User Menu for `NHTP`: Newton Hard-Thresholding Pursuit

Shenglong Zhou[†], Naihua Xiu[‡], Houduo Qi [§]

Abstract

This menu aims at providing instructions for users to use a Matlab-based package `NHTP`, an abbreviation for Newton Hard-Thresholding Pursuit. This package was created based on the algorithm proposed in the paper "*Global and Quadratic Convergence of Newton Hard-Thresholding Pursuit*"[1]. It will instruct users to set up the package and solve sparsity constrained optimization problems including compressed sensing (CS), sparse logistic regression (SLR) and some other general examples.

Keywords: Newton method, Hard-Thresholding Pursuit, sparsity constrained optimization

Mathematical Subject Classification: 90C26, 90C30, 90C90

1 Introduction

This package which can be downloaded at *https://github.com/ShenglongZhou* is programmed by Matlab language and aims at solving the sparsity constrained optimization (SCO) problems:

$$\min \ f(\mathbf{x}), \qquad \text{s.t.} \ \|\mathbf{x}\|_0 \le s, \tag{1.1}$$

where $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ is a twice continuously differentiable and lower bounded function, $s < n$ is a positive integer. $\|\mathbf{x}\|_0$ is the so-called ℓ_0-norm of $\mathbf{x}$, which refers to the number of nonzero elements in the vector $\mathbf{x}$. Several typical examples are

i) **Compressed sensing**: $f(\mathbf{x}) = \|A\mathbf{x} - \mathbf{b}\|^2$ where A is an $m \times n$ sensing matrix, $\mathbf{b} \in \mathbb{R}^n$ is the observation and $\|\cdot\|$ is the Euclidean norm in $\mathbb{R}^n$.

ii) **Sparse logistic regression (SLR)** with ℓ_2 norm regularization $f(\mathbf{x}) = \ell(\mathbf{x}) + \mu\|\mathbf{x}\|_2^2$ and

$$\ell(\mathbf{x}) := \frac{1}{m}\sum_{i=1}^{m}\left[\ln(1 + e^{\langle \mathbf{a}_i, \mathbf{x}\rangle}) - b_i\langle \mathbf{a}_i, \mathbf{x}\rangle\right],$$

where $\mathbf{a}_i \in \mathbb{R}^n, i = 1, \ldots, m$ are m samples, $b_i \in \{0, 1\}, i = 1, \ldots, m$ are responses/labels, and $\mu \ge 0$ (e.g. $\mu = m^{-1}10^{-6}$). We denote $A^\top := [\mathbf{a}_1 \ \cdots \mathbf{a}_m]$ and $\mathbf{b} := [b_1 \ \cdots \ b_m]^\top$.

iii) **Other general example:** $f(\mathbf{x})$ should be twice continuously differentiable and lower bounded. For instance, given that $Q \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$, consider

$$f(\mathbf{x}) = \mathbf{x}^\top Q\mathbf{x} + \mathbf{b}^\top\mathbf{x} - \sqrt{\|\mathbf{x}\|^2 + 1}.$$

[†]School of Mathematics, University of Southampton, Southampton SO17 1BJ, United Kingdom.(shenglong.zhou@soton.ac.uk

[‡]Department of Applied Mathematics Beijing Jiaotong University, Beijing 100044, P. R. China. nhxiu@bjtu.edu.cn

[§] School of Mathematics, University of Southampton, Southampton SO17 1BJ, United Kingdom.

2 The Main Function

The citations of the main function `NHTP.m` of package `NHTP` are:

(2.1)	`Out = NHTP(n, s, data, probname, pars)`
(2.2)	`Out = NHTP(n, s, data, probname, [])`
(2.3)	`Out = NHTP(n, s, data, probname)`
(2.4)	`Out = NHTP(n, s, data)`

For solving CS and SLR problems, *we strongly recommend to use the (2.1), (2.2) or (2.3) rather than (2.4). For other general examples, all above four citations are proper.* Corresponding inputs and output are described as follows.

Table 1: Inputs and outputs of the main function `NHTP.m`.

Inputs:		
	`n` :	Dimension of the solution $\mathbf{x}$, (required)
	`s` :	Sparsity level of $\mathbf{x}$, an integer between 1 and $n-1$, (required)
	`data` :	Input functions include the objective $f(\mathbf{x})$, its gradient and Hessian
		For CS and SLR, data also could be input in the form:
		`data.A`: $m \times n$ order measurement matrices
		`data.At`: transpose of `data.A`, i.e., `data.At=data.A'`
		`data.b`: $m \times 1$ order observation vector
	`probname`:	Name of problem, should be one of {'CS','SLR','SCO'}
	`pars`:	Parameters are all OPTIONAL
		`pars.x0`: Starting point of $\mathbf{x}$, `pars.x0=zeros(n,1)` (default)
		`pars.eta`: A positive scalar. A default one is given related to inputs
		`pars.IterOn`: Results will be shown if `pars.IterOn=1` (default)
		Results won't be shown if `pars.IterOn=0`
		`pars.Draw`: A graph will be drawn if `pars.Draw=1` (default)
		No graph will be drawn if `pars.Draw=0`
Outputs:		
	`Out.sol`:	The sparse solution $\mathbf{x}$
	`Out.sparsity`:	Sparsity level of `Out.sol`
	`Out.normgrad`:	L2 norm of the gradient at `Out.sol`
	`Out.error`:	Error used to terminate this solver
	`Out.time`:	CPU time
	`Out.iter`:	Number of iterations
	`Out.grad`:	Gradient at `Out.sol`
	`Out.obj`:	Objective function value at `Out.sol`
	`Out.eta`:	Final `eta`

3 Package Implementation

To use this solver, one needs to set up it first with opening '`startup.m`' file by Matlab and then running it to add the path. The file contains codes:

```
clc; close all; clear all; warning off
addpath(genpath(pwd));
```

Now we are ready to solve each examples mentioned in Introduction.

3.1 Solving CS Problems

Cleary, based on the citation (2.1) of `NHTP`, the important input is the `data` whose structure has three required elements: `data.A`, `data.At` and `data.b`. There are two ways to input `data`: using your data or our data.

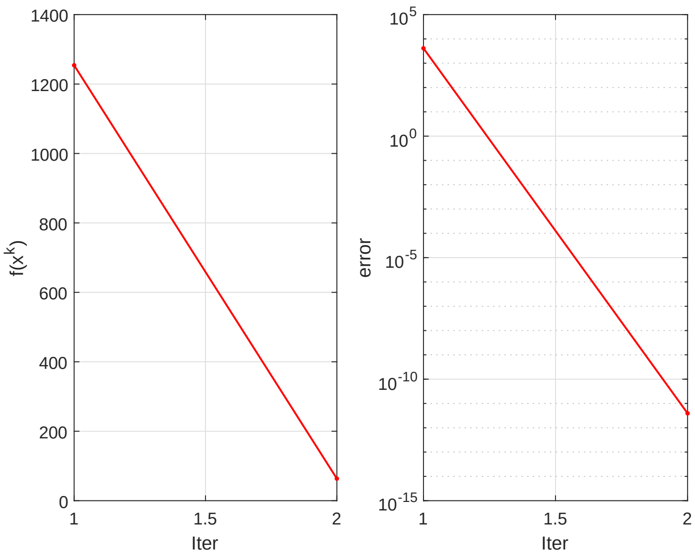
I) The first way is to put `data` as follows. Type (or copy) the following codes in a new command window (or alternatively you can simply open `demon_CS.m` and run it):

```
clc; clear; close all;                      % line 1
probname ='CS';                             % line 2
n        = 256;                             % line 3
m        = ceil(n/4);                       % line 4
s        = ceil(0.05*n);                    % line 5
data.A   = randn(m,n)/sqrt(m) ;             % line 6
data.b   = rand(m,1);                       % line 7
data.At  = data.A';                         % line 8
pars.Draw= 1;                               % line 9
out      = NHTP(n,s,data,probname,pars)     % line 10
```

The second line defines the name of the problem that will be solved by `NHTP`. In line 10, `pars` could be empty, namely, `pars=[]`. In this way, users can solve CS problems with their own interested data. For example, we use `NHTP` to solve the `DriverFaces` data. Type or copy the following codes in a new command window (Or open file `demon_CS_RealData.m`).

```
clc; clear; close all;                                   % line 1
load 'DrivFace.mat';                                     % line 2
load 'nlab.mat';    % load 'nlab.mat';                   % line 3
[m,n]    = size(A);                                      % line 4
s        = 10;                                           % line 5
data.A   = A;                                            % line 6
data.b   = y;                                            % line 7
data.At  = data.A';                                      % line 8
pars.Draw= 1;                                            % line 9
out      = NHTP(n,s,data,'CS',pars)                      % line 10
fprintf('\nSample size:    m=%4d,n=%4d\n', m,n);         % line 11
fprintf('CPU time:         %6.3fsec\n',  out.time);      % line 12
fprintf('Objective value:  %5.3e\n\n', out.obj);         % line 13
```

The outputs have the form as below which corresponds to outputs in Table 1 and contains a figure where definitions of ‘`error`’ can be found in [1].




```
sparsity:  10
normgrad:  558.3507
   error:  3.9233e-12
    time:  0.0387
    iter:  2
    grad:  [6400x1 double]
     sol:  [6400x1 double]
     obj:  63.5703
     eta:  3.3305e-05
```

II) The second way is to use the `data` generated by `compressed_sensing_data.m` which can be found in this package. Similarly, type or copy the following codes in a new command window (or alternatively you can simply open `demon_CS.m` and run it):

```
clc; clear; close all;                                        % line 1
probname = 'CS';                                              % line 2
n        = 256;                                               % line 3
m        = ceil(n/4);                                         % line 4
s        = ceil(0.05*n);                                      % line 5
data     = compressed_sensing_data('GaussianMat', m,n,s,0);   % line 6
pars.Draw= 1;                                                 % line 7
out      = NHTP(n,s,data,probname,pars)                       % line 8
```

In line 6, `compressed_sensing_data.m` generated Gaussian type matrix. This data generation function handle also provides two other types of matrices: ‘`PartialDCTMat`’ (Partial DCT matrix) and ‘`ToeplitzCorMat`’ (Toeplitz Correlated matrix). To see `NHTP` solves CS problems under these measurement matrices, just replace the code in line 6. For example,

```
data = compressed_sensing_data('PartialDCTMat', m,n,s,0);
```

3.2 Solving SLR Problems

Similarly, there are two ways to input `data`: using your data or our data.

I) The first way is to put `data` as follows. Type or copy the following codes in a new command window (or alternatively you can simply open `demon_SLR.m` and run it):

```
clc; clear; close all;                                                   % line 1
probname ='SLR';                                                         % line 2
n        = 1000;                                                         % line 3
m        = ceil(n/5);                                                    % line 4
s        = ceil(0.05*n);                                                 % line 5
I0       = randperm(n);  I  = I0(1:s);                                   % line 6
x        = zeros(n,1); x(I) = randn(s,1);                                % line 7
data.A   = randn(m,n);                                                   % line 8
data.At  = data.A';                                                      % line 9
q        = 1./(1+exp(-data.A(:,I)*x(I)));                                % line 10
data.b   = zeros(m,1);                                                   % line 11
for i    = 1:m; data.b(i) = randsrc(1,1,[0 1;1-q(i) q(i)]); end          % line 12
pars.Draw= 1;                                                            % line 13
out      = NHTP(n,s,data,probname,pars)                                  % line 14
```

The second line defines the name of the problem that will be solved by `NHTP`. Lines 6-12 generate the data `data.A`, `data.At` and `data.b`. In this way, users can solve LSR problems with their own interested data. For example, we use `NHTP` to solve the `newsgroup` data. Type or copy the following codes in a new command window (Or open file `demon_SLR_RealData.m`).

```
clc; clear; close all;                                   % line 1
prob     = 'newsgroup';     %'colon-cancer'              % line 2
measure  = load(strcat(prob,'.mat'));                    % line 3
label    = load(strcat(prob,'_label.mat'));              % line 4
label.b(label.b==-1)= 0;                                 % line 5
[m,n]    = size(measure.A);                              % line 6
normtype = 1;                                            % line 7
if m    >= 1000; normtype=2; end                         % line 8
data.A   = normalization(measure.A,normtype);            % line 9
data.At  = data.A';                                      % line 10
data.b   = label.b;                                      % line 11
s        = ceil(0.1*m);                                  % line 12
pars.Draw= 1;                                            % line 13
out      = NHTP(n,s,data,'SLR',pars);                    % line 14
```

II) The second way is to use the `data` generated by `logistic_random_data.m` which can be found in this package. Similarly, type the following codes in a new command window (or alternatively you can simply open `demon_SLR.m` and run it):

```
clc; clear; close all;                                        % line 1
probname = 'SLR';                                             % line 2
n        = 1000;                                              % line 3
m        = ceil(n/5);                                         % line 4
s        = ceil(0.05*n);                                      % line 5
data     = logistic_random_data('Correlated',m,n,s,0.5);      % line 6
pars.Draw= 1;                                                 % line 7
out      = NHTP(n,s,data,probname,pars)                       % line 8
```

In line 6, `logistic_random_data.m` generated Gaussian type matrix with correlated data. This data generation function handle also provides Gaussian type matrix with two other types of data: '`Indipendent`' and '`Weakly-Indipendent`'. To see `NHTP` solves SLR problems under these measurement matrices, just replace the code in line 6. For example,

```
data = compressed_sensing_data('Indipendent',m,n,s,0.5);
```

3.3 Solving General SCO Problems

To solve a general SCO problem, we first need write a function file as the input `data`. As described in Table 1, for general problems, the input `data` should contain the objective function $f(\mathbf{x})$, its gradient and Hessian. Let use the problem in Introduction as an example, namely,

$$f(\mathbf{x}) = \mathbf{x}^\top Q\mathbf{x} + \mathbf{b}^\top \mathbf{x} - \sqrt{\|\mathbf{x}\|^2+1} \tag{3.1}$$

with

$$n=2, \qquad s=1, \qquad Q=\begin{bmatrix} 6 & 5 \\ 5 & 8 \end{bmatrix}, \qquad \mathbf{b}=\begin{bmatrix} 1 \\ 9 \end{bmatrix}.$$

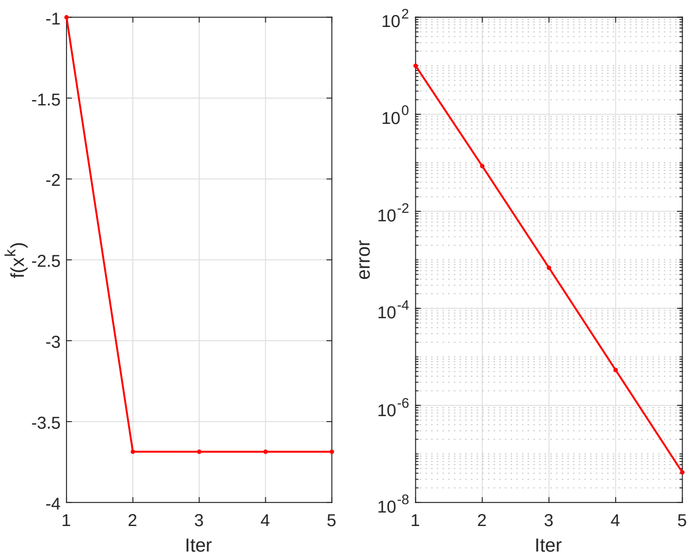
The way to define the objective function, gradient and Hessian should share the following form:

```
function  out  = simple_ex4_func(x,fgH)             % line 1
 a = sqrt(sum(x.*x)+1);                             % line 2
 switch fgH                                         % line 3
    case 'obj'   % objective function               % line 4
          out  = x'*[6 5;5 8]*x+[1 9]*x-a;          % line 5
    case 'grad'  % gradient                         % line 6
          out  = 2*[6 5;5 8]*x+[1; 9]-x./a;         % line 7
    case 'hess'  % Hessian matrix                   % line 8
          out  = 2*[6 5;5 8]+(x*x'-a*eye(2))/a^3;   % line 9
 end                                                % line 10
end                                                 % line 11
```

Here we name the function `simple_ex4_func` and save it as `simple_ex4_func.m`. Then put this file in a proper path so that `NHTP.m` could call it. For convenience, put it with `NHTP.m` in a same folder. After doing this, then type the following code in a new command window (or you can simply open `demon_General_SCO.m` and run it):

```
clc; clear; close all;                            % line 1
probname = 'SCO';                                 % line 2
n        = 2;                                     % line 3
s        = 1;                                     % line 4
data     = @(var,flag)simple_ex4_func(var,flag);  % line 5
pars.Draw= 1;                                     % line 6
out      = NHTP(n,s,data,probname,pars)           % line 7
```

Here we use `probname = 'SCO'` to call `NHTP.m` solve this general example. Unlike the `data` in CS or SLR problem where `data` is a structure input, `data` here is a function handle which should be defined by using `@(var,flag)` as in line 5. The outputs have the form as below which corresponds to outputs in Table 1 and contains a figure.




```
sparsity:   1
normgrad:   4.944
   error:   4.1752e-08
    time:   0.0030
    iter:   5
    grad:   [2x1 double]
     sol:   [2x1 double]
     obj:   -3.6864
     eta:   0.0738
```

4 Useful Notes

For the sake of the efficiency of this package, there are some useful suggestions:

a) *For CS problems:* the input matrix A is better to be normalized such that its each column has a unit length. This can be done by using

```
A = normalization(A,3);
```

where the function `normalization.m` is provided by this package.

b) *For SLR problems:*

b.1) the data label $\mathbf{b}$ should have elements from $\{0, 1\}$. So for those data with labels from $\{-1, 1\}$, replace the label -1 by 0;

b.2) when the sample size $A \in \mathbb{R}^{m\times n}$ satisfies $m \leq n \leq 1000$, sample-wise normalization has been conducted so that each sample has mean zero and variance one, and then feature-wise normalization has been conducted so that each feature has mean zero and variance one. This can be done by using `A = normalization(A,1);`.

b.3) when the sample size $A \in \mathbb{R}^{m\times n}$ satisfies $n > 1000$, it is suggested to be feature(column)-wisely scaled to $[-1, 1]$. This can be done by using `A = normalization(A,2);`.

c) *Choice of* ***`pars.eta`****:* this is an important parameter of this package. Basically, a larger value of `pars.eta` might lead to a better solution but might take much longer computational time. By contrast, a smaller value of `pars.eta` could make the solver to obtain a solution much faster, but solution may be a local one. We presented a default `pars.eta` which is related to the input example. This choice might be proper for those examples included in this package. For other examples that are not solved here, adjusting this `pars.eta` would give a better result. The general rule of adjusting it is as follows. *If $f(\mathbf{x})$ has too large value or $\nabla f(\mathbf{x})$ has too large elements (e.g., being greater that 10), it suggests to use a smaller* ***`pars.eta`*** *(e.g., smaller than 0.1).* For example, in (3.1), $\nabla f([1; 1]) = [22.4226; 34.4226]$, we could just use `pars.eta` $= 0.1$ which generates the optimal solution very fast. Again, we need emphasize that this rule might not be true for all examples.

References

[1] S. Zhou, N. Xiu and H. Qi, Global and Quadratic Convergence of Newton Hard-Thresholding Pursuit, available at *https://arxiv.org/abs/1901.02763*, 2018.