Student ID: 8444507 Name: M. Kashif Hussain

Scientific Computing Project 2
Compressed Sensing

This report will explore two algorithms known as:-

1. Gradient Descent.
2. Normalised Iterative Hard Thresholding (NIHT).

The aim of this report is to show how these algorithms are used in the field of compressed sensing. The Gradient Descent method is used for solving linear least squares problems and serves as a basis for my implementation of NIHT. The latter algorithm is used to be able to accomplish sparse recovery given certain conditions in which we will discover the phase transition phenomenon which governs the performance of the algorithm. Provided also are figures and tables to aid your understanding of the algorithms mentioned above. The majority of the code I have used can be seen in the appendix of this report.

Section 1: Understanding Compressed Sensing

1.1. What is Compressed Sensing?

Compressed sensing is a signal processing technique which is used to efficiently acquire and reconstruct a signal by finding solutions to underdetermined linear systems (i.e. fewer equations than unknowns). (Compressed Sensing, n.d.) As current technology depends heavily on the acquisition and processing of many signals/data (i.e. MRI scans), the process of retrieving compressible signals and reconstructing them can be greatly improved by acquiring them in a compressed form. From this, we can produce an excellent approximation of the actual data through the use of various methods. Compressible data is data which approximates the data being compressed accurately but has far few nonzero entries. When the signal/data acquisition process is linear, the problem is reduced to solving a linear system of equations. As a result the problem of Compressed Sensing is to find a *k-sparse* solution x to an underdetermined system of linear equations commonly written in the form:

$$Ax = b, \text{ where } A \in \mathbb{R}^{nxm},\ x \in \mathbb{R}^{mx1}, b \in \mathbb{R}^{nx1} \text{ and } m < n.$$

We refer to a vector as *k-sparse* if at most k co-ordinates are non-zeo

1.2. Applications and Achievements of Compressed Sensing (Foucart & Holger, 2013)

Compressed sensing techniques are used in a wide variety of applications. One of these is Magnetic Resonance Imaging (MRI) a common technology in medical imaging. It is used for various tasks such as brain imaging, heart imaging, examining blood vessels and more. The time taken to produce high-resolution images can range between several minutes up to hours depending on the task. In emergency situations, compressive sensing's usefulness is highlighted by its ability to speed up the imaging process.

An everyday object which makes use of compressive sensing are mobile phones. The camera sensor makes use of compressive sensing to reduce the energy taken to acquire the image by a large factor. Radar is another area which compressive sensing is used. A radar device is used to send a signal to a plane, for example, which scatters the signal. A receiving antenna is then used to receive the scattered signals. Compressive sensing deals with compressing these "pulses"

Student ID: 8444507 Name: M. Kashif Hussain

Section 2: Gradient Descent Method

2.1. What is Gradient Descent? (Lotz)

Given a least squares problem:

$$\text{minimize } ||Ax - b||^2 \ , \ A \in \mathbb{R}^{mxn}, \ m \geq n,$$

we are able to solve the above through the use of iterative methods. One such iterative method is called "Gradient Descent"

Give a function $f(x)$ and a starting point x_0 we try to minimise the function by taking steps in the direction of the negative gradient i.e. the *steepest descent:*

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

My program makes use of the following algorithm:

1. Start with a guess x_0.
2. Loop through the following:

$$\propto_i = \frac{r_i^T r_i}{r_i^T A^T A r_i} \ , \qquad x_{i+1} = x_i + \propto_i r_i \ , \qquad r_{i+1} = r_i - \propto A^T A r_i \ ,$$

until $||r_{i+1}||$ is smaller than some tolerance or a maximum iteration bound has been reached.

2.2 A Gradient Descent Implementation

Below is a snippet of code I have used to implement the "Gradient Descent" algorithm. The function norm is located in the Appendix.

```
//Computes the gradient descent given a matrix A, vector b,
//initial guess x, max no. of iterations and the tolerance
int SDLS(const Matrix& A, const MVector& b, MVector& x,
       int maxIterations, double tol)
{
       //Record results to a file
       ofstream record;
       record.open("Trajectory Points Only.txt");
       if (!record) return 1;

       //Initialize residual vector
       MVector r = A.transpose()*((b - (A*x)));
       double alpha = 0;
       //Loop until the norm of the residual is smaller than the tolerance
       for (int iter = 0; iter < maxIterations; iter++)
       {
              alpha = (r*r) / ((A*r) * (A*r));
              x = x + (alpha*r);
              record.width(10); record << x;
              r = r - alpha*(A.transpose()*((A*r)));
              if (norm(r) < tol)
                     return iter + 1;

       }
       //If loop finishes, i.e. max iterations performed then return -1
       // to show non-convergence for the no. of iterations performed
       return -1;
}
```

The above code performs the algorithm as described. When the norm of the residual vector is less than the specified tolerance, the loop terminates. In the case of the loop failing to converge on a value for x, it exits the loop and returns -1 indicating that the algorithm did not converge on a vector for x within the number of iterations the algorithm was looped for. Using matrix rules, I have simplified the calculation for alpha.

2.3 A Gradient Descent 3x2 Matrix Example

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

We will solve this $3x2$ least-squares problem with my implementation of the steepest descent algorithm.

By modifying my `main()`with the following code:

```
int main()
{
	//Create 3x2 matrix
	Matrix A(3, 2);
	A(0, 0) = 1; A(0, 1) = 2;
	A(1, 0) = 2; A(1, 1) = 1;
	A(2, 0) = -1; A(2, 1) = 0;

	//Create 3x1 vector b
	MVector b(3);
	b[0] = 10; b[1] = -1; b[2] = 0;

	//Create 2x1 initial guess vector x
	MVector x(2);
	x[0] = 0; x[1] = 0;

	//Record number of iterations
	int noOfIterations = SDLS(A, b, x, 1000, 1e-6);
	//Store trajectory points
	ofstream record;
	record.open("Trajectory Points Only.txt");
	if (!record) return 1;
	//Print out iterations, and vector x achieved.
	cout << "Iterations: " << noOfIterations << endl;
	cout << "x: " << x;


	return 0;

}
```

I am able to run the algorithm with:

`Number of iterations = 1000 and Tolerance =` $1x10^{-6}$.

We get the table displayed on the following page.

As we can see, there are 39 results due to the 39 iterations the algorithm took for it to converge on a vector for x.

Student ID: 8444507 Name: M. Kashif Hussain

Value of $x(0,0)$	**Value of $x(1,0)$**
0.998532	2.37151
-1.50369	3.42508
-1.08907	4.4098
-2.12807	4.84728
-1.95591	5.25616
-2.38733	5.43782
-2.31585	5.6076
-2.49499	5.68303
-2.4653	5.75352
-2.53969	5.78484
-2.52736	5.81412
-2.55825	5.82712
-2.55313	5.83928
-2.56596	5.84468
-2.56383	5.84972
-2.56916	5.85197
-2.56827	5.85406
-2.57049	5.85499
-2.57012	5.85586
-2.57104	5.85625
-2.57088	5.85661
-2.57127	5.85677
-2.5712	5.85692
-2.57136	5.85699
-2.57133	5.85705
-2.5714	5.85708
-2.57139	5.8571
-2.57142	5.85712
-2.57141	5.85713
-2.57142	5.85713
-2.57142	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714
-2.57143	5.85714

Figure 1: Table of x vector produced by SD algorithm

$\boldsymbol{Iterations}: 39$

$\boldsymbol{Initial\ x}: \begin{bmatrix}0\\0\end{bmatrix}.$

$\boldsymbol{End\ x}: \begin{bmatrix}-2.57143\\5.85714\end{bmatrix}.$

2.4 Graphs Demonstrating Convergence

The graphs below clearly demonstrate the convergence of points for this $3x2$ system.

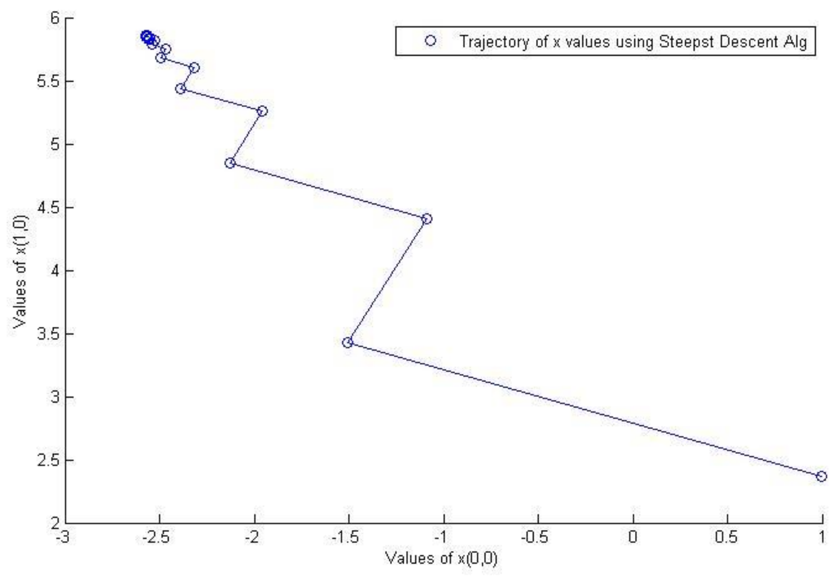
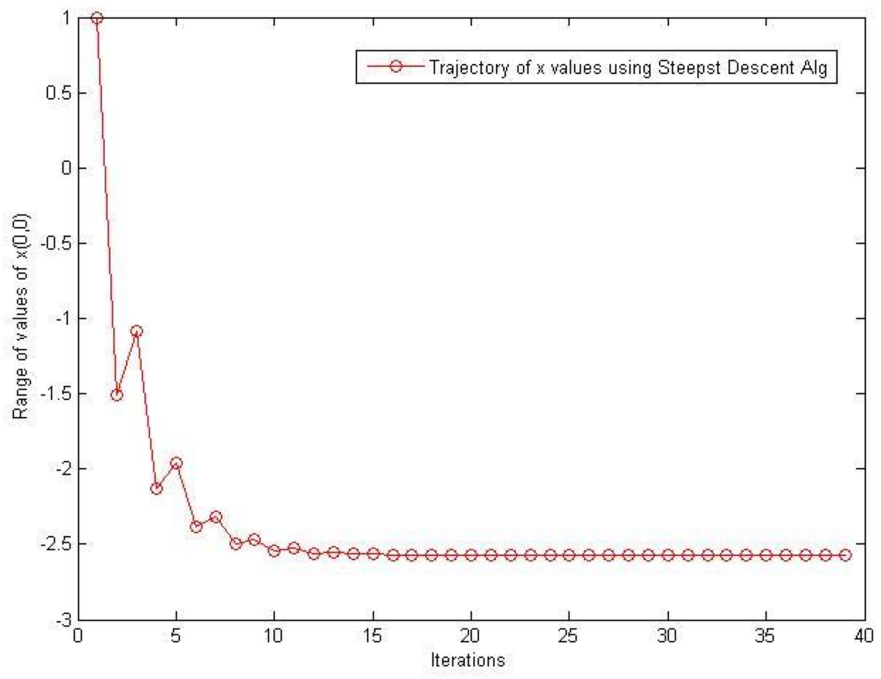


Figure 2: Line and Scatter graph showing convergence of 3x2 matrix

The graph to the left shows the points converging to a single area.



The graph to the left shows what the algorithm does to the value of $x(0,0)$ over successive iterations.

Figure 3: Graph of trajectory of x(0,0) values vs. iterations

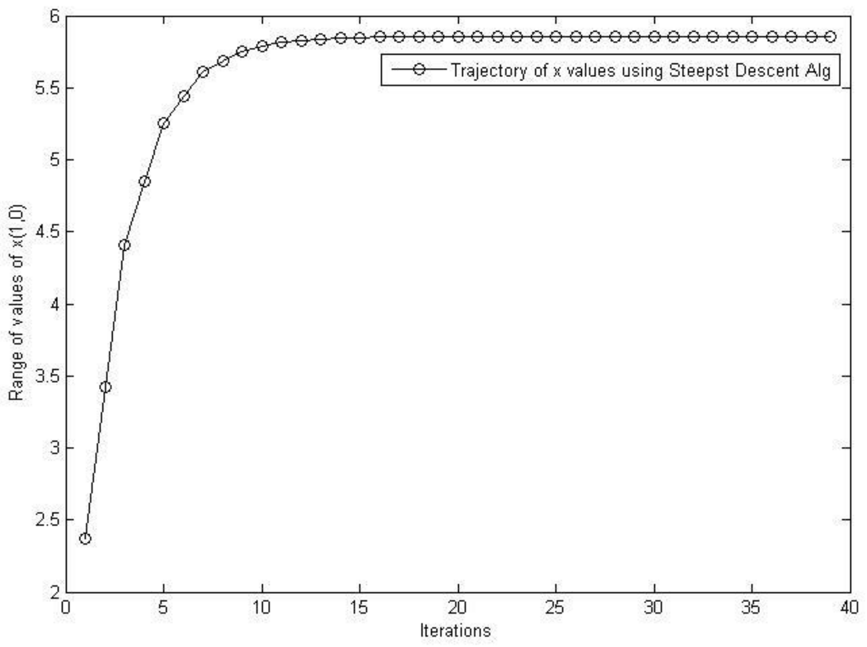


Figure 4: Graph of trajectory of x(1,0) values vs. iterations

The graph to the left shows what the algorithm does to the value of $x(1,0)$ over successive iterations.

Section 3: Normalised Iterative Hard Thresholding Algorithm

3.1 What is NIHT?

In order to solve the following:

$$Ax = b, \text{ where } A \,\epsilon\, \mathbb{R}^{nxm},\ x \,\epsilon\, \mathbb{R}^{mx1}, b \,\epsilon\, \mathbb{R}^{nx1} \text{ and } m < n,$$

we can make use of the NIHT algorithm.

Similar to the "Gradient Descent" algorithm, my program makes use of the following:

1. Loop through the following:

$$\propto_i = \frac{r_i^T r_i}{r_i^T A^T A r_i}\ ,\qquad x_{i+1} = H_k(x_i + \propto_i A^T(b - Ax_i))\ ,\qquad r_{i+1} = A^T(b - Ax_i)\,,$$

until $||r_{i+1}||$ is smaller than some tolerance or a maximum iteration bound has been reached.

Here, H_k is our thresholding function is performed at each step which sets all but the absolute largest k elements of x_i to 0.

In example:

If K = 4, $H_k((-4,1,7,-8,2,3)^T) = (-4,0,7,-8,0,3)^T$

If the algorithm converges, we expect a k-sparse solution. The algorithm does not always succeed in finding the correct solution. However, for most matrices (chosen randomly according to some distribution), when the number of equations is large enough, the algorithm solves the problem to a satisfactory standard and produces a sparse solution.

3.2 The NIHT Implementation

Below is a snippet of code I have used to implement the NIHT algorithm. The function norm is located in the Appendix.

```
/Computes the NIHT given a matrix A, vector b,
//initial guess x, threshold k, max no. of iterations and the tolerance
int NIHT(const Matrix& A, const MVector& b, MVector&x, int k,
       int maxIterations, double tol) {

       // Initialise starting vector
       x = A.transpose()*b;

       // Get s largest values
       vector<int> I = x.threshold(k);

       //Calculate the residual of the vector
       MVector r = A.transpose()*((b - (A*x)));
       double alpha = 0;
```

```
	//Loop which performs the NIHT algorithm until norm of residual is less than given tolerance
	for (int iter = 0; iter < maxIterations; iter++)
	{
		alpha = (r*r) / ((A*r) * (A*r));
		x = x + alpha*(A.transpose()*(b - (A*x)));
		//Get next largest values
		I = x.threshold(k);
		r = A.transpose()*((b - (A*x)));

		if (norm(r) < tol)
			return iter + 1;
	}
	//Max iterations reached
	return -1;

}
```

The above code performs the NIHT algorithm. When the norm of the residual vector is less than the specified tolerance, the loop terminates. In the case of the loop failing to converge on a value for x, it exits the loop and returns -1 indicating that the algorithm did not converge on a vector for x within the number of iterations the algorithm was looped for. At each step we see the threshold function as described by the above algorithm.

3.3 The Thresholding Step

Below is the function I use to calculate the threshold.

```
vector<int> threshold(int k)
	{
		//k+1 because threshold(0) = 1 non-zero value therefore size = 1.
		vector<int> nonZero(k+1);
		vector<double> orig(this->size());

		//Create same vector but with absolute valuea
		for (int i = 0; i < this->size(); i++)
		{
			orig[i] = abs(this->v[i]);
		}
		//sort vector in descending order with comparison function
		nth_element(orig.begin(), orig.begin() +k, orig.end(), greater<double>());

		//Keep count of number of k values
		int count = 0;

		//Loop through original vector and set everything less than
		//kth largest to 0. If kth largest or higher, store index
		for (int i = 0; i < this->size(); i++)
		{
			if (abs(this->v[i]) < orig[k])
				this->v[i] = 0;
			else
			{
				nonZero[count] = i;
				count++;

			}

		}

		return nonZero;
	}
```

The above algorithm returns the indices of the vectors which are greater than the kth largest. Initially, a copy of the current mvector is created which stores the absolute values of each element in the original vector. It is then sorted through the “nth_element” function in descending order using a comparison function located in the appendix which sorts the vector in descending order. We then loop through the current mvector and replace all values smaller than the kth largest with 0’s and record the index of those which are equal or greater and return these indices in the form of a vector.

The thresholding step does take a long time to compute as my function copies the current vector and storing its absolute values. Then the vector is sorted with the expected running time of the “nth_element” function being $O(n)$ with a worst case of $O(n^2)$. As a result, the threshold function does take a long time to compute as the vector passed to it grows in size.

3.4 The Phase Transition Phenomenon

A phase transition is a sharp change in the character of a computational problem as its parameters vary (Lotz). For compressed sensing, this refers to the probability that the x vector is recovered with high probability. So there is a threshold for the parameters for which the x vector is recovered with high chance in the NIHT algorithm. If this threshold is not reached, the recovery of the vector fails with high probability.

Below is an example with vectors of length $n = 200,\ k = 20$ and $k = 50$ where k refers to the sparsity of the vector. I will determine the point R_0 such that for $R > R_0$, the NIHT algorithm recovers the sparse vector with a very high degree of success. I will also set the maximum number of iterations to 200 and tolerance to $1x10^{-6}$

We modify the main() as follows so that we run each value for R

```
int NIHTtotal;
int T = 5;
//Threshold value which is K = 20 (i.e. 19 for me) as 0 =1
int  k = 19;
//Calculate the amount of times recovered
int countOfRecovery = 0;
//countOfRecovery/T
double percentageRecovery;

//Record results to a file
ofstream tally;
tally.open("Success Tally K = 20.txt");
if (!tally) return 1;

ofstream results;
results.open("Results K = 20.txt");
if (!results) return 1;

for (int i = 4; i < 200; i+=5)
{
       for (int j = 0; j < T; j++)
       {
              Matrix testMatrix(i, 200);
              testMatrix.initialize_normal();
              testVec.initialize_normal(k);
              testB = testMatrix*testVec;
              NIHTtotal = NIHT(testMatrix, testB, testVec, k, 200, 1e-6);
              results.width(10); results << i;
              results.width(10); results << NIHTtotal << endl;
              if (NIHTtotal > -1)
                     countOfRecovery++;
       }
       percentageRecovery = double(countOfRecovery / double(T));
       tally.width(10); tally << i;
```

```
            tally.width(10); tally << countOfRecovery;
            tally.width(10); tally << percentageRecovery << endl;
            countOfRecovery = 0;
    }
```

The above function starts with a 4x200 matrix (increasing the amount of rows by 5 each time) and performs the NIHT algorithm 5 times to produce an average and stores it in a file. Below are tables showing the probability of successful recovery for $k = 20$ and $k = 50$ respectively.

K = 20 (Rows)	Successes	Probability	K = 50 (Rows)	Successes	Probability
4	0	0	4	5	1
9	0	0	9	5	1
14	0	0	14	5	1
19	0	0	19	2	0.4
24	0	0	24	0	0
29	0	0	29	0	0
34	0	0	34	0	0
39	0	0	39	0	0
44	0	0	44	0	0
49	0	0	49	0	0
54	0	0	54	0	0
59	0	0	59	0	0
64	0	0	64	0	0
69	0	0	69	0	0
74	0	0	74	0	0
79	0	0	79	0	0
84	0	0	84	0	0
89	1	0.2	89	0	0
94	0	0	94	0	0
99	0	0	99	0	0
104	1	0.2	104	0	0
109	2	0.4	109	0	0
114	2	0.4	114	0	0
119	3	0.6	119	0	0
124	4	0.8	124	0	0
129	4	0.8	129	0	0
134	4	0.8	134	0	0
139	5	1	139	0	0
144	5	1	144	0	0
149	4	0.8	149	1	0.2
154	5	1	154	0	0
159	5	1	159	0	0
164	5	1	164	0	0
169	4	0.8	169	2	0.4
174	4	0.8	174	2	0.4
179	5	1	179	4	0.8
184	5	1	184	5	1
189	5	1	189	4	0.8
194	5	1	194	5	1
199	5	1	199	5	1

Figure 5: Tables of NIHT for K = 20 and K = 50

Considering an "Overwhelming Probability" of successful retrieval as 80%, we can see that for $K = 20$, we have reached that probability when $R \geq 124$. Similarly for $K = 50$, we have $R \geq 179$.

3.5 NIHT Example Graphs

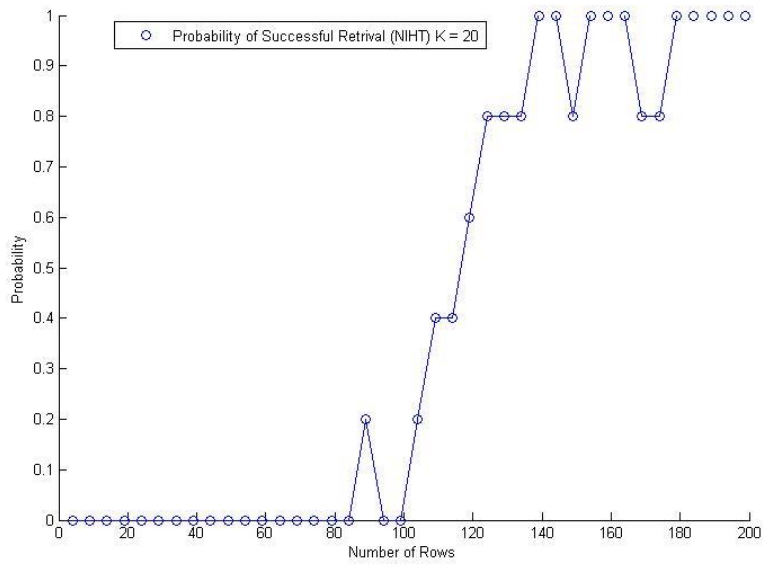


Figure 6: Graph for Probability of successful retrieval for K = 20

The NIHT algorithm has a high probability of successful retrieval when the number of rows in the matrix is greater than or equal to 124

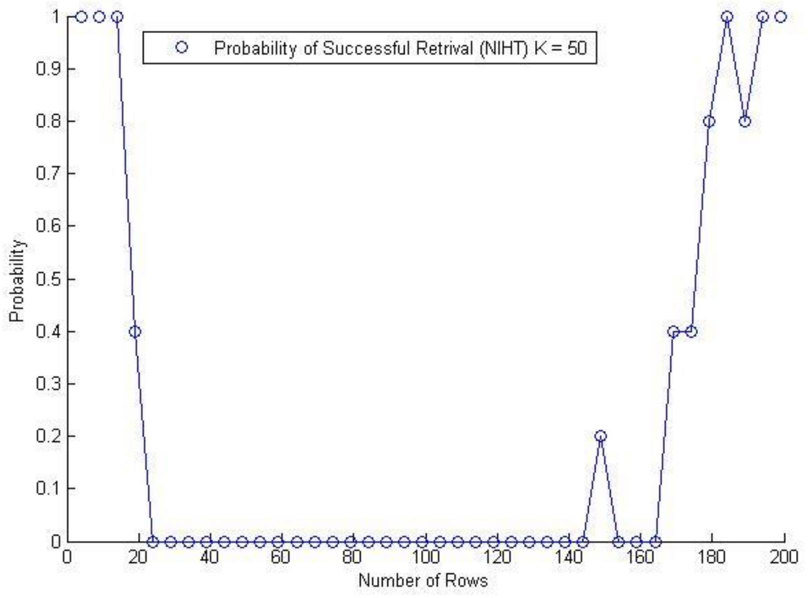


Figure 7: Graph for Probability of successful retrieval for K = 50

The NIHT algorithm has a high probability of successful retrieval when the number of rows in the matrix is greater than or equal to 179.

Note also while the number of rows is greater than or equal to 4 but less than 19 , the algorithm has a high probability of successful retrieval which may be due to the algorithm having to find a small vector x such that $Ax = b$ and drops until we get to the "Phase Transition".

Student ID: 8444507 Name: M. Kashif Hussain

I decided to loop through each matrix 5 times due to the computational cost of doing 10 or more especially for the latter, larger matrices input into the NIHT algorithm. This is mostly due to the thresholding operation taking up most of the computation time. By looping through 5 times, we can calculate the probability of retrieval as if the NIHT algorithm return -1, the algorithm couldn’t find a solution in 200 iterations such that the norm of the residual was less than the tolerance.

As the number of times I ran the test for each matrix was 5, if the code generated an ill-conditioned matrix there was a high chance that the NIHT algorithm would fail to converge to the vector within the tolerance for the allotted number of iterations. Keeping the tolerance constant, if I set the number of iterations of the matrix to a larger value, we would have a longer computation time for the algorithm but a much more accurate view of where the phase transition occurs and probably a higher retrieval percentage past the phase transition number. Ultimately, however, if the algorithm was to come across an ill-condition matrix, it may not converge even with an extra 100 or more iterations. The algorithm solves the problem for most matrices according to the distribution chosen.

Appendix

Function norm:

```
//calculates the norm of a vector
double norm(const MVector &x) {
	double total = 0;
	for (int i = 0; i < x.size(); i++) {
		total += (x[i] * x[i]);
	}

	return sqrt(total);

}
```

Function SDLS (Gradient Descent) :

```
//Computes the gradient descent given a matrix A, vector b,
//initial guess x, max no. of iterations and the tolerance
int SDLS(const Matrix& A, const MVector& b, MVector& x,
	int maxIterations, double tol)
{
	//Record results to a file
	ofstream record;
	record.open("Trajectory Points Only.txt");
	if (!record) return 1;

	//Initialize residual vector
	MVector r = A.transpose()*((b - (A*x)));
	double alpha = 0;
	//Loop until the norm of the residual is smaller than the tolerance
	for (int iter = 0; iter < maxIterations; iter++)
	{
		alpha = (r*r) / ((A*r) * (A*r));
		x = x + (alpha*r);
		record.width(10); record << x;
		r = r - alpha*(A.transpose()*((A*r)));
		if (norm(r) < tol)
			return iter + 1;

	}
	//If loop finishes, i.e. max iterations performed then return -1
	// to show non-convergence for the no. of iterations performed
	return -1;
}
```

Student ID: 8444507 Name: M. Kashif Hussain

Function NIHT (Normalised Iterative Hard Thresholding):

```
//Computes the NIHT given a matrix A, vector b,
//initial guess x, threshold k, max no. of iterations and the tolerance
int NIHT(const Matrix& A, const MVector& b, MVector&x, int k,
	int maxIterations, double tol) {

	// Initialise starting vector
	x = A.transpose()*b;

	// Get s largest values
	vector<int> I = x.threshold(k);

	//Calculate the residual of the vector
	MVector r = A.transpose()*((b - (A*x)));
	double alpha = 0;

	//Loop which performs the NIHT algorithm until norm of residual is less than given tolerance
	for (int iter = 0; iter < maxIterations; iter++)
	{
		alpha = (r*r) / ((A*r) * (A*r));
		x = x + alpha*(A.transpose()*(b - (A*x)));
		//Get next largest values
		I = x.threshold(k);
		r = A.transpose()*((b - (A*x)));

		if (norm(r) < tol)
			return iter + 1;
	}

	return -1;

}
```

Student ID: 8444507   Name: M. Kashif Hussain

mvector.h:

```
#ifndef MVECTOR_H // the 'include guard'
#define MVECTOR_H // see C++ Primer Sec. 2.9.2

#include <vector>
#include <algorithm>
#include <math.h>
#include <functional>      // For greater<int>( )
using namespace std;
// 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) {}

	//https://msdn.microsoft.com/en-us/library/7s2yb954.aspx
	bool UDgreater(int a, int b) {
		return abs(a) > abs(b);
	}

	//Prints out a vector
	friend ostream& operator<<(ostream& out, const  MVector &w) {
		//out << "(";
		for (int i = 0; i < w.size() - 1; i++)
			out << w[i] << ", ";
		out << w[w.size() - 1] << endl;
		//out << w[w.size() - 1] << ")" << endl;
		return out;

	};

	//The threshold function which returns only the kth largest
	//elements and set those below to 0
	vector<int> threshold(int k)
	{
		//k+1 because threshold(0) = 1 non-zero value therefore size = 1.
		vector<int> nonZero(k+1);
		vector<double> orig(this->size());

		//Create same vector but with absolute valuea
		for (int i = 0; i < this->size(); i++)
		{
			orig[i] = abs(this->v[i]);
		}
		//sort vector in descending order with comparison function
		nth_element(orig.begin(), orig.begin() +k, orig.end(), greater<double>());

		//Keep count of number of k values
		int count = 0;
		//Loop through original vector and set everything less than
		//kth largest to 0. If kth largest or higher, store index
		for (int i = 0; i < this->size(); i++)
		{
			if (abs(this->v[i]) < orig[k])
				this->v[i] = 0;
			else
			{
				nonZero[count] = i;
				count++;

			}

		}


	//return index of kth largest or higher values
		return nonZero;
	}


	MVector MVector::sub(const vector<int> &I) const
	{
		//return the subvector indexed by the elements in I
		MVector subVector(I.size());
		for (int i = 0; i < I.size(); i++)
			subVector[i] = this->v[I[i]];
		//cout << "Sub Vector: " << subVector << endl;
		return subVector;
	}

	//Box-Müller transform
	double rand_normal()
	{
		static const double pi = 3.141592653589793238;
		//Adding 1 prevents issue of -Nan(ind) as we can no longer get 0
		double u = (rand() + 1) / static_cast<double>(RAND_MAX + 1.0);
		double v = (rand() + 1) / static_cast<double>(RAND_MAX + 1.0);
		return sqrt(-2.0*log(u))*cos(2.0*pi*v);
	}

	//Create a randomized normal vector with threshold k
	void initialize_normal(int k)
	{
		int vecSize = size();
		for (int i = 0; i< vecSize; i++)
			this->operator[](i) = rand_normal() / sqrt(vecSize);
		vector<int> I = this->threshold(k);

	}


	// access element (lvalue)
	double &operator[](int index) { return v[index]; }

	// access element (rvalue)
	double operator[](int index) const { return v[index]; }

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

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

Student ID: 8444507 Name: M. Kashif Hussain

```
//Multiply two vectors
double operator*(const MVector& A, const MVector& B) {
	double result = 0.;
	for (int i = 0; i < A.size(); i++)
		result += (A[i] * B[i]);
	return result;
}

//Subtract two vectors
MVector operator-(const MVector& A, const MVector& B) {

	MVector C(A);
	for (int i = 0; i < C.size(); i++)
		C[i] -= B[i];
	return C;
}

//Add two vectors
MVector operator+(const MVector& A, const MVector& B) {

	MVector C(A);
	for (int i = 0; i < C.size(); i++)
		C[i] += B[i];
	return C;
}

//Multiply scalar with vector i.e. aB
MVector operator*(const double& a, const MVector& B) {

	MVector C(B);
	for (int i = 0; i < C.size(); i++)
		C[i] *= a;
	return C;
}
#endif
```

Student ID: 8444507   Name: M. Kashif Hussain

Matrix.h:

```
#ifndef MATRIX_H // the 'include guard'
#define MATRIX_H
#include <vector>
#include "mvector.h"
#include <cstdlib>
using namespace std;

// Class that represents a mathematical matrix
class Matrix
{
public:
	// Constructors and destructors (see also vector class)
	explicit Matrix() : N(0), M(0) {}
	Matrix(int n, int m) : N(n), M(m), A(n,vector<double>(m)) {}

	double operator()(int i, int j) const {
		//Get the matrix
		return A[i][j];
	}

	double& operator()(int i, int j) {
		//Write to matrix
		return A[i][j];
	}

	// Code to return the transpose of the matrix
	Matrix transpose() const
	{
		Matrix transposed(Cols(), Rows());
		for (int i = 0; i < Rows(); i++)
			for (int j = 0; j< Cols(); j++)
				transposed(j,i) = this->operator()(i, j);
		return transposed;
	}

	Matrix Matrix::sub(const vector<int>& I) const
	{
		// Return submatrix of columns indexed by the elements in I
		Matrix subMatrix(this->Rows(), I.size());
		for (int j = 0; j < I.size(); j++)
			for (int i = 0; i < this->Rows(); i++)
				subMatrix(i, j) = this->operator()(i, I[j]);
		return subMatrix;

	}
	//Box-Müller transform
	double rand_normal()
	{
		static const double pi = 3.141592653589793238;
		//Adding 1 prevents issue of -Nan(ind) as we can no longer get 0
		double u = (rand() + 1) / static_cast<double>(RAND_MAX + 1.0);
		double v = (rand() + 1) / static_cast<double>(RAND_MAX + 1.0);
		return sqrt(-2.0*log(u))*cos(2.0*pi*v);
	}
	//Creates a randomized normal matrix
	void initialize_normal()
	{
		 N = Rows();
		 M = Cols();
		for (int i = 0; i < N; i++)
			for (int j = 0; j < M; j++)
				// over sqrt(N) to get variance  = 1/N
				A[i][j] = rand_normal()/sqrt(N);

	}

	// size of matrix
	int Rows() const { return N; }
	int Cols() const { return M; }

protected:
	unsigned N, M; // Matrix dimensions
	vector <vector<double> > A; // Store as a vector of vectors
};

Matrix operator*(const Matrix& A, const Matrix& B) {
	// Does Matrix multiplication A*B
	Matrix C(A.Rows(), B.Cols());
	//Checks if matrix dimenensions match
	if (A.Cols() != B.Rows())
	{
		cout << "The dimensions of the two matrices do not match" << endl;
		throw;
	}

	for (int i = 0; i < A.Rows(); i++)
		for (int j = 0; j < B.Cols(); j++)
			for (int k = 0; k < A.Cols(); k++)
				C(i, j) += A(i, k)*B(k, j);
	return C;
}

//Does matrix multiplication to get vector A*b
MVector operator*(const Matrix& A, const MVector& B) {

	MVector C(A.Rows());
	if (A.Cols() != B.size())
	{
		cout << "The dimensions of the matrix and vector do not match" << endl;
		throw;
	}
	for (int i = 0; i < A.Rows(); i++)
	{
		C[i] = 0;
		for (int j = 0; j < A.Cols(); j++)
		{
			C[i] += (A(i, j)*B[j]);
		}
	}
	return C;
}
```

```
//Prints the matrix
ostream& operator<<(ostream& out, const Matrix& A)
{
	int N = A.Rows();
	int M = A.Cols();

	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < M; j++)
		{
			out << A(i, j) << ", ";
			if (j >= M - 1)
				out << endl;
		}
	}
	return out;
}
#endif
```

MATLAB Code:

For Steepest Descent Graphs:

```
function plots1()
data = csvread('Trajectory Points Only.txt');
scatter(data(:,1), data(:,2));
legend('Trajectory of x values using Steepst Descent Alg');
xlabel('Values of x(0,0)');
ylabel('Values of x(1,0)');
line(data(:,1), data(:,2));

function plots2()
data = csvread('Trajectory Points Only.txt');
figure
plot(1:1:size(data(:,1),1), data(:,1), 'r-o');
legend('Trajectory of x values using Steepst Descent Alg');
xlabel('Iterations');
ylabel('Range of values of x(0,0)');
%hold on;
figure
plot(1:1:size(data(:,1),1), data(:,2), 'k-o');
legend('Trajectory of x values using Steepst Descent Alg');
xlabel('Iterations');
ylabel('Range of values of x(1,0)');
```

For NIHT Graphs:

```
function plots4()
data = load('Success Tally K = 20.txt');
scatter(data(:,1), data(:,3));
legend('Probability of Successful Retrival (NIHT) K = 20');
xlabel('Number of Rows');
ylabel('Probability');
line(data(:,1), data(:,3));

function plots5()
data = load('Success Tally K = 50.txt');
scatter(data(:,1), data(:,3));
legend('Probability of Successful Retrival (NIHT) K = 50');
xlabel('Number of Rows');
ylabel('Probability');
line(data(:,1), data(:,3));
```

Bibliography

Compressed Sensing. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Compressed_sensing

Foucart, S., & Holger, R. (2013). *A Mathematical Introduction to Compressive Sensing.* Birkhauser.

Lotz, D. M. (n.d.). Compressed Sensing.