AMSC 663/664 Final Report

Memory Efficient Signal Reconstruction from Phaseless Coefficients of a Linear Mapping

Naveed Haghani
nhaghan1@math.umd.edu

Project Advisor:
Dr. Radu Balan
rvbalan@cscamm.umd.edu
Professor of Applied Mathematics, University of Maryland
Department of Mathematics
Center for Scientific Computation and Mathematical Modeling
Norbert Wiener Center

Table of Contents

Table of Figures

Introduction

Background

A recurring problem in signal processing involves signal reconstruction using only the magnitudes of the coefficients of a linear transformation. This problem has applications in the fields of speech processing and x-ray crystallography. In speech processing, it is common to work with a speech signal's spectrogram. Working with the spectrogram provides the ability to perform various audio manipulations. The challenge then becomes to retrieve a processed signal's discrete-time signal, as the spectrogram does explicitly carry any phase information with regards to the signal. In x-ray crystallography, the diffraction pattern of an x-ray beam will deliver the magnitudes of a transformed signal of electron density levels. Obtaining the desired electron density information requires the phaseless retrieval of the original signal.

The project depicted in this paper implements and tests an iterative, recursive least squares algorithm described in Balan[5] to perform phaseless reconstruction from the magnitudes of the coefficients of a linear transformation. Testing is done on synthetically generated input data created using random number generation. A random input vector is generated and passed through a transformation algorithm. The transformed signal is then passed to the iterative, recursive least squares algorithm to reconstruct the original signal. Following that, post-processing is done on the results.

The implementation is programmed in MATLAB. The implementation will be designed to prioritize memory efficiency. Memory efficiency, in this regard, applies primarily to the storage of the resulting linear system involved in reconstruction. The linear system will be on the order of $1{,}000 \times 1{,}000$. Avoiding the costly storage of this system and deriving its contents when needed will be the primary focus during implementation of the algorithm. Following the algorithm's completion, the program's performance is studied with regards to time efficiency, accuracy, and scalability with problem size.

Problem Setup

Given an n-dimensional complex signal, $x \in \mathbb{C}^n$, that has been passed through a redundant linear transformation, $T(x)$, the objective is to reconstruct x from the element by element squared modulus of the transformed signal. The transformed signal will be labeled as follows:

$$T(x) = c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_m \end{bmatrix} \in \mathbb{C}^m \tag{1}$$

The transformed signal lies in the m dimensional complex space, where $m = R \cdot n$. R here represents the level of redundancy in the transformation $T(x)$.

The element by element squared modulus of c is represented by α:

$$\alpha = \begin{bmatrix} |c_1|^2 \\ |c_2|^2 \\ \vdots \\ \vdots \\ |c_m|^2 \end{bmatrix} \epsilon \, \mathbb{R}^m \tag{2}$$

c has been transformed into the real space to produce α. Since it lies in the real space, α does not carry any phase information of the original signal, fitting the criterion for phaseless reconstruction.

The resulting vector will be passed into the iterative, recursive least square algorithm, but not before adding a variable amount of Gaussian noise. The resulting input to the algorithm is labeled y and is defined by:

$$y = \alpha + \sigma \cdot \nu \tag{3}$$

Where ν is random noise drawn from a standard normal distribution and σ is the desired standard deviation. y is the vector of transformation magnitudes with simulated noise. The iterative, recursive least squares algorithm will use the input y to produce an approximation of x, labeled $\hat{x}$. The entire process works as follows:

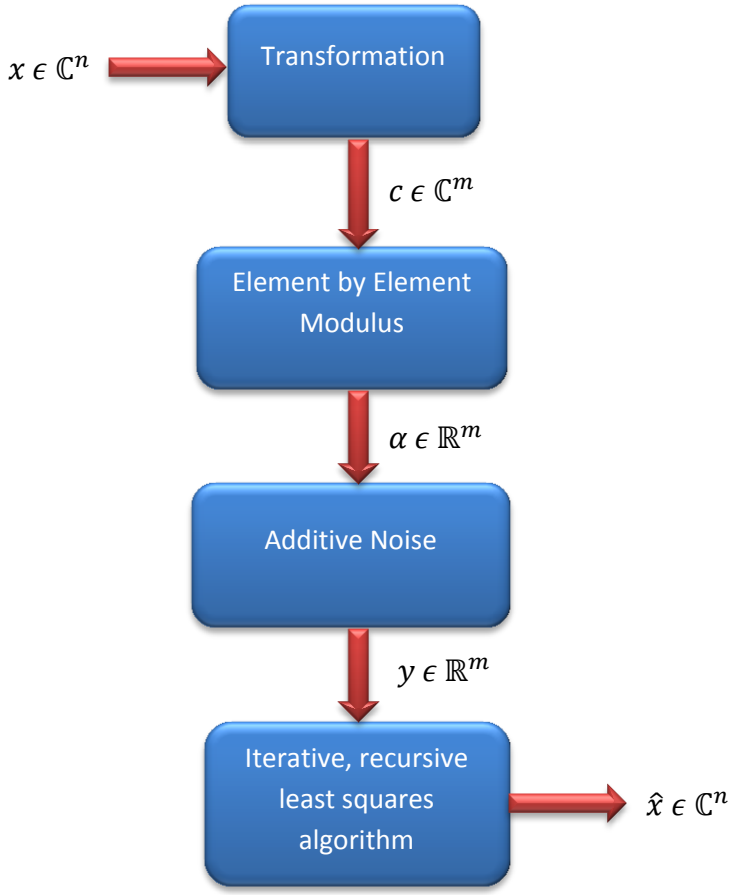

Figure 1

After $\hat{x}$ is obtained, the estimation is passed to post-processing to study certain output trends with regards to varying signal to noise ratios in y.

Transformation

The transformation used in the implementation is a weighted discrete Fourier transform. In the transformation, each element of x is first multiplied by a complex weight, w_i. Then the discrete Fourier transform is taken on the resulting vector. This is repeated R times, each time with an independent set of weights.

$$B_j = Discrete\ Fourier\ Transform\left\{\begin{bmatrix} w_1^{(j)} & & 0 \\ & \ddots & \\ 0 & & w_n^{(j)} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}\right\}, for\ j = [1, R] \tag{4}$$

The resulting transformation output c is a composite of each of the $B_{[1,R]}$ transformations, making c lie in the $m = R \cdot n$ complex dimensional space.

$$c = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ \vdots \\ B_R \end{bmatrix} \epsilon\, \mathbb{C}^m \tag{5}$$

The transformation, $T(x)$, can also be defined in terms of m unique frame vectors of length n labeled $f_{1:m}$. In such case, the transformation would be the composite of the scalar product of the input singal, x, with each of the m frame vectors:

$$T(x) = \begin{bmatrix} \langle x, f_1 \rangle \\ \langle x, f_2 \rangle \\ \vdots \\ \langle x, f_m \rangle \end{bmatrix} \tag{6}$$

Where the scalar product of two complex vectors, a and b, of length n is defined as:

$$\langle a, b \rangle = \sum_{i=1}^{n} a_i \cdot \bar{b}_i \tag{7}$$

For the case of the weighted discrete Fourier transform, the frame vector formulation for $T(x)$ would be:

$$f_k = conj\left\{\frac{1}{\sqrt{R \cdot n}}\begin{bmatrix} w_1^{(j)} \cdot 1 \\ w_2^{(j)} e^{-i2\pi r \cdot \frac{1}{n}} \\ \vdots \\ \vdots \\ w_n^{(j)} e^{-i2\pi r \cdot \frac{n-1}{n}} \end{bmatrix}\right\} for\ k = [1, m] \quad (8)$$

$$where\ \ j = ceiling\left(\frac{k}{n}\right), r = (k-1)(mod\ n), and\ i = \sqrt{-1}$$

After c is obtained from the weighted discrete Fourier transform, α is obtained by taking the modulus squared of each element of c. Finally, Gaussian noise is added to α to produce y, the input to the iterative, recursive least squares algorithm.

Algorithm

The reconstructive algorithm to be implemented has been introduced and described in Balan[5]. It consists of two primary processes, the initialization and the iterative solver. The algorithm serves as a least squares solver that is designed to minimize $\|y - \hat{\alpha}\|^2$, where $\hat{\alpha}$ is the α value in equation (2) obtained from inputting the current estimation, $\hat{x}$, into the preprocessing transformation.

Initialization

Initialization starts with finding the principal eigenvalue, a_1, and its associated eigenvector, e_1, of a matrix Q defined by:

$$Q = \sum_{k=1}^{m} y_k f_k f_k^* \ \text{[5]} \quad (9)$$

Where f_k, defined earlier, is the k[th] frame vector of $T(x)$.

Before the principal eigenpair is retrieved, the following modification is performed on Q:

$$Q^+ = Q + q \cdot I \quad (10)$$
$$where\ q = \|y\|_\infty, \qquad I = identity$$

This modification ensures that Q^+ is positive definite, subsequently ensuring that the power method for finding the principal eigenvector will converge.

Once this eigenpair is discovered the first estimation, $\hat{x}^{(0)}$, can be initialized as [5]:

$$\hat{x}^{(0)} = e_1 \sqrt{\frac{(1-\rho) \cdot a_1}{\sum_{k=1}^{m} |\langle e_1, f_k \rangle|^4}} \quad (11)$$

$$where\ \rho\ is\ a\ constant\ bewteen\ 0\ and\ 1$$

Two additional parameters, μ and λ, are initialized as [5]:

$$\mu_0 = \lambda_0 = \rho \cdot a_1 \tag{12}$$

After initialization, the algorithm moves on to the iterative process.

Iteration

Through each pass of the iterative process a linear system is solved to obtain a new approximation $\hat{x}$. The linear system is constructed in the real space. Instead of working with $\hat{x}$, the algorithm works with $\xi = \begin{bmatrix} real(\hat{x}) \\ imag(\hat{x}) \end{bmatrix}$, the composite of the real values of $\hat{x}$ and the imaginary values of $\hat{x}$. The linear system which is symmetric and positive definite is defined as:

$$A\xi^{(t+1)} = b \tag{13}$$

$$where\ \ A = \sum_{k=1}^{m} \left(\Phi_k \xi^{(t)}\right) \cdot \left(\Phi_k \xi^{(t)}\right)^* + (\lambda_t + \mu_t) \cdot I \text{ [5]} \tag{14}$$

$$b = \left(\sum_{k=1}^{m} y_k \cdot \Phi_k + \mu_t \cdot I\right) \cdot \xi^t \text{ [5]} \tag{15}$$

$$\Phi_k = \varphi_k \varphi_k^T + J\varphi_k \varphi_k^T J^T,\ where\ \ \varphi_k = \begin{bmatrix} real(f_k) \\ imag(f_k) \end{bmatrix}\ and\ J = \begin{bmatrix} 0 & -I \\ I & 0 \end{bmatrix} \text{ [5]}$$

$\xi^{(t)}$ is the composite of the real and imaginary components of the current approximation $\hat{x}^{(t)}$, and $\xi^{(t+1)}$ is the composite of the real and imaginary components of the next approximation $\hat{x}^{(t+1)}$.

Following that, the parameters are updated for the following iteration:

$$\lambda_{t+1} = \gamma \lambda_t \text{ [5]} \tag{16}$$

$$\mu_{t+1} = max\left(\gamma \mu_t, \mu^{min}\right) \text{ [5]} \tag{17}$$

$$where\ 0 < \gamma < 1$$

This process is repeated until the following stopping criterion is met:

$$\sum_{k=1}^{m} \left| y_k - \left|\langle \hat{x}^{(t)}, f_k \rangle\right|^2 \right|^2 > \sum_{k=1}^{m} \left| y_k - \left|\langle \hat{x}^{(t-1)}, f_k \rangle\right|^2 \right|^2 \tag{18}$$

This stopping criterion is essentially checking whether $\|y - \hat{\alpha}\|^2$ is below a given tolerance.

The flow of the algorithm is represented in figure (2) below.

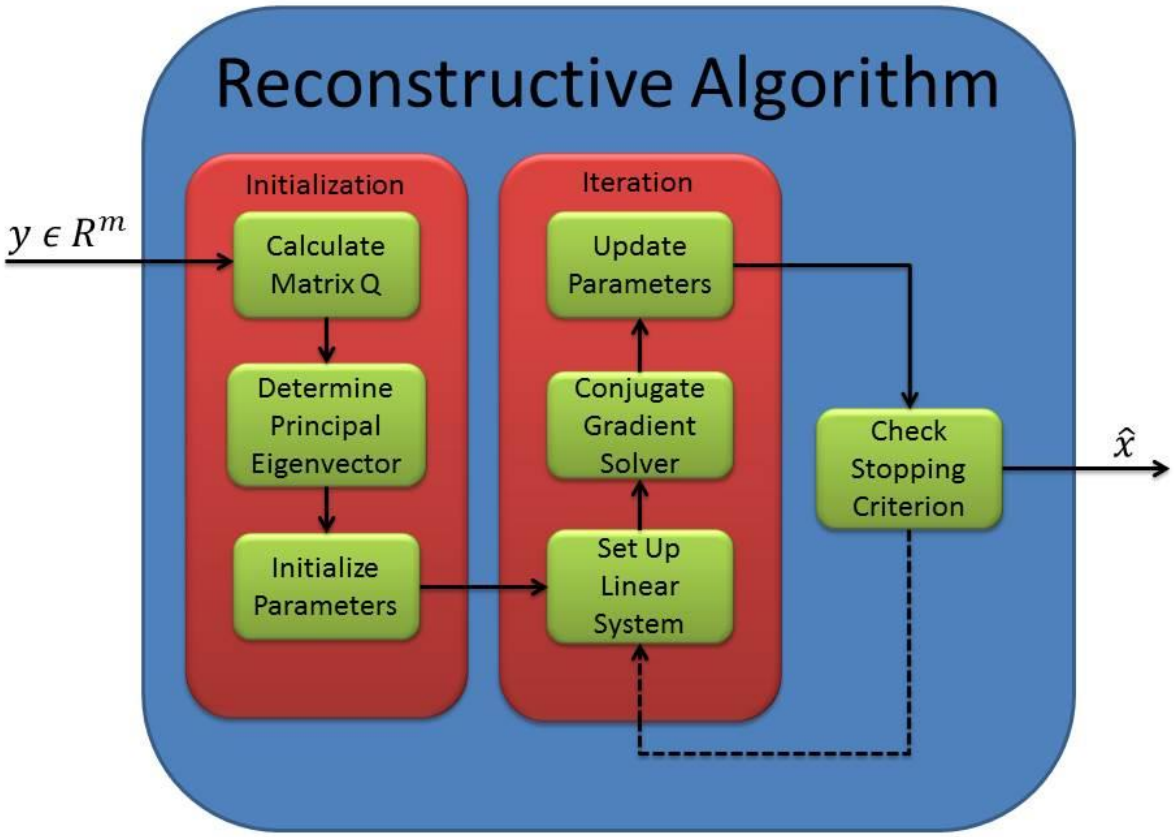

Figure 2

A transformed vector y is inputted to the algorithm. The algorithm runs through the initialization phase then to the iterative phase. After each pass of the iterative phase, a stopping criterion is checked. If the criterion is met, the algorithm delivers its current approximation, otherwise the iterative phase is repeated.

Memory Efficient Implementation

The formulations described are dependent on the frame vector representation, $f_{1:m}$, of the transformation $T(x)$. This is an nxm complex matrix. For large n, for example $n{\sim}10{,}000$, the storage of these frame vectors is very costly. Furthermore, the matrix Q required in the initialization phase is an nxn complex matrix, and the matrix A required in the iterative phase is a $2nx2n$ real matrix. The storage of these matrices would also be very costly for large problem sizes.

Avoiding such large storage requirements is critical for implementation on large problem sizes. Therefore, the variables described so far have been reformulated in terms of the transformation $T(x)$

instead of its associated frame vectors. Where the transformation $T(x)$ is required, its formulation represented by the fast Fourier transform will be used instead of the frame vectors. This will avoid, altogether, the storage of $f_{1:m}$.

For the Q^+ matrix required in the initialization phase, the matrix times a given vector u can be reformulated as:

$$Q^+ \cdot u = T^*\big(y .* T(u)\big) + \|y\|_\infty \cdot u \tag{19}$$

For the A matrix required in the iterative phase, the matrix-vector product of the matrix A and a given vector u can be redefined as:

$$A \cdot u = \begin{bmatrix} Re\left\{T^*\left(real\left\{T(u) .* conj\{T(x^{(t)})\}\right\} .* T(x^{(t)})\right) + (\lambda+\mu)\cdot u\right\} \\ Im\left\{T^*\left(real\left\{T(u) .* conj\{T(x^{(t)})\}\right\} .* T(x^{(t)})\right) + (\lambda+\mu)\cdot u\right\} \end{bmatrix} \tag{20}$$

And the right hand side, b, of the linear system in the iterative phase can be reformulated as:

$$b = \begin{bmatrix} Re\left\{T^*\left(y .* T(x^{(t)})\right) + \mu \cdot x^{(t)}\right\} \\ Im\left\{T^*\left(y .* T(x^{(t)})\right) + \mu \cdot x^{(t)}\right\} \end{bmatrix} \tag{21}$$

T^* in the given formulations represents the associated adjoint of the transformation $T(x)$. It is implemented as:

$$T^*(c) = \sum_{k=1}^{R} \frac{1}{\sqrt{R \cdot n}} \cdot n \cdot \overline{w^k} \cdot ifft\big(c_{(k-1)\cdot n+1:k\cdot n}\big) \tag{22}$$
$$where\ ifft\ represents\ the\ inverse\ Fourier\ transform$$

The given formulations have no dependence on the frame vector representation of $T(x)$. Furthermore, since the formulations produce products for $Q^+ \cdot u$ and $A \cdot u$, the matrices Q and A do not require storage either. In this case, however, the principal eigenvalue of Q^+ and the linear system involving A must both be solved without their explicit formulations. This will be done using the power method for determining the principal eigenvalue of Q^+ and the conjugate gradient method for solving the linear system involving A.

Implementation

Data Creation

The complex input vector $x \in \mathbb{C}^n$ will be generated synthetically using random number generation. Each element of x will consist of a randomly generated normal component and a randomly generated imaginary component. Both random numbers will be distributed normally about 0 with variance 1. 5 different realizations of x will be generated and saved for repeated use.

The weights used in the weighted transformation will also be synthetically generated using random number generation. Each element of w will have a random normal component and a random imaginary component, each distributed normally about 0 with variance 1. There will be 5 different realizations of each set $w^{([1,R])}$.

The noise, v, added to α to produce y will be generated randomly as well. Each element will be distributed normally about 0 with variance 1. There will be 1,000 different realizations of noise, v.

Principal Eigenvalue (Initialization)

During the initialization stage of the iterative, recursive least squares algorithm, the principal eigenvalue of a matrix Q must be obtained. To achieve this, the power method for obtaining the principal eigenvector will be used. The power method starts with an initial approximation of the associated eigenvector, $e^{(0)}$. For the purposes of this implementation, $e^{(0)}$ will be set to an array of random numbers. Each element will be distributed normally about 0 with variance 1.

From $e^{(0)}$ the algorithm will repeat as follows:

$$Repeat\colon e^{(t+1)} = \frac{Q^+ \cdot e^{(t)}}{\|Q^+ \cdot e^{(t)}\|}$$

$$where\ e^{(t)}\ is\ the\ current\ approximation, e^{(t+1)}\ is\ the\ following\ approximation$$

$$stop\ when\ \left\|e^{(t+1)} - e^{(t)}\right\| < tolerance$$

With the selection of an appropriate tolerance, this algorithm should produce an adequate approximation for the principal eigenvector, e_1, of the matrix Q^+. The associated eigenvalue a_1 is then calculated for the unmodified matrix Q. It is calculated by the equation:

$$a_1 = \frac{\|Q^+ \cdot e_1\|}{\|e_1\|} - \|y\|_\infty$$

Conjugate Gradient (Iteration)

Through each iteration of the iterative, recursive least squares algorithm, a $2n \times 2n$ linear system must be solved. This would be cumbersome to solve exactly and would jeopardize the priority of memory efficiency. Instead, the conjugate gradient method of solving linear systems will be used. The conjugate gradient method is an iterative method for solving symmetric, positive definite linear systems, and since A is a symmetric, strictly positive matrix whose lowest eigenvalue is bounded below by $\lambda_t + \mu_t$, the conjugate gradient method can be used to solve the linear system $A \cdot \xi^{t+1} = b$.

The conjugate gradient method works by taking the residual of an approximate solution to a linear system and reducing it by moving the solution along several different conjugate directions. Two vectors p^1 and p^2 are considered to be conjugate with respect to a matrix A if they satisfy the following condition:

$$p^{1^T} \cdot A \cdot p^2 = 0$$

For a given matrix in $\mathbb{R}^n$, there are always n linearly independent conjugate directions. Traveling along all directions produces the exact solution to the system. However, if during that time the approximation converges to within a given tolerance of the solution, the process can be concluded at that time with a sufficiently close approximation.

The algorithm will be initialized as [7]:

$$r^{(0)} = b - A\hat{x}^{(0)}$$

$$p^{(0)} = r^{(0)}$$

Where $\hat{x}^{(k)}$ is the approximate solution at the kth iteration, $r^{(k)}$ is the residual at the kth iteration, and $p^{(k)}$ is the kth conjugate direction. $\hat{x}^{(0)}$ is initialized to the current approximation of the iterative, recursive least squares algorithm, represented by $\xi^{(t)}$.

Each iteration repeats as [7]:

$$repeat: \alpha = \frac{\langle r^{(k)}, r^{(k)} \rangle}{{p^{(k)}}^T A p^{(k)}}$$

$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \alpha p^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \alpha A p^{(k)}$$

$$p^{(k+1)} = r^{(k+1)} + p^{(k)} \frac{\langle r^{(k+1)}, r^{(k+1)} \rangle}{\langle r^{(k)}, r^{(k)} \rangle}$$

$$until \left\| r^{(k)} \right\|^2 < tolerance$$

In each iteration, the solution moves along the conjugate direction $p^{(k)}$ a distance α. The iterations are repeated until the magnitude of the residual of the current approximation is less than a given tolerance.

Coding

The entire algorithm from preprocessing through post-processing is programmed in MATLAB. Implementing the discrete Fourier transform is done using MATLAB's $fft()$ command. $fft()$ implements a fast Fourier transform. Random numbers are generated using MATLAB's $randn()$ command, which generates random variates according to a standard normal distribution. Common random numbers are used to generate the random noise vectors for each instance. Seeding is controlled using MATLAB's $rng()$ command.

The function performing the iterative recursive least squares algorithm is labeled $LS\_Algorithm()$. A hierarchy of the function and its subfunctions is shown in figure (3) below.

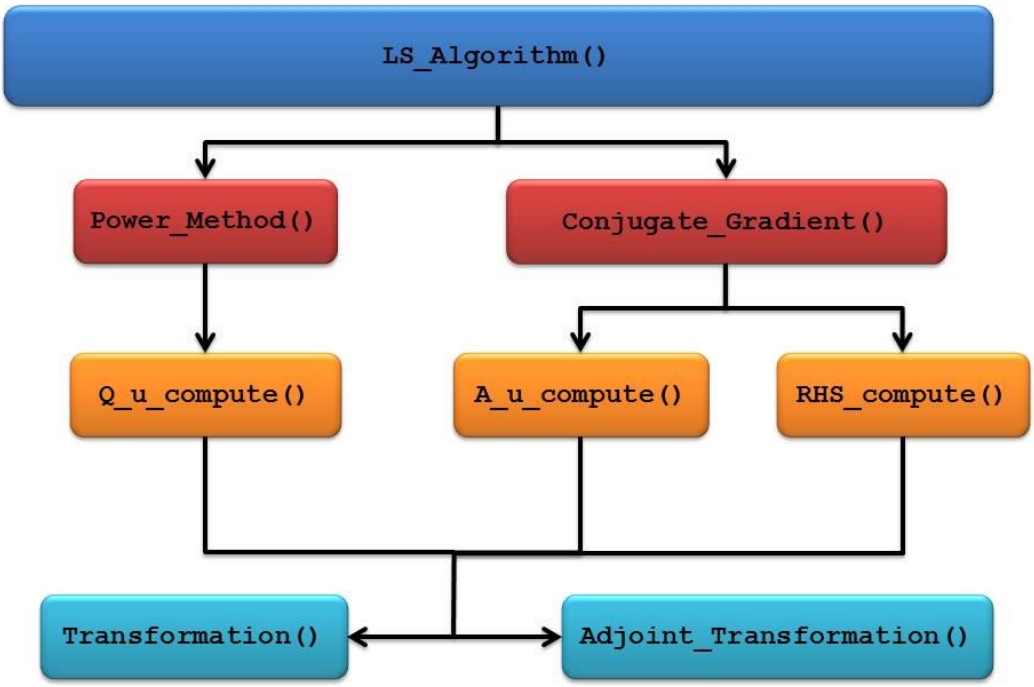

Figure 3

$LS\_Algorithm()$ is called to perform the iterative recursive least squares algorithm. Within it, there are calls to $Power\_Method()$ and $Conjuagte\_Gradient()$. $Power\_Method()$ uses the power method to determine the principal eigenvector of the Q^{+} matrix. Within $Power\_Method()$ there is a call to $Q\_u\_compute()$, which performs the formulation in equation (19). $Conjuagte\_Gradient()$ employs the conjugate gradient method to solve the linear system described in equation (13). It includes calls to both $A\_u\_compute()$, which calculates the result of equation (20), and $RHS\_compute()$, which

calculates the result of equation (21). $Q\_u\_compute()$, $A\_u\_compute()$, and $RHS\_compute()$ all perform calls to $Transformation()$ and $Adjoint\_Transformation()$ to compute their results.

Reconstruction is performed for each noise realization of each signal to noise ratio of a transformed signal. Parallelization is applied over calls to the reconstructive algorithm. MATLAB's $parfor$ is used to employ a parallel for-loop over all noise realizations for a given signal to noise ratio. To ensure common random numbers are being used, independent seeding is used for each noise realization. The random number seed for a given noise realization is set to the index of that particular noise realization. So the seed for first noise realization would be one, two for the second, and so on.

Parameters

Below is a list of required parameters and the associated values they will be set to during large scale testing.

$\boldsymbol{n}$	Signal size	1,000
$\boldsymbol{L}$	Number of noise realizations	1,000
$\boldsymbol{R}$	Transformation redundancy	8
$\boldsymbol{\gamma}$	Rate of convergence parameter	0.95
$\boldsymbol{\rho}$	Initialization weight for regularization parameters	0.9
$\boldsymbol{PM\ tol}$	Power Method stopping tolerance	10^{-8}
$\boldsymbol{CG\ tol}$	Conjugate Gradient stopping tolerance	10^{-14}
$\boldsymbol{\mu_{\min}}$	Minimum μ value	$\frac{\mu_0}{10}$

Post-processing

After a transformed signal has been passed through reconstruction, it is ready for post-processing. In post-processing several key metrics are assessed. The reconstructed outputs for all noise realizations for a given signal to noise ratio are used to calculate the bias of the output mean from the original signal, the variance of the output, and the mean squared error of the output. As well, the Cramer Rao Lower Bound is calculated for each signal to noise ratio. This is done for each input setup.

Due to the large potential number of noise realizations and the large signal length, storing the output of each noise realization is too costly. To calculate the desired metrics, however, all that is needed is the vector sum of the output signal over all noise realizations and the vector sum of the modulo-squared elements of the output signal over all noise realizations. These values are labeled as x_sum and $x2\_sum$ respectively and are formulated as follows:

$$x\_sum = \sum_{k=1}^{L} \hat{x}^{(k)} \tag{23}$$

$$where\ L = \#\ of\ noise\ realizaions$$
$$and\ \hat{x}^{(k)}\ is\ the\ output\ of\ the\ k^{th} noise\ realization$$

$$x2\_sum = \sum_{k=1}^{L} \begin{bmatrix} \left|\hat{x}_1^{(k)}\right|^2 \\ \left|\hat{x}_2^{(k)}\right|^2 \\ \vdots \\ \vdots \\ \left|\hat{x}_n^{(k)}\right|^2 \end{bmatrix} \tag{24}$$

Using only these two vectors as the output reduces the output load to two n length vectors for each signal to noise ratio. Furthermore, all the desired metrics can be calculated in terms of these two sums. The bias is formulated as follows:

$$Bias = \left\| x - \sum_{i=1}^{L} \hat{x}^{(i)} \right\|^2 \tag{25}$$

$$bias\_vec = \frac{x\_sum}{L} - x \tag{26}$$

$$Bias = \sum_{i=1}^{n} |bias\_vec_i|^2 \tag{27}$$

The variance is formulated as follows:

$$Var = \frac{1}{L-1} \sum_{i=1}^{L} \left[\left\| \frac{1}{L} \sum_{k=1}^{L} \hat{x}^{(k)} - \hat{x}^{(i)} \right\|^2 \right] \tag{28}$$

$$Var = \frac{\sum_{i=1}^{n}\left(x2\_sum_i - \frac{|x\_sum_i|^2}{L}\right)}{L-1} \tag{29}$$

And the mean squared error is formulated as follows:

$$MSE = \frac{1}{L}\sum_{i=1}^{L}\left[\left\|x - \hat{x}^{(i)}\right\|^2\right] \tag{30}$$

$$MSE = \left(1 - \frac{1}{L}\right)\cdot Var + Bias \tag{31}$$

The Cramer Rao lower bound can be calculated from the inverse of a modified fisher information matrix, $\tilde{A}$, which is formulated as:

$$\tilde{A} = \sum_{k=1}^{m} \Phi_k \xi \xi^T \Phi_k^T + \frac{1}{\|x\|^2}\cdot J\xi\xi^T J^T \text{ [5]} \tag{32}$$

The Cramer Rao lower bound derived from the trace of $\tilde{A}^{-1}$:

$$CRLB = 10\cdot log_{10}\left(\frac{\sigma^2}{4}\cdot\left(trace\left(\tilde{A}^{-1}\right) - 1\right)\right) \tag{33}$$
$$where\ \sigma\ is\ the\ standard\ deviation\ of\ the\ noise$$

$trace\left(\tilde{A}^{-1}\right)$ is cumbersome to calculate directly from the definition of $\tilde{A}$. Instead $trace\left(\tilde{A}^{-1}\right)$ is calculated by the following:

$$trace\left(\tilde{A}^{-1}\right) = \sum_{k=1}^{2n}\left\langle \tilde{A}^{-1}e_k, e_k\right\rangle \tag{34}$$
$$where\ e_k\ is\ the\ k^{th}\ canonical\ basis\ vector$$

And $\tilde{A}^{-1}$ multiplied by a given vector u can be calculated by:

$$\tilde{A}\cdot u = A\cdot u + Jxi^T\cdot u\cdot Jxi \tag{35}$$

$$where\ Jxi = \frac{1}{\sqrt{\sum_k |x_k|^2}}\begin{bmatrix} -imag(x) \\ real(x)\end{bmatrix}$$

Here A is the same as in equation (14).

These calculations are done for each signal to noise ratio. Once they are all calculated the results are all plotted together against the signal to noise ratio. This is done for all input setups.

Validation

Method

Validation for the iterative, recursive least squares implementation can be done on the individual modules within the algorithm, including the power method implementation and the conjugate gradient implementation. Using a smaller sample data set with n on the order of 100 rather than 10,000, the power method can be substituted with MATLAB's $eig()$ function. $eig()$ will reliably deliver the principal eigenvalue that was sought after by the power method implementation. The power method implementation can then be run on the same sample data in order to compare the results. If the results are comparable, the power method module will be validated.

A similar procedure can be done for the conjugate gradient implementation. On a small data set with n on the order of 100, the conjugate gradient module can be substituted with MATLAB's $mldivide()$. $mldivide()$ will provide the exact solution to the linear system. This exact solution can be used to compare with the results obtained using the conjugate gradient implementation. Comparable results would provide validation. The conjugate gradient implementation can be further validated on large data sets as well. This is done by letting the conjugate gradient run through all possible iterations. For a system of size $n \times n$, the conjugate gradient method ensures absolute convergence to the true solution in n steps. Rather than returning a result within a certain tolerance, the implementation can be made to run through all iterations regardless. The result will serve as the true solution to validate against.

The memory efficient implementation represented by equations (19), (20), and (21) will programmed and referred to as the efficient implementation. It will be compared against the frame vector implementation formulated by equations (9), (14), and (15) which will be called the sample implementation. The results of these two implementations can be compared against one another for small, sample problem sizes of $n \sim 100$.

Results

The power method and conjugate gradient method were both programmed and validated as described. The conjugate gradient method reliably produces results comparable to MATLAB's $mldivide()$ with a slight amount of round-off error ($\sim 10^{-30}$) even when the conjugate gradient method is run through all iterations. The power method successfully produces the principal eigenvector as desired, however the eigenvector differs from the result of MATLAB's $eig()$ in that it is consistently off by a multiplicative complex constant. Both eigenvectors, though, are associated with the same eigenvalue, the principal eigenvalue.

Since the initial approximation of $\hat{x}$ is dependent on the principal eigenvector of Q, the initial approximations of the sample implementation and the efficient implementation are off by a multiplicative phase factor, the same phase factor by which the two eigenvectors differed. This constant difference perpetuates through all iterations of the least squares algorithm, thus making the final results

of the sample implementation and the efficient implementation equivalent in magnitude but off by a phase factor.

The results of the sample implementation and efficient implementation are compared for three data sets with $n = 100$. For each testing setup, the signal to noise ratio in y is set to $10\ dB$. The plot below shows each element of the output of both implementations plotted on the complex plane:

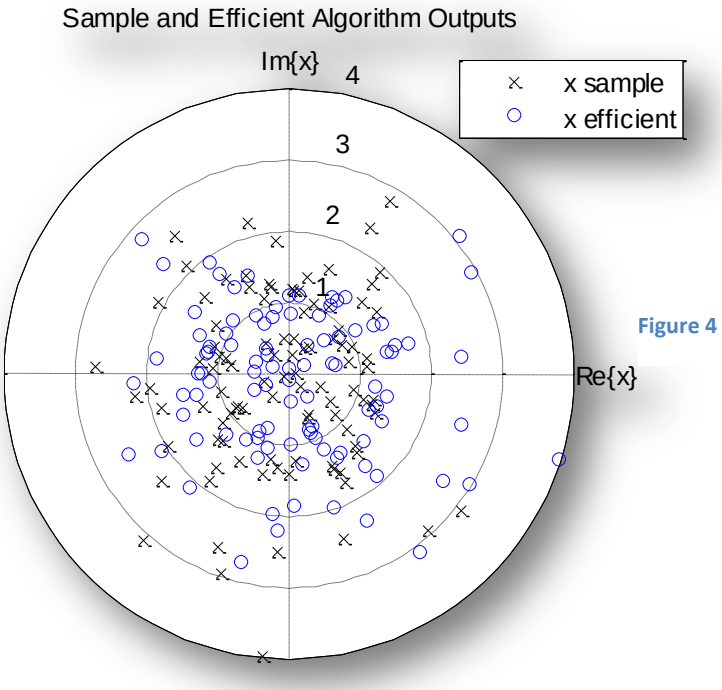

Figure 4

It can be seen that the outputs do not line up. The norm difference between the principal eigenvector of the efficient implementation and the principal eigenvector of the sample implementation was 1.2525. The phase difference between the eigenvectors of the two implementations was $e^{-i*1.3535}$. Shifting the output of the efficient implementation by this phase factor results in the following plot:

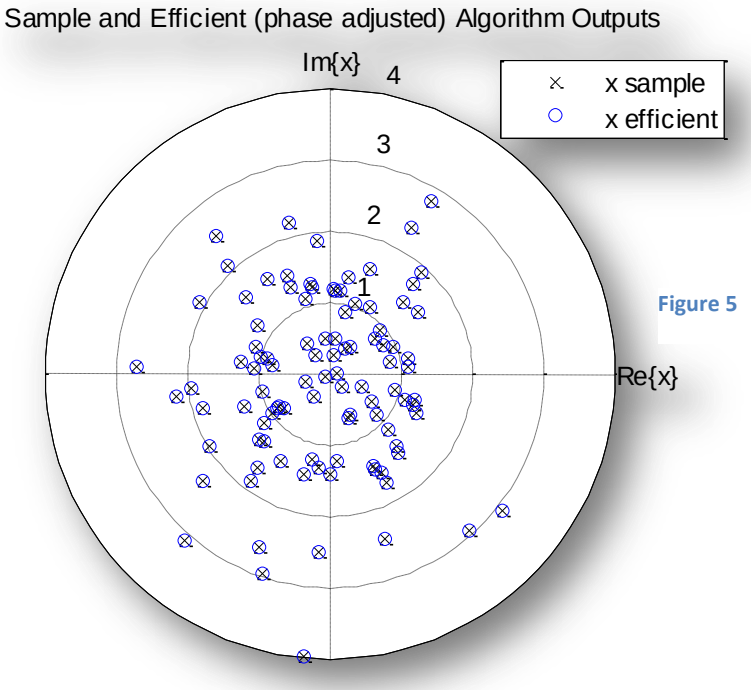

Figure 5

The outputs now line up, showing that the two outputs are only off by the phase factor introduced during the eigenvector retrieval in the initialization phase.

The same process is repeated for two more sample data sets with $n = 100$.

In this data set, the norm difference in the principal eigenvectors of Q was 0.3665 and the phase difference was $e^{i*0.3686}$. After a phase shift in the output of the efficient implementation, the outputs line up.

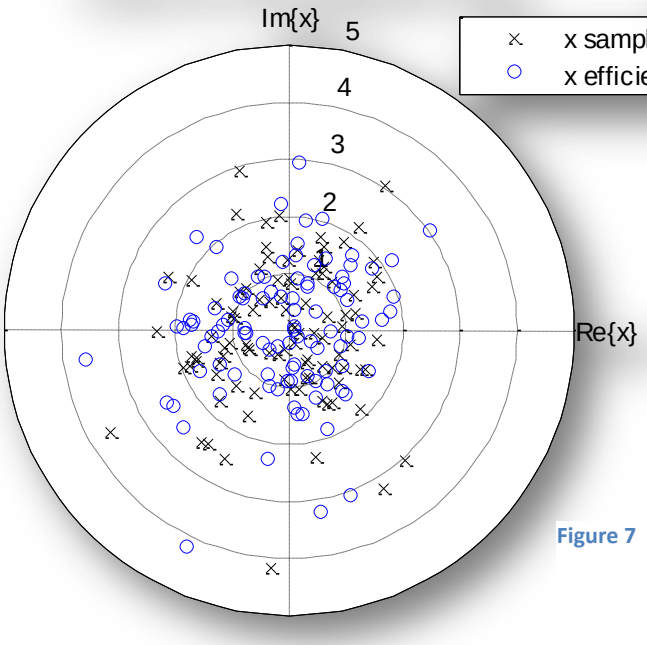

Figure 7

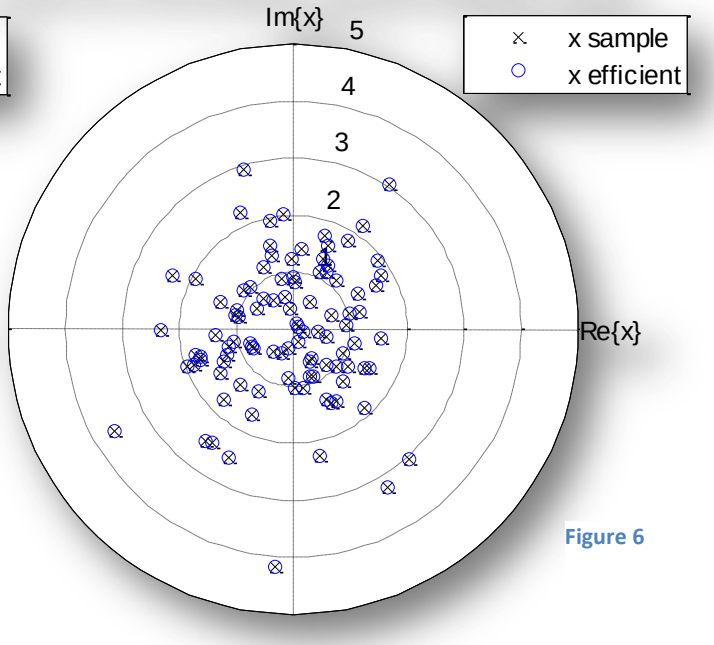

Figure 6

In this data set, the norm difference in the principal eigenvectors of Q was 0.9232 and the phase difference was $e^{-i*0.9596}$. Once more, after a phase shift in the output of the efficient implementation, the outputs line up.

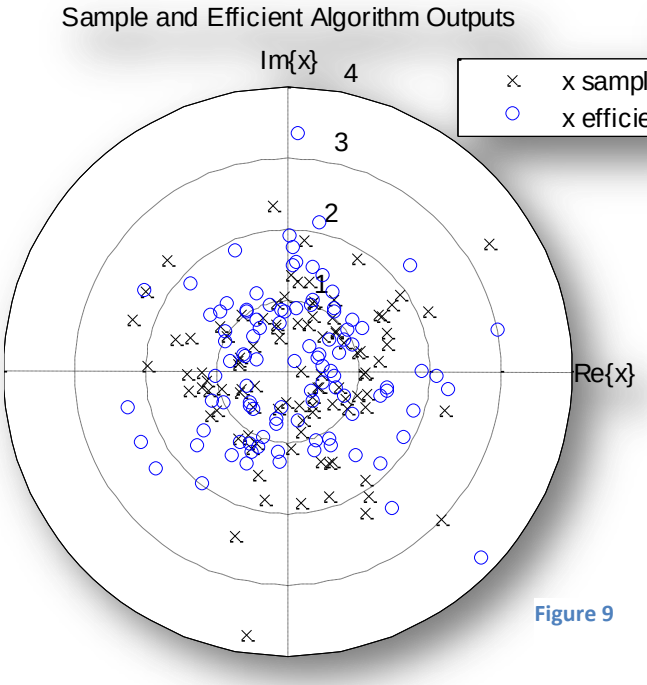

Figure 9

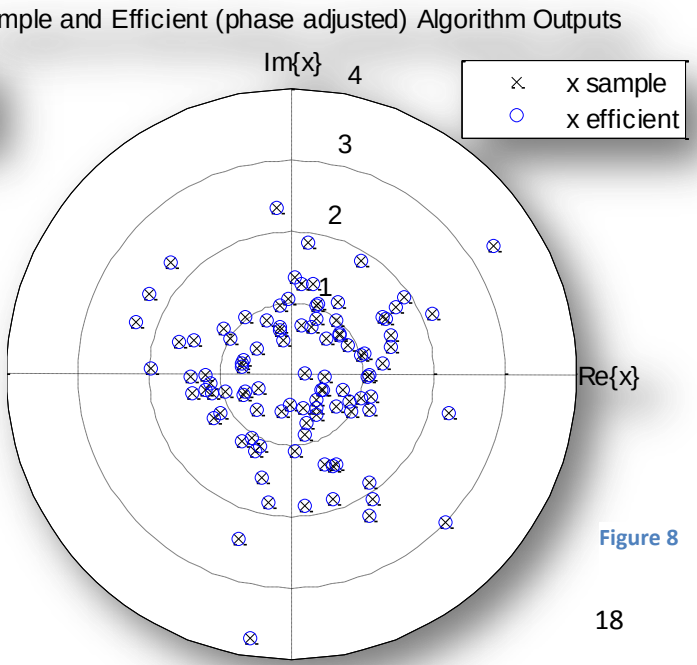

Figure 8

Testing

Preliminary Testing

A few preliminary tests are run on the resulting program to study certain aspects of its behavior. First, a visual look was taken at the output of the efficient implementation for a problem size of $n = 10{,}000$ and a signal to noise ratio of $10\ dB$. The magnitudes of the approximation $\hat{x}$ and the original signal x are plotted alongside one another for each of the n elements. A small portion of the plot is shown in the figure below.

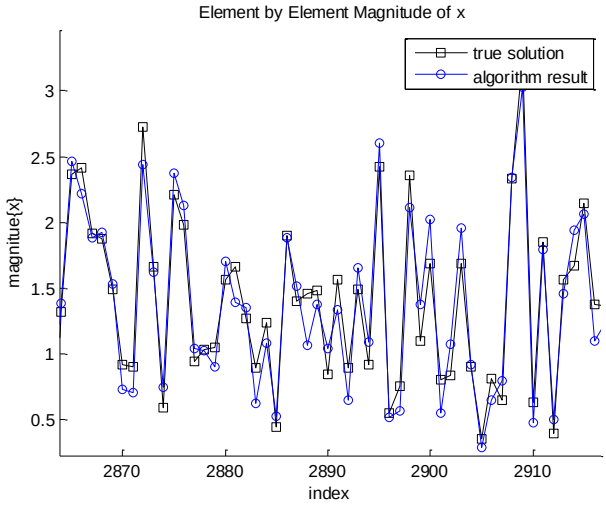

Figure 10

The approximate solution $\hat{x}$ is represented in blue while the original signal x is represented in black. For 10 dB signal to noise the approximate signal follows the same trend as the true solution very closely. Below is a plot of the same results over the portion containing the point with the highest inaccuracy.

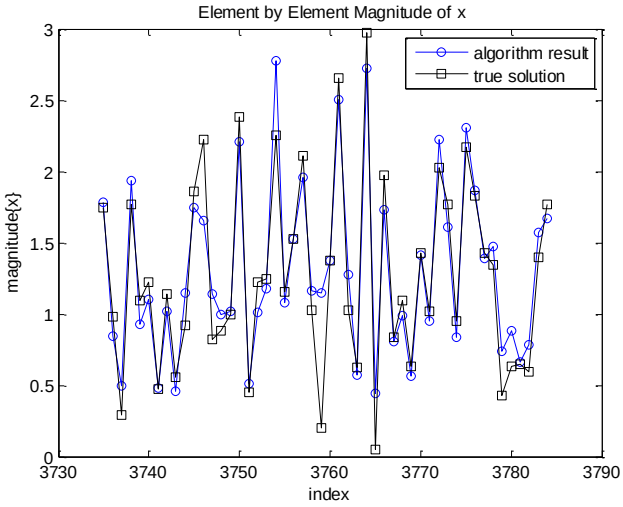

Figure 11

The highest inaccuracy occurs at an index around 3760.

Power Method and Conjugate Gradient Tolerances

Next the convergence properties of the power method and the conjugate gradient method were studied. For a given dataset with $n = 10{,}000$, both methods were run for various stopping tolerances and the number of iterations required to reach completion was recorded. The resulting output for the power method is shown below.

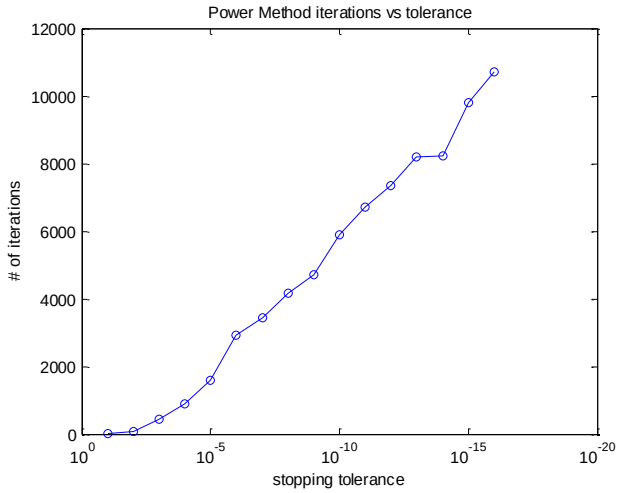

Figure 12

The stopping tolerance is plotted on the horizontal axis on a logarithmically decreasing scale. The power method requires a considerable amount of iterations. For a stopping tolerance of 10^{-10} the power method requires about $6{,}000$ iterations. From the graph it can be concluded that the error in the power method decays exponentially as the number of iterations is increased. In other words, increasing the number of iterations provides diminishing returns in the accuracy of the power method.

The required iterations of the conjugate gradient method were also plotted against its stopping tolerance. The resulting plot is shown below.

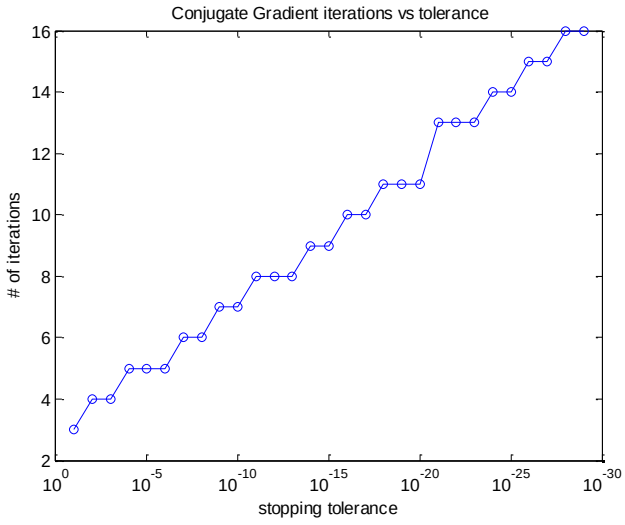

Figure 13

The conjugate gradient method requires very few iterations and converges to high accuracy very quickly. For a stopping tolerance of 10^{-20} the conjugate gradient method requires 11 iterations. This is beneficial because the conjugate gradient method is called numerous times through the execution of the least squares algorithm, once through each pass of the iterative phase. The trend of the error of the conjugate gradient method is similar to that of the power method in that the error also decays exponentially as the number of iterations is increased. However, the conjugate gradient method's error decays much more rapidly than the error in the power method.

Looking at the number of iterations for the power method to converge for various tolerance levels, it is useful to know how these different tolerance levels would affect the final algorithm output. To study this effect, the error, defined in equation (18), was recorded at each iteration through the course of the iterative portion of the algorithm. This was done for power method tolerances of 1e-14, 1e-12, 1e-10, and 1e-8. The error trends for each of the tolerances are plotted together below for $n = 10{,}000$ and a signal to noise ratio of $-30\ dB$.

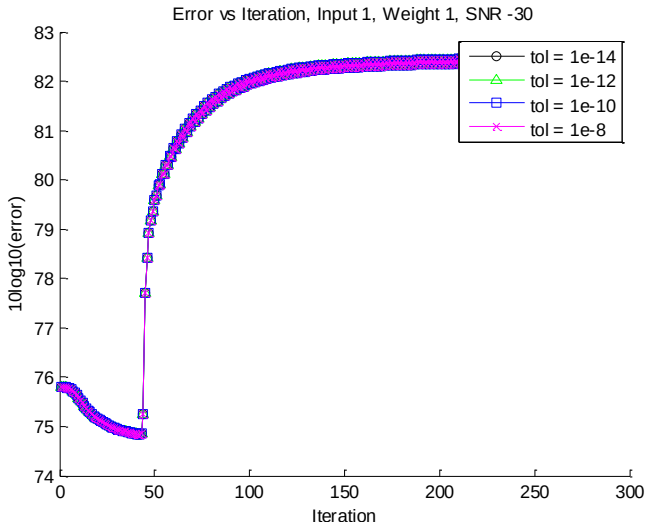

Figure 14

It can readily be seen that the error trends line up almost exactly. Not only does reducing the power method tolerance from 1e-14 to 1e-8 not affect the minimum error achieved through the course of the algorithm, it does not affect the trend in the error through the course of the succeeding iterations. Results for a signal to ratio of $0\ dB$ are show below.

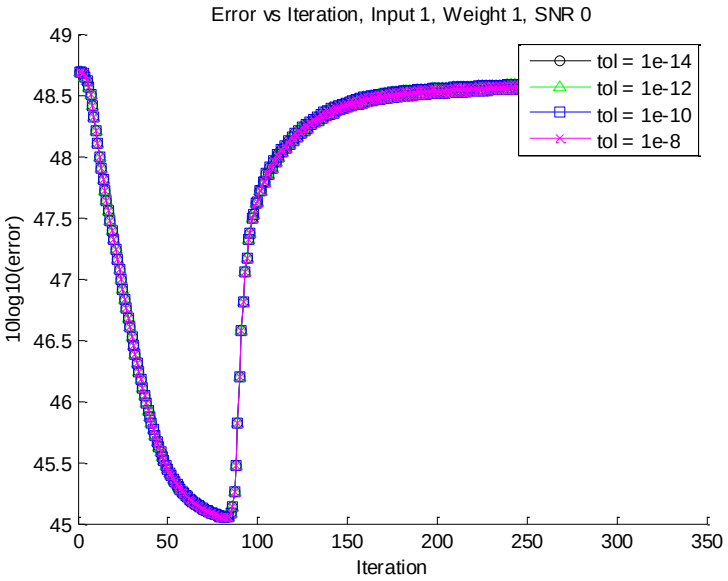

Figure 15

The same consistency is shown for $0\ dB$. The results for a signal to noise ratio of $30\ dB$ are shown below.

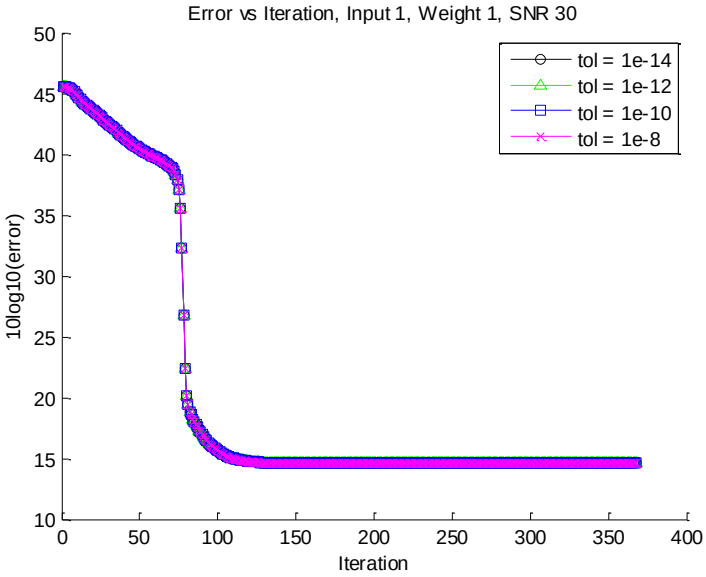

Figure 16

Again there is consistency across all the tolerances tested. By this result, it can be concluded that there is no notable loss in algorithm accuracy when adjusting the power method tolerance between 1e-8 and 1e-14. Furthermore, in figure (12) it was shown that the power method requires almost double the number of iterations to converge when increasing the tolerance from 1e-8 to 1e-14. Therefore, it is beneficial to use a power method tolerance of 1e-8 to reduce program runtime with no notable loss in the accuracy of the final approximation.

From the plots, the stark difference in error trends can be noted for varying signal to noise ratios. It is evident that higher signal to noise ratios require a greater number of iterations for the program to converge to its optimal value. For lower signal to noise ratios, the algorithm does not attain nearly as much improvement from initialization. It converges to its optimal value quickly and then diverges in the subsequent iterations. As the signal to noise ratio is increased, this process is prolonged. The algorithm takes longer to achieve its optimal value and then takes longer to diverge from there afterwards.

Memory Load

One of the primary goals in writing the efficient implementation is memory efficiency. Therefore, the memory load of the efficient implementation is compared against the memory load of the sample implementation at corresponding parts in the algorithm. MATLAB's $whos()$ function was used to track all the variables in a function's workspace at a given time. $whos()$ delivers the memory requirements of each variable stored. Summing up all the load of each variable will produce to total memory load of the program at a specific time. Both implementations were studied for a problem of size $n = 100$. The memory load was checked at the end of one iteration of the iterative phase of the least squares algorithm. A visual representation of where the load was examined is shown below.

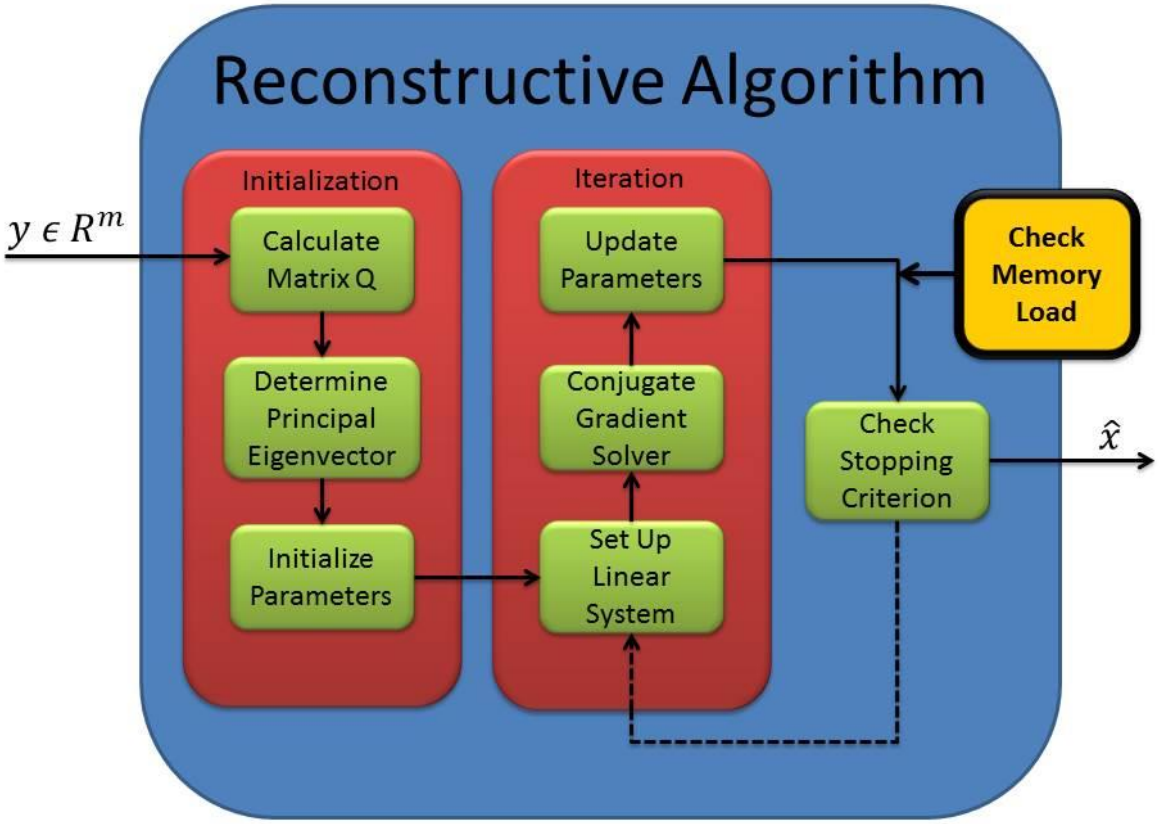

Figure 17

The results are graphed below.

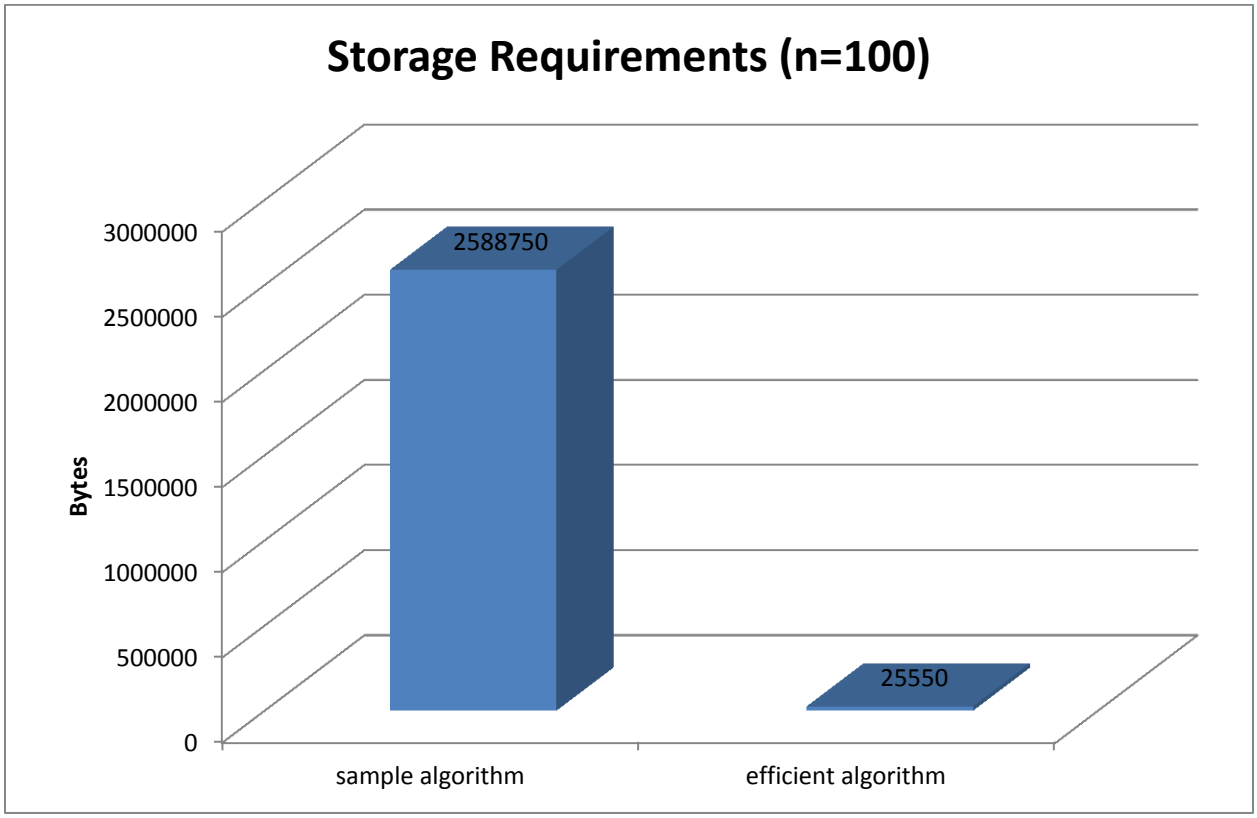

Figure 18

The sample algorithm requires significantly more storage than the efficient algorithm. At the given point in the algorithm, the sample algorithm's storage requirements are over 2.5 megabytes while the efficient algorithm requires only about 25 kilobytes.

The large storage disparity between the algorithms can be attributed primarily to the sample algorithm's storage of the transformation frame vectors, $f_{1:m}$, and the A matrix defined in equation (14). $f_{1:m}$ is an nxm matrix of complex numbers. Both the real and imaginary elements of each complex number are stored as double precision floating point numbers, requiring 8 bytes for each (16 in total for each complex number). For $n = 100$ and $R = 8$, the memory requirements for $f_{1:m}$ would thus be $16 \cdot 100 \cdot 800 = 1{,}280{,}000$ bytes. Similarly, A is a matrix of $2nx2n$ double precision floating point numbers. The storage requirements for A in the same problem setup would be $8 \cdot 200 \cdot 200 = 320{,}000$ bytes. Avoiding this storage is what allows the efficient algorithm to run on large problem sizes of $n{\sim}10{,}000$.

Program Runtime

Finally, the time performance of the efficient implementation of the least squares algorithm is investigated on a problem size of $n = 10{,}000$. The MATLAB profiler was run on a call to $LS\_Algorithm()$. The results are shown in the figure below.

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
LS_Algorithm	1	181.935 s	0.377 s	
Power_Method	1	94.735 s	5.717 s	
Q_u_compute	12717	89.018 s	4.378 s	
Conjugate_Gradient	219	86.267 s	2.215 s	
A_u_compute	7957	82.393 s	7.321 s	
adjTransformation	20893	82.302 s	82.302 s	
Transformation	29071	79.532 s	79.532 s	
RHS_compute	219	1.659 s	0.093 s	

Figure 19

"Total Time" represents the time from call to return of each function summed over all calls. "Self Time" represents the time spent within the given function that is not spent within any other functions called from within that given function. The right column provides a visual representation of the results. The dark blue represents "Self Time" while the dark blue added to the light blue represents "Total Time".

It can first be noted that $LS\_Algorithm()$ required 181.935 seconds to complete in this instance. Only 0.377 seconds were spent directly within $LS\_Algorithm()$, the rest of the time was spent within function calls. $Power\_Method()$ is called once and its total time to completion is 94.735 seconds, over half the total runtime of $LS\_Algorithm()$. Still, a vast majority of the program's time is spent directly within $Transformation()$ and $adjTransformation()$. A visual representation of the relative "Self Times" as a percentage of total algorithm runtime are shown in the pie chart below.

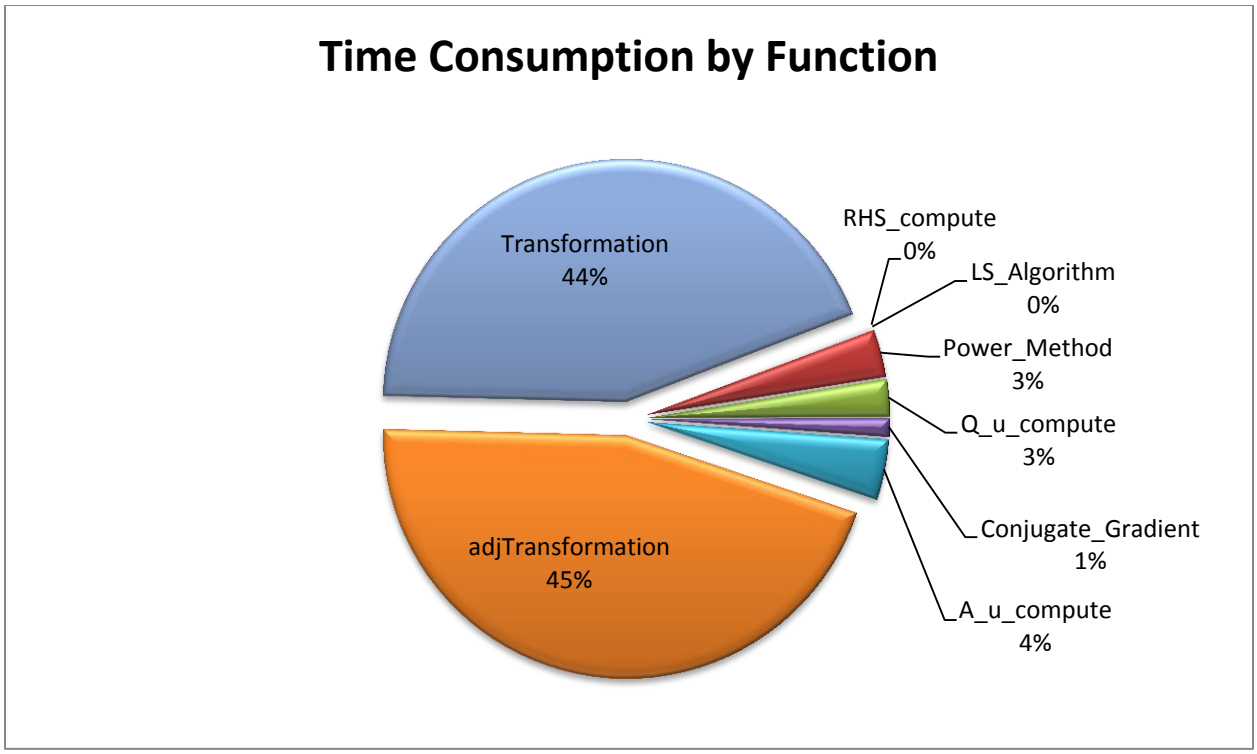

Figure 20

The pie chart shows that the time spent directly within $Transformation()$ and $adjTransformation()$ encompasses nearly4 90% of the runtime of $LS\_Algorithm()$. The next most significant time consuming function is $A\_u\_compute()$ which only takes of 4% of the runtime.

Database Testing

There are 5 different sets of input data, each of which can be passed through 5 uniquely weighted transformations to produce a unique α. The vector y is generated by adding noise to α. The noise vector v can be weighted by σ to produce a desired signal to noise ratio in y.

The goal in testing is to test each input on multiple signal to noise ratio levels ranging from -30 decibels to 30 decibels, in 10 decibel increments. The appropriate signal to noise ratio will be set by adjusting σ in the following equation:

$$SNR_{dB} = 10 \cdot log_{10}\left[\frac{\sum_{k=1}^{m}|c_k|^2}{\sigma^2 \sum_{k=1}^{m}|v_k|^2}\right] \tag{18}$$

Given a certain transformed input, there are 1,000 different noise variations that can be used for each signal to noise ratio level. This produces 1,000 output samples for a specific input at a given signal to noise ratio level. From this data, the mean squared error of the output can be studied in relation to the

signal to noise ratio. As well, the bias of the mean of the output and the variance of the output can be studied against the signal to noise ratio.

For a signal of length $n = 1{,}000$ with 1,000 noise realizations, the following results were obtained for a single input setup:

SNR	$10 \cdot log_{10}(Bias)$	$10 \cdot log_{10}(Variance)$	$10 \cdot log_{10}(MSE)$	CRLB
-30	32.13	43.78	44.07	59.04
-20	32.39	39.29	40.09	49.04
-10	32.27	35.56	37.23	39.04
0	22.17	30.24	30.87	29.04
10	-2.21	19.16	19.19	19.04
20	-18.88	9.05	9.05	9.04
30	-30.63	-0.96	-0.96	-0.96

Plotting the results together gives the following graph:

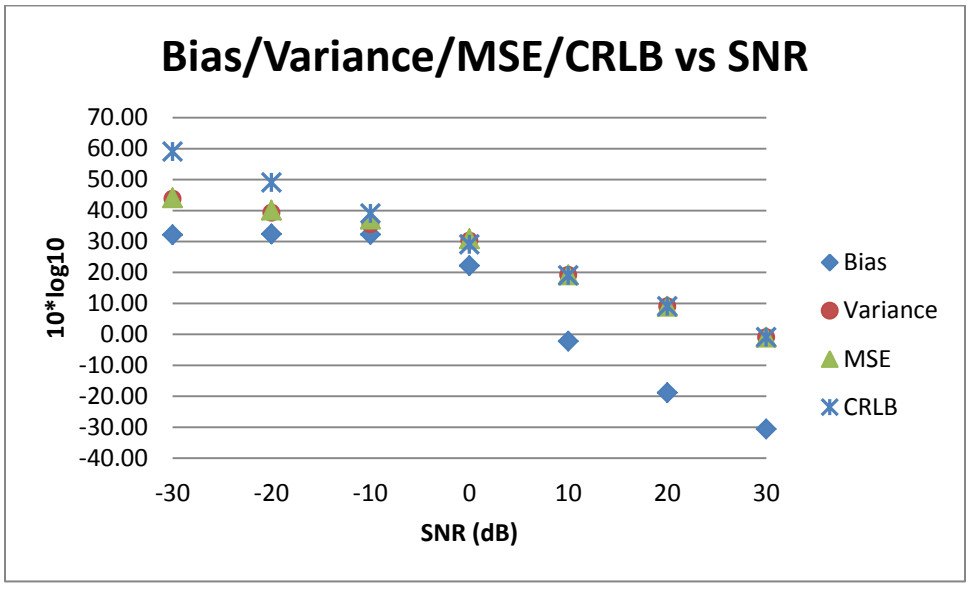

Figure 21

It is clear that the accuracy and precision improve as the level of noise is reduced. The bias continually improves as the signal to noise ratio is brought up to 30 dB. The variance and mean squared error follow similar trends in this regard. Once the signal to noise ratio falls below -10 dB, the rise in the bias

levels off. The variance and the mean squared error continue to rise, however. Furthermore, when the signal to noise ratio falls below 0 dB the Cramer Rao lower bound starts to separate from the variance. The variance in that region lies below the Cramer Rao lower bound, indicating a substantial amount of bias has been introduced into the estimator.

Algorithm Analysis

Parameters

To give insight on the algorithm characteristics, a study was done on two of the fundamental parameters of the algorithm, R and γ. R representes the level of redundancy in the linear transformation. γ represents the rate of convergence of the iterative portion of the algorithm. Higher γ values lead to slower convergence. R and γ were both varied over several values and the resulting effects on the bias, the variance, and the mean squared error as well as the number of iterations required for the reconstructive algorithm to converge were studied. R was varied through values of 4, 6, 8, and 12. γ was varied through values of .9, .95, and .99.

The results on the output metrics when varying R are shown in the plots below. These tests were done for $n = 1{,}000$ over 1,000 noise realizations.

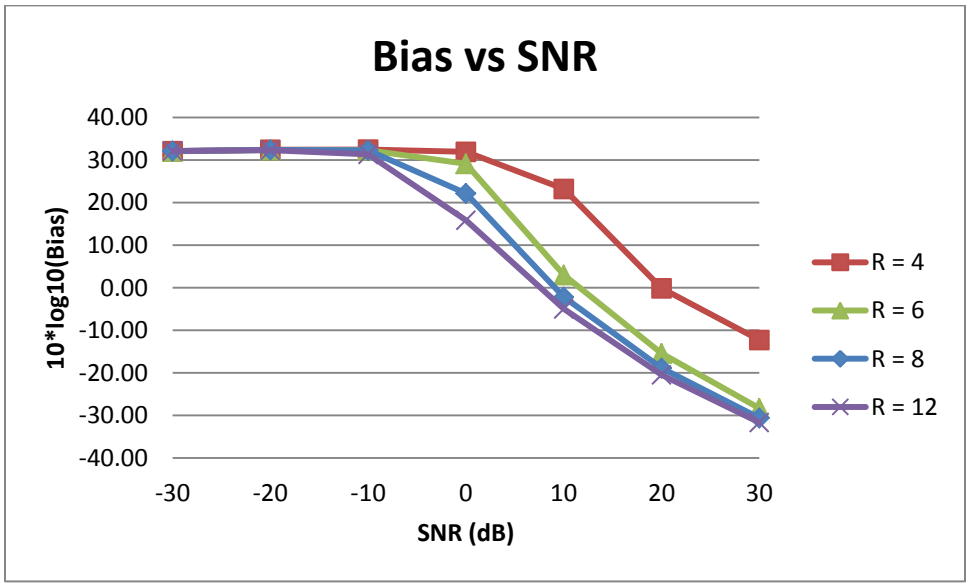

Figure 22

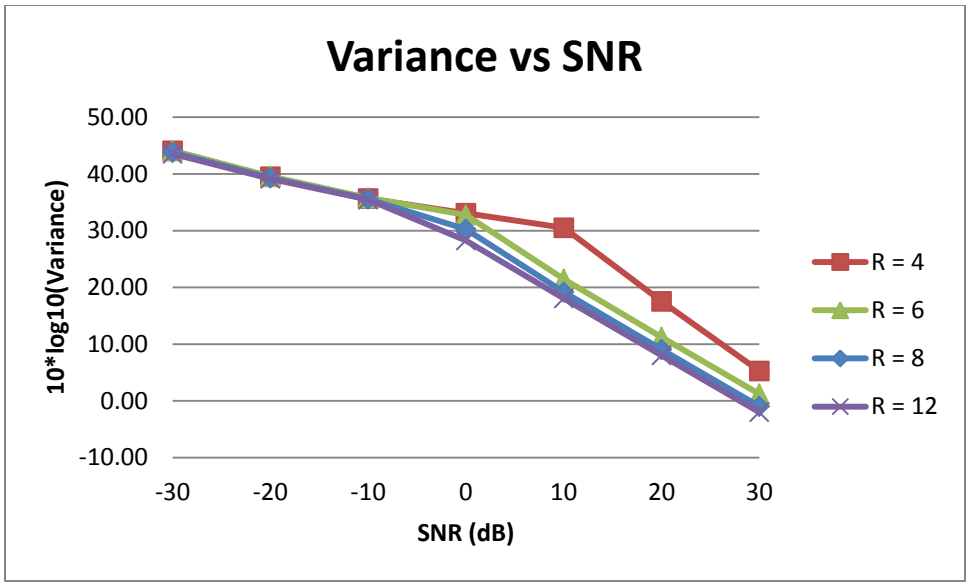

Figure 23

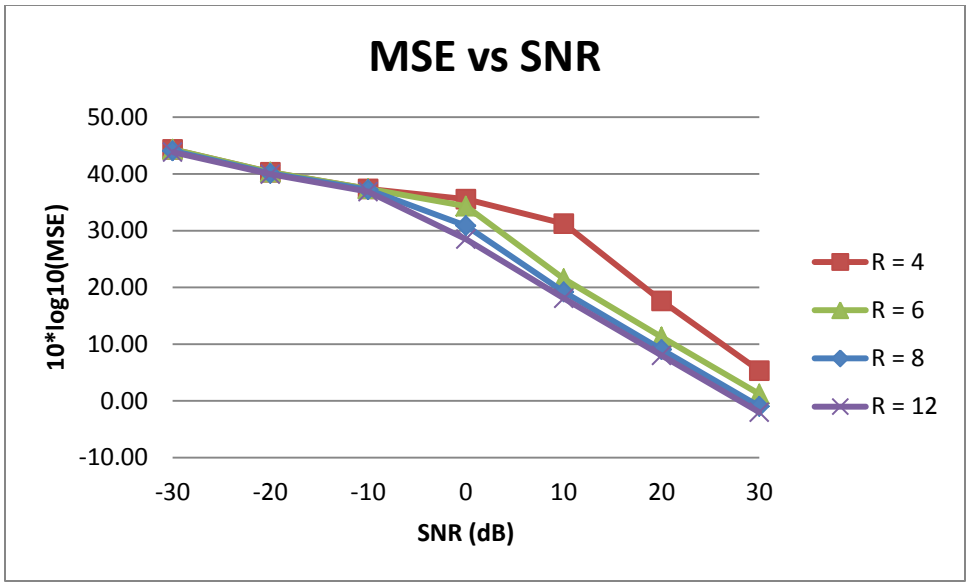

Figure 24

It is clear that higher R values produce greater accuracy and precision in the output for high signal to noise ratios. However, this benefit diminishes and becomes negligible for negative signal to noise ratios. R represents the level of redundancy in the linear transformation. Higher redundancy provides more information about the original signal during reconstruction. This explains the greater accuracy for higher R values. The lack of benefit, though, for low signal to noise ratios could be due to the algorithm's inability to adequately reconstruct the original signal regardless of the level of redundancy.

The number of iterations required for the reconstructive algorithm to converge for varied R values are listed below.

	Iterations/Realization (n = 1,000)			
SNR	**R = 4**	**R = 6**	**R = 8**	**R = 12**
-30	42.6	43.2	43.5	43.0
-20	45.3	46.5	47.1	47.3
-10	52.2	54.7	55.9	56.9
0	69.1	71.2	73.9	92.2
10	2599.0	348.3	340.1	335.2
20	700.9	361.7	346.1	344.4
30	486.6	360.6	348.0	344.0

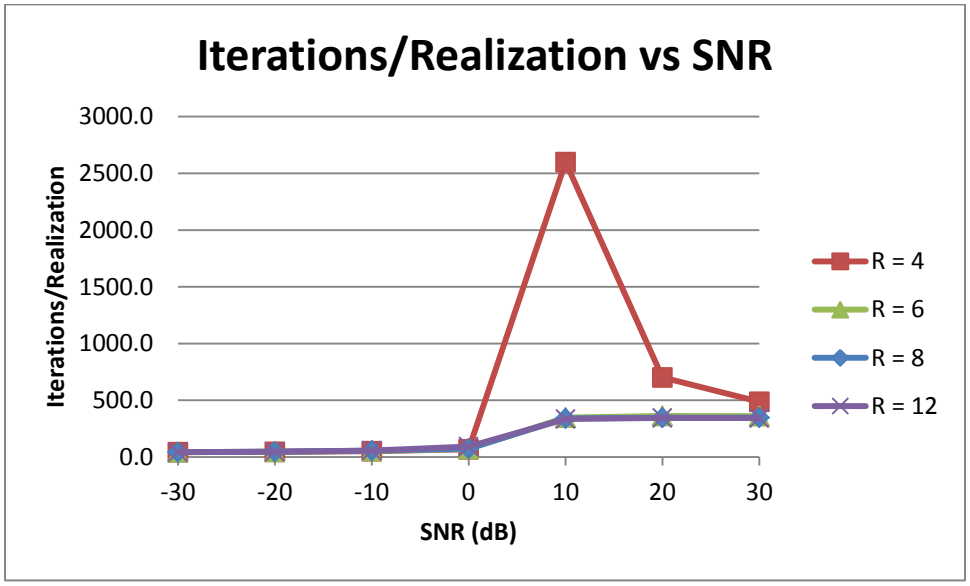

Figure 25

Most notable on the plot is the behavior of the $R = 4$ trend. For signal to noise ratios from -30 to 0, the $R = 4$ trend behaves just like the others. However, at 10 dB signal to noise, there is a huge spike in required iterations. Furthermore, this behavior is not exclusive to this data set. Running the same tests on other data sets more often than not delivers this sort of pattern. It is still unknown why there is a spike for 10 dB signal to noise, but the spike diminishes for 20 dB and 30 dB signal to noise. Further investigation into the algorithm's behavior for low redundancy transformations is needed to help explain this trend.

Taking the $R = 4$ results out of the plot provides better insight into the general trend. This plot is shown below.

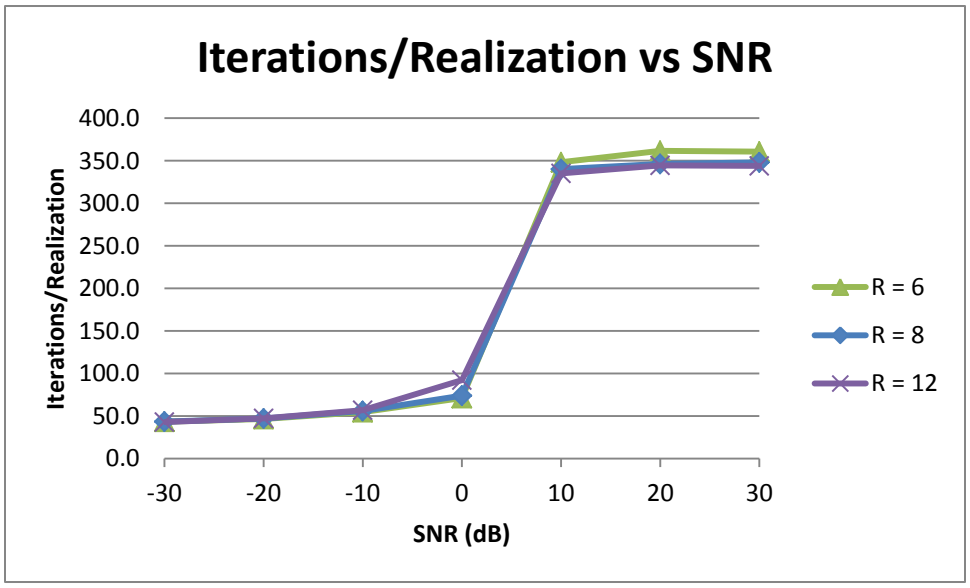

Figure 26

Here the number of iterations seems to remain relatively constant over two regions; positive signal to noise, and nonpostitive signal to noise. Furthermore, the trends for each R value seem to be relatively similar, the number of iterations remains close to 50 for negative signal to noise, and once the signal to noise hits 10 dB the number of iterations jumps to around 350. This is what is expected given the error trends for different signal to noise ratios that was presented earlier in figures (14), (15), and (16).

The results on the output metrics when varying γ are shown on the plots below. These tests were done for $n = 1{,}000$ over 1,000 noise realizations.

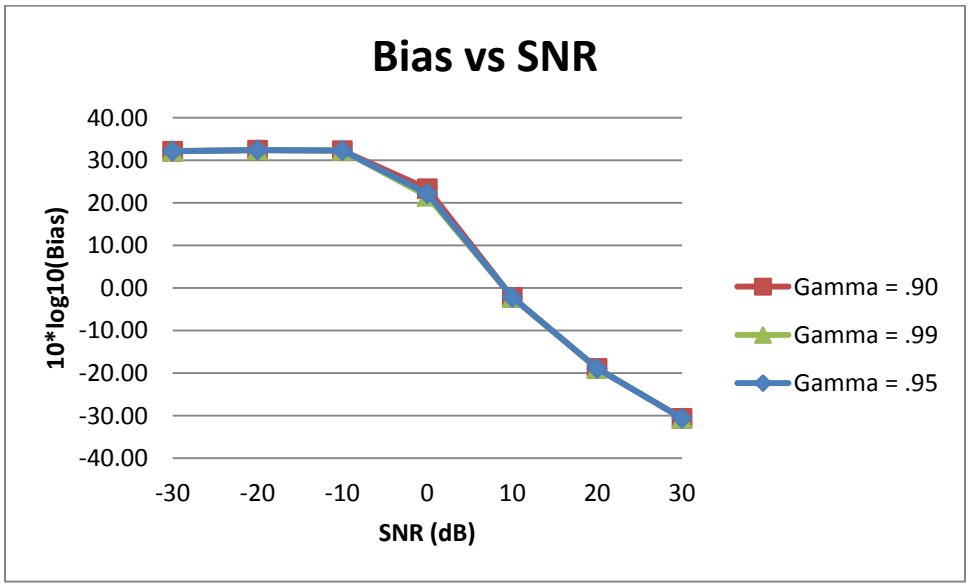

Figure 27

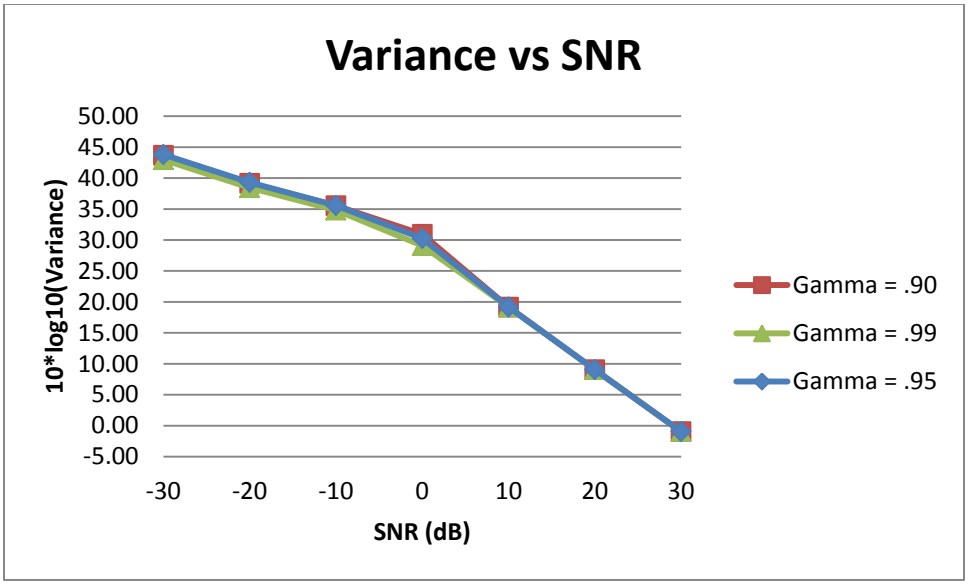

Figure 28

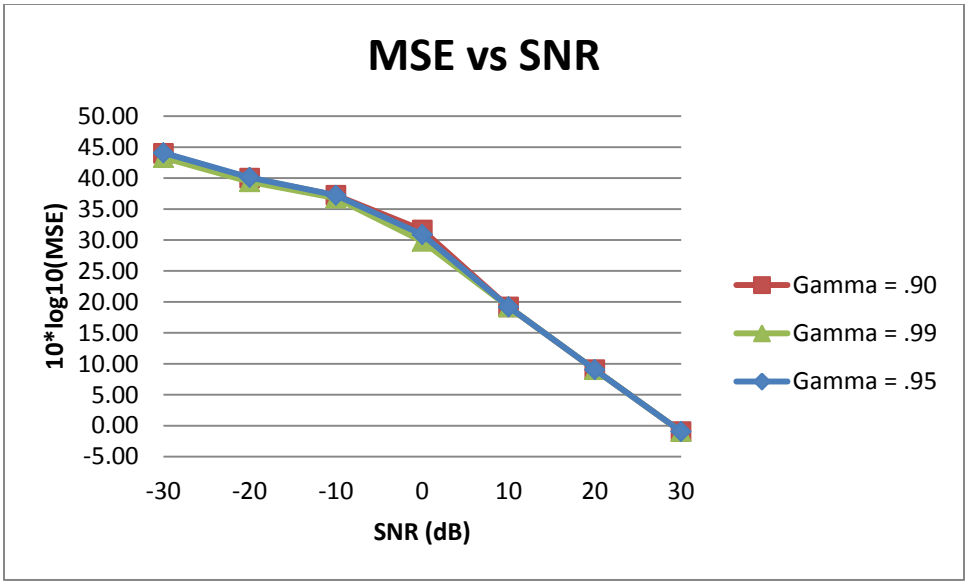

Figure 29

The trends for each γ value follow each other very closely, especially for high signal to noise ratios. Once the signal to noise ratio falls to 0 dB and below, a small separation in variance and mean squared error is introduced. The most relative separation is present actually at 0 dB. This could be because reconstruction is difficult enough at 0 dB that the algorithm gets the added benefit of smaller iterative steps but not too difficult that the benefit becomes negligible.

The number of iterations required to complete reconstruction for varied γ values is listed in the table below.

	Iterations/Realization (n = 1,000)		
SNR	**Gamma = .99**	**Gamma = .95**	**Gamma = .90**
-30	150.9	43.5	22.4
-20	160.3	47.1	23.6
-10	193.7	55.9	29.3
0	283.4	73.9	44.0
10	1592.5	340.1	171.5
20	1584.2	346.1	175.7
30	1579.1	348.0	173.0

Below is a plot of the results.

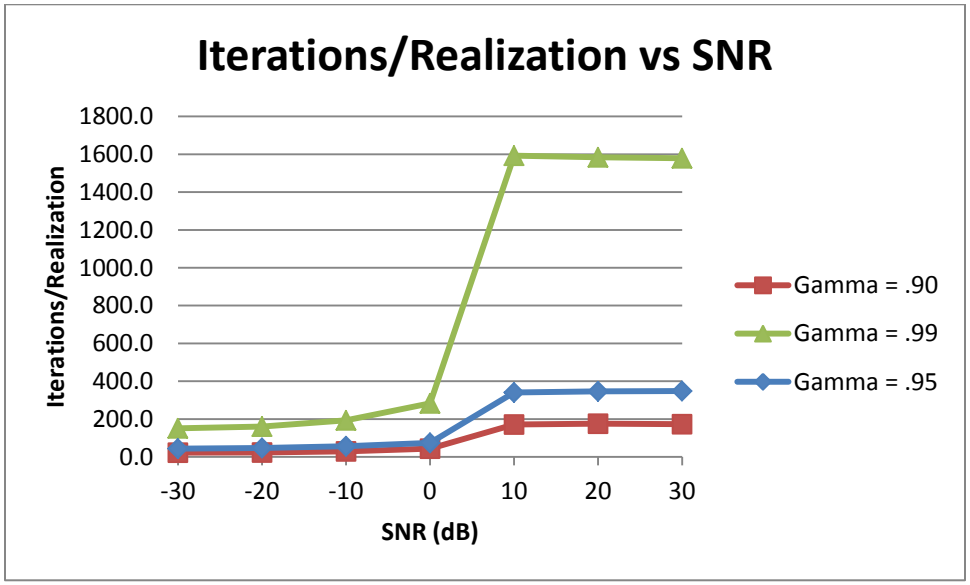

Figure 30

Higher values of γ require more iterations from the reconstructive algorithm for convergence. This is supported by the fact that γ controls the step size between iterations of the algorithm. Specifically, γ controls the rate of decay of the regularization parameters λ and μ. The faster the decay, the more quickly the algorithm can converge to its solution. What's notable about the trend in γ is the increase in iterations required when transitioning from negative signal to noise ratios to positive signal to noise ratios. Moreover, the great disparity in this transition jump between $\gamma = .99$ and $\gamma = .95, .90$. While there is a jump of nearly 300 iterations between SNR = 0 dB and SNR = 10 dB for $\gamma = .95$, there is jump

of approximately 1300 iterations for $\gamma = .99$. Therefore the cost of increase in γ becomes very significant for positive signal to noise ratios.

Computational Complexity

The computational complexity of the reconstructive algorithm was studied with respect to input size. Running reconstruction 100 consecutive times on various input sizes and timing the results yielded the following output:

n value	Total Time (s)	# of trials	Time/trial (s)
4000	3514.3	100	35.143
5000	4892.2	100	48.922
6000	6364.3	100	63.643
7000	7669.4	100	76.694
8000	8775.6	100	87.756
9000	11364.8	100	113.648
10000	11934.7	100	119.347

The time per realization results are plotted below.

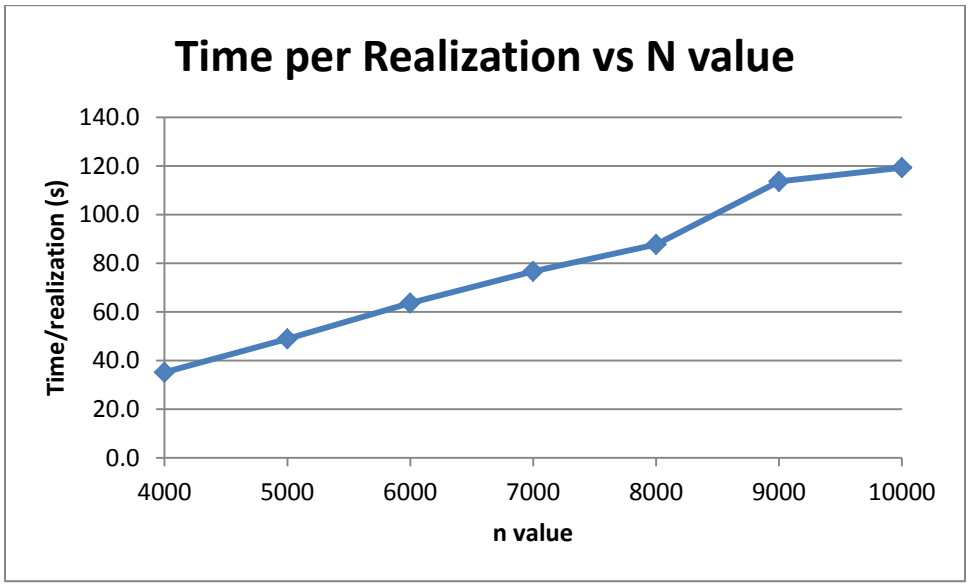

Figure 31

The trend looks virtually linear for large n. This is expected given the information previously presented. From figure (20) it is seen that the algorithm spends approximately 89% of its in $Transformation()$ and $adjTransformation()$. These functions primarily perform repeated fast Fourier transforms and inverse fast Fourier transforms. The fast Fourier transform has order $n \cdot \log(n)$, which far large n would look similar to the results seen here.

Given that algorithm spends an overwhelming majority of its time performing fast Fourier transforms, there is potential for speedup if the fast Fourier transform can be optimized for time efficiency. This can be done in MATLAB using the Fastest Fourier Transfrom in the West library. It is called upon using the $fftw()$ command. When given certain parameters, this command finds the optimal fast Fourier transform algorithm to use for the given platform. Upon the next call to $fft()$ MATALB will perform a search for the most efficient fast Fourier algorithm and then use that algorithm for all subsequent fast Fourier transforms.

Below is a performance comparison between the original fast Fourier transform and the optimized version. The time to complete 5 million Fourier transforms is shown for several vector lengths in the plot below.

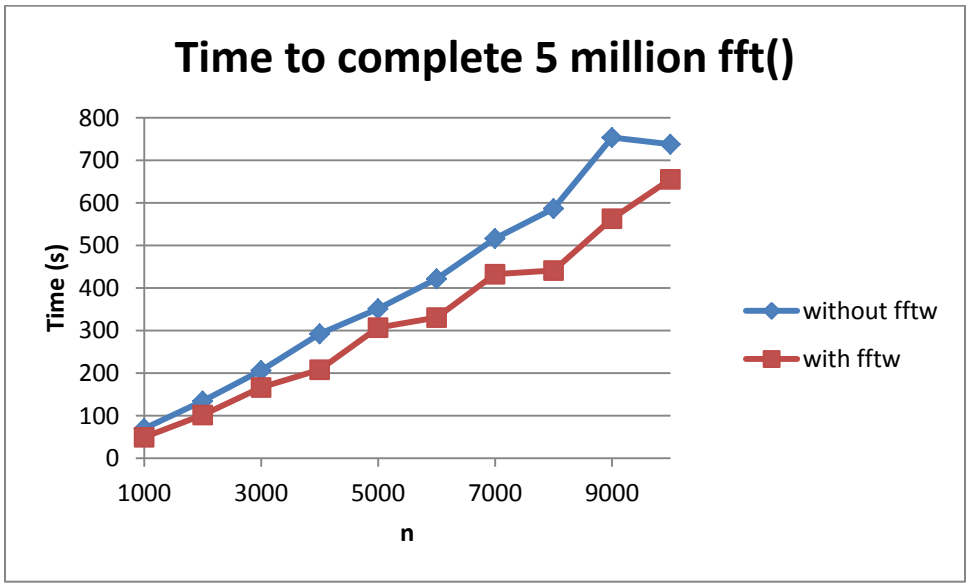

Figure 32

There trend suggests that on average there is a constant proportional increase in performance for all vector lengths. A chart of the relative speedups at each vector length is shown below.

n	Speedup
1000	1.41
2000	1.32

3000	1.24
4000	1.40
5000	1.14
6000	1.28
7000	1.19
8000	1.33
9000	1.34
10000	1.13

When implementing $fftw()$ in the reconstructive algorithm, the goal is to achieve speedups comparable to what is seen here. Upon the first implementation of $fftw()$ in the reconstructive algorithm, there was no noticeable speedup. This was due to the fact that each fast Fourier transform was being called within a separate parallel thread while the initial call to $fftw()$ was made in the original workspace. Thus, the individual threads did not recognize any call to $fftw()$.

After adjusting the program to make individual calls to $fftw()$ for each parallel thread, a marginal speedup was observed. Running the reconstructive algorithm on a signal of length of 10,000 yielded a speedup of 1.025 when implementing $fftw()$. This does not match up to the speedups seen in figure (32). This could still be due to parallelization. The speedups seen in figure (32) were for $fft()$ calls when all system resources were available. When running the reconstructive algorithm, all calls to $fft()$ are made in parallel, thus only the resources of that particular node are available. This could affect the potential speedup of $fftw()$, though more investigation is needed.

Timeline

October	✓ Post-processing framework ✓ Database generation
November	✓ MATLAB implementation of iterative recursive least squares algorithm
December	✓ Validate modules written so far
February	✓ Implement power iteration method ✓ Implement conjugate gradient
By March 15	✓ Validate power iteration and conjugate gradient
March 15 – April 15	✓ Test on synthetic databases ✓ Extract metrics
April 15 – end of semester	✓ Write final report

Deliverables

The project produced several deliverables. They are listed below:

- Proposal presentation
- Written proposal
- Midterm presentation
- Midterm report
- Final presentation
- Final report
- MATLAB code
- Input data
- Output data
- Output plots

References

[1] Allaire, Grâegoire, and Sidi Mahmoud Kaber. *Numerical linear algebra*. Springer, 2008.

[2] R. Balan, On Signal Reconstruction from Its Spectrogram, Proceedings of the CISS Conference, Princeton, NJ, May 2010.

[3] R. Balan, P. Casazza, D. Edidin, On signal reconstruction without phase, Appl.Comput.Harmon.Anal. 20 (2006), 345-356.

[4] R. Balan, Reconstruction of signals from magnitudes of redundant representations. 2012.

[5] R. Balan, Reconstruction of signals from magnitudes of redundant representations: the complex case. 2013.

[6] Christensen, Ole. "Frames in Finite-dimensional Inner Product Spaces."*Frames and Bases*. Birkhäuser Boston, 2008. 1-32.

[7] Shewchuk, Jonathan Richard. "An introduction to the conjugate gradient method without the agonizing pain." (1994).