AMSC 663/664 Mid-year 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 Weiner Center

Table of Contents

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 not carry in an obvious way 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 will be 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 $10{,}000 \times 10{,}000$. 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, $x \epsilon \mathbb{C}^n$, 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:

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

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

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

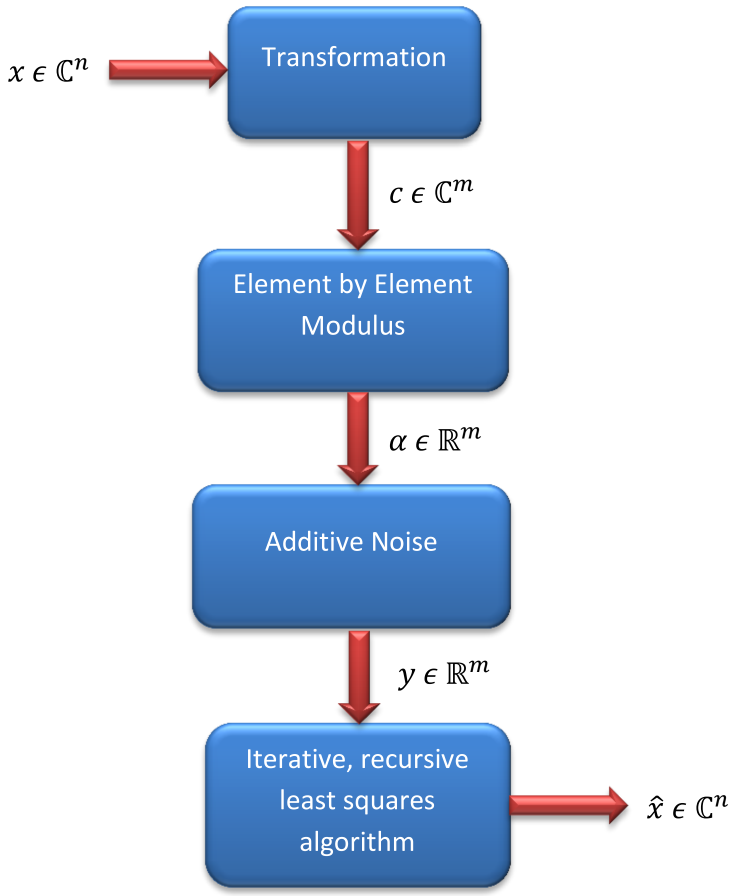
$$\alpha = \begin{bmatrix} |c_1|^2 \\ |c_2|^2 \\ \vdots \\ \vdots \\ |c_m|^2 \end{bmatrix} \epsilon\, \mathbb{R}^m \tag{2}$$

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 Gaussian noise and σ is a weight factor used to achieve a desired signal to noise ratio for testing. 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:



$x \,\epsilon\, \mathbb{C}^n$
Transformation
$c \,\epsilon\, \mathbb{C}^m$
Element by Element Modulus
$\alpha \,\epsilon\, \mathbb{R}^m$
Additive Noise
$y \,\epsilon\, \mathbb{R}^m$
Iterative, recursive least squares algorithm
$\hat{x} \,\epsilon\, \mathbb{C}^n$

After $\hat{x}$ is obtained, the estimation is passed into a post processing framework to study certain output characteristics, namely certain 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:

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

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

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:

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

Where f_k, defined earlier, is the kth frame vector of $T(x)$.

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

$$Q^+ = Q + q\cdot I \tag{10}$$
$$where\ q = \|y\|_\infty, \qquad I = identity$$

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]:

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

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

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:

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

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

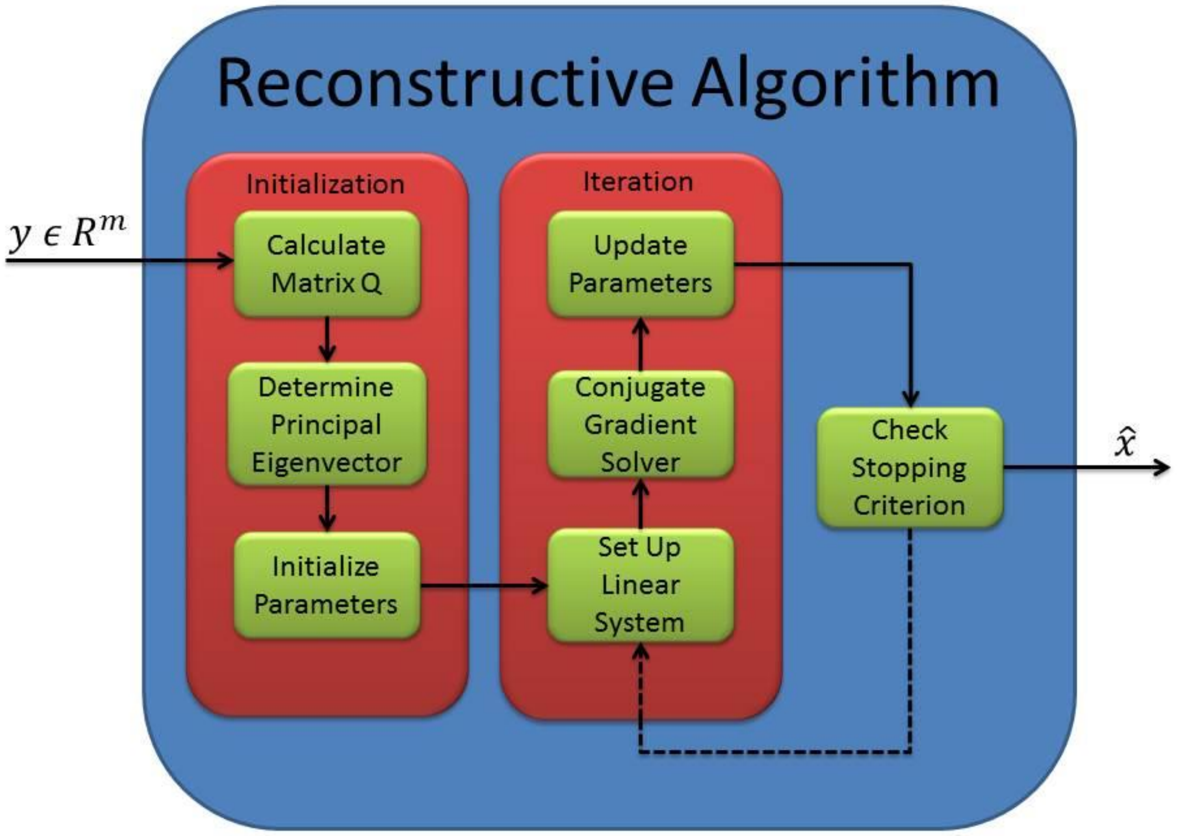
$$where\ 0 < \gamma < 1$$

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

$$\sum_{k=1}^{m} \left| y_k - \left| \left\langle x^{(t)},\ f_k \right\rangle \right|^2 \right|^2 \leq\ \kappa m \sigma^2,\ where\ \kappa\ is\ a\ constant < 1 \text{ [5]} \tag{18}$$

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

The flow of the algorithm is represented in the figure below.



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~$10{,}000$, 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:

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

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:

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

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

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

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

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

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 $x \epsilon \mathbb{C}^n$ 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. n, which is the vector length of x, will be on the order of 10,000. 10 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 10 different realizations of each set $w^{([1,R])}$.

The noise, ν, 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 10,000 different realizations of noise, ν.

Principal Eigenvalue (Initialization)

During the initialization stage of the iterative, recursive least squares algorithm, the principal eigenvalue of a matrix Q must be calculated. 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:

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

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

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

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 iterations converge to within a given tolerance of the solution, the process can be concluded at that time with a sufficient 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]:

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

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

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

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

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

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 implemented in MATLAB. The iterative, recursive least squares algorithm will be programmed to run in parallel on different input vectors. This will increase the time efficiency of the program as it runs over multiple input data sets.

To implement the discrete Fourier transform, MATLAB's $fft()$ function will be used. $fft()$ implements a fast Fourier transform. Random numbers will primarily be generated using MATLAB's $randn()$ command, which generates normally distributed random variates. For generating the random noise variants however, a linear congruential generator will be implemented. The linear congruential generator works as follows:

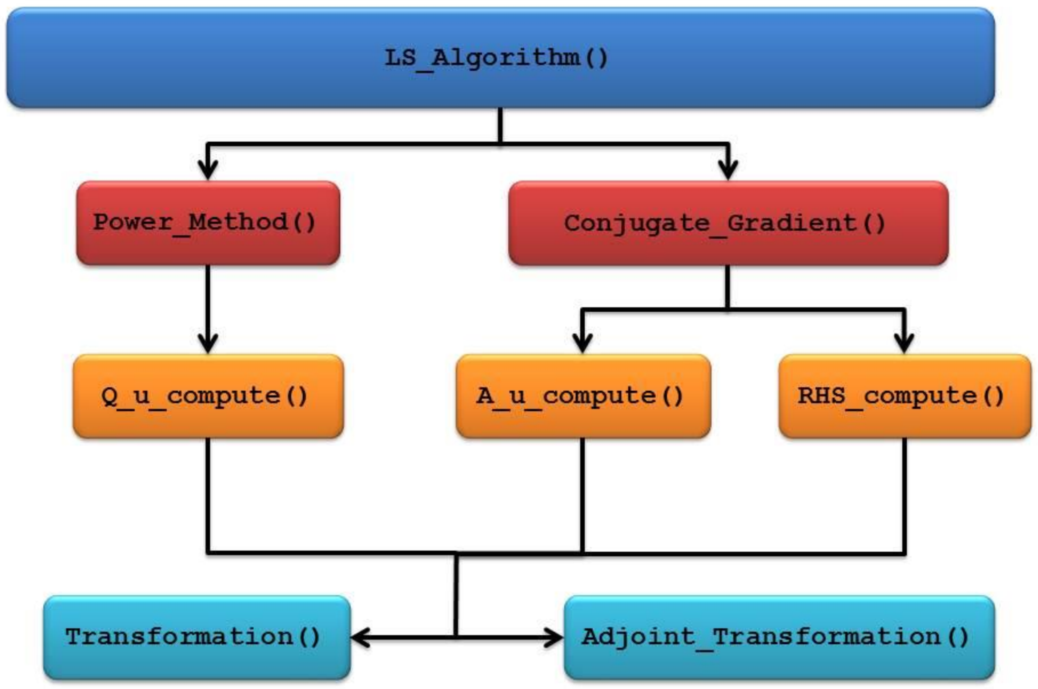
$$Z_{k+1} = (aZ_k + c)(mod\ M)$$

Here M is the modulus, a is the multiplier, c is the increment, and Z_0 is the seed. For a given integer Z_k, the associated random number U_k would equal:

$$U_k = \frac{Z_k}{M}$$

Using a linear congruential generator has the advantage that if the same seed is used in different testing implementations, then the same exact sequence of random numbers would be produced in each case. Since there will be 10,000 different realizations of noise vectors, it will be beneficial to generate them each time the program runs, rather than saving and loading each realization.

To preserve memory efficiency, implementing the iterative, recursive least squares algorithm is written using the formulations described in equations (19), (20), and (21). The memory efficient algorithm has the following function structure:



$LS\_Algorithm()$ 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.

Parameters

Below is a list of the various parameters required in implementation and the associated values they will be set to.

$\boldsymbol{n}$	$\mathbf{10,000}$
$\boldsymbol{R}$	8
$\boldsymbol{\gamma}$	0.95
$\boldsymbol{\rho}$	0.5
$\boldsymbol{\kappa}$	0.5
PM stopping tolerance	10^{-14}
CG stopping tolerance	10^{-14}
$\boldsymbol{\mu_{\min}}$	$\frac{\mu_0}{10}$

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.

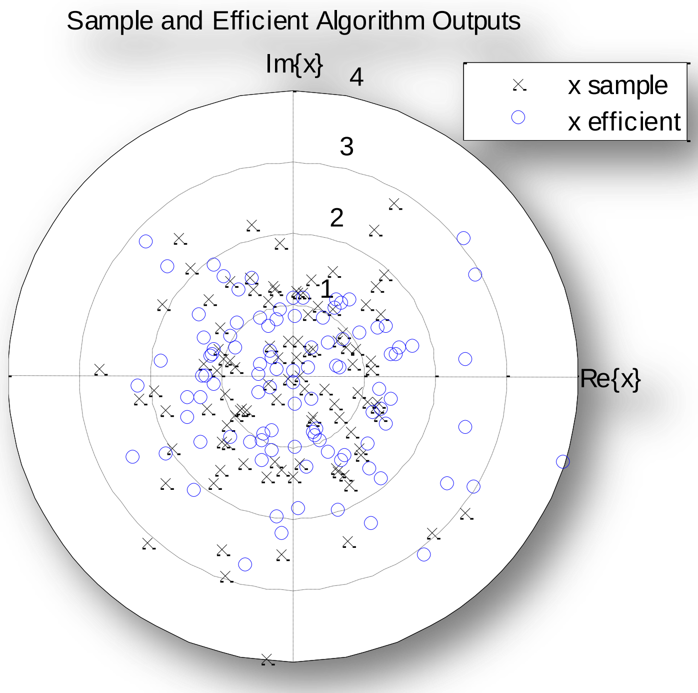
The memory efficient implementation represented by equations (19), (20), and (21) will programmed and referred to as the efficient implementation. It will be compared against the frame vector implementation formulated by equations (9), (14), and (15) which will be called the sample implementation. The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

Results

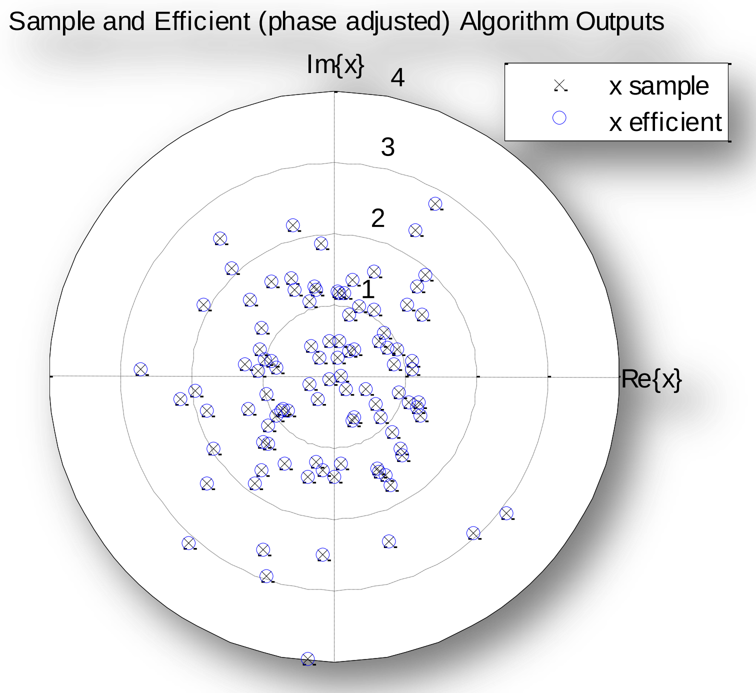
The power method and conjugate gradient method were both programmed and validated as described. The conjugate gradient method reliably produces results comparable to MATLAB's $mldivide()$ with a slight amount of round-off error ($\sim 10^{-30}$) even when the conjugate gradient method is run through 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:



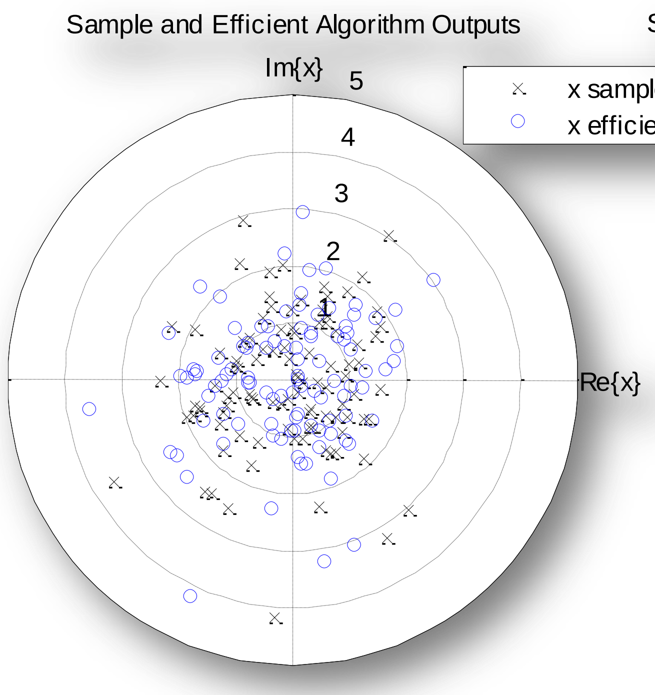
It can be seen that the outputs do not line up. The norm difference between the principal eigenvector of the efficient implementation and the principal eigenvector of the sample implementation was 1.2525. The phase difference between the eigenvectors of the two implementations was $e^{-i*1.3535}$. Shifting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

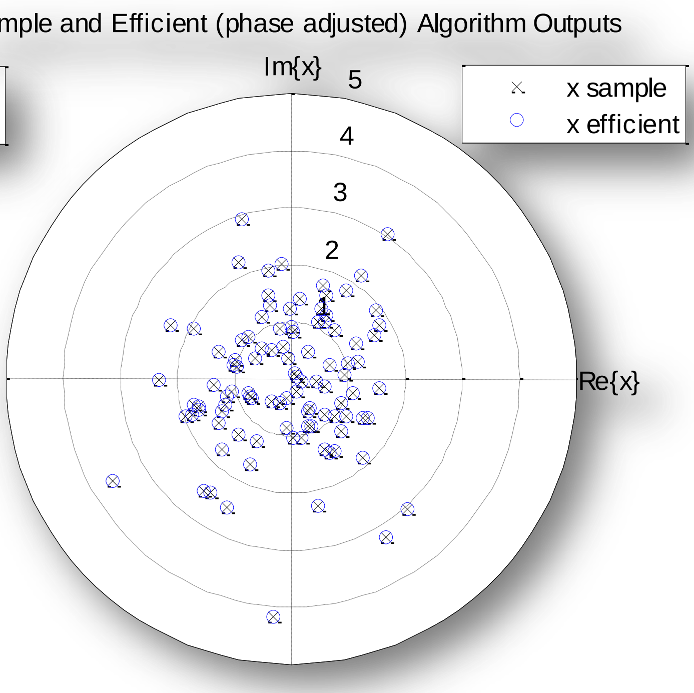


The outputs now line up, showing that the two outputs are only off by the phase factor introduced during the eigenvector retrieval in the initialization phase.

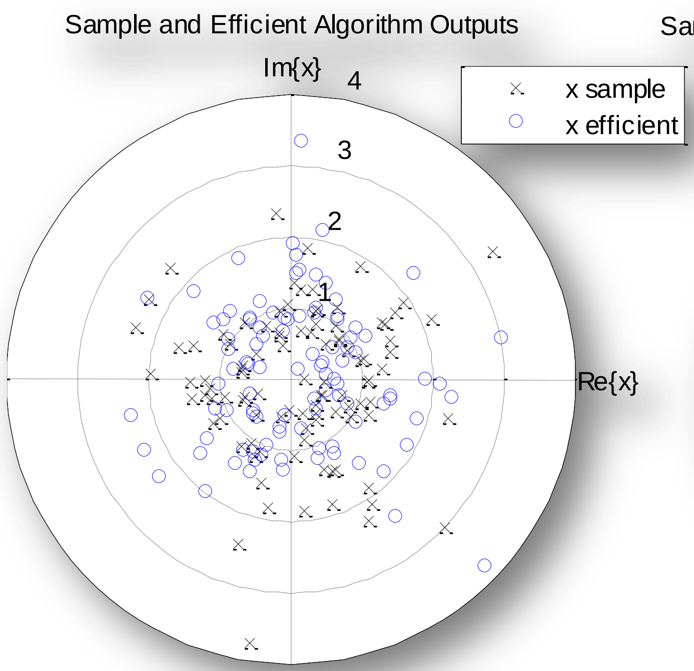
The same process is repeated for two more sample data sets with $n = 100$.

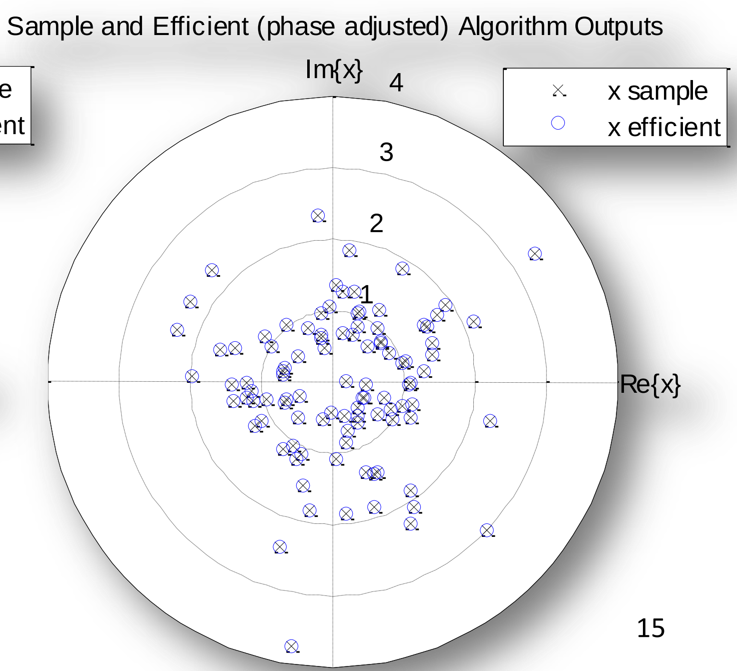
In this data set, the norm difference in the principal eigenvectors of Q was 0.3665 and the phase difference was $e^{i*0.3686}$. After a phase shift in the output of the efficient implementation, the outputs line up.





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.

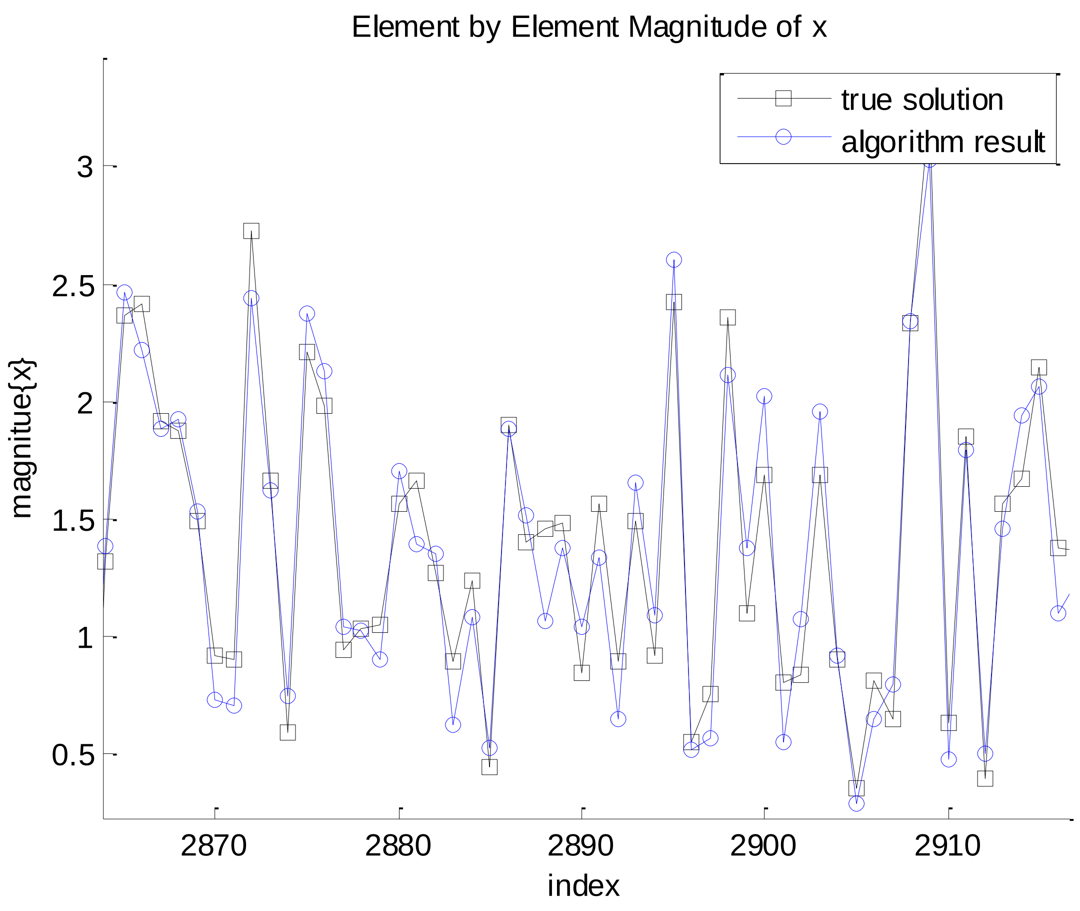




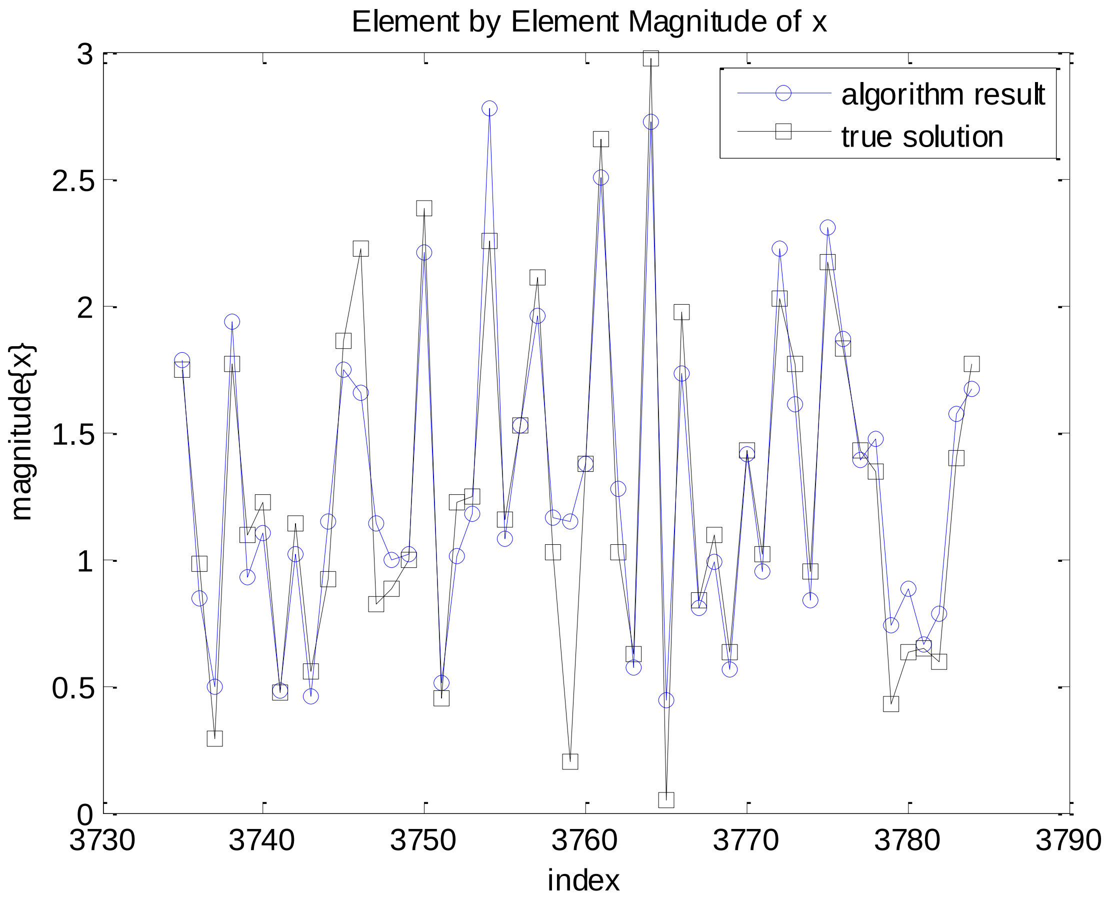
Testing

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 in y 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.

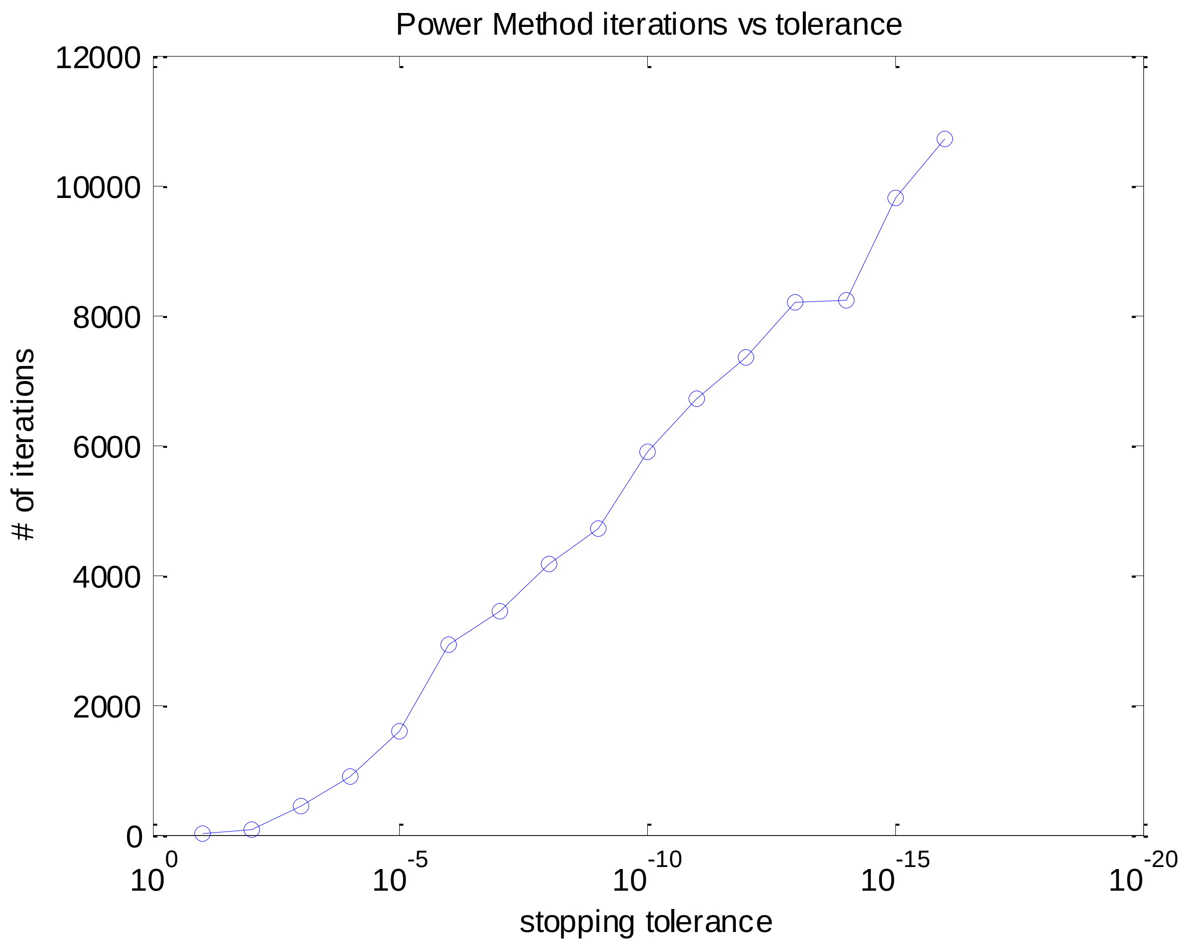


The approximate solution $\hat{x}$ is represented in blue while the original signal x is represented in black. Below is a plot of the portion containing the point with the highest inaccuracy.



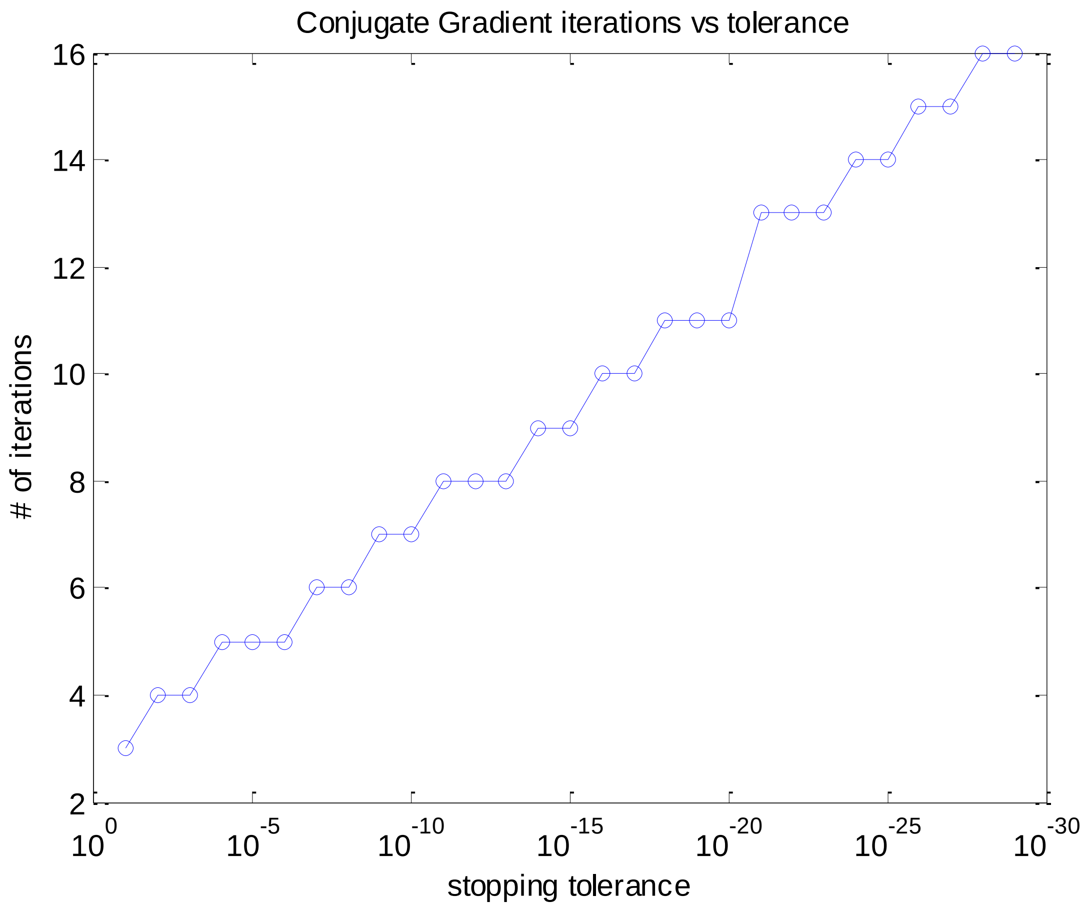
The highest inaccuracy occurs at an index around 3760.

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.



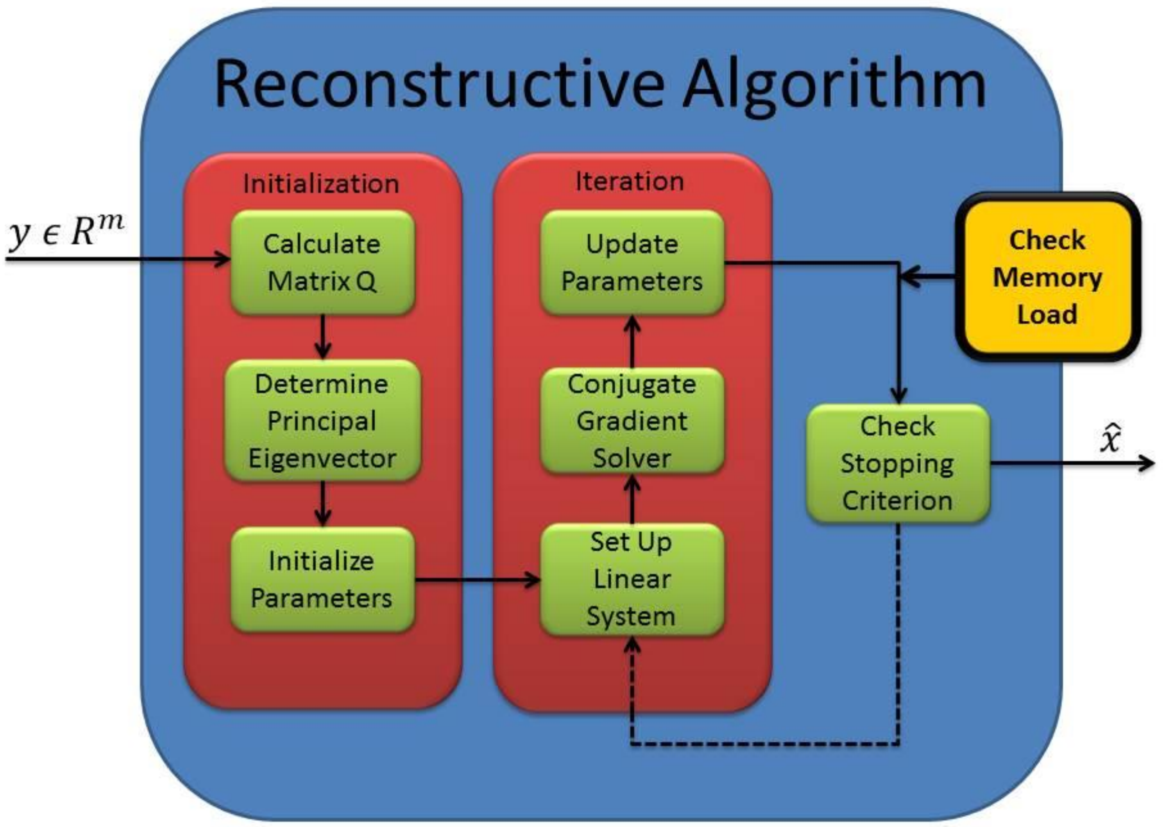
The stopping tolerance is plotted on the horizontal axis on a logarithmically decreasing scale. The power method requires a considerable amount of iterations. For a stopping tolerance of 10^{-10} the power method requires about $6{,}000$ iterations. From the graph it can be concluded that the error in 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.

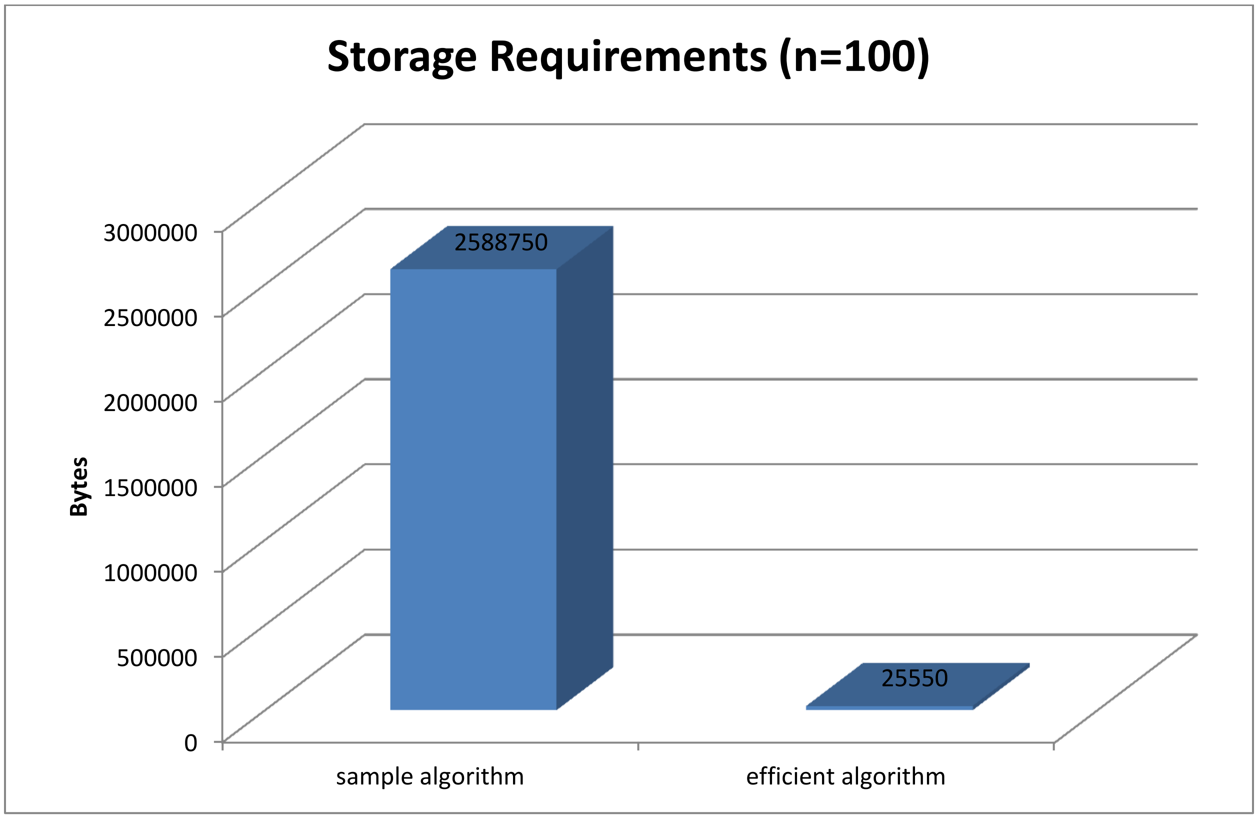


The conjugate gradient method requires very few iterations and converges to high accuracy very quickly. For a stopping tolerance of 10^{-20} the conjugate gradient method requires 11 iterations. This is beneficial because the conjugate gradient method is called numerous times through the execution of 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.

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.



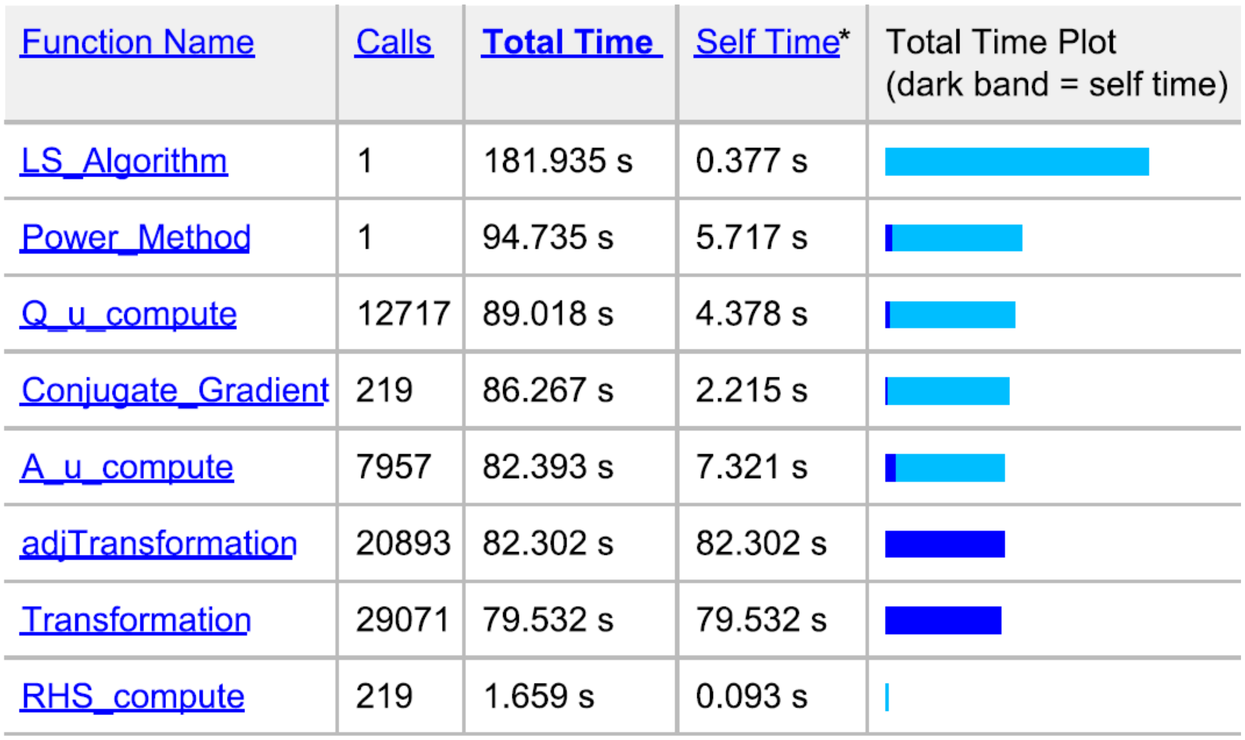
The results are graphed below.



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 $n \sim 10{,}000$.

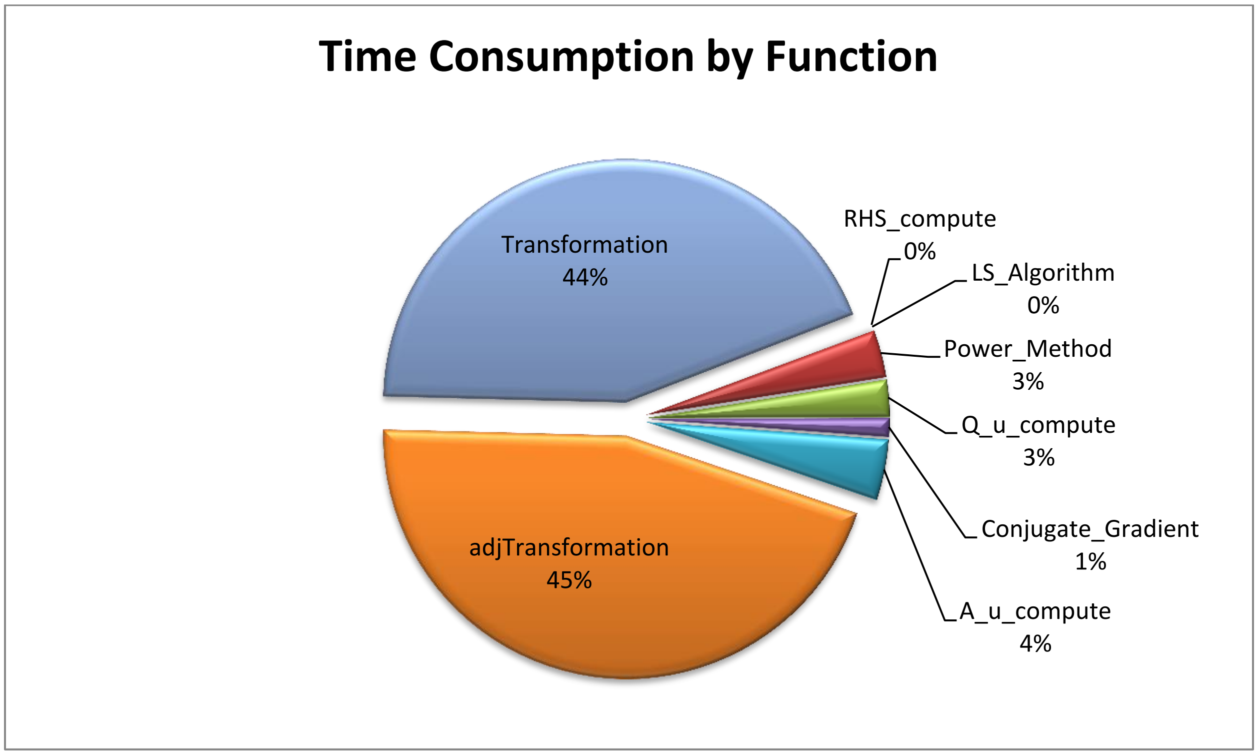
Finally, the time performance of the efficient implementation of the least squares algorithm is investigated on a problem size of $n = 10{,}000$. The MATLAB profiler was run on a call to $LS\_Algorithm()$. The results are shown in the figure below.



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	

"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.



The pie chart shows that the time spent directly within $Transformation()$ and $adjTransformation()$ encompasses nearly 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 will be 10 different sets of input data, each of which can be passed through 10 different 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 will be to test each input on a multiple of signal to noise ratio levels ranging from -30 decibels to 30 decibels, in 5 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 10,000 different noise variations that can be used for each signal to noise ratio level. This will produce 10,000 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.

Timeline

So far the recursive least squares algorithm has been program and validated. This includes the implementation of the power method and the conjugate gradient method. The databases required for testing have also been generated. Future work will involve programming the post processing framework and running tests. An updated schedule of the progress so far is shown below. A check mark indicates a task that has been completed.

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 will produce several deliverables. They are listed below:

- Proposal presentation
- Written proposal
- Midterm presentation
- Final presentation
- Final report
- MATLAB program
- Input data
- Output data
- Output charts and graphs

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).