ANALYSIS OF NONNEGATIVE LEAST SQUARES ALGORITHMS

A Thesis

by

JASON STANLEY SATHYAKUMAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Freddie Witherden
Committee Members,	Ignacio Rodriguez-Iturbe
	Daniele Mortari
Head of Department,	Sharath Girimaji

May 2021

Major Subject: Ocean Engineering

ABSTRACT

Nonnegative least squares problems (NNLS) which are least squares solutions that are constrained to take nonnegative values often arise in many applications like image processing, data mining, etc. There have been several approaches to solve such a problem like the active set method by Lawson and Hanson, FNNLS by Bro and Jong, the Quasi-Newton minimization method, and Randomized projections methods. In this thesis, we evaluated the performance properties of all these algorithms by implementing them in MATLAB and compared the results. The results obtained showed that Randomized projections seem to work very efficiently, producing results around 3 times faster than Quasi-Newton method with a relative error of 3.25% for randomly generated matrices using MATLAB.

DEDICATION

This thesis is dedicated to my family and friends for their valuable support.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Freddie, for guiding me and helping me throughout this research. I would also like to thank my committee members, Dr. Ignacio, and Dr. Daniele, for their valuable comments on my thesis.

I would also like to thank my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my mother, father, and sister for their encouragement.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Professor Dr. Freddie Witherden [Advisor, Ocean Engineering] and Professor Dr. Ignacio Rodriguez-Iturbe [Home Department – Ocean Engineering] and Professor Dr. Daniele Mortari [Outside Department – Aerospace Engineering].

All work for the thesis was completed independently by the student.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

LS	Least Squares
NNLS	Non-negative Least Squares
FNNLS	Fast Non-negative Least Squares
SVD	Singular Value Decomposition

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER I

INTRODUCTION

MOTIVE OF RESEARCH

Non-negative least square methods are used in various fields like image processing, data mining, etc. There have been several algorithms that have been proposed over the years to solve this kind of problem. Some of them include the Lawson and Hanson algorithm (1974), FNNLS by Bro and Jong (1997), Quasi-Newton method (2007), and the random projections method (2009). Many of these algorithms were proposed nearly 10 years ago. With the improvement of processors and software over the years, we wanted to see if there were significant improvements in the performance of these algorithms individually and in comparison with each other.

The motive of this research was to implement these algorithms and compare the performance parameters of the algorithms such as the computational time and the residual error.

FORMAT OF RESEARCH

This research is divided into five sections as follows :

Chapter 1 describes the motive of the research.

Chapter 2 gives an introduction to least squares and the methods for solving a least squares problem.

Chapter 3 presents the non-negative least squares problem and gives a detailed explanation of the algorithms that are used in this research.

Chapter 4 describes the tests that were performed and the results that were obtained for it.

Chapter 5 provides a summary of the research.

CHAPTER II

LEAST SQUARES

The Least Squares methods are used to find the approximate solution for an overdetermined system of equations. They are called linear least squares when the given system of equations is linear.

MATHEMATICAL DEFINITION

Given a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $b \in \mathbb{R}^{m\times 1}$, the vector $x \in \mathbb{R}^{n\times 1}$ that minimizes the L2-norm $\|Ax - b\|^2$ is called as the least squares solution to the system of equations $Ax = b$. The norm $\|Ax - b\|^2$ is called as the residual norm denoted by r = $\|Ax - b\|^2$.

$$x = \arg\min \ \|Ax - b\|^2$$

Example:

Given the system of equations $Ax = b$ where $A = \begin{bmatrix}1 & 3\\ 2 & 1\end{bmatrix}$ and $b = \begin{bmatrix}2\\ -1\end{bmatrix}$, we can easily find the solution to the system by solving the equations $1x + 3y = 2$ and $2x + 1y = -1$ which would give us the solution $\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}-1\\ 1\end{bmatrix}$.

By adding another equation $2x - 2y = 3$ to the existing system of equations, we get $A = \begin{bmatrix}1 & 3\\ 2 & 1\\ 2 & -2\end{bmatrix}$ and $b = \begin{bmatrix}2\\ -1\\ 3\end{bmatrix}$. The earlier solution we obtained clearly does not solve the third equation $2(-1) - 2(1) = -4 \neq 3$. This is an example of an

overdetermined system of equations where we have more than the required number of independent equations needed to determine the unknowns. In other words, the number of equations (3 in this case) is higher than that of the number of unknowns (2, x and y in this case). This is where we use a least squares solution to solve the overdetermined system of equations. The solution to this new system would be $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.68 \\ -0.12 \end{bmatrix}$ and the residual norm would be 3.1305 which is lesser than the residual norm obtained for $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ which is equal to 7.

GEOMETRIC INTERPRETATION

The least squares solution can be viewed as the point on the column space of A that is at the least distance from point b.

Consider the overdetermined system of equations from the previous example. The column space of matrix A is spanned by the column vectors $a_1 = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$ and $a_2 = \begin{bmatrix} 3 \\ 1 \\ -2 \end{bmatrix}$. Taking the array x as x = $\begin{bmatrix} x \\ y \end{bmatrix}$, the product Ax can be expanded as $Ax = [a_1\ a_2].\begin{bmatrix} x \\ y \end{bmatrix} = [a_1]x + [a_2]y$ which is the linear combination the column vectors of A. Hence for any real vector x, Ax would be a vector in the column space of A pointing to a point in the plane spanned by $a_1$and a_2. The norm $\|Ax - b\|^2$ would be the square of the distance between this point and the point b = (2,-1,3). We know that the shortest distance of a point from a plane is given by the length of the perpendicular from the

point to the plane. From Figure 1, we can observe that the point on the plane which is at the least distance from b is (0.32, 1.24, 1.60). The vector joining this point from (0,0,0) is given by Ax = $\begin{bmatrix}0.32\\1.24\\1.60\end{bmatrix}$ for x = $\begin{bmatrix}0.68\\-0.12\end{bmatrix}$.

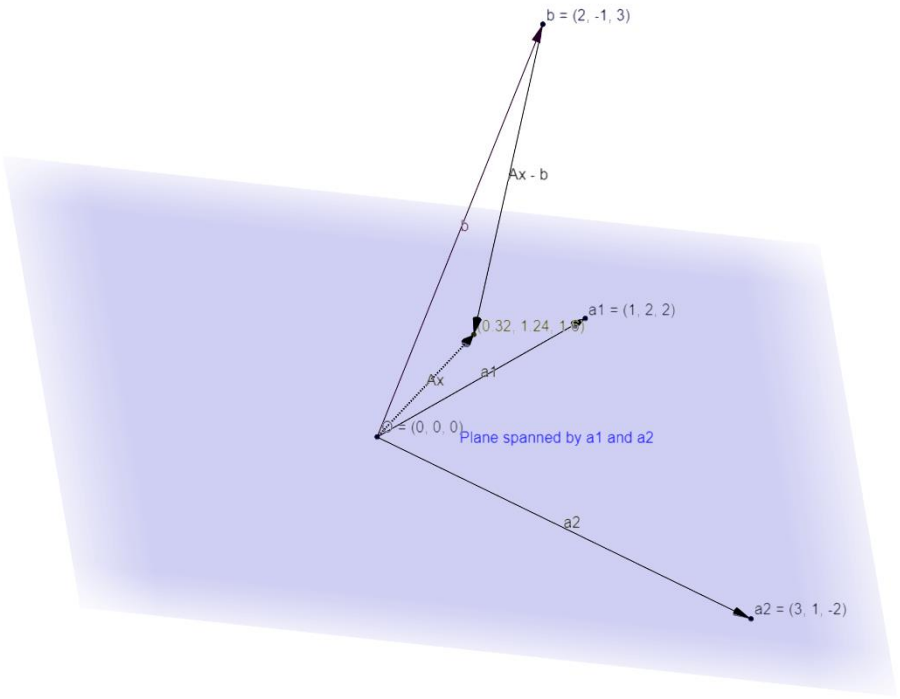

Figure 1. Geometric Interpretation of the Least Squares solution. Least Squares solution is the vector x for which Ax is at the least distance from b.

The Uniqueness of Least Squares solution:

The least squares solution always produces the least norm for $\|Ax - b\|^2$. To prove this, say the least squares solution for the system is $\hat{x}$. The norm of this solution is given by $\|Ax - b\|^2$. For any other vector $x \neq \hat{x}$, we have:

$$\begin{aligned}\|Ax-b\|^2 &= \|Ax-A\hat{x}+A\hat{x}-b\|^2\\ &= \|A(x-\hat{x})+(A\hat{x}-b)\|^2\\ &= \|A(x-\hat{x})\|^2+\|A\hat{x}-b\|^2+2\,A(x-\hat{x})^T(A\hat{x}-b)\\ &= \|A(x-\hat{x})\|^2+\|A\hat{x}-b\|^2\\ &> \|A\hat{x}-b\|^2\end{aligned}$$

$A(x-\hat{x})^T(A\hat{x}-b)$ is zero because $A(x-\hat{x})^T$ is orthogonal to $(A\hat{x}-b)$. This is because $A(x-\hat{x})$ is a vector that lies in the column space of A (as $x \neq \hat{x}$, $A(x-\hat{x})$ is a linear combination of the column space of A) and $(A\hat{x}-b)$ is a vector that is perpendicular to the column space of A (as observed in the previous example). So $(A\hat{x}-b)$ is orthogonal to $A(x-\hat{x})$ or vice versa.

APPLICATIONS

Least Sqaures methods have various applications in the field of statistics, control systems, image processing, etc. Some of their simple applications are explained below:

1) Polynomial fitting:

   Given the points $(x_1, y_1), (x_2, y_2) \ldots (x_m, y_m)$ we can find the best fit polynomial of degree less than 'n' by solving the least squares problem Ax = b, where:

$$A = \begin{bmatrix} 1 & x_1 & {x_1}^2 & . & . & {x_1}^{n-1} \\ 1 & x_2 & {x_2}^2 & . & . & {x_2}^{n-1} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_m & {x_m}^2 & . & . & {x_m}^{n-1} \end{bmatrix}$$ is known as the Vandermonde matrix

$$x = \begin{bmatrix} c_0 \\ c_1 \\ . \\ . \\ c_{n-1} \end{bmatrix} \text{ and } b = \begin{bmatrix} y_1 \\ y_1 \\ . \\ . \\ y_m \end{bmatrix}$$

Then the approximating polynomial passing through $(x_1, y_1), (x_2, y_2)$ …. (x_m, y_m) with degree 'n-1' would then be given by :

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^n$$

2) Deblurring of Images:

Least squares problem is used in deblurring images from the deblurring model equation which is given by $y = Ax + v$, where x is the unknown image, y is the observed image, A is the blurring matrix, and v is the noise. Images are of the size MxN pixels stored as a single MNx1 vector in x and y.

To solve, we minimize $\|Ax - y\|^2 + \lambda (\|D_V x\|^2 + \|D_h x\|^2)$, where the first term is called the data fidelity term and the second term penalizes the differences between the values at neighboring pixels:

$\|D_V x\|^2 + \|D_h x\|^2 = \sum_{i=1}^{M} \sum_{j=1}^{N-1} (X_{i,j} - X_{i,j+1})^2 + \sum_{i=1}^{M-1} \sum_{j=1}^{N} (X_{i,j} - X_{i+1,j})^2$

where X is the MxN image stored in the MNx1 vector x. In this case, minimizing $\|Ax - y\|^2$ would be a least squares problem.

METHODS TO SOLVE LEAST SQUARES

Forming the Normal Equations:

Since we try to solve for the vector x that minimizes the norm $\|Ax - b\|^2$, we need to find the vector for which the gradient of the objective function ($f(x) = \|Ax - b\|^2$) is zero.

$$f(x) = \|Ax - b\|^2$$
$$= (Ax - b)^T(Ax - b)$$
$$= (x^T A^T - b^T)(Ax - b)$$
$$= x^T A^T Ax - x^T A^T b - b^T Ax + b^T b$$

We know that $(x^T A^T b)^T = b^T (A^T)^T (x^T)^T = b^T Ax$. Also, the size of $x^T A^T b$ (x^T- 1xn and b – mx1) and $b^T Ax$ (b^T-1xm and x-nx1) are both 1x1. Hence, $x^T A^T b = b^T Ax$. This simplifies our **objective function** as:

$$\boldsymbol{f(x) = x^T A^T Ax - 2\, x^T A^T b + b^T b}$$

Taking gradient of the objective function we get:

$$\nabla f(x) = 2A^T Ax - 2A^T b$$
$$= 2A^T(Ax - b)$$

Thus, to get the least squares solution we need to solve the equation:

$$A^T(Ax - b) = 0$$

(Because at the minimum $\nabla f(x) = 0$)

Simplifying the above equation we get:

$$A^T Ax = A^T b$$

These above equations are called as the normal equations of the least squares problem. The coefficient matrix $A^T A$ is called as the **Gram matrix** of A.

If A has linearly independent columns then:

- $A^T A$ is non-singular.
- The normal equations have a unique solution (say $\hat{x}$), given by :

$$\hat{x} = (A^T A)^{-1} A^T b$$

The matrix $(A^T A)^{-1} A^T = A^+$ is called the **pseudo-inverse** of the matrix A.

Computational cost for forming the normal equations:

- To compute $A^T A$, for each element we do 'm' products and '$m - 1$' addition operations. A total of $n^2 \times (2m - 1) = O(mn^2)$ $(n^2 - \text{ total elements})$ flops would be required to find all elements. Since, $A^T A$ is a symmetric matrix ($(A^T A)^T = A^T A$), computing either one of the upper or lower triangle of the matrix should be enough to determine it. Hence, it requires $O(mn^2)$ flops.
- To compute $A^T b$, for each element we do 'm' products and '$m - 1$' addition operations. A total of n (total elements) $\times (2m - 1) = O(mn)$ flops would be required to find all elements.

Hence, a total of $O(mn^2) + O(mn) = O(mn^2)$ would be required for computing the normal equations.

- To solve the normal equations directly using the pseudo-inverse, we need $O(n^3)$ flops to compute the inverse of $A^T A$ and $O(n^2)$ flops to evaluate the product of

$(A^T A)^{-1}$ and $A^T b$. This is generally not a computationally efficient method to solve the least squares methods especially when A is a very large matrix.

Solving Normal Equations:

There are different methods to solve the normal equations after forming or computing them. Some of them are as follows:

Cholesky Factorization:

Cholesky factorization is the decomposition of a positive-definite matrix into a product of lower triangular matrix and its transpose. In our case, we can factorize our $A^T A$ matrix into:

$$A^T A = LL^T$$

$$\text{where } L - \text{lower triangular matrix}$$

This decomposition (to find the L matrix) takes $O(mn^2)$ to compute.

Substituting this in the normal equations, we get:

$$LL^T x = A^T b$$

Which reduces to

$$Lw = A^T b$$

$$\text{where } w = L^T x$$

After solving for w, we find x by solving:

$$L^T x = w$$

QR Factorization:

QR factorization is the decomposition of a matrix into a product of an orthogonal matrix Q and an upper triangular matrix R. Using QR decomposition, our A matrix becomes:

$$A = QR$$

The normal equations reduce to

$$x = (A^T A)^{-1} A^T b$$

$$x = (R^T Q^T QR)^{-1} R^T Q^T b$$

$$x = (R^T R)^{-1} R^T Q^T b,$$

$$\text{where } Q^T Q = I \text{ (orthogonality)}$$

$$x = R^{-1} Q^T b$$

$$Rx = Q^T b$$

Now, we just have to solve the system of equations $Rx = Q^T b$, where we have a reduced upper triangular system with R being an upper triangular matrix.

This approach is more stable than the Cholesky method and is considered as the standard method for solving least squares problems.

SVD Decomposition:

SVD decomposition is used for decomposing the matrix A into a product of matrices $U\Sigma V^T$, where U- mxm and V- nxn are orthogonal matrices and Σ is a mxn diagonal matrix.

Using SVD decomposition the normal equations reduce to

$$x = (A^T A)^{-1} A^T b$$

$$x = (V\Sigma U^T U\Sigma V^T)^{-1} V\Sigma U^T b$$

$$x = (V\Sigma^2 V^T)^{-1} V\Sigma U^T b,$$

where $U^T U = I$ (orthogonality)

$$x = V\Sigma^{-1}\Sigma^{-1}\Sigma U^T b$$

$$x = V\Sigma^{-1} U^T b$$

We solve the final equation to find the least squares solution using SVD decomposition. Similarly, like the QR factorization, we don't have to evaluate the $A^T A$ product to solve the problem.

Among the three methods to solve the least squares problem:

1) All the methods take $O(mn^2)$ operations to solve.
2) QR and SVD decompositions are more stable than the Cholesky methods.
3) Cholesky is the fastest, and QR is faster than the SVD method.
4) Hence, QR decomposition is the most efficient method to solve a least squares problem.

CHAPTER III

NON-NEGATIVE LEAST SQUARES

The Non-negative Least Squares methods are least squares solutions but with a constraint that our solution vector needs to be non-negative.

IMPORTANCE OF NON-NEGATIVE LEAST SQUARES

Non-negative least squares solutions are important because having negative values in the solution no sense in certain cases. For example, if we're solving equations involving chemical reaction we don't want our chemical concentrations to be negative. Another example is when we're dealing with images in the least squares problems, we know that the blocks in images represent pixels with some color code values in them which need to be non-negative. Hence, we need this constraint to ensure that the solution is non-negative.

One idea to enforce Non-negativity is to set the negative values in the final solution to be zero. This method is bad as it doesn't give a least squares solution to the problem with non-negative values. An example of why such an approach is bad is given below.

Example:

Consider an example :

$$A = \begin{bmatrix} 7 & 9 \\ 5 & 6 \\ 4 & 6 \end{bmatrix} \text{ and } b = \begin{bmatrix} 7 \\ 9 \\ 10 \end{bmatrix}$$

The least squares solution, (call it x_{LS}) was found to be $\begin{bmatrix} -2.56 \\ 3.11 \end{bmatrix}$.

If we want our solution to be non-negative, one of the ways would be to set the negative values to zero. By doing so we get

$x = \begin{bmatrix} 0 \\ 3.11 \end{bmatrix}$ and the norm $\|Ax - b\|^2$ = 609.56

But there exists a non-negative vector, $x = \begin{bmatrix} 0 \\ 1.16 \end{bmatrix}$ for which we have a norm $\|Ax - b\|^2 = 25.23$. Hence, setting the negative value obtained from the least squares solution to zero is not the best method to find the non-negative least squares solution to the system as we do not get the least norm for the solution vector (not a 'least square solution' to the system). Also, such an approach can cause problems when used in a multiway algorithm (such as the Alternating least squares method) where it could cause the algorithm to diverge [2].

MATHEMATICAL DEFINITION

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^{m \times 1}$, the vector $x \in \mathbb{R}^{n \times 1}$ that minimizes the L2-norm $\|Ax - b\|^2$subject to the condition $x_i > 0 \; \forall \; i \in [1, n]$ is called as the non-negative least squares solution to the system of equations $Ax = b$.

$$x = \arg\min \; \|Ax - b\|^2$$

$$\text{subject to } \; x_i \geq 0 \; \forall \; i \in [1, n]$$

Quadratic Form of the Objective Function:

We know that the objective function for least squares is given by:

$$f(x) = x^T A^T Ax - 2\, x^T A^T b + b^T b$$

For non-negative least squares problem, we have the same objective function with a constraint that $x_i \geq 0 \; \forall \; i \in [1, n]$.

Since $A^T A$ is symmetric and positive definite, we have a convex quadratic problem. This means that the function always has a global minimum even when it is constrained to take non-negative values.

METHODS TO SOLVE NON-NEGATIVE LEAST SQUARES

There have been several algorithms proposed over the years to solve a non-negative least squares problem. They can be broadly divided into:

1) Active set methods: The first algorithm for solving the NNLS problem was proposed by Lawson and Hanson (which was published in their book 'Solving Least Squares Problems') in 1974 [1, 5]. This algorithm was later modified by Bro and Jong [2] (called the FNNLS algorithm) in 1997 which improved the speed of the original algorithm by a good amount for large matrices.
2) Iterative methods: Since NNLS problem is a quadratic optimization problem, methods like Newton's methods can be used. The Quasi-Newton method [3] proposed in 2007 improved the speed of the FNNLS algorithm by nearly 10

times for large matrices of size 6000x3600 and higher. Other iterative algorithms include the sequential coordinate-wise approach.

3) Sampling methods: Some algorithms like the randomized NNLS [4] speeds up any standard NNLS solver by decomposing the main problem into a new problem involving lesser rows/equations that take lesser time to solve than the main problem.
4) Other methods: Algorithms like interior-point methods can also be used to solve the NNLS problems [4].

Lawson and Hanson algorithm:

The Lawson and Hanson algorithm is an active set algorithm. It involves the use of two sets: Active – which contains the variables that violate the non-negativity constraint and Passive – which contains the other variables. The algorithm solves for the least square solution for the passive set system by moving a variable from the active set to the passive set every iteration. The active set element with the most negative gradient is chosen at each iteration. After the least square solution is obtained at every step, the most negative value in the solution vector is moved to zero and a scale of this distance was used to move the other values, thereby ensuring that the solution stays non-negative. This value which was the most negative is sent back to the active set. This procedure continues till there are no elements in the active set or till the objective function gradient of the active set elements is positive. All the steps of this algorithm is shown in Figure 2 followed with an example case to illustrate this method. The code for this algorithm which was implemented in MATLAB can be found in APPENDIX A.

Active set method

A . Initialization

Requires input matrices : $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^{m \times 1}$

1. Initialize $x = 0 \in \mathbb{R}^{n \times 1}$, Z = {1,2…n} (Active set indices), P = Null (Passive set indices)

2. Compute $w = A^T(b - Ax)$ (The negative of the gradient of $||Ax - b||^2$)

B . Main loop (Proceeds to find and change values of x in the active set which can be increased to minimize the norm)

1. Proceed if Z = Null or if $w_i \geq 0 \; \forall \; i \in Z$. Else, end the algorithm and return x.

2. Set $t = j$ for which $w = \max\{w\}$

3. Move the index t, from set Z to P

4. Calculate the least squares solution z only for the indices present in the passive set using the submatrix A_P (containing only the columns that belong to the passive set) i.e., solve $A_P z \cong b$

C. Inner loop (Proceeds to find the negative values coming from the least squares solution and increase the x values in the passive set by a factor each such that the most negative value is set to 0)

1. Proceed if $\min(z_p) \leq 0 \; \forall \; p \in P$. Else, go to step A2.

2. Set $\alpha = -\min(\frac{x_p}{x_p - z_p}) \quad \forall \; p \in P$

3. $x = x + \alpha(x - z)$

4. Move indices whose x values are 0 from set P to set Z. Go to step A2.

Figure 2. Lawson & Hanson algorithm. A brief overview of the Lawson and Hanson active set algorithm.

Example:

Take the example: $A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \\ 2 & -2 \end{bmatrix}$ and $b = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$.

Initialization:

A1. We initialize $x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, Z = [1 2] (active set indices) and P = [ ] (passive set – Null at start).

A2. First, we find the negative gradient vector $w = A^T(b - Ax)$:

$$w = \begin{bmatrix} 1 & 2 & 2 \\ 3 & 1 & -2 \end{bmatrix} \text{x} \left(\begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} - 0 \right) = \begin{bmatrix} 6 \\ -1 \end{bmatrix}$$

We normally move the value at the index 'i' of the active set, which will decrease the norm by the highest amount. In this case the most negative gradient is at position 1, so we choose that index from the active set and move it to the passive set.

Iteration 1:

B1 – B3. Updating our Z and P vectors, we get:

Z = [2] and P = [1]

B4. Now, we find the least squares solution for the active set system:

$$A_P = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} \; and \; b = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$$

Which gives the value z_P = [0.667]

$z_Z = [0]$ (as we don't use the active set variables)

$z = \begin{bmatrix} 0.667 \\ 0 \end{bmatrix}$ (z at first iteration)

Since z_P is non-negative, we skip the inner loop C, set $x = z$ and go to step A2. (If z_P was negative, we move the value to 0 and move the other elements by the step C2. We then move the index 1 from the passive set to the active set)

Iteration 2:

A2. $w = A^T(b - Ax)$:

$$w = \begin{bmatrix} 1 & 2 & 2 \\ 3 & 1 & -2 \end{bmatrix} \text{x } \left(\begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} - \begin{bmatrix} 1 & 3 \\ 2 & 1 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} 0.667 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} 0 \\ -1.667 \end{bmatrix}$$

Now, we have

$w_Z = [-1.667]$ (Checking for B1 step to enter the B loop)

This means that the active set gradient is greater than 0. So, we cannot increase the value of x for this index as it increases the norm (increases $f(x)$ as $\nabla f(x) > 0$). We cannot decrease this value to decrease the norm either as we have a non-negativity constraint.

Hence, the algorithm stops here as Z = [2] and $w_Z < 0$.

Solution vector $x = \begin{bmatrix} 0.667 \\ 0 \end{bmatrix}$.

FNNLS algorithm by Bro and Jong:

This algorithm modified the Lawson and Hanson method by pre-computing the matrices $A^T A$ and $A^T b$ [2].

If we look at the steps A2 and B4 of the Lawson and Hanson algorithm:

A2. $w = A^T(b - Ax)$

B4. Solving $(A^P)^T A^P y = (A^P)^T b$

We have to evaluate these matrix products $A^T(b - Ax)$ and $(A^P)^T\ A^P$ every iteration. This can become computationally expensive when the number of iterations become larger. For example, a randomly generated matrix problem of size 6000x3600 in MATLAB took about 178 iterations to solve the problem.

Replacing these steps with:

$$w = (A^T b) - (A^T A)x$$

$$A^T A(P, P)\ x = (A^T b)(P)$$

We can reduce the time taken to solve the problem over large iterations by pre-computing $A^T A$ and $A^T b$. The code for this algorithm in MATLAB can be found in APPENDIX B.

Quasi-Newton algorithm:

This iterative method that is used for finding the minima of multi-dimensional functions works very well for solving the NNLS problem because of the convex quadratic form of the objective function.

This method is a modification to the Newton's method where it uses an approximate gradient scaling matrix S_k that is evaluated at every iteration k rather than computing $\nabla^2 f(x)^{-1}$ at every step which could be computationally expensive.

Newton's method is a second-order method that has a better convergence rate than the gradient descent method. In these methods, we start from a point (the origin, in most of the cases), find the direction that is opposite to that of the gradient, and move in that direction based on a step size. This way we get closer to the minima at every step. Identifying the direction and finding the best step size at every iteration is crucial to the

convergence speed of the algorithm. In addition, the solution vector is made non-negative by projecting the negative values to zero at each update. The full steps of the Gradient Descent and Newton's methods are shown in Figure 3.

Gradient Descent and Newton's methods:

A. Initialization part

Requires input matrices: $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^{m \times 1}$

1. Initialize x=0 $\in \mathbb{R}^{n \times 1}$, $p = A^T b, Q = A^T A$ and k=1.

B. Main loop

while $(x(k) - x(k-1) < tolerance)$ (Checking for convergence)

1. *Gradient descent method*: $x_k = [x_{k-1} - t\nabla f(x_{k-1})]_+$
   Newton's method: $x_k = [x_{k-1} - t(\nabla^2 f(x_{k-1}))^{-1}(\nabla f(x_{k-1}))]_+$ (Converges faster than GD)

Where $\nabla f(x) = Qx - p$ and $\nabla^2 f(x) = Q$.

t is determined based on backtracking line search

Figure 3. Gradient descent & Newton's algorithm. Used for finding the minima of convex functions. We can use this to find the minimum of the objective function $f(x)$.

The Quasi-Newton method for solving the NNLS problem proposed by Kim, Sra, & Dhillon [3] partitioned the x vector variables into free and fixed set at every iteration. It then solved the problem over the free set at every iteration.

Fixed variable set (z) at the kth iteration:

$$z_{ind}^k = \{i \mid x_i^k = 0 \ \& \ \nabla f(x^k)_i > 0\}$$

Free variable set (y) at the kth iteration:

$$y_{ind}^{k} = \{1, 2, 3 \dots n\} - z_{ind}^{k}$$

When $x_i^k = 0$ and $\nabla f(x^k)_i > 0$, decreasing x_i^k would decrease the norm ($f(x)$) but it would make x_i^k take a negative value which violates the nonnegativity constraint. On the other hand, increasing it would increase the objective function value so it is better to keep it fixed for the current iteration. The free variable set which consists of the variables that do not belong to the free can be increased so that their norm can be reduced.

At each iteration:

$x_i^k = \begin{bmatrix} y^k \\ z^k \end{bmatrix}$, $\nabla f(x^k) = \begin{bmatrix} \nabla f(y^k) \\ \nabla f(z^k) \end{bmatrix}$, where $y \in free\ set$ and $z \in fixed\ set$

We need to solve:

$$\text{minimize}_y \ \frac{1}{2} \ \|A[y; z] - b\|^2$$

$$\text{subject to } \ y > 0, z = 0$$

The problem over free variable set reduces to:

$$\text{minimize}_y \ g^k(y) = \tfrac{1}{2}\|\bar{A}y - b\|^2, \quad \text{subject to } y > 0$$

The solution vector for free variables y^{k+1} is given by:

$$y^{k+1} = y^k + \alpha(\gamma^k - y^k)$$

Where $\gamma^k = P(y^k - \beta \bar{S}^k \nabla f(y^k))$ is an intermediate step (Quasi-Newton update step). The previous equation gives a point y^{k+1} which is in the direction joining the points y^k and γ^k where the function $g^k(y)$ is minimum. α and β are line search

parameters obtained from minimization rule and Armijo backtracking line search respectively. $\bar{S}^k$ is the gradient scaling matrix for the free set which is evaluated at the end of every iteration.

Limited Minimization Rule (for determining α):

The α value is obtained by minimization rule, where α is the value between 0 and 1 for which the function $g\left(y^k + \alpha(\gamma^k - y^k)\right)$ is minimum.

$$\alpha = \text{argmin}_{\alpha \in [0,1]} \; g(y^k + \alpha d),$$

$$\text{where } \; d = \gamma^k - y^k$$

To obtain the value of α, we take the derivative of the function with α and equate it to zero and solve for the value α. This can be done because we get a quadratic function in single variable α along the direction y^k to γ^k.

$$\frac{dg}{d\alpha} = d^T \nabla g(y^k + \alpha d) = d^T(\bar{A}^T \bar{A}(y^k + \alpha d) - \bar{A}^T b) = 0$$

$$d^T \bar{A}^T \bar{A} y^k + \alpha(d^T \bar{A}^T \bar{A} d) = d^T \bar{A}^T b$$

$$\alpha = \frac{(\bar{A}d)^T(b - \bar{A}y^k)}{\|\bar{A}d\|^2}$$

where $\bar{A} = A(:, y^k_{ind})$ (A matrix over free variable set)

Armijo Line Search (for determining β):

The line search value β is important because it makes a better choice of a step size than having a constant step size (say a value of 1). This is because when we get closer to the solution, β has to be very small in order for the algorithm to converge to the expected solution. If it remains high as the initial value chosen, then the minimization

rule applied to the current point and the projected point due to β would make it require many iterations to get very close to the solution.

The β value is determined iteratively till this inequality that is specified below is satisfied.

$$g^k(y^k) - g^k\left(\gamma^k(s^m\sigma)\right) \geq \tau\, \nabla g^k(y^k)^T(\gamma^k(s^m\sigma) - y^k)$$

Where: $\sigma > 0$ is the starting point specified, $s \in (0,1)$ is multiplied to σ at every iteration till the condition is satisfied and $\tau \in (0,0.5)$ is multiplied to the gradient. The line with this new slope starting from $g^k(y^k)$ intersects the curve at a point that is the upper limit of the value β.

Gradient Scaling Matrix update (S^k):

The gradient scaling matrix S^k is updated at each step using the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update given by:

$$S_{k+1} = S_k + \left(1 + \frac{u^T A^T A S_k A^T A u}{u^T A^T A u}\right)\frac{uu^T}{u^T A^T A u} - \frac{S_k A^T A u u^T + u u^T A^T A S_k}{u^T A^T A u}$$

where $u = x^{k+1} - x^k$ and S^k is the gradient scaling matrix from the previous iteration starting from $S^1 = I_n$ (identity matrix of size nxn).

This method ensures that we get a matrix that is positive definite and symmetric like the $\nabla^2 f(x)^{-1}$ matrix that we use this approximation for.

$\nabla^2 f(x) = A^T A$ for our NNLS problem

(This requires $O(mn^2)$ operations for computation)

If we look at the term $u^T A^T A S_k A^T A u$ in the BFGS update, we can combine the products like Au and then multiply it with A^T and similarly for $(Au)^T A(S_k(A^T(Au)))$,

grouping the products in such a way that we always multiply a matrix with a vector. This reduces the computational cost of evaluation of $\nabla^2 f(x)^{-1}$ at every step to $O(mn)$. The main steps of the Quasi-Newton algorithm is shown in Figure 4.

Quasi-Newton algorithm:

A. Initialization:

1. Initialize $x = zeros\ \mathbb{R}^{n\times 1}$, $S^1 = I_n$ and $k = 0$.

B. Main Loop

Repeat till $\|Ax - b\|_{k-1} - \|Ax - b\|_k < tolerance$

1. Find fixed (z^k) and free (y^k) variable set.
2. Find appropriate β value using line search and obtain $\gamma^k = P(y^k - \beta \bar{S}^k \nabla f(y^k))$.
3. Find the value of α using minimization rule and then update $y^{k+1} = y^k + \alpha(\gamma^k - y^k)$.
4. Update the x^{k+1} vector and evaluate the $\|Ax - b\|_{k+1}$ norm.
5. Do the BFGS update to find the gradient scaling matrix S^{k+1} and set $k = k + 1$.

Figure 4. Quasi-Newton Algorithm. Code for this algorithm implemented in MATLAB is shown in APPENDIX C.

Example:

Take the same example: $A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \\ 2 & -2 \end{bmatrix}$ and $b = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$.

Initialization:

A1. We initialize $x^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $S^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$\|Ax_1 - b\| = 3.7417$

Main Loop:

This loop is executed whenever the stopping criteria is not satisfied.

Iteration 1:

B1. First, we find the fixed and free variable set:

$$\nabla f(x^1) = A^T(Ax^1 - b) = \begin{bmatrix} -6 \\ 1 \end{bmatrix}$$

Since $x^1(2) = 0$ and $[\nabla f(x^1)](2) = 1 > 0$, the index 2 of x goes to the fixed variable set and index 1 goes to the free variable set.

We have, $y^1_{ind} = [1]$, $z^1_{ind} = [2]$, $y^1 = [0]$ and $z^1 = [0]$.

For this example, we take a constant value of $\beta = 1$. For larger matrix problems we need to do line search to obtain the β value as $\beta = 1$ could be large in some cases causing the problem to take longer time to converge to the solution.

B2. For $\beta = 1$, we get γ^1:

$$\gamma^1 = P(0 - 1.[1].[-6]) = P([6]) = [6]$$

The $\bar{S}^k$ matrix is the S^k matrix for free variable set i.e., $\bar{S}^k = S^k(y^k_{ind}, y^k_{ind})$.

Since $S^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\bar{S}^1 = S^k(y^1_{ind}, y^1_{ind}) = [1]$.

$P(x)$ returns a non-negative projection (sets negative values to 0) of x to maintain the non-negativity constraint.

B3. Now we find α using the minimization rule equation and get $\alpha = 0.1111$.

$$y^2 = y^1 + \alpha(\gamma^1 - y^1) = [0] + 0.1111.([6] - [0]) = [0.667]$$

B4. Hence, we get $x^2 = \begin{bmatrix} y^2 \\ z^2 \end{bmatrix} = \begin{bmatrix} 0.667 \\ 0 \end{bmatrix}$ and $\|Ax_2 - b\| = 3.1623$

B5. Doing the BFGS update we get:

$$S^2 = \begin{bmatrix} 0.1235 & -0.1111 \\ -0.1111 & 1.0000 \end{bmatrix}$$ (Symmetric as expected)

We check for the stopping criterion: $\|Ax_{k-1} - b\| - \|Ax_k - b\| < tolerance$.

We take a tolerance value of 10^{-6} (or $1e-6$), which tells the program to stop if the function value doesn't change more than 0.000001 by value i.e., the function converges to the solution for up to 6 figures.

$$\|Ax_1 - b\| - \|Ax_2 - b\| = 3.7417 - 3.1623 = 0.5794$$

This value is greater than the $tolerance = 10^{-6}$. Hence, we go through the main loop again.

Iteration 2:

B1. The fixed and free variable set:

$$\nabla f(x^2) = A^T(Ax^2 - b) = \begin{bmatrix} -8.8 \times 10^{-16} \\ 1.666667 \end{bmatrix}$$

Since $x^2(2) = 0$ and $[\nabla f(x^2)](2) = 1 > 0$:

$y^2_{ind} = [1]$, $z^2_{ind} = [2]$, $y^2 = [0.667]$ and $z^2 = [0]$.

B2. For $\beta = 1$, we get γ^2:

$$\gamma^2 = P([0.667] - 1.[0.1235].[-8.8 \times 10^{-16}]) \approx P([0.667]) \approx [0.667]$$

B3. We find α using the minimization rule equation and get $\alpha = 0.8889$.

Using the value of α we get: $y^3 \approx 0.667$.

B4. Hence, we get $x^3 = \begin{bmatrix} y^3 \\ z^3 \end{bmatrix} \approx \begin{bmatrix} 0.667 \\ 0 \end{bmatrix}$ and $\|Ax_3 - b\| \approx 3.1623$

B5. Doing the BFGS update we get:

$$S^3 \approx \begin{bmatrix} 0.1235 & -0.1111 \\ -0.1111 & 1.0000 \end{bmatrix}$$

Stopping criterion: $\|Ax_2 - b\| - \|Ax_3 - b\| = -4.4409 \times 10^{-16} < tolerance$.

Hence, the program ends here, returning the result $x = x^3 = \begin{bmatrix} 0.667 \\ 0 \end{bmatrix}$.

Random Projections algorithm:

Some methods involve sampling of the rows of the matrix A and vector b to construct a smaller problem with lesser rows and the same columns. If $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^{m \times 1}$, these sampling algorithms choose r $(r < m)$ rows out of the m rows and this way they reduce the complexity of solving the problem. For example, least squares problem that would take $O(mn^2)$ to compute would be reduced to $O(rn^2)$. These methods can be applied to any NNLS algorithm to increase the speed of computing the reduced problem. The problem that arises with such methods is the loss in accuracy because it solves lesser equations than the given equations. Another challenge faced by these algorithms is to determine a computationally efficient method to form this induced problem. The sampling algorithm proposed by Drineas, Mahoney & Muthukrishnan [6] involves the calculation of the L2-norms of the left singular matrix of A to determine the probability of selecting the rows. This is not fast since the calculation of this probability alone would cost $O(mn^2)$ operations.

The method proposed by Boutsidis & Drineas [4] which is based on a fast Johnson–Lindenstrauss Transform, makes use of the Hadamard matrix H to form the induced problem. The running time of this algorithm is given by $O(mn \log r) + T_{NNLS}(r, n)$, where T_{NNLS} is the time taken to solve the reduced problem (rxn) using a

standardized NNLS solver. In this algorithm, the relative error bounds (which depends on the size of the smaller probelm) of the norm $\|Ax - b\|$ produced by this algorithm with respect to the NNLS algorithm used can be predicted with a high probability.

The relative error bound is given by:

$$\|A\tilde{x} - b\|^2 \leq (1 + \epsilon) \min_{x \geq 0} \|Ax - b\|^2$$

$where\ \tilde{x} - randomized\ algorithm\ solution, x - standard\ NNLS\ solver\ solution.$

This inequality holds with a probability that is at least 0.5

In this method, the matrices A and b are pre-multiplied by S, H and D matrices which are given by:

Normalized Hadamard matrix (H_m):

$$H_m = \frac{1}{\sqrt{m}} \begin{bmatrix} H_{m/2} & H_{m/2} \\ H_{m/2} & -H_{m/2} \end{bmatrix}, \text{ where } H_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix}$$

Diagonal matrices S, D: ($\forall\ i \in [1, m]$)

$$S_{ii} = \begin{cases} \sqrt{m/r} & with\ probability\ r/m \\ 0 & otherwise \end{cases}$$

$$D_{ii} = +1\ with\ a\ probability\ 0.5, else - 1$$

By doing such a transform, we can obtain the error bound with a probability of 0.5 which is shown by Boutsidis & Drineas [4]. The value of r in our implementation was taken to be $n + 20$ as it was said to give the best results according to the literature. All the steps for this algorithm is shown in Figure 5. Our code for this algorithm in MATLAB is given in APPENDIX D.

Randomized NNLS algorithm:

1. Check if the number or rows m is a power of 2 (since Hadamard matrix is a mxm matrix with m=2^k (k=1,2,3…). If it is not pad A and b with rows of 0s till the total rows become a power of 2.

2. Initialize the normalized Hadamard matrix of mxm.

3. Initialize a diagonal matrix S of size mxm, where $S_{ii} = \sqrt{m/r}, with\ probability \frac{r}{m}, 0\ otherwise.$

4. Evaluate the matrix $\tilde{H}$ containing non-zero rows of SH (This has an expectation of r rows).

5. Create a diagonal matrix D of size mxm, where $D_{ii} = +1$ with probability 1/2, -1 otherwise.

6. Solve the NNLS problem: $min\left\|\tilde{H}DA\tilde{x} - \tilde{H}Db\right\|_2^2$ using the any standard NNLS solver with inputs A as $\tilde{H}DA$ and b as $\tilde{H}Db$ (NNLS solver – Quasi-Newton in our case).

Figure 5. Randomized NNLS Algorithm.

CHAPTER IV

TEST AND RESULTS

TEST

Firstly, the results produced by the $lsqnonneg(A, b)$ function in MATLAB was verified to see if it was same as the Lawson and Hanson algorithm with our implementation. The results obtained were similar to that of our implementation.

The Lawson and Hanson, FNNLS, Quasi-Newton and Random projections algorithms were implemented in MATLAB. Fully dense random matrices were generated using MATLAB's randi function for the tests averaged over 20 iterations. The computational time taken to solve the problem by each algorithm was measured using cputime in MATLAB. The residual $\|Ax - b\|_2$ were also calculated for each algorithm to check for accuracy. The matrix sizes chosen for these tests are shown in Table 1.

Table 1. The list of matrix problems (labelled 1 to 8) used in the tests for m rows and n columns.

Matrices	1	2	3	4	5	6	6	8
m	2800	3600	4400	5200	6000	7200	8800	10400
n	2000	2400	2800	3400	3600	4800	5600	6800

RESULTS

Computational Time:

The results obtained for computational time are shown in Table 2 and the plot

showing cputime for every matrix problem for each algorithm is shown in Figure 6. The Lawson and Hanson algorithm took the most amount of time to solve the problems, followed by FNNLS, Quasi-Newton and Randomized algorithms. For the matrix problem 1 where mxn – 2800x2000 the Quasi-Newton took longer time than FNNLS method. Randomized algorithm solved the problems the fastest, taking around or less than 10s for matrix sizes up to 10400x6800. For the 10400x6800 case, randomized algorithm solved the problem nearly 3 times faster than the Quasi-Newton method. The results for computational time are shown in Figure 6.

Table 2. Cputime results (time in seconds) of the algorithms for each of the matrix problems.

Matrices	1	2	3	4	5	6	6	8
Lawson and Hanson	5.25	10	15.281	22.75	29.484	54.844	72.766	124.08
FNNLS	1.8281	3.6563	5.7813	9.875	11.828	27.266	31.703	47.25
Quasi-Newton	2.3594	3.2969	4.0375	5.6375	6.5203	13.391	19.078	30.57
Randomized NNLS	0.7672	1.125	1.4906	2.1234	2.4063	4.2344	5.6547	11.944

Residual Norm:

The results obtained for the residual norm $\|Ax - b\|_2$ are listed in Table 3 and the plot is shown in Figure 7. The results were as expected for the first two algorithms because FNNLS is just a slight modification of the Lawson and Hanson algorithm. However, the Quasi-Newton method only agreed with the Lawson and Hanson algorithm results for up to 3 significant figures. To match more significant figures in the

final residual norm, the Quasi-Newton algorithm had to take more iterations as the line search was producing very small steps. The randomized algorithm produced the highest norm because it solves the smaller subset of equations from $Ax = b$.

Table 3. Residual norm $\|Ax - b\|_2$ results of the algorithms for each of the matrix problem.

Matrices	1	2	3	4	5	6	6	8
Lawson and Hanson	147.01	166.83	182.42	198.83	216.53	236.9	262.9	285.45
FNNLS	147.01	166.83	182.42	198.83	216.53	236.9	262.9	285.45
Quasi-Newton	147.16	166.97	182.58	198.92	216.62	236.92	262.92	285.56
Randomized NNLS	149.58	171.77	188.32	203.61	221.54	242.35	267.99	291.67

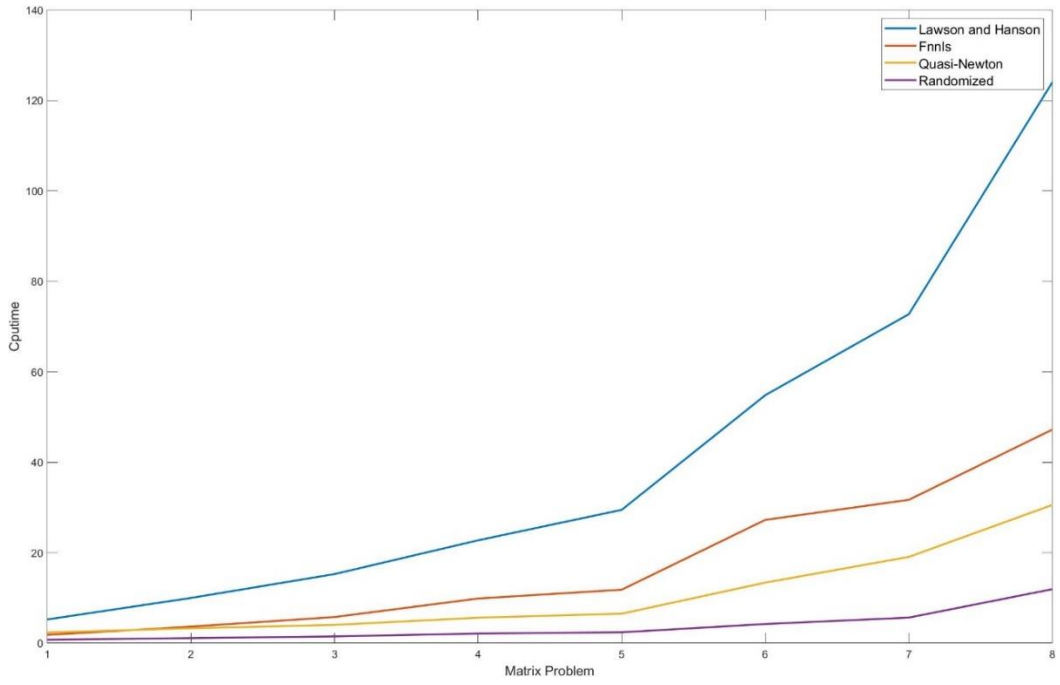

Figure 6. Time vs matrix problem number. Shows the cputime taken to solve the problem.

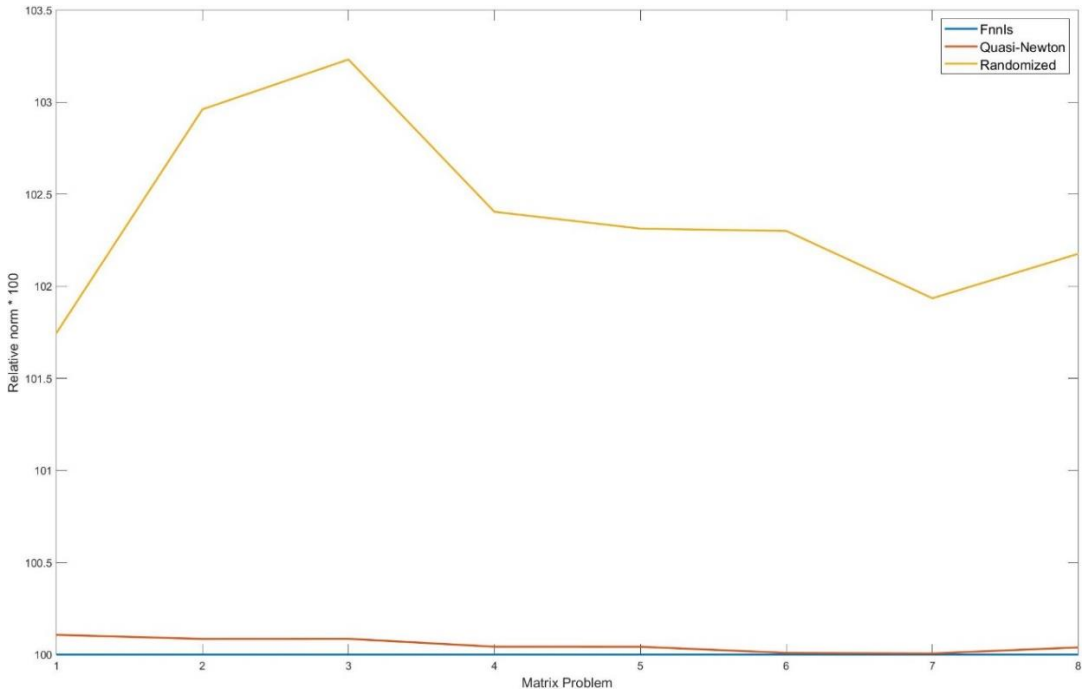

Figure 7. Relative norm vs matrix problem number. Relative norm is taken with respect to Lawson & Hanson algorithm (Relative norm = $100 * \|Ax_a - b\|_2 / \|Ax_{LH} - b\|_2$, x_a- solution for each algorithm).

The computational time results for a matrix size 6000x3600 shows that our Quasi-Newton code solves the problem in 30s which is about of 1.57 the time taken by FNNLS (47s). The results obtained by Kim & Sra & Dhillon [3] in 2006, showed that the time taken to solve the same problem by Quasi-Newton took 294s which was about 10 times faster than FNNLS (3076s). This difference in speedup that we obtained for the Quasi-Newton with respect to FNNLS method in comparison to their result could be because of our line search, which may not be producing the best step sizes to converge to the solution faster.

CHAPTER V

CONCLUSIONS

All the algorithms are faster than they were 15 years ago. This could be due to the better processors available now. From this research, we see that the Quasi-Newton method performs well up to 3 significant digits in comparison to the Lawson & Hanson and FNNLS algorithms. To get results that match beyond 3 significant figures, the Quasi-Newton method takes a lot of time because the iterations become higher as the line search step sizes are lower. Line search is the area that could be looked at to provide better step sizes so that the overall speed of the algorithm is reduced while obtaining results with high significant figures with that of the expected solution. The results obtained in our research for the 3 significant figures show that Quasi-Newton algorithm starts taking lesser time than the FNNLS method for matrix sizes above 3600x2400. The randomized algorithm is the fastest method to produce the results though there would be some errors in the residual norm for the result. The maximum and minimum relative error seems to be around 3.5% and 2% respectively for an average of 25 iterations, which is very less given the speed at which it solves the problem. The only problem where Lawson and Hanson algorithm would perform better is for very low dimension matrix problems which are not ideal in real-world applications.

REFERENCES

[1] C. L. Lawson and R. J. Hanson, Solving least squares Problems, Prentice Hall, (1987), pp. 121-132 & 158-162.

[2] R. Bro, S. D. Jong, A fast non-negativity-constrained least squares algorithm, Journal of Chemometrics, Vol. 11, No. 5, (1997), pp. 393–401.

[3] D. Kim, S. Sra, and I. S. Dhillon, A new projected quasi-Newton Approach for the non-negative least squares problem, Technical Report TR-06-54, Computer Sciences, The Universityof Texas at Austin, (2006).

[4] C. Boutsidis, P. Drineas, Random projections for the nonnegative least-squares problem, Linear Algebra and its Applications, Volume 431, Issues 5–7, (2009), pp. 760-771.

[5] Y. Luo and R. Duraiswami. Efficient parallel nonnegative least squares on multicore architectures. SIAM Journal on Scientific Computing, 33 (5), (2011), pp. 2848–2863.

[6] P. Drineas, M. Mahoney, S. Muthukrishnan, Sampling algorithms for l2 regression and applications, in: ACM-SIAM Symposium on Discrete Algorithms, (2006), pp. 1127–1136.

APPENDIX A

LAWSON & HANSON ALGORITHM CODE

(Our Implementation in MATLAB)

```
tl = cputime;
%Initialization
n = size(A,2);
P = [];
N = 1:n;

x = zeros(n,1);
w = A'*(b - A*x);   %Negative Gradient
wN = w(N);
iters = 0;

%Main Loop
while(~(isempty(N) || all(wN<=1e-12)))
    %Moving active and passive set indices
    wmax = max(wN);
    t = find(w==wmax);
    P = [P t];
    N(N==t) = [];

    %Solving the LS problem
    z = zeros(n,1);
    clear("Ap");
    Ap = A(:,P);
    zp = Ap\b;
    z(N) = zeros(length(N),1);
    z(P) = zp;

    %Inner loop handling the non-negativity constraint
    while(min(zp)<=0)
        xp = x(P);
        ratio = zeros(length(xp),1);
        for i=1:length(xp)
            ratio(i) = xp(i)/(xp(i)-zp(i));
        end
        qind = zp<=0;
        [alpha,pq] = min(ratio(qind));
        q = P(pq);
        x = x + alpha*(z-x);
        temp = P(x(P)==0);
        N = [N temp];
        P(P==temp) = [];

        clear("Ap");
        Ap = A(:,P);
        zp = Ap\b;
        z(N) = zeros(length(N),1);
        z(P) = zp;
    end
    x=z;
    w = A'*(b - A*x);
    wN = w(N);
    iters = iters + 1;
end
tlhnnls = cputime - tl;
```

APPENDIX B

FNNLS ALGORITHM CODE

(Our Implementation in MATLAB)

```
t1 = cputime;
%Precomputing AtA and Atb
AtA = A'*A;
Atb = A'*b;

n = size(AtA,2);
P = [];
N = 1:n;
x = zeros(n,1);
w = (Atb) - (AtA)*x;
wN = w(N);
iters = 0;

%Main Loop
while(~(isempty(N) || all(wN<=0)))
    %Moving active and passive set indices
    wmax = max(wN);
    t = find(w==wmax);
    P = [P t];
    N(N==t) = [];

    %Solving the LS problem
    z = zeros(n,1);
    An = AtA(P,P);
    bn = Atb(P);
    zp = An\bn;
    z(N) = zeros(length(N),1);
    z(P) = zp;

    %Inner loop handling the non-negativity constraint
    while(min(zp)<=0)
        xp = x(P);
        ratio = zeros(length(xp),1);
        for i=1:length(xp)
            ratio(i) = xp(i)/(xp(i)-zp(i));
        end
        qind = zp<=0;
        [alpha,pq] = min(ratio(qind));
        q = P(pq);
        x = x + alpha*(z-x);
        temp = P(x(P)==0);
        N = [N temp];
        P(P==temp) = [];
        An = AtA(P,P);
        bn = Atb(P);
        zp = An\bn;
        z(N) = zeros(length(N),1);
        z(P) = zp;
    end
    x=z;
    w = (Atb) - (AtA)*x;
    wN = w(N);
    iters = iters +1;
end
tsfnnls = cputime - t1;
```

APPENDIX C

QUASI-NEWTON ALGORITHM CODE

(Our Implementation in MATLAB)

```
t = cputime;
n = size(A,2);
S(:,:,1) = eye(n,n);
x(:,1) = zeros(n,1);
k = 1;
Ax_b = A*x(:,k) - b;
l2norm(k) = norm(Ax_b);


while(1)
    %Fixed and Free set evaluation
    x_zero = find(x(:,k)==0);
    F = A'*Ax_b;
    z_ind = intersect(x_zero,find(F>=0));
    z = x(z_ind,k);
    y_ind = setdiff(1:n,z_ind);
    y = x(y_ind,k);

    %Direction
    Sn = S(y_ind,y_ind,k);  %Sn,An - matrix for free set vars
    An = A(:,y_ind);
    dfy = F(y_ind);    %Gradient of f
    Any_b = An*y - b;
    dgy = An'*(Any_b); %Gradient of g

    %Line search parameters
    g_0 = 0.5*(norm(Any_b)^2);
    Sn_dfy = Sn*dfy;
    if(k<=3)
        beta = 1;
    end
    if(k>3)
        beta = stepsize(An,b,y,g_0,Sn_dfy,dgy);
    end

    ga = y - beta*(Sn_dfy);
    %ga(ga<0) = 0;

    %Minimization rule
    d = ga - y;
    denom = norm(An*d)^2;
    if(denom == 0)
        a1= 1;
    else
        a1 = d'*(-dgy)/denom;
    end
    a1(a1>1) = 1;
    a1(a1<0) = 0;

    %x Update
    y = y + a1*(ga-y);
    y(y<0)=0;
    k = k+1;
    x(y_ind,k) = y;
    x(z_ind,k) = z;
```

```
    %BFGS update
    u = x(:,k)-x(:,k-1);
    S(:,:,k) = BFGS(A,u,S(:,:,k-1));
    Ax_b = A*x(:,k) - b;
    l2norm(k) = norm(Ax_b);

    %Stopping criteria
    if(abs(l2norm(k-1) - l2norm(k)) < 1e-5)
        break;
    end
end
xq = x(:,k);
tq = cputime - t;
```

BFGS Update for S_k:

```
function BFGS = BFGS(A,u,S)
    Au = A*u;
    denom = norm(Au)^2;
    SAtAu = S*(A'*Au);
    SAtAuut = SAtAu*u';
    uutAtAS = SAtAuut';
    numer = (Au'*A)*SAtAu;
    BFGS = S + (1 + numer/denom)*(u*u')/denom - (SAtAuut + uutAtAS)/denom;
end
```

Armijo Line Search for β:

```
function b = stepsize(An,b,y,g_0,Sn_dfy,dgy)
    %Line search parameters
    tau = 0.2;
    sigma = 0.1e-4;
    s = 0.7;
    m = 0;

    while(1)
        beta = (s^m)*sigma; %Beta update
        ga = y - beta*(Sn_dfy);
        %ga(ga<0) = 0;
        g_1 = 0.5*(norm(An*ga-b)^2);
        condition = g_0 + tau*(dgy'*(ga - y));

        %Checks for the condition
        if(g_1<condition)
            break;
        end
        m = m+1;
    end
    b = beta;
end
```

APPENDIX D

RANDOMIZED ALGORITHM CODE

(Our Implementation in MATLAB)

```
[m,n] = size(A);
%Padding the rows with 0s to get power of 2
pad = ceil(log2(m));
m2 = 2^pad;
A(m+1:m2,:) = zeros(m2-m,n);
b(m+1:m2) = zeros(m2-m,1);
[n,d] = size(A);
r = d+20;  %Setting r value (r = d+20 is best as per Boutsidis et al. [4])
tstart = cputime;

%D matrix
dd = 2*randi(2,1,n)-3;   %assigns +1 or -1 with equal probability

%S matrix
s = zeros(1,n);
s(randsample(n,r)) = sqrt(n/r);
ind = find(s~=0); %used to make mat multiplication easier

%Hadamard Submatrix
H = (1/sqrt(n))*hadamard(n); %multiply with * to normalize it

%SHD product
i = 1:n;
i = i(ind);
for i=i
    H(i,:) = H(i,:)*s(i);
end
H = H(ind,:);

HD = zeros(size(H));
for j=1:size(H,2)
    HD(:,j) = H(:,j)*dd(j);
end

%Induced system
An = HD*A;
bn = HD*b;
tprep = cputime-tstart;

tnew = cputime;
x_rand = quasi(An,bn);  %Calling the Quasi-Newton NNLS solver
tsmall = cputime - tnew;
```