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

Shenglong Zhou[†], Naihua Xiu[‡] and 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 titled "*Global and Quadratic Convergence of Newton Hard-Thresholding Pursuit*"[1]. It instructs users to set up the package and solve sparsity constrained optimization problems including compressed sensing, sparse logistic regression and some other general examples.

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

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

1 Introduction

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

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

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

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

ii) *Sparse logistic regression* (SLR): $f(x) = \ell(x) + \mu\|x\|_2^2$ and

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

where $a_i \in \mathbb{R}^n$ and $b_i \in \{0,1\}, i = 1, \ldots, m$ are m samples and responses/labels respevtively, and $\mu \geq 0$ (e.g. $\mu = m^{-1}10^{-6}$). We denote $A^\top := [a_1 \ \cdots a_m]$ and $b := [b_1 \ \cdots \ b_m]^\top$.

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

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

[†]School of Mathematics, University of Southampton, Southampton SO17 1BJ, United Kingdom.
[‡]Department of Applied Mathematics Beijing Jiaotong University, Beijing 100044, P. R. China
[§]shenglong.zhou@soton.ac.uk, nhxiu@bjtu.edu.cn, h.qi@soton.ac.uk

2 The Main Function

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

(2.1) `Out = NHTP(n, s, func, pars)`

(2.2) `Out = NHTP(n, s, func)`

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 x, (required)	
	`s` :	Sparsity level of x, an integer between 1 and $n-1$, (required)	
	`func`:	function handle, define the function value, gradient, Hessian of $f(x)$	
	`pars`:	Parameters are all OPTIONAL	
		`pars.x0`:	Starting point of x, `pars.x0=zeros(n,1)` (default)
		`pars.eta`:	A positive scalar. A default one is given related to inputs
		`pars.display`:	Display results in each step if `pars.display=1` (default)
			Results won't be displayed if `pars.display=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 x	
	`Out.sparsity`:	Sparsity level of `Out.sol`	
	`Out.normgrad`:	Euclidean 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.obj`:	Objective function value at `Out.sol`	

2.1 Construction of `func`

Before implementing `NHTP`, we need create the function handle `func`, which has the form as

```
[out1,out2]= func(x,fgh,T1,T2);
```

For inputs, `x` is the variable in $\mathbb{R}^n$. `fgh` has two options: `'ObjGrad'` and `'Hess'`. `T1` and `T2` are two subsets of $\{1,2,\cdots,n\}$. For outputs,

`fgh`	`out1`	`out2`
`'ObjGrad'`	$f(x)$	$\nabla f(x)$
`'Hess'`	$(\nabla^2 f(x))_{T1,T1}$	$(\nabla^2 f(x))_{T1,T2}$

where $(\nabla^2 f(x))_{T1,T2}$ is the submatrix containing `T1` rows and `T1` columns of Hessian matrix $\nabla^2 f(x)$. Now we use two examples to illustrate the construction of `func`.

- For CS problem: $f(x) = \|Ax-b\|^2$, we first create a Matlab function m-file (which also can be found in this package through the path `NHTP-->examples-->compressed_sensing`) by typing or copying following codes into a blank Matlab script file:

Table 3: M-file `compressed_sensing.m` to define CS problems.

```
function [out1,out2] = compressed_sensing(x,fgh,T1,T2,data)   % line 1
    Tx = find(x);                                              % line 2
    Ax = data.A(:,Tx)*x(Tx);                                   % line 3
    switch fgh                                                 % line 4
    case 'ObjGrad'                                             % line 5
        Axb  = Ax-data.b;                                      % line 6
        out1 = sum(Axb.*Axb)/2;                                % line 7
        if  nargout>1                                          % line 8
        out2 = data.At*Axb;                                    % line 9
        end                                                    % line 10
    case 'Hess'                                                % line 11
        out1 = data.At(T1,:)*data.A(:,T1);                     % line 12
        if nargout>1                                           % line 13
        out2 = data.At(T1,:)*data.A(:,T2);                     % line 14
        end                                                    % line 15
    end                                                        % line 16
end                                                            % line 17
```

Here, the important input `data` is a structure that has three basic elements:

(`data.A`, `data.At`, `data.b`),

where `data.A` $= A$, `data.At` $= A^\top$, `data.b` $= b$. If `fgh='ObjGrad'`, `out1`$= f(x)$ by line 7 or `out2`$= \nabla f(x) = A^\top(Ax-b)$ by line 9. If `fgh='Hess'`, then `out1`$= A_{T1}^\top A_{T1}$ by line 12 and `out2`$= A_{T1}^\top A_{T2}$ by line 14. Then to construct `func`, one could just use

```
func = @(x,fgh,T1,T2)compressed_sensing(x,fgh,T1,T2,data);
```

- For a general problem: $f(x) = x^\top Qx + b^\top x - \sqrt{\|x\|^2+1}$, where $Q = [6\ 5; 5\ 8]$ and $b = [1; 9]$, we first create a Matlab function m-file (which also can be found in this package through the path `NHTP-->examples-->general_sco`) by typing or copying following codes into a blank Matlab script file:

Table 4: M-file `simple_ex4_func.m` to define a general problem.

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

The above function file defines the objective function value, gradient and Hessian matrix of f. However to construct `func`, we need define sunmatrix of Hessian. So we also need following Matlab function m-file (which also can be found in the package through the path `NTTP-->examples-->general_sco`).

Table 5: M-file `general_example.m` to define a general problem with sunmatrix of Hessian.

```
function [out1,out2] = general_example(x,fgh,T1,T2,data)   % line 1
    switch fgh                                              % line 2
    case 'ObjGrad'                                          % line 3
          out1 = data(x,'obj');                             % line 4
          if nargout>1                                      % line 5
          out2 = data(x,'grad');                            % line 6
          end                                               % line 7
    case  'Hess'                                            % line 8
          H    = data(x,'hess');                            % line 9
          out1 = H(T1,T1);                                  % line 10
          if nargout>1                                      % line 11
          out2 = H(T1,T2);                                  % line 12
          end                                               % line 13
          clear  H;                                         % line 14
    end                                                     % line 15
end                                                         % line 16
```

Then to construct `func`, one could just use

```
data = @(x,fgh)simple_ex4_func(x,fgh);
func = @(x,fgh,T1,T2)general_example(x,fgh,T1,T2,data);
```

Alternatively, one could construct `func=exfunc` by following codes directly.

Table 6: M-file `exfunc.m` to define a general problem directly.

```
function [out1,out2] = exfunc(x,fgh,T1,T2)                  % line 1
a = sqrt(sum(x.*x)+1);                                      % line 2
    switch fgh                                              % line 3
    case 'ObjGrad'                                          % line 4
          out1 = x'*[6 5;5 8]*x+[1 9]*x-a;                  % line 5
          if nargout>1                                      % line 6
          out2 = 2*[6 5;5 8]*x+[1; 9]-x./a;                 % line 7
          end                                               % line 8
    case  'Hess'                                            % line 9
          H    = 2*[6 5;5 8]+(x*x'-a*eye(2))/a^3;           % line 10
          out1 = H(T1,T1);                                  % line 11
          if nargout>1                                      % line 12
          out2 = H(T1,T2);                                  % line 13
          end                                               % line 14
          clear  H;                                         % line 15
    end                                                     % line 16
end                                                         % line 17
```

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 to the current directory. The file contains codes:

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

To solve a problem, we need define its functions: objective function, gradient and Hessian. This can be done by `func`. While `func` might involve inputting data, such as A and b in CS problem. This data is stored into `data`. Therefore, the whole procedure of implementing this package is summarized as follow: (1) generate or collect `data`, then (2) define the function `func` and next (3) call `NHTP` to solve this problem. Now we solve each examples mentioned in Introduction.

3.1 Solving CS Problems

We have already created CS function `func` through Table 3. Then we generate `data`, namely A and b. As we mentioned before, `data` is a structure that has three basic 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
n        = 256;                                                   % line 2
m        = ceil(n/4);                                             % line 3
s        = ceil(0.01*n);                                          % line 4
data.A   = randn(m,n)/sqrt(m) ;                                   % line 5
data.b   = randn(m,1)/sqrt(m);                                    % line 6
data.At  = data.A';                                               % line 7
pars.eta = 1;                                                     % line 8
func     = @(x,fgh,T1,T2)compressed_sensing(x,fgh,T1,T2,data);    % line 9
out      = NHTP(n,s,func,pars)                                    % line 10
```

Lines 5-7 generate the `data` containing A and b. The function `func` in line 9 is defined by `compressed_sensing` created in Table 3. Last line calls the solver to solve this problem. For another CS problem with real data `DriverFaces`, type or copy the following codes in a new command window (Or open file `demon_CS_RealData.m` and run it).

```
clc; clear; close all;                                            % line 1
load 'DrivFace.mat';                                              % line 2
load 'nlab.mat';    % load 'identity.mat';                        % line 3
[m,n]    = size(A);                                               % line 4
s        = 10;                                                    % line 5
data.A   = A/sqrt(m);                                             % line 6
data.b   = y/sqrt(m);                                             % line 7
data.At  = data.A';                                               % line 8
func     = @(x,fgh,T1,T2)compressed_sensing(x,fgh,T1,T2,data);    % line 9
out      = NHTP(n,s,func)                                         % line 10
```

II) The second way is to use the `data` generated by `compressed_sensing_data.m` which can be found in the package through the path `NTTP-->examples-->compressed_sensing`. 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
n        = 256;                                                   % line 2
m        = ceil(n/4);                                             % line 3
s        = ceil(0.01*n);                                          % line 4
data     = compressed_sensing_data('GaussianMat',m,n,s,0);        % line 5
func     = @(x,fgh,T1,T2)compressed_sensing(x,fgh,T1,T2,data);    % line 6
out      = NHTP(n,s,func)                                         % line 7
```

In line 5, `compressed_sensing_data.m` generates 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 5. For example,

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

3.2 Solving SLR Problems

Similarly, to solve SLR problems, we need `data` and `func`. Again, `func` can be create by

```
func = @(x,fgh,T1,T2)logistic_regression(x,fgh,T1,T2,data);
```

where the function handle `logistic_regression` (or the Matlab m-file `logistic_regression.m`) can be found in the package through the path `NTTP-->examples-->logistic_regression`. Same to CS problem, `data` is a structure that has three basic elements:

(`data.A`, `data.At`, `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_SLR.m` and run it):

```
clc; clear; close all;                                            % line 1
n        = 1000;                                                  % line 2
m        = ceil(n/5);                                             % line 3
s        = ceil(0.05*n);                                          % line 4
I0       = randperm(n);  I  = I0(1:s);                            % line 5
x        = zeros(n,1); x(I) = randn(s,1);                         % line 6
data.A   = randn(m,n);                                            % line 7
data.At  = data.A';                                               % line 8
q        = 1./(1+exp(-data.A(:,I)*x(I)));                         % line 9
data.b   = zeros(m,1);                                            % line 10
for i    = 1:m;data.b(i)=randsrc(1,1,[0 1;1-q(i) q(i)]);end       % line 11
func     = @(x,fgh,T1,T2)logistic_regression(x,fgh,T1,T2,data);   % line 12
out      = NHTP(n,s,func)                                         % line 13
```

Lines 7-11 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 SLR with the `newsgroup` data, a real-world 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
data.A   = normalization(measure.A,1+(m>=1000));                % line 8
data.At  = data.A';                                             % line 9
data.b   = label.b;                                             % line 10
s        = ceil(0.1*m);                                         % line 11
pars.eta = 5;                                                   % line 12
func     = @(x,fgh,T1,T2)logistic_regression(x,fgh,T1,T2,data); % line 13
out      = NHTP(n,s,func,pars)                                  % line 14
```

Lines 2-5 load the data A and b. They are then stored into `data` by lines 8-10. After data collection, we construct the function handle `func` to define the SLR problem in line 13 and solve it by calling `NHTP` in last line.

II) The second way is to use the `data` generated by `logistic_random_data.m` which can be found in the package through the path `NTTP-->examples-->logistic_regression`. 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
n        = 1000;                                                % line 2
m        = ceil(n/5);                                           % line 3
s        = ceil(0.05*n);                                        % line 4
pars.eta = 5;                                                   % line 5
data     = logistic_random_data('Correlated',m,n,s,0.5);        % line 6
func     = @(x,fgh,T1,T2)logistic_regression(x,fgh,T1,T2,data); % line 7
out      = NHTP(n,s,func,pars)                                  % line 8
```

In line 6, `logistic_random_data.m` generates Gaussian sample with correlated data. This data generation m-file 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, for example

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

with

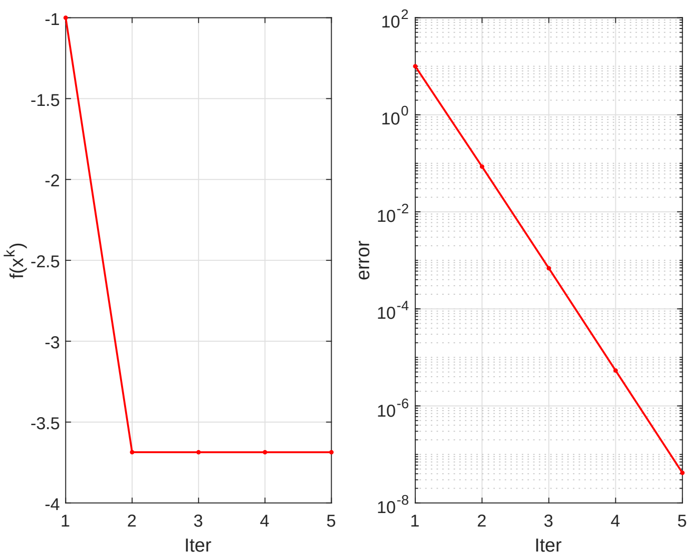
$$n = 2, \qquad s = 1, \qquad Q = \begin{bmatrix} 6 & 5 \\ 5 & 8 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ 9 \end{bmatrix}.$$

we need construct the file `func` first. As shown in Subsection 2.1, there are two ways to do it.

- In the first way, Tables 4 and 5 define `func`. 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
`n         = 2;`	% line 3
`s         = 1;`	% line 4
`pars.et1 = 0.1;`	% line 5
`data      = @(x,fgh)simple_ex4_func(x,fgh);`	% line 6
`func      = @(x,fgh,T1,T2)general_example(x,fgh,T1,T2,data);`	% line 7
`out       = NHTP(n,s,func,pars)`	% line 8

  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 `@(x,fgh)` as in line 6. 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
```

- In the second way, Table 6 defines `func=exfunc` directly. Then type the following code in a new command window:

`clc; clear; close all;`	% line 1
`n         = 2;`	% line 3
`s         = 1;`	% line 4
`pars.et1 = 0.1;`	% line 5
`out       = NHTP(n,s,@exfunc,pars)`	% line 6

The difference between the first and second way to construct `func` is that when it comes to solve a different example, the first way allows you only to change the `data` while the second way need you change `func`. Apparently, changing `data` seems to be easier than changing `func`.

4 Useful Notes

For the sake of the efficiency of this package, we summarize 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 in this package.

b) *For SLR problems:*

b.1) the data label 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 $m \le n \le 1000$, 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 $f(x)$ has too large value or $\nabla f(x)$ 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.