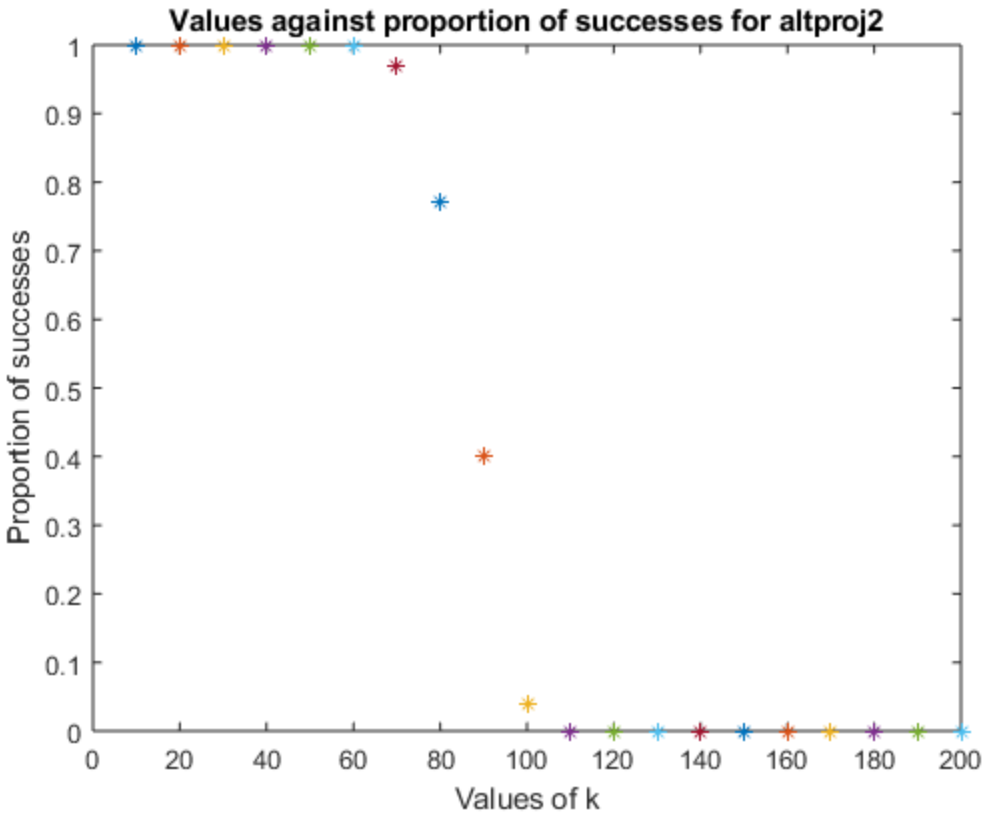
Exercise 3.6

```
type alt_proj2.m
alt_proj2(30,200,400)
```

```
function successfailure = alt_proj2(k,m,n)
%%ALT_PROJ2 Runs the code of the alt_proj algorithm and measures the
%size of the resulting error in approximating the initial solution.

    epsilon = 10^(-4); % The benchmark for the error
    iteratederror = alt_proj(k,m,n);
    if (iteratederror >= epsilon)
        successfailure = false; % False if error too large
    else
        successfailure = true; % True if error sufficiently small
end

ans =

  logical

   1
```

Exercise 3.7

```
for k = 10:10:200 % Looping over the values of k
    count(k) = 0; % Initial success count is nothing
    for j = 1:100
        count(k) = count(k) + alt_proj2(k,200,400); % Adding 1 or 0
    end
    plot(k,(count(k))/100,'*') % Plotting the individual points
    hold on
end
hold off
title('Values against proportion of successes for altproj2');
xlabel('Values of k');
ylabel('Proportion of successes');
```



For $m = 200$, $n = 400$, the recovery against the fixed epsilon seems to be perfect for $10 \leq k \leq 70$, i.e. sparse solutions are approximated with great accuracy. Between 70 and 110, the success rate is strictly decreasing, and from this point onwards it becomes 0. Hence, the algorithm works very well for smaller values of k and less so as we increase it towards 200.

Exercise 3.8

We modify the sparsify function to display the closest vector with at most k coefficients in the range $[0, 1]$. This is achieved by sorting the coefficients in size, and realizing that if any of the biggest k coefficients is bigger than 1, it must be set to 1; if it is negative, the best one can do is setting 0. Naturally, if the coefficients are in the desired range to begin with, we do not alter them in any way.

```
type sparsify2.m
x = randn(20,1)
sparsify2(x,7)
```

```
function x = sparsify2(x,k)
%%SPARSIFY2 Returns a vector with at most k zeros closest to the
%initial solution, under the costraint that the non-zero
%coefficients are in [0,1].

    [xsort ind] = sort(x, 'descend'); % Sort entries by size
    x(ind(k+1:end)) = 0; % Set the smallest entries to 0
for i = 1:k
```

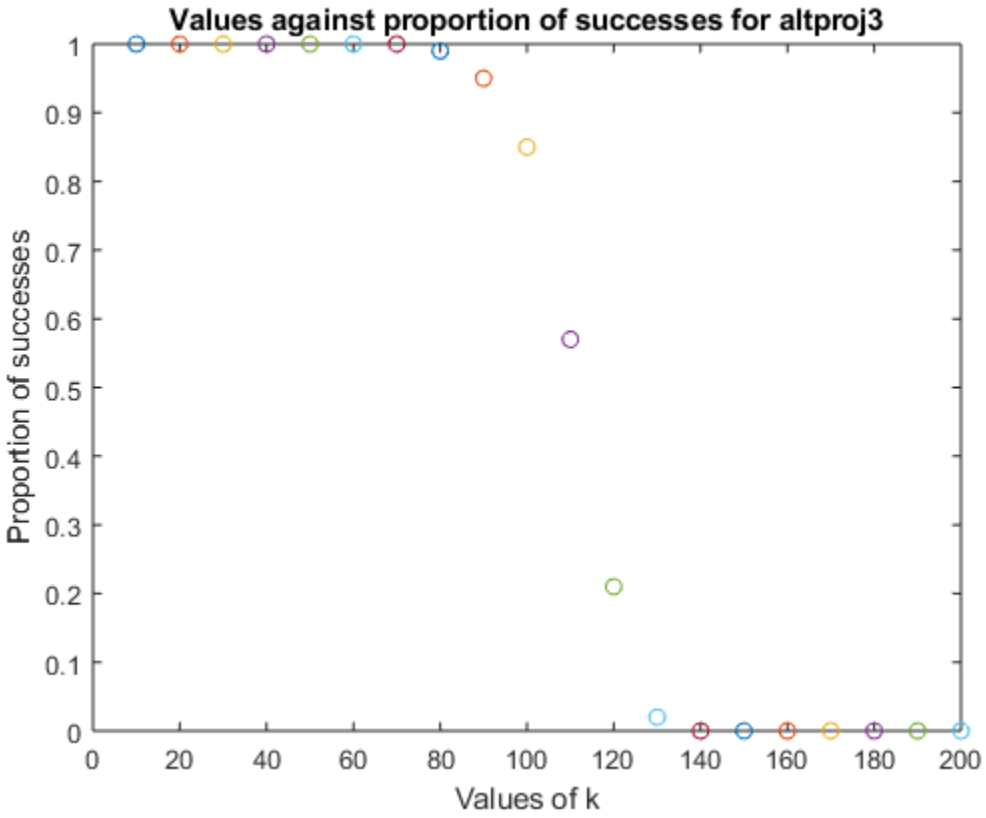
```
    if x(ind(i)) > 1
        x(ind(i)) = 1; % Setting entries bigger than 1 to 1.
    elseif x(ind(i)) < 0
        x(ind(i)) = 0; % Setting entries smaller than 0 to 0.
    else
        ; % Do nothing if entries are in [0,1]
    end
end
end

x =

    0.2971
    0.7623
    1.0227
    0.5151
   -0.8772
    0.8404
   -1.8453
    0.0984
    0.5367
   -0.0712
   -0.1263
    1.0268
    0.8265
    1.3805
   -0.1258
    1.1110
   -0.1452
    0.6767
   -0.4078
   -0.4127


ans =

         0
    0.7623
    1.0000
         0
         0
    0.8404
         0
         0
         0
         0
         0
    1.0000
    0.8265
    1.0000
         0
    1.0000
         0
         0
```

```
     0
     0
```

Exercise 3.9

```
type alt_proj3.m

for k = 10:10:200 % Looping over the values of k
    count(k) = 0; % Initial success count is nothing
    for j = 1:100
        count(k) = count(k) + alt_proj3(k,200,400); % Adding 1 or 0
    end
    plot(k,(count(k))/100, 'o') % Plotting the individual points
    hold on
end
hold off
title('Values against proportion of successes for altproj3');
xlabel('Values of k');
ylabel('Proportion of successes');


function successfailure = alt_proj3(k,m,n)
%%ALT_PROJ3 Runs the alternating projection algorithm with the
%improved version of the sparsify algorithm, and measures the
%resulting error against a fixed "benchmark" epsilon.

    % Initial setup of Problem 1
    A = randn(m,n);
    x_star = randomvector(n,k);
    b = A * x_star;
    B = pinv(A);

    % Conditions specific to the iterations/initial vector
    x = zeros(n,1); % An nx1 column vector of zeroes
    i = 0; % Initial count to keep track of while loop iterations
    I = 500; % Number of iterations

    epsilon = 10^(-4); % The benchmark for the error

    while (i < I)
        x_hyphen = x + B * (b - A * x); % Step 1
        x = sparsify2(x_hyphen,k); % Step 2
        i = i + 1; % Step 3
    end

    x_tilde = x;
    iteratederror = (norm(x_tilde-x_star))/(norm(x_star)); % Error

    if (iteratederror >= epsilon)
        successfailure = false; % A false logical if error too large
    else
        successfailure = true; % A true if error sufficiently small
```

end



Much like the previous algorithm, the success rate is perfect against the given value of epsilon for $10 \leq k \leq 70$, but now it is also near perfect for $k = 80$, an improvement from before. The drop from perfect success to zero success in this case is in the range $90 \leq k \leq 130$, with each success rate here greater than the corresponding one for the previous algorithm. In general, we can say that this plot is roughly a shift to the right from the original one, with the declining trend being less steep. The improvements are particularly significant for roughly $80 \leq k \leq 120$. For the largest values of k, the algorithm completely fails to improve from altproj2, as the success proportion is 0.

Published with MATLAB® R2017b