1 Introduction

This project explores of the Compressed Sensing problem, with our main focus being the Normalised Iterative Hard Thresholding algorithm (NIHT) - a gradient projection algorithm that recovers compressed data from a linear system defining our problem. We define compressed data to be an accurate representation of the data required while containing fewer non-zero entries. We will begin by implementing a simple algorithm known as the Steepest Descent method, which is used to solve linear least squares problems and serves as a basis for the NIHT algorithm.

Compressed Sensing has many applications; most widely discussed being the acquisition of data in medical imaging. While X-ray CT scans are sufficient for acquiring the image data, the high dose of radiation makes them undesirable to use. However, while MRI scanners are inefficient in recovering high resolution data sets, the use of Compressed Sensing techniques to recover a compressed data sample (hence reducing the computing cost) makes MRI more desirable as they are much safer to use than CT scans [5]. Applications of Compressed Sensing can also be found as far back as World War II (although the techniques used aren't as complex as those today), where soldiers had their blood samples pooled and tested for syphilis, thus reducing the cost required to test blood samples individually [11].

2 Classes

Remark. Due to the size of the code listings for the classes, they will be included in the appendix. Members of a class will be referenced via the listing number and the line ranges including the code discussed by (listing number | line range).

Two classes were implemented during the duration of this project, the `MVector` class (listing 11) and the `MMatrix` class (listing 12). These classes are used to represent the linear systems that our algorithms will solve. The most important member functions of these classes are the sorting functions (11 | 14-32,71-117), which are used for the thresholding step (11 | 122-133) in the NIHT algorithm, and the basic matrix manipulation functions such as transposing (12 | 65-73), matrix-vector/matrix-matrix multiplication (12 | 30-62) and the functions which initialise normal values for vectors (11 | 47-69) and matrices (12 | 19-27).

A couple of things to note here - the member functions for initalising normal values use the function `rand_normal` (listing 10) which returns a standard normal distribution value via the Box-Müller transform. While this is not a member function, it is only used inside member functions and so is included in the appendix. The second note is about the sorting functions discussed above, which make use of the `this` keyword. While this was not explicitly taught in the course, it allows for the use of member functions within member functions of the same class (which would not be possible otherwise as for instance, our `MVector` member functions treat the vector as a `std::vector<double>` and not an `MVector` as we would like).

3 What is Compressed Sensing?

Compressed Sensing is an approach to signal and image processing that allows for the acquisition of high resolution data from fewer sensors than previously thought with the use of numerical optimisation [4]. By 'previously thought' we mean that signal processing was required to follow Nyquist's Law,

Nyquist's Law. [3] *The sampling frequency should be at least twice the highest frequency contained in the signal.*

Put more simply, Nyquist's Law states that a minimum number of samples is required to recover a high resolution signal, however if the signal is sparse then we can reduce the number of measurements needed i.e. instead of recovering a high resolution sample and compressing the data, we would like to instead directly recover the compressed data [7].

Strohmer [12] outlines some of the progress that has been made in the field of Compressed Sensing. For instance, while alot of the theory regarding Compressed Sensing (along with the work done in this project) requires that the matrix used in any calculations be random, using a matrix that is dictated by the laws of the sensing process can speed up recovery algorithms such as NIHT. However, the required work to obtain these structured matrices is rather long winded (combinatorial arguments in regards to the algebraic structure need to be considered) and the performance of such matrices in calculations doesn't have the universal guarantee of the random matrices we use, hence more work needs to be done to allow a wider implementation of these structured matrices.

Strohmer [12] also discusses how Compressed Sensing has inspired the production of new data acquisition hardware. MRI is an example discussed in the introduction but another example is single pixel cameras, which allows for the use of a single pixel instead of thousands to capture images and so it does not require the use of a traditional lens [8]. However, these single pixel cameras aren't used commercially since high resolution cameras in phones and DSLR cameras need to only capture visible light, which the economically viable silicon used in cameras responds to, hence these single pixel cameras are more widely used in industry where the information we want to acquire is found in non-visible wavelengths [1].

4 Gradient Descent

Gradient Descent (or Steepest Descent) is an optimisation algorithm which uses an iterative process to find a local minimum of a function from some given point. The process involves taking steps proportional to the negative of the gradient of the function, while the iterative nature ensures we always take steps in the steepest direction possible so convergence to the minimum is optimal. The gradient descent method is used widely in machine learning today to train models and can be combined with many algorithms (such as NIHT), making it a popular optimisation strategy [6].

In our case, we would like to solve a linear least squares problem, i.e. $\min\{\|A\underline{x} - \underline{b}\|^2\}$, where A is an $m \times n$ matrix with $m \geq n$ (we will call such a matrix *tall*), $\underline{x}$ an n-vector, $\underline{b}$ an m-vector and $\|\cdot\|$ is the Euclidean norm. Hence the function we consider will be of the form

$$f(x) = \frac{1}{2}\|A\underline{x} - \underline{b}\|^2 \ , \tag{1}$$

where $\frac{1}{2}$ is a scaling factor such that the gradient function will be in a "nicer" form. Consequently, the gradient of the function f is given by

$$\begin{aligned} \nabla f &= \frac{1}{2}\nabla(\langle A\underline{x} - \underline{b}, A\underline{x} - \underline{b}\rangle) \\ &= \frac{1}{2}\nabla(\langle A\underline{x}, A\underline{x}\rangle - 2\langle A^T\underline{b}, \underline{x}\rangle + \langle \underline{b}, \underline{b}\rangle) \\ &= \frac{1}{2}(2A^TA\underline{x} - 2A^T\underline{b}) \\ &= A^T(A\underline{x} - \underline{b}) \ , \end{aligned} \tag{2}$$

where $\langle\cdot,\cdot\rangle$ denotes the dot product and A^T denotes the transpose of the matrix A. Hence we will define the *residual* at $\underline{x}$ to be $\underline{r} = -\nabla f(\underline{x})$, which will be used to define the direction of the steps taken to converge to a minimum in our algorithm. Now our iterative steps can converge to a minimum by the function

$$\underline{x}_{i+1} = \underline{x}_i - \alpha_i \nabla f(\underline{x}_i) = \underline{x}_i + \alpha_i \underline{r}_i \ , \tag{3}$$

where i denotes the i^{th} iteration of the algorithm, and α_i is a step size chosen such that the step sizes become smaller as we converge to a minimum, i.e. as the residuals $\underline{r}_i$ become smaller. Thus the step length can be taken to minimise the function $\alpha_i \longmapsto f(\underline{x}_i + \alpha_i \underline{r}_i)$. A similar argument to the one above shows that the step size can be calculated as follows

$$\alpha_i = \frac{\langle \underline{r}, \underline{r}\rangle}{\langle A\underline{r}, A\underline{r}\rangle} \tag{4}$$

Hence our Steepest Descent algorithm will successively calculate α_i, $\underline{x}_{i+1}$ and $\underline{r}_{i+1}$ with the equations (4), (3) and

$$\underline{r}_{i+1} = -\nabla f(\underline{x}_{i+1}) = A^T(\underline{b} - A\underline{x}_{i+1}) \ , \tag{5}$$

respectively, with the initial conditions

$$\begin{aligned} \underline{x}_0 &= \underline{0} \\ \underline{r}_0 &= A^T(\underline{b} - A\underline{x}_0) \\ \alpha_0 &= \frac{\langle \underline{r}_0, \underline{r}_0 \rangle}{\langle A\underline{r}_0, A\underline{r}_0 \rangle} \ . \end{aligned} \tag{6}$$

The iteration continues until $\|\underline{r}\|$ is sufficiently small or the maximum number of iterations is reached (we include this so the program doesn't run infinitely).

Listing 1 includes the function `SDLS` which implements the Gradient Descent algorithm into C++.

```
// function implementing the steepest descent algorithm, returning the number of
// iterations needed for x to a solution (i.e. number of calculations needed for
// the residual to become sufficiently small)
int SDLS(const MMatrix& A, const MVector& b, MVector& x0, int maxIterations,
    double tol, bool output = false)
{
    if (output) // if output==true, write x values for each iteration to a file
    {
        std::ofstream sdlsoutput;
        sdlsoutput.open("sdls.csv");
        MMatrix At = A.transpose(); // transpose of A
        MVector r = At * (b - A * x0); // residual at x
        double alpha; // step length
        for (int iter = 0; iter <= maxIterations; iter++)
        {
            if (r.dot(r) < tol*tol) return iter; // if ||r|| sufficiently small
                                                 // return no. of iterations
            else // calculate values for next iteration
            {
                alpha = r.dot(r) / (A*r).dot(A*r);
                x0 = x0 + r.scale(alpha);
                sdlsoutput << x0 << "\n";
                r = At*(b - A*x0);
            }
        }
        return -1; // if it fails to find a soln within the no. of maxIterations,
                   // return -1
    }
    else
    {
        MMatrix At = A.transpose(); // transpose of A
        MVector r = At * (b - A * x0); // residual at x
        double alpha; // step length
        for (int iter = 0; iter <= maxIterations; iter++)
        {
            if (r.dot(r) < tol*tol) return iter; // if ||r|| sufficiently small
                                                 // return no. of iterations
            else // calculate values for next iteration
            {
                alpha = r.dot(r) / (A*r).dot(A*r);
                x0 = x0 + r.scale(alpha);
                r = At * (b - A * x0);
            }
        }
        return -1; // if it fails to find a soln within the no. of maxIterations,
                   // return -1
    }
```

```
48 }
```

Listing 1: SDLS function implementing the Gradient Descent algorithm

The function takes an input of a matrix A and vectors $\underline{x}_0$ ($= \underline{0}$, an initial guess) and $\underline{b}$ as defined above, while also taking inputs of an integer `maxIterations` which provides a bound for the number of iterations allowed, a real number `tol`, which is used to decide whether the size of the residual $\|\underline{r}\|$ is sufficiently small at each iteration, and a boolean value `output` which writes trajectory of $\underline{x}_i$ to a file if it is input as `true`. The function returns the number of iterations taken to converge to a minimum, or it returns -1 if the maximum number of iterations allowed is reached.

We now look at the convergence of $\underline{x}$ for a 3×2 system. Consider the system

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \end{pmatrix} \quad , \quad \underline{b} = \begin{pmatrix} 10 \\ -1 \\ 0 \end{pmatrix} \quad , \tag{7}$$

and define the maximum number of iterations to be 1000 and the tolerance of $\|\underline{r}\|$ to be 10^-6. Listing 2 shows the code that inputs this into the `SDLS` function and outputs the number of iterations taken to converge to a minimum.

```
1  // task for SDLS algortihm to find convergence of x to a minimum for the system
2  // initialised below
3  int main()
4  {
5      std::srand(std::time(NULL)); // avoid same seq of random values each time the
6                                   // the program is run
7      MMatrix A(3, 2);
8      MVector b(3);
9      MVector x0(2);
10     // initialise A as in task
11     A.set(0, 0, 1.0);
12     A.set(1, 0, 2.0);
13     A.set(2, 0, -1.0);
14     A.set(0, 1, 2.0);
15     A.set(1, 1, 1.0);
16     A.set(2, 1, 0.0);
17     // initialise b as in task
18     b[0] = 10;
19     b[1] = -1;
20     b[2] = 0;
21
22     std::cout << SDLS(A, b, x0, 1000, 1.0e-06, true); // also writes x trajectory
23     return 0;                                         // to a file
24 }
```

Listing 2: Code that finds the minimum for the system (7)

The output of the `SDLS` function shows that this system converges to a minimum in 39 iterations, which seems pretty fast given our maximum iteration bound was 1000, but what happens if we change the matrix A slightly? Setting the bottom row to be $(1.8, -2)$ (call this matrix A_1) shows that the `SDLS` function converges to a minimum in just 3 iterations, a drastic improvement. Changing the matrix slightly again and setting the bottom row to be $(-2, -2)$ (call this matrix A_2) shows that the `SDLS` function converges to a minimum in 75 iterations, a deterioration over the use of `SDLS` over both of the previous systems. How can we explain this behaviour?

We consider the condition number of the matrix, which can help us answer the question "How much can a change in the right hand side of a system of simultaneous linear equations $A\underline{x} = \underline{b}$ affect the solution?" [10].

Definition. [9] Let $sing(A) := \{+\sqrt{\lambda} \mid \lambda \text{ an eigenvalue for } A^TA\}$ be the set of singular values of an $m \times n$ matrix A. We define the condition number of A to be

$$\kappa(A) = \frac{\max sing(A)}{\min sing(A)} \tag{8}$$

.

While Moler [10] states that there are many condition numbers for a matrix depending on the problem (i.e. are we trying to invert the matrix or compute it's eigenvalues?) this definition is sufficient, and computing the condition number using (8) in the definition above is equivalent to using the `cond` command in MATLAB. Listing 3 contains the MATLAB code used to calculate the condition number of the matrices A, A_1, A_2 using both the equation above and the `cond` command.

```
% initialise matrices
A = [1,2;2,1;-1,0];
A1 = [1,2;2,1;1.8,-2];
A2 = [1,2;2,1;-2,-2];
% caculating vector of singular values of matrices above
singA = sqrt(eig(transpose(A)*A));
singA1 = sqrt(eig(transpose(A1)*A1));
singA2 = sqrt(eig(transpose(A2)*A2));
% condition numbers calculated using the definition
cA = max(singA)/min(singA)
cA1 = max(singA1)/min(singA1)
cA2 = max(singA2)/min(singA2)
% condition numbers calculated using cond command
condA = cond(A)
condA1 = cond(A1)
condA2 = cond(A2)
```

Listing 3: MATLAB code for calculating the condition numbers

The output of the code above shows the condition numbers of the matrices to be

$$\kappa(A) \approx 2.55, \quad \kappa(A_1) \approx 1.07, \quad \kappa(A_2) \approx 4.12. \tag{9}$$

A matrix is defined to be *ill conditioned* if its condition number is large, and by *ill conditioned*, we mean that small changes in the right hand side of the solution $A\underline{x} = \underline{b}$ cause changes in the solution of the system in proportion to $\kappa(A)$ [10]. This agrees with the test we did on our 3×2 system, where the matrix A_1 caused a minimum to be found in the least number of iterations of the Steepest Descent algorithm, whereas A_2 required the most number of iterations of the Steepest Descent algorithm to converge to a minimum of the function. Hence the ordering (from smallest to largest, say) of the number of iterations needed to converge to a minimum agrees with the ordering of the condition numbers, and so we may conclude that the more ill conditioned a matrix is, it will require more iterations of the Steepest Descent algorithm to converge to a minimum of the function.

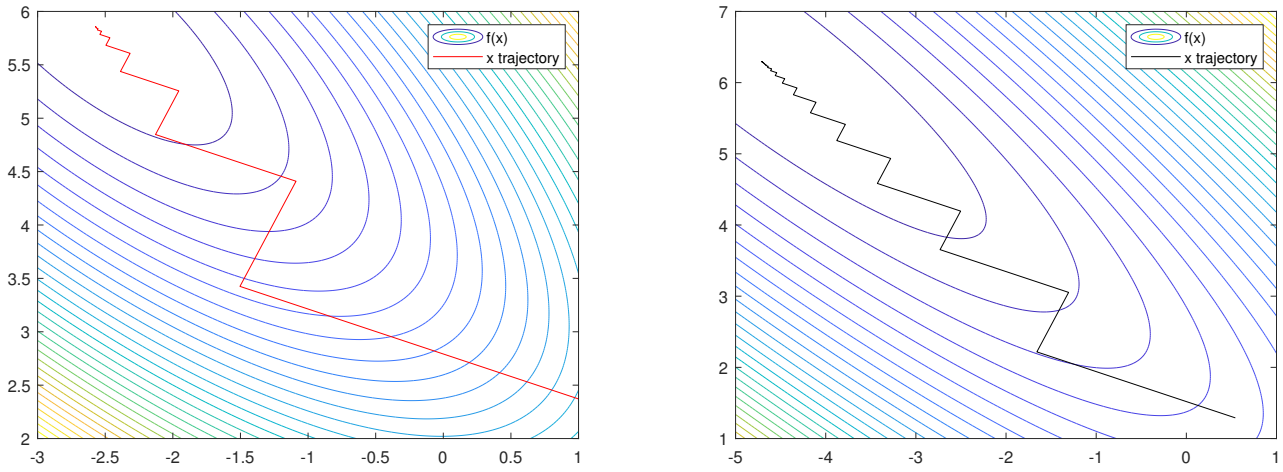
We now want to produce a plot the shows the convergence of the points $\underline{x}_i$ in the Steepest Descent algorithm to a minimum of our function $f(\underline{x})$ in (1). The trajectory of $\underline{x}_i$ ($i \in [0, \texttt{SDLS output}]$) was output using the `SDLS` function in listing 2 by setting the boolean variable `output` to `true`. Hence listing 4 shows the MATLAB code which plots trajectory of $\underline{x}_i$ with respect to the function $f(\underline{x})$, whilst figure 1 shows the plots output by the code for the systems (7) with matrices A and A_2 respectively (the system using A_1 has such a quick convergence rate that it is difficult to see the trajectory, hence the plot is omitted).

N.B. Some of the file names may have been changed between the writing by C++ and the input into MATLAB to allow for simpler/safer implementation when producing the plots.

```
% read the x trajectory file and put in correct format
Xtraj = csvread("sdls_A_xtraj.csv");
Xtraj(:,end) = [];
Xtraj;
% intialise matrix and vector for system
A = [1,2;2,1;-1,0];
b = [10;-1;0];
% create grid for contour plot
x1 = transpose(linspace(-3,1,200));
```

```
x2 = transpose(linspace(2,6,200));
[X,Y] = meshgrid(x1,x2);
fx = zeros(200);
% calculate f(x) values for contour plot
for i = 1:200
    for j = 1:200
        xi = [X(i,j);Y(i,j)];
        fx(i,j) = (norm(A*xi - b)^2)/2;
    end
end
% plot the system and x trajectory
contour(X,Y,fx,30)
hold on
plot(Xtraj(:,1),Xtraj(:,2))
```

Listing 4: MATLAB code producing the plots in figure 1



(a) (b)

Figure 1: Plots of the $\underline{x}_i$ trajectories for the systems defined by A (a) and A_2 (b)

The jaggedness of the lines of each plot represents the number of iterations taken to converge to the minimum of each function, so as we expect the plot (1b) representing the system defined by A_2 has a more jagged trajectory than the system defined by A (1a). At each point, the Steepest Descent algorithm takes steps perpendicular to the isolines, i.e. steps in the direction of the maximum value of the gradient at the point $\underline{x}_i$. We can see from the plots that a smaller condition number $\kappa(A)$ causes the contour lines to become more circular in shape, which makes sense. The more circular the contour lines, the closer to the centre (or minimum of the function) the residuals $\underline{r}_i$ will point, thus providing a faster convergence to the minimum of a system, further proof that a more well conditioned matrix (i.e. a matrix A such that $\kappa(A)$ is small) provides a faster convergence to a minimum in the Steepest Descent algorithm.

5 Iterative Thresholding and NIHT

The Normalised Iterative Hard Thresholding (NIHT) algorithm is a Compressed Sensing technique for recovering compressed data directly from a signal, but how do we define compressed data? In the context of this project, we define it as follows:

Definition. An n-vector $\underline{x}$ is k-sparse for $k < n$ if it has at most k non-zero entries.

In fact, in our case we would like to solve the system $A\underline{x} = \underline{b}$, where A is an $m \times n$ matrix such that $m < n$ (we will call such a matrix *fat*), and so we define $\underline{x}$ to be k-sparse for $k < m$ since the number of equations m is smaller than the number of unknowns n.

Aside. Blumensath and Davies [2] define the system as $A\underline{x} + \underline{e} = \underline{b}$, where $\underline{e}$ is an m-vector describing the observation noise captured by (for example) the sensors observing the system. We will not consider this in order to simplify the problem and the implementation of the NIHT algorithm in C++.

The problem that NIHT is aiming to solve against other algorithms in Compressed Sensing is that we don't know which components of $\underline{x}$ are non-zero before we aim to solve the solution. If this were the case (i.e. if we knew I was the set of indices for which these components of $\underline{x}$ were non-zero), then we could just remove all the components with indices not in I and (and the respective columns in A) to gain a a system $A_I\underline{x}_I = \underline{b}$, where A_I is the matrix A with the columns not indexed by I removed, and $\underline{x}_I$ is the vector x with the components not indexed by I removed. This system contains a *tall* matrix A_I, and so we can just use the Steepest Descent algorithm from section 4 to solve our system.

Unfortunately this is not the case, so NIHT implements a step into the Steepest Descent method called a *thresholding* operation. Let $\mathcal{H}_k(\underline{x})$ be the non-linear thresholding operator that sets all but the k largest (in absolute value) components of $\underline{x}$ to 0. Then the step (3) in the Steepest Descent algorithm is replaced by

$$\underline{x}_{i+1} = \mathcal{H}_k(\underline{x}_i + \alpha_i A^T(\underline{b} - A\underline{x}_i)) = \mathcal{H}_k(\underline{x}_i + \alpha_i \underline{r}_i) \ , \tag{10}$$

hence throughout the calculation $\underline{x}_i$ contains at most k many non-zero components.

The NIHT algorithm also relies on the fact that the *restricted isometric property* (RIP) holds for our matrix A, which means that with respect to k-sparse vector $\underline{x}$, A is "nearly orthonormal", i.e. A changes the length of the vector $\underline{x}$ "very little" given $\underline{x}$ is k-sparse. More succinctly,

Restricted Isometry Property. [2] *A matrix A satisfies the Restricted Isometry Property of order k if there exists $\delta_k < 1$ such that*

$$(1 - \delta_k)\langle \underline{x}, \underline{x}\rangle \leq \langle A\underline{x}, A\underline{x}\rangle \leq (1 + \delta_k)\langle \underline{x}, \underline{x}\rangle \ , \tag{11}$$

where $\underline{x}$ is k-sparse. The smallest δ_k such that (11) *holds is called Restricted Isometry Constant.*

There is no known efficient method to calculate the *Restricted Isometry Constant* but Blumensath and Davies [2] implement the use of random matrices into the NIHT algorithm, as normal $m \times n$ matrices (of mean 0 and variance m^{-1}) have a high probablilty of satisfying RIP, which gives some weight as to why we implement the thresholding step.

The thresholding step adds alot of computation to the algorithm versus the standard Steepest Descent method in section 4 (recall the code for `threshold` which implements the thresholding operator $\mathcal{H}_k$ is contained in the appendix at (11 | 122-133)). The function `threshold` requires that we create a copy of $\underline{x}$, so that the copy can be sorted using Quicksort and we can retrieve the k^{th} largest element of $\underline{x}$, and then set all components of the original $\underline{x}$ less than or equal to this element equal to zero. The Quicksort algorithm runs in $\mathbf{O}(n \log n)$ time, whereas the pass to set the $n - k$ elements to zero runs in $\mathbf{O}(n)$ time, adding a considerable amount of computation to our algorithm at each iteration. We could implement some methods to our sorting functions so they run quicker - for instance we only require that the k^{th} largest element be in the correct place, so we can terminate Quicksort when a pivot is placed at the $(n-k)^{\text{th}}$ index of the vector. It might be better to implement Heapsort for sorting, as after creating the maximal heap it will sort the vector - starting by placing the largest element at the end and continuing in this way, ensuring the $(n-k)^{\text{th}}$ index sorted correctly in a more regular time over the iterations. However, the implementation of the *median of three* rule in the Quicksort algorithm (i.e. choosing the median of the first, middle and last components of the vector) avoids worst case scenarios, hence making it more efficient than Heapsort. No such implementations were made as the vectors considered in this project were considerably small compared to the vectors sorted in the Sorting Algorithms project. Instead, efforts were focused on optimising the implementation of the NIHT algorithm. Listing 5 contains the function which implements the NIHT algorithm in C++.

```
1 // function implementing the normalised iterative hard thresholding algortihm,
2 // returning the number of iterations needed for x to converge to a solution (i.e.
3 // no. of calculations needed for residual to become sufficiently small)
```

```
4  int NIHT(const MMatrix& A, const MVector& b, MVector&x0, int k,
5      int maxIterations=1000, double tol=1.0e-6)
6  {
7
8      // initialise starting vector
9      x0 = A.transpose()*b; // x = A^T*b
10     x0.threshold(k); // threshold x to make k sparse
11     MMatrix At = A.transpose();
12     MVector r = At * (b - A * x0), r_ip1(x0.size()); // residuals (r_ip1 for next
                                                        // iteration)
13     double alpha, change_in_r;
14     for (int iter = 0; iter <= maxIterations; iter++)
15     {
16         if (r.dot(r) < tol*tol) { return iter; } // if ||r|| sufficiently small
                                                    // return no. of iterations
17         else
18         {
19             alpha = r.dot(r) / (A*r).dot(A*r);
20             x0 = x0 + At.scale(alpha)*(b - A * x0);
21             x0.threshold(k); // thresholding step
22             r_ip1 = At * (b - A * x0);
23             change_in_r = abs(r.dot(r) - r_ip1.dot(r_ip1)); // change in residuals
24             if (change_in_r < 1e-16) return -1; // if residuals don't sufficiently
25                                                 // change, nonconvergence occurs
26             else r = r_ip1; // continue with algorithm
27         }
28     }
29     return -1; // return -1 if it does not converge
30 }
```

Listing 5: `NIHT` function implementing the NIHT algorithm

N.B. The optimisation of the code has already been implemented above, where the algorithm terminates if the residuals aren't changing sufficiently between iterations. For clarity, lines 22-26 are replaced with the line `r = At * (b - A * x0);` for the function before optimisation.

Due to the residuals $\underline{r}_i$, the vectors $\underline{x}_i$ and the step size α_i being dependent on each other in iterative calculations, if the length of the vector $\underline{r}_i$ doesn't change sufficiently between calculations, then neither will the step size α_i, hence the vector $\underline{x}_i$ is unlikely to converge to a solution. Tests were ran to find whether insufficient changes to $\langle \underline{r}_i, \underline{r}_i \rangle$ between calculations would result in the non convergence of $\underline{x}_i$ for a system.

Listing 6 contains the function `niht_residual_test` which returns the number of iterations at which the residuals insufficiently change within a tolerance of 10^{-16} and listing 7 contains the code which tests this hypothesis for a number of systems satisfying the conditions which the NIHT algorithm requires.

```
1  // function that tests whether the residual is changing in value enough between
2  // iterations of the NIHT algorithm
3  int niht_residual_test(const MMatrix& A, const MVector& b, MVector&x0, int k,
4      int maxIterations=1000)
5  {
6      // initialise starting vector
7      x0 = A.transpose()*b; // x = A^T*b
8      x0.threshold(k);
9      MMatrix At = A.transpose();
10     MVector r = At * (b - A * x0), r_ip1(x0.size()); // residuals (r_ip1 for next
                                                        // iteration)
11     double alpha, close;
12     for (int iter = 0; iter <= maxIterations; iter++)
13     {
14         alpha = r.dot(r) / (A*r).dot(A*r);
15         x0 = x0 + At.scale(alpha)*(b - A * x0);
16         x0.threshold(k);
```

```
        r_ip1 = At * (b - A * x0);
        // if r changes insufficiently between iterations, algorithm won't converge
        // to x
        close = abs(r.dot(r) - r_ip1.dot(r_ip1));
        if (close < 1e-16) { return iter; }
        else r = r_ip1;
    }
    return -1; // return -1 if it does not converge
}
```

Listing 6: `niht_residual_test` function for testing the hypothesis

```
// residuals not changing test
int main()
{
    // residuals not changing test
int main()
{
    std::srand(std::time(NULL));
    int m, n, k, niht_it, resfail_it;
    for (int it = 1; it <= 200; it++)
    {
        n = rand() % 91 + 10; // random int in [10,100]
        m = rand() % (n-9) + 9; // random int in [9,n-1]
        k = rand() % ((m / 2) - 4) + 4; // random in [4,m/2 - 1]
        std::cout << "(m,n,k) = " << m << "," << n << "," << k << "\n";
        MMatrix A(m, n);
        A.initialise_normal(); // initialise normal matrix A
        MVector x(n), x0(n), x00(n);
        x.initialise_normal(k); // initialise normal k sparse vector x
        MVector b = A * x;
        niht_it = NIHT(A, b, x0, k);
        // if it converges to wrong x, set niht_it to -1
        if (niht_it != -1 && !(x - x0).has_converged()) niht_it = -1;
        resfail_it = niht_residual_test(A, b, x00, k);
        std::cout << "no. of iter to converge = " << niht_it <<
            " | | r stops changing at = " << resfail_it << "\n";
    }
    return 0;
}
}
```

Listing 7: Code implementing the test for the hypothesis

A fraction of the output of the code from listing 7 is included in the appendix (listing 13) but the conclusion will be explained here - most of the time, systems which did not converge to a solution (or converged to an incorrect solution) had their residuals change insufficiently within the maximum iteration bound, whereas systems which did converge to a solution always had their iteration at which the residuals insufficiently changed after the iteration at which the solution was found for the samples considered. Hence we can conclude that this hypothesis is correct and so this is implemented into the `NIHT` function to increase the efficiency of the algorithm.

6 NIHT and the Phase Transition phenomenon

In the context of computation and numerical analysis, a phase transition is a sharp change in the character of a problem as the parameters vary. With respect to NIHT and Compressed Sensing in general, we define the phase transition to occur when the change in inputs cause the successful recovery rate to go from almost 0 to almost 1 at a sharp rate.

In this report, we wish to find the points of transition with respect to m for A a random $m \times n$ matrix ($n = 200$), $\underline{x}$ a random n-vector of sparsities $k = 10, 20, 50$ and $\underline{b} = A\underline{x}$ an m-vector. We will compute

NIHT 100 times for each $m \in \{4, 9, 14, ..., 199\}$, and so the success rate $p(m)$ will be computed by $p(m) = \#\text{successes}/100$. Listing 8 shows the code which outputs the parameters m, n, k and $p(m)$ to a file which can be used for plotting.

```
// write p(m) values to a file for 4<=m<=199 (steps of 5)
// screen prints are for making sure program is running
int main()
{
    std::srand(std::time(NULL)); // avoid same seq of random values
    std::ofstream niht_successrate; // open file for writing values
    niht_successrate.open("niht_successrate.csv");
    niht_successrate << "m,n,k,p(m)" << "\n";
    int n = 200, k = 20, T = 100; // T the no. of times NIHT is run
    double pm; // pm the successful recovery rate
    for (int m = 4; m < 200; m += 5)
    {
        pm = 0.0;
        for (int t = 0; t < T; t++)
        {
            MVector x(n), x0(n); // x random, x0 for NIHT
            x.initialise_normal(k);
            MMatrix A(m, n);
            A.initialise_normal();
            MVector b = A * x;
            int it = NIHT(A, b, x0, k);
            // compare x and x0 to see if they are really the same
            if (it != -1 && (x-x0).has_converged()) {
                std::cout << "no. of iterations = " << it << "\n";
                pm += 1.0;
            }
            else std::cout << "does not converge \n";
        }
        std::cout << "m = " << m << " is done\n";
        pm /= T;
        niht_successrate << m << "," << n << "," << k << "," << pm << "\n";
    }
    return 0;
}
```

Listing 8: Code outputting $p(m)$ values to a file

N.B. The code in listing 8 only shows the output for sparsity $k = 10$. However, the only thing that is changed for the other sparsities is this k value and so the code is omitted.

Now that we have the successful recovery rates for the m required, we can plot the data and see where the phase transitions occur. Listing 9 shows the MATLAB code which produces a plot of the successful recovery rates $p(m)$ against the number of rows in A (equivalently, the number of equations) m, while figure 2 shows the plot produced by the code.

```
% produce arrays of information
A = csvread("niht_successrate_k10.csv",2);
B = csvread("niht_successrate_k20.csv",2);
C = csvread("niht_successrate_k50.csv",2);
% extract information needed from arrays
m1 = A(:,1);
pm1 = A(:,4);
m2 = B(:,1);
pm2 = B(:,4);
m3 = C(:,1);
pm3 = C(:,4);
% produce plot of p(m) against m for sparsities k = 10, 20, 50
nihtplot = plot(m1,pm1,'b',m2,pm2,'r',m3,pm3,'k')
xlabel("m (no. of rows in A)");
```

```
15 ylabel("p(m) (successful convergence rate)");
16 legend("k = 10","k = 20","k = 50");
17 ax = nihtplot.Parent;
18 set(ax,'XTick',0:20:200,'YTick',0:0.1:1);
```

Listing 9: MATLAB code producing the plot in figure 2

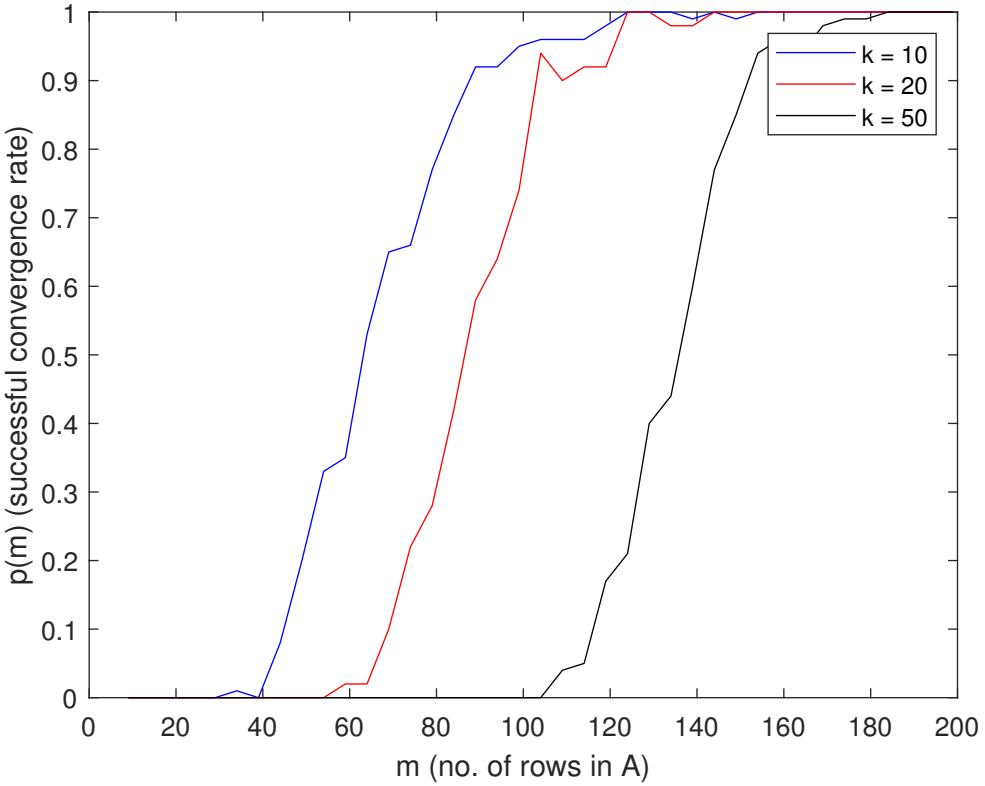


Figure 2: Plot of the recovery rate $p(m)$ against the number of rows m for vectors $\underline{x}$ of sparsity $k = 10, 20, 50$

From this plot, we can deduce that the phase transition for each of the sparsities k allows occurs for a point $m_0 \geq 2k$, which is the kind of behaviour we would expect - there are an insufficient number of equations at this point and so a solution is unlikely to be determined. At least $2k$ equations are required because intuitively, we require at least k equations to solve the system of with a k-sparse vector $\underline{x}$, however ahead of time we don't know where the k non zero components lie in $\underline{x}$ and so another k equations are needed to 'find' these non zero components. We can deduce from the graphs alone that the points m_0 of transition lie in the interval $[40, 100]$ for $k = 10$, $[65, 120]$ for $k = 20$ and $[110, 170]$ for $k = 50$. These intervals aren't very small, which is due to the fact that this is a very simple implementation of NIHT. More complex implementations of the algorithm, such as the one discussed by Blumensath and Davies [2] give rise to sharper transitions as their plots show.

Looking at the output produced by the code in listing 8 directly and defining "overwhelming probability of success" to be $p(m) \geq 0.9$, we can conclude that phase transitions occur at $m_0 \approx 88$ for $k = 10$, $m_0 \approx 103$ for $k = 20$ and $m_0 \approx 152$ for $k = 50$.

7 Conclusion

Throughout this report we have implemented the algorithms of Steepest Descent and Normalised Iterative Hard Thresholding, and shown how the conditioning of the matrix describing the system affects how the algorithm converges to a solution. We deduced directly that ill-conditioned matrices will cause the algorithms to require more iterations to converge to a solution, while matrices that were sufficiently

well-conditioned would cause the algorithm to converge to a solution almost instantly. We also discussed the positives and negatives of the thresholding step of the NIHT algorithm and showed the phase transitions of NIHT for sparsities of $k = 10, 20, 50$. The implementation of NIHT was rather simple, however the power of Compressed Sensing should not be underestimated - the discussion of its applications in the Introduction and the section on Compressed Sensing show that it is a powerful tool in data acquisition and signal processing.

8 Appendix

```
// returns a standard normal value (i.e. mean 0 and variance 1)
double rand_normal()
{
    static const double pi = 3.141592653589793238;
    double u = 0, v;
    while (u == 0) // loop to ensure u non-zero for log
        u = rand() / static_cast<double>(RAND_MAX);
    v = rand() / static_cast<double>(RAND_MAX);
    return sqrt(-2.0*log(u))*cos(2.0*pi*v);
}
```

Listing 10: `rand_normal` function

```
// Class that represents a mathematical vector
class MVector
{
public:
    /* constructors */
    MVector() : v(0) {}

    // initialise empty vector of size n
    explicit MVector(int n) : v(n) {}

    // initialise vector of size n with values x
    MVector(int n, double x) : v(n, x) {}

    /* sorting functions */

    // returns no. of elements in vector
    unsigned size() const { return v.size(); }

    // swaps two elements in a vector
    void swap(int i, int j)
    {
        double temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }

    // compares order of two elements in a vector
    bool cmp(int i, int j)
    {
        if (v[i] < v[j]) return true;
        else return false;
    }


    /* initialise random vector */

    // for an n-vector, give each place a random value in the range [xmin,xmax]
    void initialise_random(double xmin, double xmax)
    {
        size_t s = v.size();
        for (size_t i = 0; i < s; i++)
        {
            v[i] = xmin + (xmax - xmin)*rand() / static_cast<double>(RAND_MAX);
        }
    }

    // initialise random k-sparse vector with (standard) normally distributed
    // values
    void initialise_normal(int k)
```

```
50      {
51          int i = 0;
52          while (i < k)
53          {
54              int j = rand() % v.size(); // produce random index for vector v
55              if (v[j] == 0.0)
56              {
57                  v[j] = rand_normal(); // if index empty (i.e. = 0) place random
58                  i++;                  // value here
59              }
60          }
61      }
62
63      // returns the dot product of vectors v and w
64      double dot(MVector w) const
65      {
66          double dp = 0.0;
67          for (unsigned i = 0; i < v.size(); i++) dp += v[i] * w[i];
68          return dp;
69      }
70
71      // implements median of three rule, places the median of the first, last
72      // middle elements of the subsequence of the vector at the start
73      void MedOfThree(int start, int end)
74      {
75          int mid = start + (end - start) / 2;
76          if (this->cmp(end, start)) this->swap(end, start);
77          if (this->cmp(mid, start)) this->swap(mid, start);
78          if (this->cmp(end, mid)) this->swap(end, mid);
79          this->swap(start, mid);
80      }
81
82      // implements the quick sorting step - places the pivot in the correct places,
83      // all elements less than pivot before it and all elements more than pivot
84      // above it
85      int CorrectPivotPlacement(int start, int end)
86      {
87          this->MedOfThree(start, end); // choose pivot and place at start
88          int p_goes_here = start + 1; // index of pivot
89          for (int i = start + 1; i <= end; i++)
90          {
91              // if v[i] < pivot, increase index of pivot by 1
92              if (this->cmp(i, start))
93              {
94                  this->swap(p_goes_here, i);
95                  p_goes_here++;
96              }
97          }
98          p_goes_here--;
99          this->swap(start, p_goes_here); // place pivot in correct position
100         return p_goes_here; // return index of pivot
101     }
102
103     // recursive call for quick sort
104     void quick_recursive(int start, int end)
105     {
106         if (start == end) { ; } // do nothing if vector is empty or 1 dim.
107         else
108         {
109             int pivot_index = this->CorrectPivotPlacement(start, end);
110             // do quicksort on either side of pivot if possible
111             if (pivot_index != start) this->quick_recursive(start, pivot_index - 1)
    ;
112             if (pivot_index != end) this->quick_recursive(pivot_index + 1, end);
```

```
113         }
114     }
115
116     // wrapper for quick recursive
117     void quick() { this->quick_recursive(0, v.size() - 1); }
118
119     // makes all values in vector absolute
120     void absall() { for (int i = 0; i < v.size(); i++) v[i] = abs(v[i]); }
121
122     // sets all but k absolutely largest elements to 0 in the vector
123     void threshold(int k)
124     {
125         // copy vector v to w and quicksort to find kth largest element
126         MVector w(v.size());
127         w = *this;
128         w.absall();
129         w.quick();
130         // set all elements absolutely smaller than kth largest element of w to 0
131         double kthmax = w[v.size() - k];
132         for (int i = 0; i < v.size(); i++) { if (abs(v[i]) < kthmax) v[i] = 0.0; }
133     }
134
135     // returns the Euclidean norm of a vector v
136     double norm()
137     {
138         double modv = 0.0;
139         for (unsigned i = 0; i < v.size(); i++) modv += v[i] * v[i];
140         modv = std::sqrt(modv);
141         return modv;
142     }
143
144     // scales the vector v with respect to scalar a
145     MVector scale(double a)
146     {
147         MVector x(v.size());
148         for (int i = 0; i < v.size(); i++) x[i] = a * v[i];
149         return x;
150     }
151
152     // checks for convergence to the vector in the SDLS and NIHT algorithms
153     // (within an error of 1.0e-5 for each index)
154     bool has_converged()
155     {
156         for (int i = 0; i < v.size(); i++) if (abs(v[i]) > 1.0e-5) return false;
157         return true;
158     }
159
160     /* overloaded operator functions */
161     // access element (lvalue)
162     double &operator[](int index)
163     {
164         assert(index >= 0 && index < v.size());
165         return v[index];
166     }
167
168     // access element (rvalue)
169     double operator[](int index) const
170     {
171         assert(index >= 0 && index < v.size());
172         return v[index];
173     }
174
175     // set one vector to take the values of another vector
176     void operator=(MVector const w)
```

```
    {
        assert(v.size() == w.size());
        for (int i = 0; i < v.size(); i++) v[i] = w[i];
    }

    // coordinate-wise vector addition
    MVector operator+(MVector w) const
    {
        assert(v.size() == w.size());
        MVector x(v.size());
        for (int i = 0; i < v.size(); i++) x[i] = v[i] + w[i];
        return x;
    }

    // coordinate-wise vector subtraction
    MVector operator-(MVector w) const
    {
        assert(v.size() == w.size());
        MVector x(v.size());
        for (int i = 0; i < v.size(); i++) x[i] = v[i] - w[i];
        return x;
    }

    // print n-vector in the format v0,v1,....,vn,
    friend std::ostream& operator<<(std::ostream& os, const MVector& v);

private:
    std::vector<double> v;
};

// print MVector in the format v0 ,v1 ,.... , vn ,
std::ostream & operator <<(std::ostream & os, const MVector & v)
{
    for (unsigned i = 0; i < v.size(); i++) { os << v[i] << ","; }
    return os;
}
```

Listing 11: `MVector` class

```
// Class that represents a mathematical matrix
class MMatrix
{
public:
    // Constructors and destructors (see also vector class)
    explicit MMatrix() : mRows(0), nCols(0) {}
    MMatrix(int n, int m) : mRows(n), nCols(m), mtrx(n, std::vector<double>(m)) {}
    // Include other methods here

    // initialise a random matrix of values in the range [xmin,xmax]
    void initialise_random(double xmin, double xmax)
    {
        for (int i = 0; i < mRows; i++)
            for (int j = 0; j < nCols; j++)
                mtrx[i][j] = xmin + (xmax - xmin)*rand()
                / static_cast<double>(RAND_MAX);
    }

    // intialise a random matrix of normally distributed values with mean zero and
    // variance 1/m (for an m x n matrix)
    void initialise_normal()
    {
        double sd = 1.0 / sqrt(mRows);
        for (int i = 0; i < mRows; i++)
            for (int j = 0; j < nCols; j++)
```

```
26                  mtrx[i][j] = rand_normal() * sd; // place random value here
27      }
28
29
30      // matrix-matrix multiplication
31      MMatrix operator*(MMatrix B)
32      {
33          assert(nCols == B.mRows); // check if dimensions are sufficient for
34                                    // matrix- matrix multiplication
35          MMatrix C(mRows, B.nCols);
36          for (unsigned i = 0; i < mRows; i++)
37              for (unsigned j = 0; j < B.nCols; j++)
38              {
39                  double Cij = 0.0;
40                  for (unsigned k = 0; k < nCols; k++)
41                      Cij += mtrx[i][k] * B.mtrx[k][j];
42                  C.mtrx[i][j] = Cij;
43              }
44          return C;
45      }
46
47      // matrix-vector multiplication
48      MVector operator*(MVector v) const
49      {
50          assert(v.size() == nCols); // check if dimensions are sufficient for
51                                     // matrix-vector multiplication
52          MVector w(mRows);
53          for (unsigned i = 0; i < mRows; i++)
54          {
55              double wi = 0.0;
56              for (unsigned k = 0; k < v.size(); k++)
57              {
58                  wi += mtrx[i][k] * v[k];
59              }
60              w[i] = wi;
61          }
62          return w;
63      }
64
65      // return the transpose of a matrix
66      MMatrix transpose() const
67      {
68          MMatrix B(nCols, mRows);
69          for (int i = 0; i < mRows; i++)
70              for (int j = 0; j < nCols; j++)
71                  B.mtrx[j][i] = mtrx[i][j];
72          return B;
73      }
74
75      // scale a matrix with respect to a scalar a
76      MMatrix scale(double a) const
77      {
78          MMatrix B(mRows, nCols);
79          for (int i = 0; i < mRows; i++)
80              for (int j = 0; j < nCols; j++)
81                  B.mtrx[i][j] = a * mtrx[i][j];
82          return B;
83      }
84
85      // set the value in the ith row and jth column to the double a
86      void set(int i, int j, double a)
87      {
88          assert(i >= 0 && j >= 0 && i < mRows && j < nCols);
89          mtrx[i][j] = a;
```

```
90      }
91
92      // print matrix to file
93      void print()
94      {
95          std::ofstream matrix;
96          matrix.open("randmatrix.csv");
97          for (int i = 0; i < mRows; i++)
98          {
99              for (int j = 0; j < nCols; j++)
100             {
101                 matrix << mtrx[i][j] << ",";
102             }
103             matrix << "\n";
104         }
105     }
106
107     friend std::ostream& operator<<(std::ostream& os, const MMatrix& A);
108
109 private:
110     unsigned int mRows, nCols; // dimensions of matrix
111     std::vector<std::vector<double>> mtrx; // matrix a vector of double vectors
112 };
113
114 // overload << operator to print matrix
115 std::ostream & operator <<(std::ostream & os, const MMatrix & A)
116 {
117     for (unsigned i = 0; i < A.mtrx.size(); i++)
118     {
119         os << "|";
120         for (unsigned j = 0; j < A.mtrx[i].size(); j++)
121             os << A.mtrx[i][j] << ",";
122         os << "|\n";
123     }
124     return os;
125 }
```

Listing 12: `MMatrix` class

```
1  (m,n,k) = 15,16,6
2  no. of iter to converge = -1 | | r stops changing at = 206
3  (m,n,k) = 16,33,5
4  no. of iter to converge = -1 | | r stops changing at = 170
5  (m,n,k) = 29,44,11
6  no. of iter to converge = -1 | | r stops changing at = 283
7  (m,n,k) = 28,63,11
8  no. of iter to converge = -1 | | r stops changing at = 351
9  (m,n,k) = 85,87,11
10 no. of iter to converge = 51 | | r stops changing at = 65
11 (m,n,k) = 54,75,21
12 no. of iter to converge = -1 | | r stops changing at = 427
13 (m,n,k) = 10,57,4
14 no. of iter to converge = -1 | | r stops changing at = -1
15 (m,n,k) = 30,36,14
16 no. of iter to converge = -1 | | r stops changing at = -1
17 (m,n,k) = 17,23,6
18 no. of iter to converge = -1 | | r stops changing at = 355
19 (m,n,k) = 52,88,7
20 no. of iter to converge = 84 | | r stops changing at = 103
21 (m,n,k) = 23,25,4
22 no. of iter to converge = 60 | | r stops changing at = 73
23 (m,n,k) = 51,83,16
24 no. of iter to converge = -1 | | r stops changing at = 262
25 (m,n,k) = 32,59,15
```

```
no. of iter to converge = -1 | | r stops changing at = 382
(m,n,k) = 54,56,8
no. of iter to converge = 55 | | r stops changing at = 72
(m,n,k) = 67,87,13
no. of iter to converge = 84 | | r stops changing at = 106
(m,n,k) = 12,36,5
no. of iter to converge = -1 | | r stops changing at = 177
(m,n,k) = 57,89,16
no. of iter to converge = -1 | | r stops changing at = 300
```

Listing 13: Output of code in listing 7

Bibliography

[1] Ams.org. (n.d.). *Compressed Sensing Makes Every Pixel Count.* [online] Available at: https://www.ams.org/publicoutreach/math-history/hap7-pixel.pdf [Accessed 10 Dec. 2018].

[2] Blumensath, T. and Davies, M. (2010). Normalized Iterative Hard Thresholding: Guaranteed Stability and Performance. *IEEE Journal of Selected Topics in Signal Processing*, [online] 4(2), pp.298-309. Available at: https://eprints.soton.ac.uk/142499/1/BD_NIHT09.pdf [Accessed 11 Dec. 2018].

[3] Olshausen, B. (2010). *Aliasing.* [online] Rctn.org. Available at: http://www.rctn.org/bruno/npb261/aliasing.pdf [Accessed 10 Dec. 2018].

[4] Candes, E. (2010). *Compressed Sensing - A 25 Minute Tour.* [online] Raeng.org.uk. Available at: https://www.raeng.org.uk/publications/other/candes-presentation-frontiers-of-engineering [Accessed 10 Dec. 2018].

[5] Chartrand, R., Baranuik, R., Eldar, Y., Figueiredo, M. and Tanner, J. (2010). Introduction to the Issue on Compressive Sensing. *IEEE Journal of Selected Topics in Signal Processing*, [online] 4(2), pp.241-243. Available at: https://ieeexplore-ieee-org.manchester.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=5431096 [Accessed 8 Dec. 2018].

[6] Donges, N. (2018). *Gradient Descent in a Nutshell.* [online] Towards Data Science. Available at: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0 [Accessed 10 Dec. 2018].

[7] Eldar, Y. (2015). *Sampling theory.* Cambridge: Cambridge University Press, pp.390-471.

[8] Hardesty, L. (2017). *A faster single-pixel camera.* [online] MIT News. Available at: http://news.mit.edu/2017/faster-single-pixel-camera-lensless-imaging-0330 [Accessed 10 Dec. 2018].

[9] Lichtblau, D. and Weisstein, E. (n.d.). Condition Number. [online] Mathworld.wolfram.com. Available at: http://mathworld.wolfram.com/ConditionNumber.html [Accessed 11 Dec. 2018].

[10] Moler, C. (2017). *What is the Condition Number of a Matrix?.* [online] Mathworks.com. Available at: https://blogs.mathworks.com/cleve/2017/07/17/what-is-the-condition-number-of-a-matrix/ [Accessed 11 Dec. 2018].

[11] Qaisar, S., Bilal, R., Iqbal, W., Naureen, M. and Lee, S. (2013). Compressive sensing: From theory to applications, a survey. *Journal of Communications and Networks*, [online] 15(5), pp.443-456. Available at: https://ieeexplore-ieee-org.manchester.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=6674179 [Accessed 8 Dec. 2018].

[12] Strohmer, T. (2012). Measure What Should be Measured: Progress and Challenges in Compressive Sensing. *IEEE Signal Processing Letters*, [online] 19(12), pp.887-893. Available at: https://www.math.ucdavis.edu/~strohmer/papers/2012/CS_challenges.pdf [Accessed 10 Dec. 2018].