Project 2: Compressed Sensing

Richard Cotton

1 Introduction

In this report, we will investigate and analyse a relatively modern breakthrough in scientific computing called compressed sensing. The vast majority of data has lots of data points, many of which can be combined into a simpler subset of data which takes up less memory space. If we were to use the full data set without compression, trying to use the data for any processes would require a large amount time and power which are often constrained. Thus we must use a compressed dataset which still resembles the full system suitably.

The usage of compressed sensing is vital as a number of parameters within a system are often negligible, meaning that they have minimal impact on the solution of the system, causing unnecessary calculations. It is important that the amount of data removed when compressing is not so significant that it removes the main features of the system, causing unviable/incorrect solutions to be found. [3]

This work is important in many fields within our data-filled world. From small case usage such as medical imaging technology and radar systems where the compressing needs to be significant, all the way down to person to person usage such as mobile phone photo compression. The applications of these algorithms are vital for storing data effectively and efficiently. [5]

We begin our analysis with the gradient descent algorithm as described by Liu [2]. This is a method used for systems where the number of equations are greater than the number of variables. Therefore, this system cannot always be solved exactly, requiring a least squares analysis to find the best solution possible. We continue our analysis by considering systems where the number of variables is greater than the number of equations. This means that the system is under-determined. Thus we require the solution vector x to be sparsely filled so that not all equations are considered at once. As the solution is not always unique with under-determined systems, we must take care that the solution the algorithm finds is the correct solution for the system when completing our analysis.

2 Gradient Descent Method

We begin our investigation by considering an algorithm called Gradient Descent. In this problem, we shall consider a linear system of equations. This system can be represented in the following form

$$Ax = b$$

where A is the matrix of coefficients, x is a vector of unknown variables and b is the solution vector.

For this problem, we shall consider $m \times n$ matrices where $m \geq n$. This condition produces matrices which are "tall and skinny", whereby the number of equations to be solved is larger than the number of variables. A simple iterative method called Gradient Descent can be implemented to solve this least squares problem.

Given a function $f(x)$ and an initial condition x_0, the function is minimised by taking steps in the negative gradient direction. This is given iteratively by

$$x_{i+1} = x_i - \alpha_i \nabla f(x_i)$$

with step length α_i. The residual at a point x is given by $r_i = A^T(b - Ax_i)$. [4]

We wish to choose the value of α at each iteration such that the successive residuals are orthogonal to each other. The reasoning behind this is due to the gradient projected current direction of movement r_i having a negative value. This is no longer the case when $r_{i+1} \cdot r_i = 0$.

We note that r_{i+1} can be written as $r_i - A^T A \alpha_i r_i$. Then

$$\begin{aligned} 0 &= r_i^T r_{i+1} \\ &= r_i^T (r_i - A^T A \alpha_i r_i). \end{aligned}$$

Expanding the brackets,

$$r_i^T r_i = r_i^T A^T A \alpha_i r_i.$$

Rearranging for α, we get

$$\alpha_i = \frac{r_i^T r_i}{r_i^T A^T A r_i}.$$

Thus the gradient descent algorithm takes the following form.

We start with an initial guess of $x_0 = 0$ and calculate $r_0 = A^T(b - Ax_0)$. Then the following values are successively computed:

$$\begin{aligned} \alpha_i &= \frac{r_i^T r_i}{r_i^T A^T A r_i}, \\ x_{i+1} &= x_i + \alpha r_i, \\ r_{i+1} &= A^T(b - Ax_{i+1}). \end{aligned}$$

This algorithm is executed until either a maximum number of iterations are performed or until the vector norm of the residual, $\|r_{i+1}\|$ is smaller than some given tolerance. For the purpose of our calculations, we take the maximum iterations to be 100 and the tolerance for $\|r_{i+1}\|$ is set to be 10^{-6}. [2]

The conditioning on the matrix A plays a strong role in the convergence to a solution for the least squares problem. We shall illustrate this idea with a number of examples with varying matrices A.

2.1 Example 1

We begin our examples with the following matrix A and vector b and attempt to find the value of x such that the calculated residuals are minimised.

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix}.$$

We run the SDLS algorithm as seen in Appendix A.4, recording the coordinates of x and plotting the evolution with the vector x. We may take the initial solution x_0 as any point as this will only affect the number of iterations taken to reach the solution marginally. In our case, we shall take the origin as the initial point for all our examples. As seen in Figure 1, the value of x converges to the solution by "zig-zagging" along the lines of steepest descent. As mentioned before, the trajectory appears this way as successive gradients are orthogonal to each other.

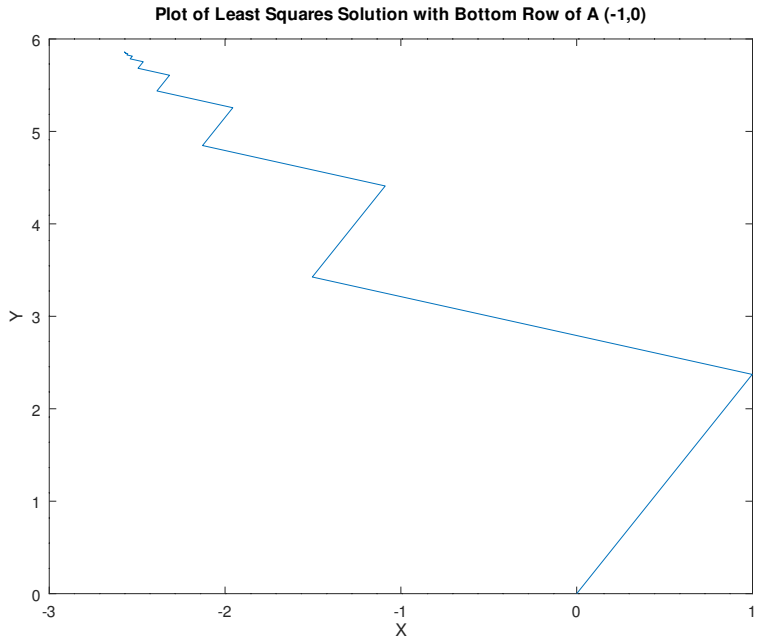

Figure 1: Plot of convergence of solution with bottom row of $A(-1, 0)$

Upon running the SDLS algorithm, we find that the solution is converged to after 40 iterations, with the algorithm terminating when $||r_{i+1}|| < 10^{-6}$.

2.2 Example 2

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 1.8 & -2 \end{pmatrix}, \quad b = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix}.$$

In this example, we consider the same solution vector b, but change the bottom row of the matrix A to $(1.8, -2)$. The trajectory of the solution can be seen from Figure 2. We run the SDLS algorithm and find the final solution which is converged to after only 4 iterations.

The main reason for the speed of convergence in this example is due to the value of $-\nabla f$, which sends x_0 to the next value x_1 which is very close to the correct solution. From this point, the successive few points calculated takes the approximate solution to within the desired tolerance of 10^{-6} very quickly. This initial iteration bypasses the need for the lengthy "zig-zagging" process, resulting in a much faster convergence of solution.

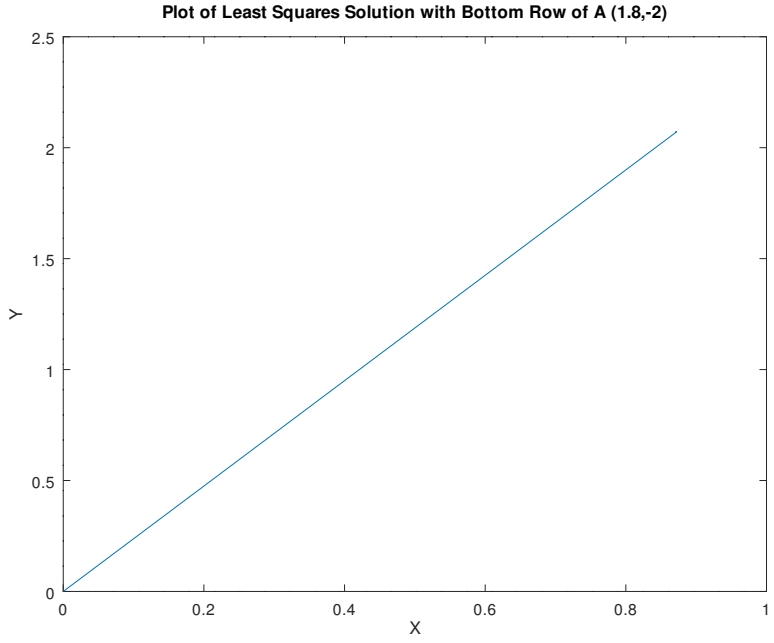

Figure 2: Plot of convergence of solution with bottom row of $A(1.8, -2)$

2.3 Example 3

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -2 & -2 \end{pmatrix}, \quad b = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix}.$$

In this example, we again keep b the same and change the bottom row of A to $(-2, -2)$. Running the SDLS algorithm again, we find the solution is converged to after 76 iterations.

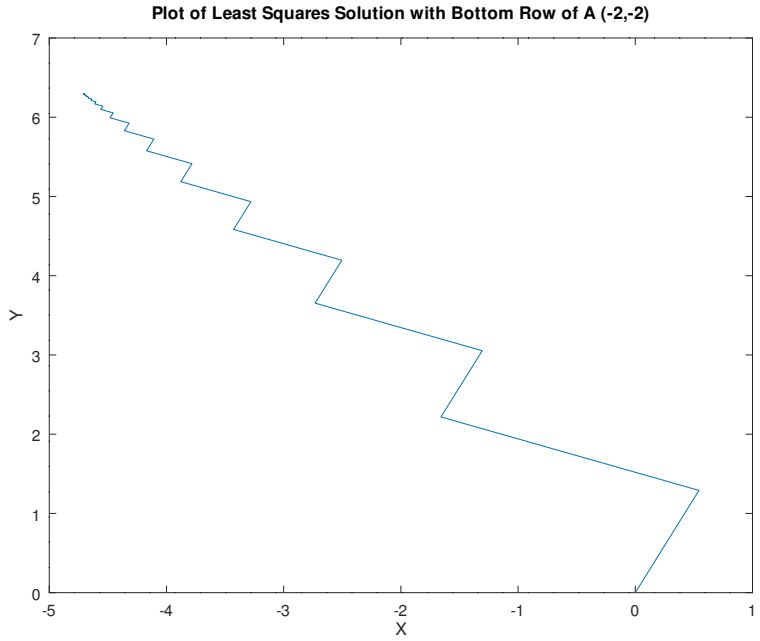

Figure 3: Plot of convergence of solution with bottom row of $A(-2, -2)$

As can be seen from the three examples above, changing the coefficients of the matrix A has drastic consequences on the speed of convergence to a solution and the value of the final solution converged upon. The main factor which affects the speed of convergence of the system is the conditioning of the matrix.

We shall use the 2-norm of a non-singular matrix, denoted by $\|A\|_2$. This value is calculated using the command `cond()` in Octave. The condition number gives a representation as to how small changes in the matrix affects the solution found when solving systems of the form $Ax = b$. For example, the closer the conditioning number is to one, the less sensitive the solution will be to minor variations to the matrix entries.

The results of calculating the condition number for our three examples can be found in the table below.

Bottom row of A	Cond(A)	Iterations to reach solution
$(1.8, -2)$	1.0662	4
$(1, 0)$	2.5473	40
$(-2, -2)$	4.1231	76

As can be seen above, the number of iterations taken to reach a solution successfully varies greatly depending on the final set of coefficients given by the bottom row of A. In addition, there

is a clear correlation between the condition number and the number of iterations required to reach the solution. [6]

3 Normalised Iterative Thresholding Algorithm

We now consider the reverse situation of the dimensions on the matrix A. In this case, we consider the system

$$Ax = b$$

where the vector of unknown variables x and the solution vector b are the same as in Section 2, but the $m \times n$ matrix A has the condition $m \leq n$. This produces matrices which look "long and fat". We also impose the constraint that x is k-sparse for $k < m$. This is done to ensure that the problem is sufficiently determined.

We shall now discuss and implement the Normalised Iterative Thresholding Algorithm (NIHT). This algorithm is similar to the SDLS algorithm described in Section 2, but at each step, a thresholding operation is performed. This operation is applied to the vector x during each iteration, which sets all but the largest k elements (by absolute value) of the vector to zero. The reasoning for this thresholding operation is to ensure only the most significant values of the solution matrix are included in the system for which the algorithm is applied to. This is done as elements which have less significant coefficients are much less likely to have an impact of the solution of the system and is used in applications to ensure the amount of memory used is not significant. [1]

The algorithm is described as follows. We initialise the starting vector $x_0 = A^T b$ and threshold this vector with the k largest values remaining in the vector. We also calculate the initial residual $r_0 = A^T(b - Ax)$. We then perform the following steps until the vector norm of the residuals $\|r_{i+1}\|$ is less than some tolerance or the maximum number of iterations is reached. We use the same parameters of tolerance 10^{-6} and maximum iterations 1000.

The iterative step is described below:

$$\begin{aligned} \alpha &= \frac{r_i^T r_i}{r_i^T A^T A r_i}, \\ x_{i+1} &= \mathcal{H}(x_i + \alpha r_i), \\ r_{i+1} &= A^T(b - A x_{i+1}) \end{aligned}$$

where $\mathcal{H}$ is the thresholding operation which sets all but the largest k elements (by absolute value) to 0.

In addition to this, at the end of each iteration, we check whether the distance between the 2-norm of the current and previous residual is less than some tolerance, in our case 10^{-7}. If this is the case, we know that the residuals are not changing sufficiently, implying non-convergence. This step ensures that we do not waste time and memory iterating to find a solution which will never be converged upon.

An implementation of the algorithm can be found in Appendix A.5. In our implementation of the thresholding step, we take the absolute value of each element of the vector, sort it using the C++ `sort` routine and find the kth largest value of the vector. Then we run through the vector and check whether each element is smaller than the kth largest value. This operation requires running through the vector twice which is $\mathcal{O}(n)$ as well as a sorting step which is $\mathcal{O}(n \log(n))$. Thus the overall complexity of the thresholding operation is $\mathcal{O}(n \log(n))$.

4 Phase Transition Phenomenon

We shall now investigate a phenomenon called phase transition. This is a drastic change in the behaviour of a system as one or more parameters are changed. We shall investigate this phenomenon by keeping the values of the sparsity level k and the number of variables $n = 200$ fixed and vary the number of equations m. Here, we fill the entries of the matrix A, with normally distributed random variables with mean zero and variance $1/m$. The vector x is also randomly generated with mean zero and variance one. We then calculate $b = Ax$ and run the NIHT algorithm.

For each value of k, we shall run the NIHT algorithm 100 times and record the number of times that a solution is successfully recovered. We then calculate the ratio

$$p(m) = \frac{\#\text{successful recoveries}}{100}.$$

This is then plotted as a function of m for the values of $k = 10, 20$ and 50.

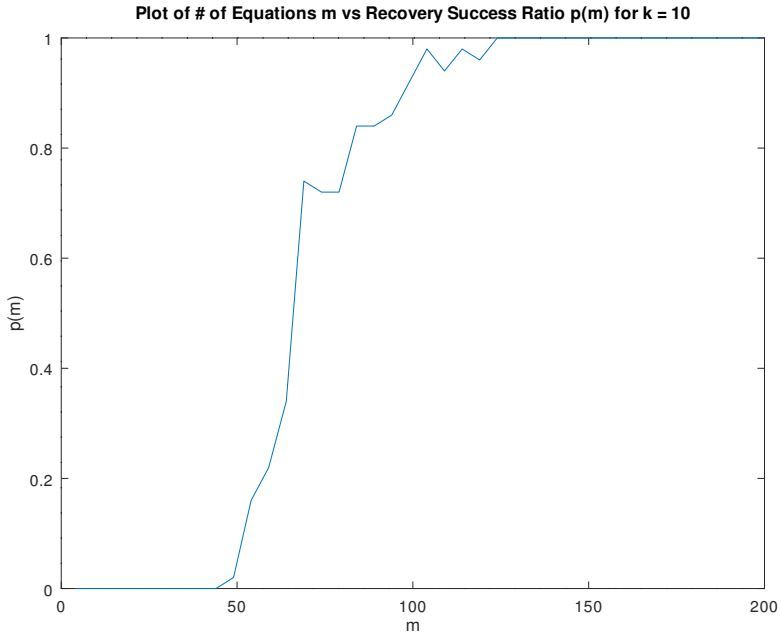

Figure 4: Plot of number of equations vs solution recovery success rate for $k = 10$

From Figure 4, we see that that the first point of successful recovery occurs just below $m = 50$ and sharply increases to $p = 0.8$ at $m = 70$, with the first point where recovery is always achieved occurring at $m = 120$. Thus we find that the range it takes for m to increase from $p = 0$ to 1 is 70.

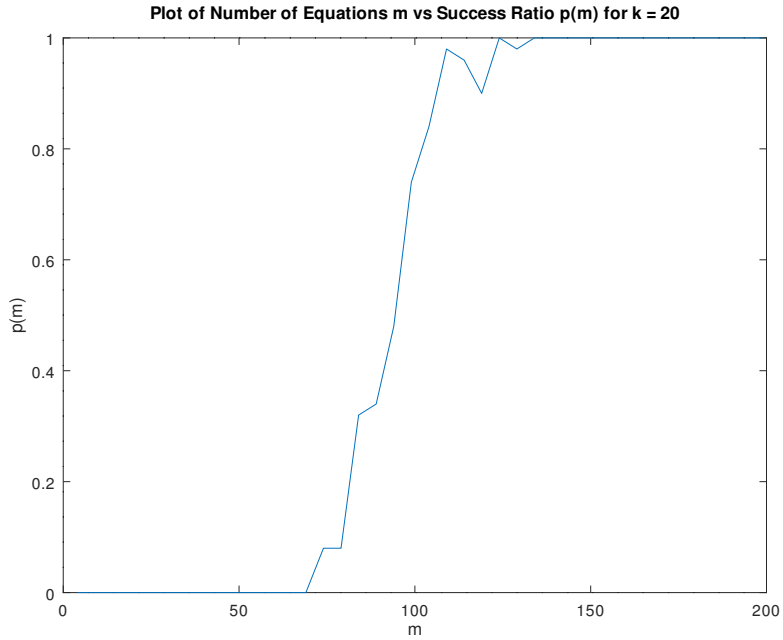

Figure 5: Plot of number of equations vs solution recovery success rate for $k = 20$

Increasing the value of k to 20, we again run the NIHT algorithm and plot the empirical recovery probabilities. The results of this can be seen in Figure 5. Here we find that first point of successful probability occurs at $m = 70$ and reaches the constant success rate at around $m = 130$. Thus the range in which the probability increases from 0 to 1 is 60, a slight decrease from when $k = 10$.

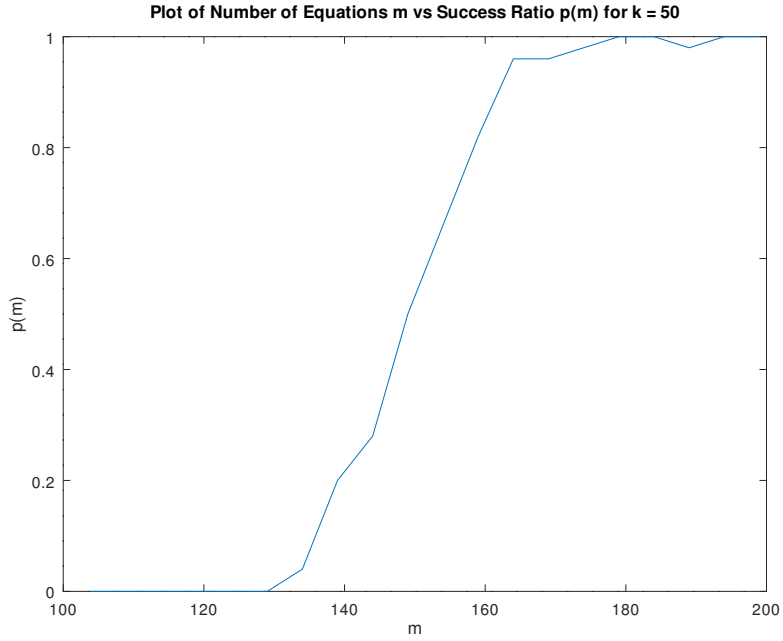

Figure 6: Plot of number of equations vs solution recovery success rate for $k = 50$

In the case of $k = 50$, we find that the solution vector x converges to a solution, but this is

not always the correct solution. Thus our code produces "false-positive" solutions to the system of equations. However, we note that when $m > 2k$, then for most matrices, the solution x is unique. Thus in this case, we look for solutions of x only in the case where $m > 100$. The results of this can be found in Figure 6.

Here, we see that the first point of success occurs at $m = 130$ and reaches close to constant success at around $m = 180$. Thus the range in which the recovery rate changes is 50.

From our analysis, we find that as we increase the sparsity level k of the solution vector x calculated at each step, the range in which the solution recovery rate $p(m) \in (0, 1)$ decreases significantly. We find that for $k = 10$, the range is of width 70, for $k = 20$, the range is 60 and for $k = 50$, the range is 50. Thus, we must increase the value of the sparsity variable exponentially to get an equal reduction in recovery range. Therefore, we must take great care when selecting the sparsity value for our system of equations to ensure we achieve the correct solution at all times.

5 Conclusion

In this report, we investigated and implemented the problem of solving linear systems of equations for differing sizes of the coefficient matrix A. We began by looking at the case where the number of equations to be solved is larger than the number of variables. This condition ensures the system is not fully solvable and thus, we must find the solution which minimises the errors given. We discussed the gradient descent method which is an iterative method used to solve this least squares problem. This method was then implemented in `C++` and tested using a number of coefficient matrices A.

From this programming, we found that altering the coefficients of A has a large impact on the speed of convergence to an appropriate solution. We then considered the condition number of the matrix A which represents the rate at which the solution vector x changes compares to a change in the vector b. From this, we found a clear correlation between the condition number and the number of iterations to reach the solution from our three examples.

We then considered an alteration of the problem whereby the matrix A has more variables than equations. We also imposed the sparsity condition which ensures the problem is sufficiently determined. A thresholding operation was also considered and implemented as this ensured only the k most significant values within the solution vector are carried forward.

We then investigated a phenomenon called phase transition, which is a large change in the behaviour of a system upon varying certain parameters. We ran the NIHT algorithm with a fixed number of variables and sparsity condition k to find a sharp change in the probability of success as the number of equations increases. There is a fine balance between choosing the sparsity value given a number of equations from the system, as the time taken to perform the thresholding operation is not insignificant. Thus we must minimise the sparsity value enough that the solution is found with a high success chance, but still ensuring the run time is not excessive.

Appendix A Source code listings

A.1 Vector Class

```
#ifndef mVector_h
#define mVector_h

#include <vector>
#include <math.h>
#include <cstdlib>
#include <algorithm>

//Normally distrubuted value calculator
double rand_normal()
{
    static const double pi = 3.141592653589793238;
    double u = 0;
    while (u == 0) // loop to ensure u nonzero, for log
    {
        u = rand() / static_cast<double>(RAND_MAX);
    }
    double v = rand() / static_cast<double>(RAND_MAX);
    return sqrt(-2.0*log(u))*cos(2.0*pi*v);
}


// Class that represents a mathematical vector
class MVector
{
public:
    // constructors
    MVector() {}
    explicit MVector(int n) : v(n) {}
    MVector(int n, double x) : v(n, x) {}
    MVector(std::initializer_list<double> l) : v(l) {}

    // access element (lvalue) (see example sheet 5, q5.6)
    double &operator[](int index)
    {
        if(index+1 > v.size() || index < 0){
            std::cout << "Error: Index outside range" << std::endl;
            exit(-1);
        }
        else{
            return v[index];
        }
    }

    // access element (rvalue) (see example sheet 5, q5.7)
    double operator[](int index) const {

        if(index+1 > v.size() || index < 0){
            std::cout << "Error: Index outside range" << std::endl;
            exit(-1);
        }
        else{
            return v[index];
        }
    }

    int size() const { return v.size(); } // number of elements
```

```
58
59      void resize(int n){
60          v.resize(n);
61      }
62
63      //Return standard norm value of a vector
64      double VectNorm(){
65          double NormVal = 0.0;
66
67          for(int i = 0; i < v.size(); i++){
68              NormVal = NormVal + (v[i] * v[i]);
69          }
70          NormVal = sqrt(NormVal);
71
72          return NormVal;
73      }
74
75
76      // Set all but k largest elements to zero
77      void threshold(int k){
78
79          w = v;
80
81          for(int i = 0; i < w.size(); i++){
82              w[i] = abs(w[i]);
83          }
84
85          std::sort(w.begin(), w.end());
86
87          //kth largest element
88          double LargestVal =  w[w.size() - k];
89
90          for(int i = 0; i < v.size(); i++){
91              if(abs(v[i]) < LargestVal){
92                  v[i] = 0;
93              }
94          }
95      }
96
97
98      MVector initialise_normal(int k){
99          MVector returnVector(v.size());
100
101         //fill all elements with normally distributed values and indices
        vector
102         for(int i = 0; i < v.size(); i++){
103             returnVector[i] = rand_normal();
104         }
105
106         for(int i = 0; i < v.size() - k; i++){
107             int randIndex = rand() % v.size();
108
109             if(!(returnVector[randIndex] == 0)){
110                 returnVector[randIndex] = 0;
111             }else{
112                 k = k-1;
113             }
114         }
115
116         return returnVector;
117     }
```

```
118
119
120 private:
121     std::vector<double> v;
122     std::vector<double> w;
123
124 };
125
126
127 // Overload the << operator to output MVectors to screen or file
128 std::ostream& operator<<(std::ostream& os, const MVector& v){
129     int n = v.size();
130     os << "(";
131     for(int i = 0; i < n-1; i++){
132         os << v[i] << ", ";
133     }
134     os << v[n-1] << ")" << std::endl;
135     return os;
136 }
137
138
139
140 #endif
```

A.2 Matrix Class

```
1  #ifndef mMatrix_h
2  #define mMatrix_h
3
4  #include <vector>
5  #include <math.h>
6  #include <cstdlib>
7  #include <algorithm>
8
9  // Class that represents a mathematical matrix
10 class MMatrix
11 {
12 public:
13     // constructors
14     MMatrix() : nRows(0), nCols(0) {}
15     MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n * m, x)
       {}
16
17     // set all matrix entries equal to a double
18     MMatrix &operator=(double x)
19     {
20         for (unsigned i = 0; i < nRows * nCols; i++) A[i] = x;
21         return *this;
22     }
23
24     // access element, indexed by (row, column) [rvalue]
25     double operator()(int i, int j) const
26     {
27         if(i+1 > nRows || j+1 > nCols || i < 0 || j < 0 ){
28             std::cout << "Error: Index outside range" << std::endl;
29             exit(-1);
30         }
31         else{
32             return A[j + i * nCols];
33         }
34
```

```
35      }
36
37      // access element, indexed by (row, column) [lvalue]
38      double &operator()(int i, int j)
39      {
40          if(i+1 > nRows || j+1 > nCols || i < 0 || j < 0 ){
41              std::cout << "Error: Index outside range" << std::endl;
42              exit(-1);
43          }
44          else{
45              return A[j + i * nCols];
46          }
47      }
48
49      // size of matrix
50      int Rows() const { return nRows; }
51      int Cols() const { return nCols; }
52
53
54      //Transpose a matrix
55      MMatrix transpose() const {
56          MMatrix returnMatrix(nCols, nRows);
57
58          for(int i = 0; i < nCols; i++){
59              for(int j = 0; j < nRows; j++){
60                  returnMatrix(i,j) = A[i+j*nCols];
61              }
62          }
63          return returnMatrix;
64      }
65
66      //Initialise Matrix with normally distributed entries of mean zero,
        variable 1/m
67      MMatrix initialise_normal() {
68          MMatrix returnMatrix(nRows, nCols);
69
70          for(int i = 0; i < nRows; i++){
71              for(int j = 0; j < nCols; j++){
72                  returnMatrix(i,j) = rand_normal() / sqrt(nRows);
73              }
74          }
75          return returnMatrix;
76      }
77
78
79  private:
80      unsigned int nRows, nCols;
81      std::vector<double> A;
82  };
83
84
85  // Overload the << operator to output MMatrix to screen or file
86  std::ostream& operator<<(std::ostream& os, const MMatrix& m){
87      int rowsize = m.Rows();
88      int columnsize = m.Cols();
89
90      for(int i = 0; i < rowsize; i++){
91          for(int j = 0; j < columnsize; j++){
92              std::cout.width(10);
93              os << m(i,j);
94          }
```

```
95          os << std::endl;
96      }
97      return os;
98  }
99
100
101
102 #endif /* mMatrix_h */
```

A.3 Operator Overloads

```
1  #ifndef OperatorOverloads_h
2  #define OperatorOverloads_h
3
4  //Overloading Matrix/Vector Product
5  MVector operator*(const MMatrix& A, const MVector& v){
6      MVector returnVect(A.Rows());
7      double temp = 0.0;
8
9      for(int i = 0; i < A.Rows(); i++){
10         temp = 0.0;
11         for(int j = 0; j < v.size(); j++){
12             temp  = temp + A(i,j) * v[j];
13         }
14         returnVect[i] = temp;
15     }
16     return returnVect;
17 }
18
19 //Overloading Vector/Matrix Product
20 MVector operator*(const MVector& v, const MMatrix& A){
21     MVector returnVect(A.Cols());
22     double temp = 0.0;
23
24     for(int i = 0; i < A.Cols(); i++){
25         temp = 0.0;
26         for(int j = 0; j < v.size(); j++){
27             temp  = temp + v[j] * A(j,i);
28         }
29         returnVect[i] = temp;
30     }
31     return returnVect;
32 }
33
34 //Overloading Matrix/Matrix product
35 MMatrix operator*(const MMatrix& A, const MMatrix& B){
36     MMatrix returnMatrix(A.Rows(),B.Cols());
37     double temp = 0.0;
38
39     for(int i = 0; i < A.Rows(); i++){
40         for(int j = 0; j < B.Cols(); j++){
41             temp = 0.0;
42             for(int k = 0; k < A.Cols(); k++){
43                 temp = temp + A(i,k) * B(k,j);
44             }
45             returnMatrix(i,j) = temp;
46         }
47     }
48     return returnMatrix;
49 }
50
```

```
51
52 //Overloading subtraction of 2 vectors
53 MVector operator-(MVector Vect1, MVector Vect2){
54     MVector returnVect(Vect1.size());
55     if(Vect1.size() == Vect2.size()){
56         for(int i = 0; i < Vect1.size(); i++){
57             returnVect[i] = Vect1[i] - Vect2[i];
58         }
59     }
60     return returnVect;
61 }
62
63
64 //Overloading dot product of 2 vectors
65 double operator*(MVector Vect1, MVector Vect2){
66     double returnInt = 0;
67     if(Vect1.size() == Vect2.size()){
68         for(int i = 0; i < Vect1.size(); i++){
69             returnInt = returnInt + (Vect1[i] * Vect2[i]);
70         }
71     }
72     return returnInt;
73 }
74
75 //Overloading scalar/vector product
76 MVector operator*(double a, const MVector& v){
77     MVector returnVector(v.size());
78
79     for(int i = 0; i < v.size(); i++){
80         returnVector[i] = a * v[i];
81     }
82
83     return returnVector;
84 }
85
86
87 //Overloading addition of 2 vectors
88 MVector operator+(MVector Vect1, MVector Vect2){
89     MVector returnVect(Vect1.size());
90     if(Vect1.size() == Vect2.size()){
91         for(int i = 0; i < Vect1.size(); i++){
92             returnVect[i] = Vect1[i] + Vect2[i];
93         }
94     }
95     return returnVect;
96 }
97
98 //Overloading scalar/Matrix product
99 MMatrix operator*(double a, const MMatrix& A){
100     MMatrix returnMatrix(A.Rows(),A.Cols());
101
102     for(int i = 0; i < A.Rows(); i++){
103         for(int j = 0; j < A.Cols(); j++){
104             returnMatrix(i,j) = a * A(i,j);
105         }
106     }
107     return returnMatrix;
108 }
109
110
111
```

```
112 #endif /* OperatorOverloads_h */
```

A.4 SDLS

```
 1 #ifndef SDLS_h
 2 #define SDLS_h
 3
 4 //Steepest Descent Algorithm
 5 int SDLS(const MMatrix& A, const MVector& b, MVector& x, int maxIterations,
       double tol){
 6     int CurrentIter = 1;
 7     double alpha = 0.0;
 8
 9     MVector xVals(maxIterations);
10     MVector yVals(maxIterations);
11
12     xVals[0] = 0;
13     yVals[0] = 0;
14
15     MVector r = A.transpose() * (b - A * x);
16
17     while(r.VectNorm() > tol && CurrentIter <= maxIterations){
18         alpha = (r * r) / (r * A.transpose() * A * r);
19         x = x + alpha * r;
20         r = r - alpha * A.transpose() * A * r;
21
22         xVals[CurrentIter] = x[0];
23         yVals[CurrentIter] = x[1];
24
25         CurrentIter = CurrentIter + 1;
26     }
27
28     if(CurrentIter == maxIterations+1){
29         std::cout << "Max iterations reached without solution found" << std
       ::endl;
30         return maxIterations;
31     }
32     else{
33         return CurrentIter;
34     }
35 }
36
37
38 #endif /* SDLS_h */
```

A.5 NIHT

```
 1
 2 #ifndef NIHT_h
 3 #define NIHT_h
 4
 5
 6 //Normalised Iterative Thresholding Algorithm
 7 int NIHT(const MMatrix& A, const MVector& b, MVector&x, int k, int
       maxIterations, double tol){
 8     int CurrentIter = 1;
 9     double alpha = 0.0;
10     double rNormPrev = 0.0;
11
12     // Initialise starting vector
13     x = A.transpose()*b;
```

```
14      x.threshold(k);   // Get k largest values
15
16      MVector r = A.transpose() * (b - A * x);
17
18
19      while(r.VectNorm() > tol && CurrentIter <= maxIterations){
20          rNormPrev = r.VectNorm();
21          alpha = (r * r) / (r * A.transpose() * A * r);
22          x = x + alpha * r;
23          x.threshold(k);
24          r = A.transpose() * (b - A * x);
25
26          //Tests convergence of successive vector norms
27          if(abs(r.VectNorm() - rNormPrev) < 1e-7){
28              return 0;
29          }
30
31          CurrentIter = CurrentIter + 1;
32      }
33
34      if(CurrentIter == maxIterations+1){
35          //std::cout << "Max iterations reached without solution found" <<
        std::endl;
36          return 0;
37      }
38      else{
39          return 1;
40      }
41  }
42
43  #endif /* NIHT_h */
```

References

[1] S. Foucart and H. Rauhut. (2013) A mathematical introduction to compressive sensing, (336) Applied and Numerical Harmonic Analysis. Birkhuser, Basel. Springer.

[2] C. Liu. (2013). An Optimally Generalized Steepest-Descent Algorithm for Solving Ill-Posed Linear Systems, Journal of Applied Mathematics, (2013), Article ID 154358. Available at: https://doi.org/10.1155/2013/154358, (Accessed 2nd December 2019).

[3] I. Orovi, Vladan Papic, Cornel Ioana, Xiumei Li, and Srdjan Stankovic, (2016). Compressive Sensing in Signal Processing: Algorithms and Transform Domain Formulations, Mathematical Problems in Engineering, (2016), Article ID 7616393, Available at: https://doi.org/10.1155/2016/7616393, (Accessed 3rd December 2019).

[4] J. Shewchuk, An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, Edition 1 $\frac{1}{4}$. Carnegie Mellon University, Pittsburgh, 1994.

[5] S. Sreeharitha, 2018, Compressive Sensing Recovery Algorithms and Applications- A Survey. (396). Available at: https://doi.org/10.1088/1757-899X/396/1/012037 (Accessed 7th December 2019).

[6] Moler, C. (2017). 'What is the Condition Number of a Matrix?', Mathworks, 17th July. Available at https://blogs.mathworks.com (Accessed: 9th December 2019).