A Guide to the Sparse-Simplex MATLAB Package

Amir Beck and Yonina C. Eldar

December 2, 2012

This short note briefly describes the usage of the functions in the sparse-simplex package which are based on the paper

> Amir Beck and Yonina C. Eldar, "Sparsity Constrained Nonlinear Optimization: Optimality Conditions and Algorithms"

The package contains several m-files implementing method for solving the optimization problem

$$\text{(P)} \quad \min\{f(\mathbf{x}) : \|\mathbf{x}\|_0 \le s\}.$$

1 The IHT Method

A well-known solution method is the so-called iterative hard-thresholding (IHT) method whose recursive update formula is

$$\mathbf{x}^{k+1} = P_{C_s}\left(\mathbf{x}^k - \frac{1}{L}\nabla f(\mathbf{x}^k)\right).$$

The m-file implementing the IHT method is

```
IHT.m
```

For example, suppose that $f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$ where $\mathbf{A}$ and $\mathbf{b}$ are generated by the following commands:

```
randn('seed',314);
A=randn(10,10);
b=A*[1;-1;1;zeros(7,1)];
```

then obviously the optimal solution of (P) is $\mathbf{x}^* = (1,-1,1,0,0)^T$. The input for the `IHT` function consists of the function, its gradient, s, L, an initial guess $\mathbf{x}_0$ and the number of iterations. We can invoke 200 iterations of IHT with a randomly chosen initial vector by the commands

```
randn('seed',146);
x0=randn(10,1);
v=IHT(@(x)norm(A*x-b)^2,@(x)2*A'*(A*x-b),3,2*max(eig(A'*A))+0.1,x0,200)
```

Note that we have chosen L to be slightly larger than the Lipschitz constant of the gradient of the function. The output is the expected one

```
v =
    0.9999
   -1.0000
    1.0000
         0
         0
         0
         0
         0
         0
         0
```

Of course, the IHT method might converge to non-optimal points. For example, starting with another initial point we get convergence to a non-optimal point.

```
randn('seed',147);
y0=randn(10,1);
w=IHT(@(x)norm(A*x-b)^2,@(x)2*A'*(A*x-b),3,2*max(eig(A'*A))+0.1,y0,200)
w =
         0
         0
         0
         0
   -0.9510
   -0.5630
         0
   -0.1714
         0
         0
```

2 The Greedy Sparse-Simplex Method

To invoke the greedy sparse-simplex method on problem (P), two MATLAB functions should be constructed. The first one is the objective function and the second is a function that performs one-dimensional optimization with respect to each of the coordinates and outputs the index of the variable causing the largest decrease, its optimal value and the corresponding objective function value. For the least squares problem $\min \|\mathbf{Ax} - \mathbf{b}\|^2$, the MALTAB function

```
f_LI.m
```

is a simple implementation of the least squares term (with input $\mathbf{A}, \mathbf{b}, \mathbf{x}$), for example:

```
f_LI(A,b,y0)
ans =
  160.8177
```

The second required function is

```
g_LI.m
```

The arguments for this function are $(\mathbf{A}, \mathbf{b}, \mathbf{x}, S)$. S is a subset of indices on which the search is performed. For example:

```
out=g_LI(A,b,x0,1:10)
out =
   -0.0318   23.3425   10.0000
```

This result means that if we change x_{10} to have value -0.038, the new objective function value will be 23.3425. Indeed,

```
>> f_LI(A,b,x0)
ans =
   67.6136
>> x0_new=x0;
>> x0_new(10)=out(1);
>> f_LI(A,b,x0_new)
ans =
   23.3425
```

If we want to restrict the search for the indices set $\{1, 4, 7, 9\}$, then we can just write

```
>> g_LI(A,b,x0,[1,4,7,9])
ans =
   -1.2412   65.8406    9.0000
```

and in this case it is best to optimize with respect to to x_9. After having these two functions, we can now invoke the main function

```
greedy_sparse_simplex.m
```

To employ the greedy sparse-simplex with initial vector $\mathbf{x}_0$ and maximum of 200 iterations we run the command

```
>> X=greedy_sparse_simplex(@(x)f_LI(A,b,x),@(x,S)g_LI(A,b,x,S),3,200,x0);
iter=  1  fun_val = 13.61871   change = 1
iter=  2  fun_val = 1.54145   change = 0
iter=  3  fun_val = 1.24098   change = 0
iter=  4  fun_val = 0.84902   change = 0
iter=  5  fun_val = 0.66883   change = 0
iter=  6  fun_val = 0.49099   change = 0
```

```
    .                  .                  .
    .                  .                  .
    .                  .                  .
iter= 66  fun_val = 0.00000   change = 0
iter= 67  fun_val = 0.00000   change = 0
```

Note that the method did not require the maximum of 200 iterations, but satisfied a stopping criteria at the end of the 67th iteration. The function `greedy_sparse_simplex` is not restricted to the least squares function, but can be employed on any function, but for that the user must satisfy an objective function, and a one-dimensional minimizer such as `f_LI` and `g_LI`. Another example of an objective function that can be found in the package is the function

$$f_{\mathrm{QU}}(\mathbf{x}) = \sum_{i=1}^{m}((\mathbf{a}_i^T\mathbf{x})^2 - c_i)^2,$$

where $\mathbf{a}_i \in \mathbb{R}^n$. This function is implemented in

```
f_QU.m
```

Let us consider a 20×10 example in which the optimal solution is $\mathbf{x} = (0,0,1,0,2,0,-10,0,0,0)^T$.

```
randn('seed',314);
A=randn(20,10);
x_real=zeros(10,1);
x_real(3)=1;
x_real(5)=2;
x_real(7)=-10;
c=(A*x_real).^2;
```

Obviously,

```
>> f_QU(A,c,x_real)
ans =
     0
```

As before, we also have a one-dimensional minimization function

```
g_QU.m
```

This function also uses two other auxiliary functions that are responsible for solving simultaneously several scalar minimizations of one-dimensional quartic functions.

```
solve_cubic.m
solve_minimum_quartic.m
```

For example, the objective function value of the vector of all ones is

```
>> f_QU(A,c,ones(10,1))
ans =
  7.4168e+005
```

Invoking the function `g_QU` we obtain

```
>> out=g_QU(A,c,ones(10,1),1:10);
>> out(1)
ans =
   -9.4314
>> out(2)
ans =
  3.2078e+004
>> out(3)
ans =
     7
```

This means that if we change the value of the seventh variable to be -9.4314, then the new objective function will be `3.2078e+4` (which is btw, more than ten times lower than the value of the function on the vector of all ones). Invoking the greedy sparse-simplex method with initial vector $(1, 1, 1, 0, 0, 0, 0, 0, 0, 0)^T$ results with the correct solution:

```
>>[X,fun_val]=greedy_sparse_simplex(@(x)f_QU(A,c,x),@(x,S)g_QU(A,c,x,S),...
p,500,[1;1;1;ones(7,1)]);
iter=  1  fun_val = 32078.43241   change = 0
iter=  2  fun_val = 20204.87905   change = 0
iter=  3  fun_val = 15330.39511   change = 0
iter=  4  fun_val = 10362.79145   change = 0
iter=  5  fun_val = 9518.23586   change = 0
iter=  6  fun_val = 8492.85147   change = 0
iter=  7  fun_val = 7577.44331   change = 0
    .                  .             .
    .                  .             .
    .                  .             .
iter=497  fun_val = 0.00008   change = 0
iter=498  fun_val = 0.00008   change = 0
iter=499  fun_val = 0.00008   change = 0
iter=500  fun_val = 0.00007   change = 0
>> X(:,end)
ans =
    0.0001
    0.0000
    1.0000
    0.0001
    1.9998
   -0.0001
  -10.0002
    0.0002
```

```
   -0.0001
    0.0001
```

3 The Partial Sparse-Simplex Method

The partial sparse-simplex method is implemented in the m-file

`partial_sparse_simplex.m`

The input arguments are the same as the one of `greedy_sparse_simplex` expect for one additional input argument which is the gradient function. For the function f_{QU}, the gradient is implemented in the m-file

`gradient_QU.m`

For example, invoking the partial sparse-simplex method with the same setting as the last example (of the greedy sparse-simplex method) also results with the correct solution:

```
>>[X,fun_val]=partial_sparse_simplex(@(x)f_QU(A,c,x),@(x)gradient_QU(A,c,x),...
@(x,S)g_QU(A,c,x,S),p,500,[1;1;1;ones(7,1)]);
iter=  1  fun_val = 32078.43241 change = 0
iter=  2  fun_val = 20204.87905 change = 0
iter=  3  fun_val = 15330.39511 change = 0
iter=  4  fun_val = 10362.79145 change = 0
iter=  5  fun_val = 9518.23586 change = 0
iter=  6  fun_val = 8492.85147 change = 0
    .          .              .
    .          .              .
    .          .              .
iter=498  fun_val = 0.00008 change = 0
iter=499  fun_val = 0.00008 change = 0
iter=500  fun_val = 0.00007 change = 0
>> X(:,end)
ans =
    0.0001
    0.0000
    1.0000
    0.0001
    1.9998
   -0.0001
  -10.0002
    0.0002
   -0.0001
    0.0001
```

As a last example, let us invoke the partial sparse-simplex problem on linear least squares problem ($f = f_{\rm LI}$) with an initial vector whose support is completely different than the true support (last 10 indices instead of the first 10). The algorithm finds the correct solution.

```
>>randn('seed',314);
>>A=randn(200,100);
>>x=[ones(10,1);zeros(90,1)];
>>b=A*x;
>>x_initial=[zeros(90,1);randn(10,1)];
>>[X,fun_val]=partial_sparse_simplex(@(x)f_LI(A,b,x),@(x)2*A'*(A*x-b),...
    @(x,S)g_LI(A,b,x,S),10,500,x_initial);
iter=  1  fun_val = 2531.91899 change = 1
iter=  2  fun_val = 2148.50550 change = 1
iter=  3  fun_val = 1871.38289 change = 1
iter=  4  fun_val = 1645.16590 change = 1
iter=  5  fun_val = 1410.71129 change = 0
iter=  6  fun_val = 1251.96140 change = 1
iter=  7  fun_val = 1007.72918 change = 1
iter=  8  fun_val = 820.43212 change = 1
iter=  9  fun_val = 620.07693 change = 1
iter= 10  fun_val = 461.79364 change = 0
iter= 11  fun_val = 331.87055 change = 1
iter= 12  fun_val = 248.76544 change = 0
iter= 13  fun_val = 89.85229 change = 1
iter= 14  fun_val = 56.83251 change = 0
iter= 15  fun_val = 27.64811 change = 0
iter= 16  fun_val = 19.79799 change = 0
    .             .              .
    .             .              .
    .             .              .
iter= 57  fun_val = 0.00000 change = 0
iter= 58  fun_val = 0.00000 change = 0
iter= 59  fun_val = 0.00000 change = 0
```