Compressed Sensing

Will Woolfenden

March 3, 2024

1 Introduction

This report is concerned with iterative methods for solving systems of linear equations (SLEs), focusing on singular systems which are either over or under-determined. We are interested in compressive sensing, where we want to reconstruct a solution to an SLE given an insufficient amount of information to solve with common iterative methods. To start, we look at an implementation of the gradient descent method for solving the linear least squares problem for an overdetermined system. We then focus on an implementation of normalised iterative hard thresholding, with a focus instead on recovering a sparse solution to an SLE. Compressed sensing arises in areas of signal processing, such as image and audio compression. For example, we may have the goal of reducing the information in an image as much as possible, such that we can reliably reconstruct the original image.

This report covers the results produced using C++ implementations of steepest descent and normalised iterative hard thresholding. All code implementations are provided in the Appendix. The testing function has been commented into several sections in order to isolate bodies of tests. These can be uncommented, compiled and executed by the reader for verification. All code was compiled successfully on Ubuntu 22.04 using the g++ 11 compiler, with $-$`O2` optimisation, native hardware priority $-$`march = native` and accelerated floating point arithmetic $-$`ffast - math`.

2 Steepest Descent

2.1 Least Squares

We are interested in solving the least squares problem $\min_x ||Ax - b||_2$. The solution is computed by solving the *normal equations* $A^{\mathrm{T}}Ax = A^{\mathrm{T}}b$. Our results compare different paths taken by the steepest descent algorithm to solve the normal equations for a least squares problem. We first look at minimising $||Ax - b||_2$ for

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

This system is over-determined, i.e. the number of constraints exceeds the number of variables x_i. We can analyse the least squares problem using the singular value decomposition $A = U\Sigma V^{\mathrm{T}}$. It can be shown that the first r columns of the SVD span the range of A. We can use this knowledge to evaluate convergence of the least squares problem. Namely, we can only find a solution which solves $Ax = b$ when b belongs to range(A). Otherwise, there is no such x that will solve this system

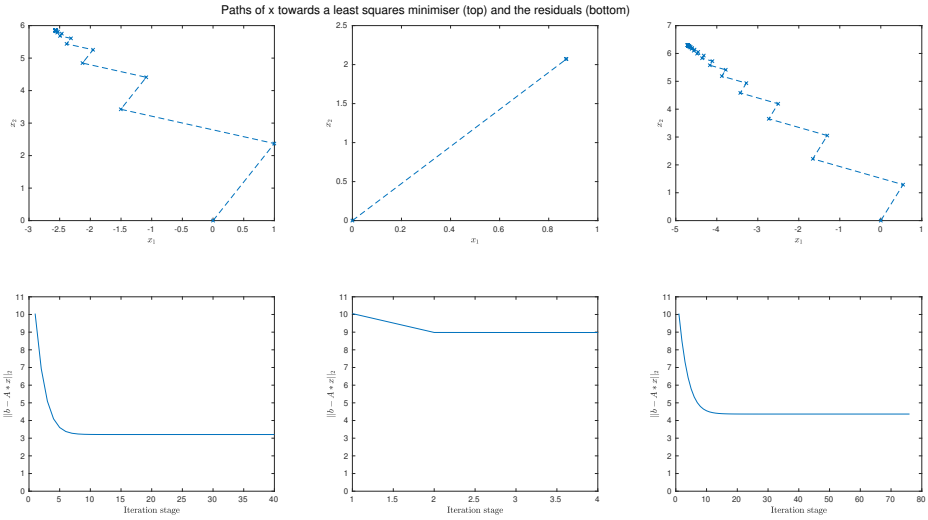

Figure 1: Gradient Descent implementations. Left: matrix $A = [1, 2; 2, 1; -1, 0]$. Middle: matrix $A = [1, 2; 2, 1; 1.8, -2]$. Right: matrix $A = [1, 2; 2, 1; -2, -2]$. Top row is plots of x for each step of the Steepest Descent algorithm. Bottom row shows the $L2$-norm $||b - Ax||_2$. Stopping criterion is for the residual $L2$-norm $r_i = ||A^{\mathrm{T}}(b - Ax_i)||_2$ to be below a given threshold.

of linear equations. If this solution does not exist, then we search for the solution which minimises $||Ax - b||_2$, which is the Euclidean distance from Ax to b. For a minimum $L2$-norm solution x^*, we have that $b - Ax^*$ must be orthogonal to any vector in range(A). We can use this criterion to test our solution.

2.2 Singular Value Decomposition Examples

Any matrix has an SVD $A = U\Sigma V^{\mathrm{T}}$. If A is $n \times m$ then so is Σ, where $U \in \mathbb{R}^{n \times n}$ and $V \in \mathbb{R}^{m \times m}$ are orthogonal matrices. The matrix Σ has the singular values σ_i on the diagonal in descending order, which are the square roots of the eigenvalues of $A^{\mathrm{T}}A$. Some of the singular values may be zero. We often write $\sigma_1 > \sigma_2 > \ldots > \sigma_r > \sigma_{r+1} = \sigma_{r+2} = \ldots = 0$. We use Octave for linear algebra results. See below:

```
octave:22> A
A =
   1   2
   2   1
  -1   0

octave:23> [U,Z,V] = svd(A)
U =
  -0.6716   0.6911  -0.2673
  -0.7000  -0.4735   0.5345
   0.2428   0.5461   0.8018

Z =
Diagonal Matrix
```

```
   3.0873         0
        0    1.2120
        0         0

V =
  -0.7497   -0.6618
  -0.6618    0.7497
```

The optimum solution obtained by our gradient descent method is $x^* \approx (-2.57, 5.86)^{\mathrm{T}}$. These results correspond to section (2) in the `main()` function. We compute the vector $b - Ax^*$ and find its inner products with the columns of U.

```
octave:27> x
x =
  -2.5714
   5.8571

octave:28> b
b =
   10
   -1
    0

octave:29> (b-A*x)'*U
ans =
-4.8408e-15   1.7471e-15  -3.2071e+00
```

The inner products with the first two columns are extremely small - they are only non-zero since we are working in finite precision arithmetic. See Appendix B for the inner products when looking at other cases. In the second case, we change the last row of A to $[1.8, -2]$ and in the final case it is $[-2, -2]$.

In every case, Ax is simply a linear combination of its two spanning vectors which we find from the SVD. Therefore if $b - Ax^*$ is orthogonal to the first two columns of U then x^* is a minimum $L2$-norm solution. See Figure 1 for visualisations of the Gradient Descent method.

2.3 Method

Each step of the Gradient Descent/Steepest Descent iteration is the step

$$x_{i+1} = x_i + \alpha_i r_i$$

where r_i is the residual

$$r_i := A^{\mathrm{T}}(b - Ax_i)$$

to the normal equations $A^{\mathrm{T}}Ax = A^{\mathrm{T}}b$, and α_i is the minimising step length

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

We differentiate $||Ax - b||_2^2$, the squared distance, with respect to x to find that a minimum is attained when x solves the normal equations.

$$\begin{aligned}
\nabla ||Ax - b||_2^2 &= \nabla \left((Ax - b)^{\mathrm{T}} (Ax - b) \right) \\
&= \nabla \left(x^{\mathrm{T}} A^{\mathrm{T}} Ax + b^{\mathrm{T}} b - x^{\mathrm{T}} A^{\mathrm{T}} b - b^{\mathrm{T}} Ax \right) \\
&= 2A^{\mathrm{T}} Ax - 2A^{\mathrm{T}} b
\end{aligned}$$

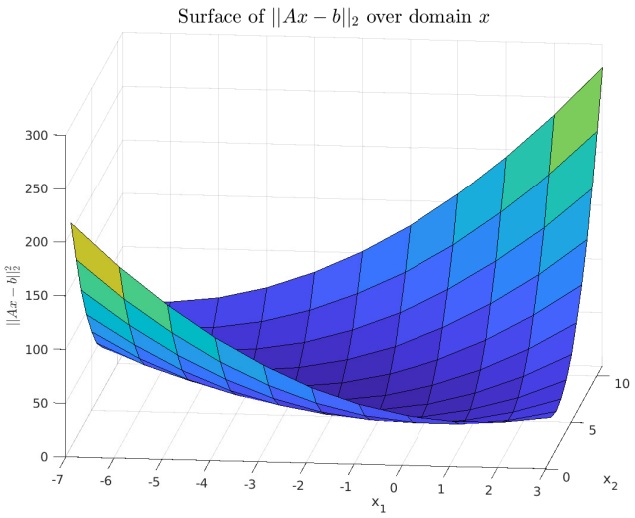

Figure 2: Graph of the objective function $f(x) = \frac{1}{2}||Ax - b||_2^2$. Case 1, where $A = [1, 2; 2, 1; -1, 0]$. We have solved for the minimiser $x^* = [-2.57, 5.85]^{\mathrm{T}}$, which is clearly in the vicinity of the minimiser for this graph.

Clearly the solution to the normal equations is equivalent to when $\nabla||Ax - b||_2^2 = 0$. This implies that our iterative method is of the form $x_{i+1} = x_i + \alpha_i\,(-\nabla f(x_i))$ where $f(x_i)$ is the $L2$-norm we are trying to minimise.

See Figure 1 for the performance of the steepest descent implementation in three similar cases. Cases 1 and 3 show a zig-zag path taken, where the residual only points in one of two directions. For case 2, the residual takes the path of a straight line and converges rapidly.

In Figure 2, we have graphed the objective function which we are trying to minimise from one case we have looked at. This convex function has a unique minimiser. The residual r_i is the negative gradient of this function at x_i, which is the *steepest descent direction* local to that point. Each iteration starts at a point x_i and finds the appropriate descent direction. Consider the contours of the convex function. In order for the step length to be minimising, we must send x_i to x_{i+1} such that r_i is tangent to the level set at x_{i+1}. Otherwise, further minimisation would be possible from x_{i+1} in direction r_i.

This requirement leads to the shapes of the paths taken in Figure 1, namely that the descent direction alternates for each step. The second case lacks this appearance because the initial steepest descent direction points directly towards the unique minimiser, thus no alternate directions are chosen. However, it takes more than one iteration to converge, potentially due to machine imprecision.

3 Normalised Iterative Hard Thresholding

3.1 Motivation

Having looked at an introduction for iterative methods for solving systems of linear equations, we now move on to the focus of this report, which is the implementation of the Normalised Iterative Hard Thresholding algorithm (NIHT) for the purpose of compressed sensing. We look at the NIHT

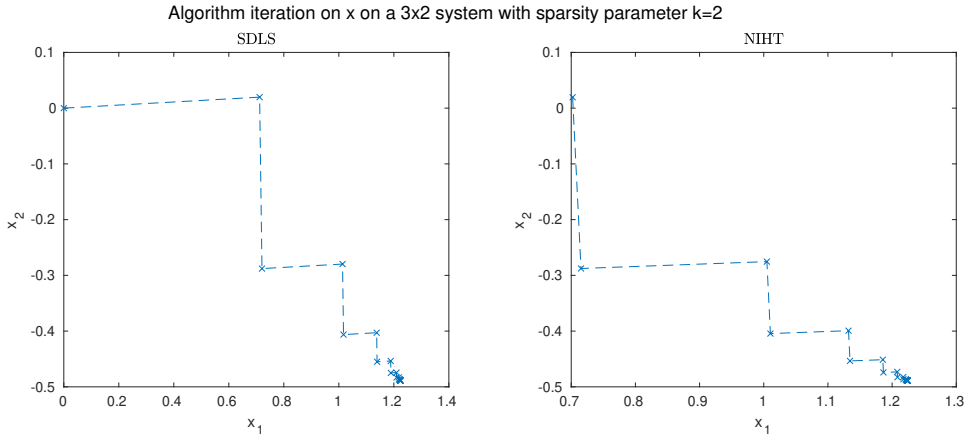

Figure 3: Map of x similar to earlier seen in Figure 1. We compare the steepest descent iteration (left) with the NIHT iteration on $k = 2$ (right). A random 3×2 matrix A and a random vector x^* of length 2 are generated and we compute $b = Ax^*$. Both algorithms solve $b = Ax$. We observe identical paths towards the optimiser in this example.

algorithm from [3]. Our NIHT algorithm implementation is one of several algorithms which are popular for compressed sensing [2]. An example of a compressed sensing problem would be the linear system $Ax = b$ to represent the data from a signal. Here, b represents observations from the signal, whereas the terms in x are the coefficients of a decomposition of the signal, such as a Fourier transformation. We focus on the sparsity of the vector x, which we can exploit to effectively recover the solution. We show that the algorithm will converge to the correct solution under particular conditions. We consider how the likelihood of convergence depends on the properties of the system.

3.2 Solutions to SLEs

The setup for our problem is different to that of least squares. Previously, we were given A and b, and had to minimise the function $||Ax - b||_2^2$. Now, we start with A and x and generate b. We are then given the problem of recovering the solution x from only A and b. Depending on properties of A and x, we may or may not recover the original vector.

Normalised iterative hard thresholding is an algorithm for finding a sparse solution to a system of linear equations. The system is often underdetermined, where $m < n$. However we include a sparsity parameter k. We say x is k-sparse if it has at most k entries which are non-zero. Then there are m equations on k non-zero unknowns and the system may be fully determined. If $m = 1$, then we have one equation on the k unknowns of x, which cannot be solved uniquely, whereas if $m = n$ then the problem can usually be solved. Recovery of x depends on both the information m and the sparsity k.

Since we start with A and x to produce $b = Ax$, we can guarantee that the system has a solution by its construction.

3.3 Method

Normalised iterative hard thresholding is a greedy algorithm, meaning that at any point in time it will make the "best" decision based on only current information. This is apparent in the thresholding

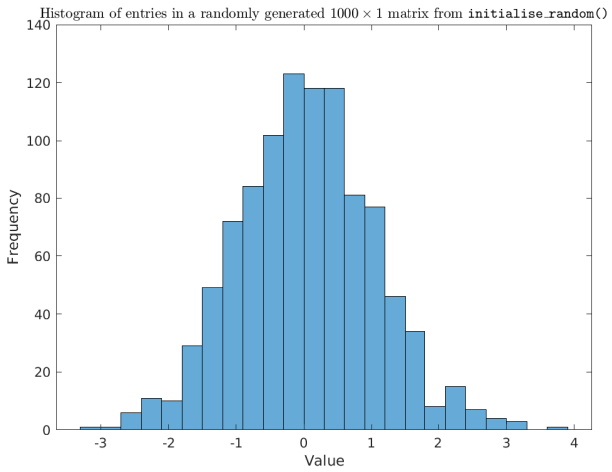

Figure 4: Figure of the values generated in a 1000×1 random matrix. The values are clearly that of a Gaussian distribution, with mean approximately 0 and variance approximately 1.

implementation, especially since the thresholding of a random vector is likely unique. Thresholding the vector means we are losing information that we would otherwise consider in an algorithm such as gradient descent.

In all our results, we consider matrices and vectors with random entries. We do this using an `initialise_normal()` method which used the C++ standard library `rand()` function and applies the Box-Müller transform to generate values randomly from a Gaussian distribution. See Figure 4 for the distribution from a random matrix. A random vector takes entries directly from the Gaussian distribution, but any random matrix takes entries $\eta/\sqrt{m}$, where η is the random variable from the Gaussian distribution and m is the number of rows of the matrix. This is such that the entries in a matrix have variance $1/m$. It is useful to be able to construct random matrices so that we can apply NIHT to different problems repeatedly. We generate an $m \times n$ matrix A and an n-vector x. We threshold x by the sparsity parameter k and compute $b = Ax$. Starting with $x_0 = A^{\mathrm{T}}b$, iterate:

$$x_{i+1} = \mathcal{H}_k\left(x_i + \alpha_i r_i\right)$$

where α_i and r_i are defined identically to how they were in the gradient descent method.

The iteration continues until $||r||_2$ is sufficiently small. Our method itself is a modification of gradient descent applied to compressed sensing [2]. See Figure 3, which shows the behaviour of both algorithms applied to the same problem. In this case, the paths are identical except for different starting points. This verifies that NIHT performs gradient descent for this example, especially since $k = 2$ is the size of x and so we consider no sparsity. It is important to note that we should not expect similar iteration performances between the two algorithms when working with very sparse and very full problems.

The function $\mathcal{H}_k(y)$ is the thresholding operation that sets all but the k greatest-in-modulus entries in y to zero. Thresholding is performed at every stage of the iteration, essentially projecting to a k-dimensional vector space. Our method `threshold(k)` copies the vector and sorts it by absolute value using our `abs_cmp()` comparison function fed to the standard C++ `sort()` function. This standard library function has time complexity $\mathcal{O}(n \log n)$ where n is the vector length [4]. In

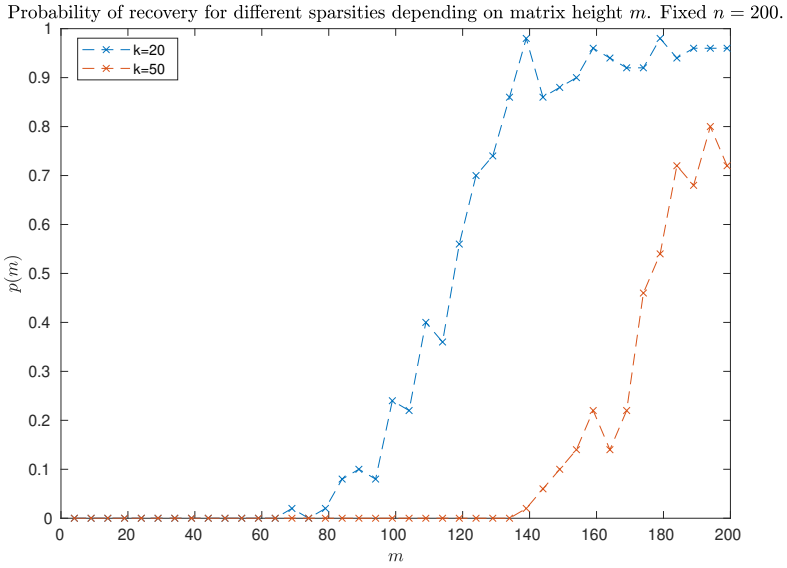

Figure 5: Plots of the likelihood of convergence for sparsities $k = 20$ and $k = 50$ on a vector x of length 200. The probability of convergence changes as a function of m. For $k = 20$, the algorithm is likely to succeed for $m > 120$. For $k = 50$, success is likely for $m > 180$. Likelihood estimated from 50 samples for each combination of k and m.

our function, we then take the k-th largest entry from the sorted vector, then iterate through the original vector and perform thresholding. Space complexity could be improved by avoiding copying the vector, but the copy allows us to perform sorting on a separate vector, get the value we need, and perform the threshold.

3.4 Testing and Results

The `NIHT()` function performs the iteration given A, b as well as a tolerance `tol` and integer `maxIterations`. The function does also take x passed by reference, but it is non constant and immediately changed to $x_0 = A^{\mathrm{T}}b$ as an initial guess. We give x as an argument for the solution to be returned in, since the function returns the `int` number of iterations taken. The algorithm can take a long time to converge for large systems, so we need to perform some convergence testing. We have implemented a clause in the function for this. Before anything, we compute a relaxed tolerance `laxtol` which is the square root of the tolerance. If `tol` is 10^{-6}, convergence testing considers the tolerance 10^{-3}. Our testing statement checks that $||r||_2 >$ `laxtol`, i.e. we are not close to the required tolerance, and that $|||x_{i-1}||_\infty - ||x_i||_\infty| <$ `tol`, i.e. the iteration is improving by a small amount at most. If these are satisfied then clearly the algorithm is stagnating, since we are not within the desired tolerance but x is not changing enough after each iteration. The algorithm returns zero, which we regard as failure.

The NIHT algorithm exhibits a phase transition in its likelihood to succeed depending on k and m. A phase transition is where the behaviour of the algorithm changes abruptly as the parameters change. See Figure 5, where we have plotted two results for different sparsities. For low values of m, the algorithm initially fails. Once we reach a particular value, the probability of success grows

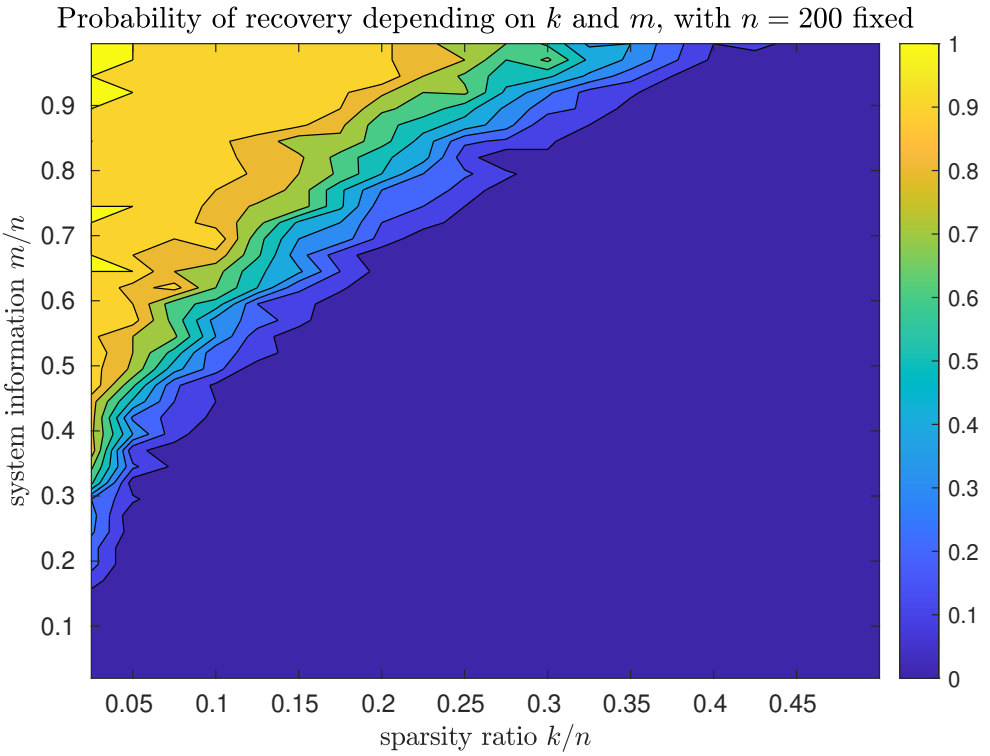

Figure 6: Contour graph of the likelihood of successful convergence as a function on both k and m. For a fixed sparsity k, we are more likely to solve the system with more information m. If we instead fix m, the algorithm appears more likely to succeed for smaller values of k. Sparsity parameters tested are $k = 5, 10, \ldots, 100$ and $m = 4, 9, \ldots, 199$. The value of n is fixed at 200. For each combination of k and m, we take 50 samples to estimate the likelihood of success.

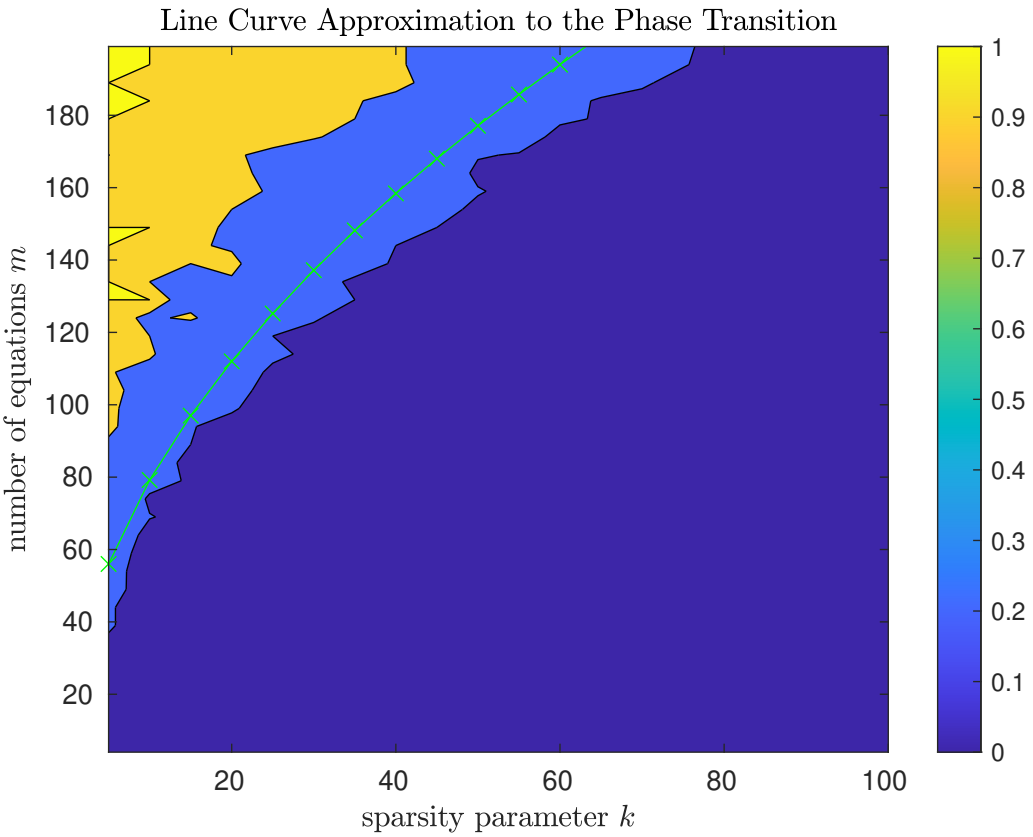

Figure 7: Line fitting to approximate the phase transition region. Replicate of Figure 6 with weaker contour information, and without scaling from dividing by $n = 200$. The line marked in green is $y = \mathcal{O}(\sqrt{k})$ a constant scaling of $\sqrt{k}$. Clearly, this curve is a close fit to the region in which the phase transition occurs.

rapidly to near certainty. Given k fixed, if the algorithm is likely to succeed for some $m = m_0$ then it is also likely to succeed for all $m > m_0$.

The likelihood of success is computed using the `stability()` function. Stability performs T executions of NIHT, each time formulating a new system, and then outputs the proportion of those executions which solved the problem successfully. Success is evaluated under two criteria, one of which is that the NIHT claims completion in > 0 iterations, since the algorithm returns zero if the solution fails. The other criterion is that the $L\infty$-norms of the original x and the newly computed solution must vary by less than 10^{-3}. Importantly, this does not check equality of the vectors. It only checks that the largest values by magnitude differ by a small amount. We implement this check because even if the algorithm succeeds, the new x may have entries that are only close to the original, by a distance relating to the tolerance given to the `NIHT()` implementation. We may be concerned that this logic could lead to improper registration of success. However, since we are working with randomised systems, the likelihood of an incorrect solution having an $L\infty$-norm virtually equal to the correct solution is extremely unlikely.

Figure 5 indicates how the number of rows affects the algorithm. For $k = 20$, we are working with a sparsity ratio of 0.1. The requirement for success is visibly $m > 120$. If we increase k to 50, we have sparsity ratio 0.25. We now require $m > 180$, which is closer to a full system.

See Figure 6, where we have constructed a contour graphic of the phase transition behaviour. Our domain is $k = 5 : 5 : 100$ and $m = 4 : 5 : 199$ in MATLAB notation. For every combination, the probability is sampled from 50 tests. This was chosen such that the entire result could be computed within a reasonable amount of time. The sparsity ratio is measured up to $k/n = 0.5$, since beyond this value we will clearly only observe failure. Likewise, the system information parameter m/n is only measured up to 199/200, since we are only testing NIHT for underdetermined systems. It is important to note that the results have a lot of noise. The region of failure is in blue and the region of success is yellow. The transition region between these disjoint subsets of the parameter space is a thin region of steep gradient. In Figure 7 we have found that, from our data, a line of the form $m = \alpha\sqrt{k}$ for scalar α approximates the curve defined by the phase transition region. The method for finding this relationship is based on results from [1], mainly being Theorem 2 on phase transitions for linear inverse problems. These results do not apply directly to the NIHT algorithm since it does not belong to the same class of methods.

Our key result from our computations is that NIHT is often successful for a system with high sparsity. If the sparsity parameter k is low, we can still find the k-sparse solution x to the system even when $m < n$ such that the system is underdetermined. Intuitively this makes sense, since we are only considering k non-zero unknowns in x. As k increases, we must increase m such that the algorithm has more information. If the problem is dense, it can be that NIHT fails to solve the system even if it is fully determined. For these denser problems, we would probably be more suited to implementing an algorithm like gradient descent, although we do not maintain the results we have for NIHT concerned with solving an underdetermined system correctly.

4 Conclusion

The normalised iterative hard thresholding algorithm is an effective method for finding a sparse solution of a linear system. We have shown how the behaviour of NIHT varies depending on a combination of the parameters of the linear system. Specifically, we have explored the properties of the phase transition. We have acknowledged that the phase transition for this method is not properly understood, but we have linked it to current results on phase transitions for optimisation methods.

We have considered the relationship between applying NIHT to find a sparse solution, and using gradient descent to solve the least squares problem. Despite being extremely similar in terms of the instructions performed, both algorithms are well suited for very different problems. The traditional least squares problem is overdetermined, and we apply gradient descent to find a solution which satisfies none of the constraints but minimises a squared distance. In comparison, we have found that the NIHT method is most effective for underdetermined sparse problems and will often fail to solve fully determined systems. Out of the scope of this project is the implementation of more effective numerical methods. The gradient descent method is helpful to understand, but is often avoided in favour of a more effective algorithm such as conjugate gradient. Compressed sensing methods that use alternative methods are explored in [2].

In the context of compressed sensing, we have explored the potential of NIHT to recover sparse solutions to linear systems. These systems often describe observations of signals for which we want to recover the entirety of the original information. Our priority is to reduce the stored information as much as possible, such that we can implement an algorithm to recover the signal entirely, and we do not consider any restrictions on the cost of the recovery algorithm we implement. We have shown that NIHT is an effective method for compressed sensing applications when the sparsity ratio is relatively low, in that it will correctly solve the underdetermined system for the original sparse vector used in formulating the problem. As the sparsity parameter increases, we are increasing the stored information and our problem becomes less about compression and more about solving a general linear system. As such, we have discussed how NIHT is less suited towards these problems, and that a method such as steepest descent is more appropriate. However we must always be aware that steepest descent has no guarantee on finding the desired solution when the problem is underdetermined. Our argument is that NIHT appears to be effective if and only if it is applied as a method in compressed sensing.

References

[1] Dennis Amelunxen et al. "Living on the edge: Phase transitions in convex programs with random data". In: *Information and Inference: A Journal of the IMA* 3 (2014), pp. 224–294.

[2] Jeffrey D Blanchard and Jared Tanner. "GPU accelerated greedy algorithms for compressed sensing". In: *Mathematical Programming Computation* 5 (2013), pp. 267–304.

[3] Thomas Blumensath and Mike E Davies. "Normalized iterative hard thresholding: Guaranteed stability and performance". In: *IEEE Journal of selected topics in signal processing* 4 (2010), pp. 298–309.

[4] *std::sort, cppreference.com.*

A C++ Code Implementations

A.1 Compiled Files

`main.cpp`

```cpp
#include <iostream>
#include <fstream>
#include <thread>

#include "mvector.h"
#include "mmatrix.h"
```

```
7
8  using namespace std;
9
10 // non-class functions
11 int SDLS(const MMatrix& A, const MVector& b, MVector& x, int maxIterations, double
       tol, bool writesFile = false, string filename = "")
12 {
13     // r is the residual for solving the normal equations
14     MVector r = A.transpose() * (b - A*x);
15
16     // block for writing data. f must be created but if !writesFile we write nothing
17     ofstream f;
18     if (writesFile && filename != "")
19     {
20         f.open(filename);
21         if (!f)
22         {
23             exception e;
24             throw e;
25         }
26         f << "Steepest Descent Algorithm implementation" << "\n";
27         f << "A = " << "\n" << A << "\n";
28         f << "b = " << b << "\n";
29         f << "Iterations for x:\n";
30         f << x << "\n";
31     }
32
33     int iter = 0;
34     while (iter < maxIterations && r.L2Norm() > tol)
35     {
36         // perform the iteration
37         double alpha = (dot(r, r))/(dot(A*r, A*r));
38         x = x + alpha * r;
39         r = r - alpha*(A.transpose()*(A*r));
40
41         if (writesFile && filename != "")
42         {
43             f << x << "\n";
44         }
45
46         iter++;
47     }
48     if (writesFile && filename != "")
49     {
50         f.close();
51     }
52
53     if (r.L2Norm() > tol)
54     {
55         // iteration fails to converge to a solution
56         return 0;
57     }
58
59     return iter;
60 }
61
62 int NIHT(const MMatrix& A, const MVector& b, MVector&x, int k, const int&
       maxIterations, const double& tol, const bool& writesFile = false, string filename
        = "")
63 {
```

```
    // compute vector norms and sqrt(tol) for convergence checking
    double x_infnorm = x.LInfNorm(), x_2norm = x.L2Norm();
    double laxtol = sqrt(tol);

    // Initialise starting vector
    x = A.transpose()*b;
    x.threshold(k); // Get k largest values
    MVector r = A.transpose() * (b - A*x);

    // block for writing data
    ofstream f;
    if (writesFile && filename != "")
    {
        f.open(filename);
        if (!f)
        {
            exception e;
            throw e;
        }
        f << "Normalised Iterative Hard Thresholding Algorithm implementation" << "\n
    ";
        f << "A is a random " << A.Rows() << " by " << A.Cols() << " matrix.\n";
        f << "x is a random " << x.size() << "-vector with sparsity " << k << ".\n";
        f << "Norm of the residual:\n";
        f << r.L2Norm() << "\n";

        // f << "Iterations for x:\n";
        // f << x << "\n";
    }

    // begin iteration
    int iter = 0;
    while (iter < maxIterations && r.L2Norm() > tol)
    {
        double alpha = (dot(r, r))/(dot(A*r, A*r));
        // compute and threshold x
        MVector xPrev = x;
        x = x + alpha*r;
        x.threshold(k);

        // cout << x << "\n";

        // compute residual
        r = A.transpose() * (b - A*x);

        // do some convergence testing
        if (abs(xPrev.LInfNorm() - x.LInfNorm()) < tol && r.L2Norm() > laxtol)
        {
            return 0;
        }

        if (writesFile && filename != "")
        {
            f << r.L2Norm() << "\n";
            // f << x << "\n";
        }
        iter++;
    }

    if (writesFile && filename != "")
```

```
    {
        f.close();
    }
    if (r.L2Norm() > tol)
    {
        // iteration has failed to converge
        return 0;
    }
    return iter;

}

double stability(int m, int n, int k, int T)
{
    /*
    Stability returns the probability p(m) of successful recovery of NIHT given
    dimensions, sparsity, and the number of times T to perform NIHT.
    */
    double successes = 0;
    for (int i = 0; i < T; i++)
    {
        // make a k-sparse vector x, make A, form b=Ax
        MVector x(n);
        x.initialize_normal(k);
        MMatrix A(m, n);
        A.initialize_normal();
        MVector b = A*x;

        double x_linfnorm_orig = x.LInfNorm();

        int iterations = NIHT(A, b, x, k, 100000, 1e-6);
        if (abs(x.LInfNorm()-x_linfnorm_orig) < 1e-3 && iterations > 0)
        {
            successes++;
        }
    }
    return successes/double(T);
}

int main()
{
    srand(time(NULL));

    // (1) generating a matrix and testing that transpose() works
    MMatrix A(2, 3);
    A(0, 0) = 1;
    A(1, 1) = 2;
    A(2, 0) = 1;

    cout << A << endl;
    A.transpose();
    cout << A << endl;

    // (2) least squares convergence
    MMatrix A(3, 2);
    A(0, 0) = 1;
    A(1, 0) = 2;
    A(2, 0) = -1;
    A(0, 1) = 2;
    A(1, 1) = 1;
```

```
    A(2, 1) = 0;

    MVector b = {10, -1, 0};

    MVector x = {0, 0};
    cout << A << endl;
    int n = SDLS(A, b, x, 1000, 1e-6, true, "sdsolveA1.txt");

    A(2, 0) = 1.8;
    A(2, 1) = -2;
    x = {0, 0};
    n = SDLS(A, b, x, 1000, 1e-6, true, "sdsolveA2.txt");

    A(2, 0) = -2;
    A(2, 1) = -2;
    x = {0, 0};
    n = SDLS(A, b, x, 1000, 1e-6, true, "sdsolveA3.txt");
    cout << n << ", " << x << ", " << A*x << endl;

    // (3) For generating a histogram of the entries for a 1000x1 random matrix
    MMatrix N(1000, 1);
    N.initialize_normal();

    ofstream f;
    f.open("normaldisttest.txt");
    if (!f)
    {
        exception e;
        throw e;
    }
    f << N << endl;
    f.close();

    // (4) perform normalised iterative hard thresholding for a single example
    double eqnratio, sparsity;
    eqnratio = 0.9, sparsity = 0.25;
    int n = 20;
    int m = eqnratio * n;
    int k = sparsity * n;
    MMatrix A(m, n);
    MVector b(m);
    MVector x(n);

    A.initialize_normal();
    x.initialize_normal(k);

    b = A*x;

    cout << A << endl;
    cout << x << endl;
    cout << b << endl;

    int iterations = NIHT(A, b, x, k, 100000, 1e-6);
    cout << "iteration NIHT performed in " << iterations << " steps\n";
    cout << x << endl;

    // (5) compute the phase transition
    int n = 200, k = 20, T = 50;

    for (int m = 4; m <= 199; m += 5)
```

```
    {
        double p = stability(m, n, k, T);
        cout << "For m = " << m << ", n = " << n
        << ", the probability of recovery for sparsity " << k << " is " << p << endl;
    }

    // (6) testing
    cout << stability(60, 50, 10, 50) << "\n";
    cout << stability(120, 100, 20, 50) << "\n";
    cout << stability(240, 200, 40, 50) << "\n";

    MMatrix A(3,2);
    int k = 2;
    A.initialize_normal();
    MVector xstar(2);
    xstar.initialize_normal(2);
    MVector xiter(2), yiter(2);

    cout << SDLS(A, A*xstar, xiter, 1000, 1e-4, true, "randomsdlstest.txt") << endl;
    cout << NIHT(A, A*xstar, yiter, k, 1000, 1e-4, true, "randomnihttest.txt") <<
    endl;

    // (7) 2d figure generator
    /*
    compute the figure as a matrix.
    n = 200,
    m ranging 4:5:199 is 5*(1:1:40)-1
    k ranging 5:5:100 is 5*(1:1:20)
    */

    ofstream f;
    string filename = "placeholder.txt";
    f.open(filename);

    // generate a results matrix with all the values and output this to a file.
    // generate a file for each value of m with the results

    int n = 200;
    int T = 50;
    int rows = 40, cols = 20;
    for (int m = 1; m <= rows; m++)
    {
        int M = 5 * m - 1;
        for (int k = 1; k <= cols; k++)
        {
            int K = 5*k;
            double p = stability(M, n, K, T);

            cout << "For m = " << M << ", n = " << n
            << ", the probability of recovery for sparsity " << K << " is " << p <<
    endl;

            // index A by row m, column k
            f << "A(" << M << ", " << K << ") = " << p << "\n";

        }
    }
    f.close();

    return 0;
```

```
300 }
```

mmatrix.cpp

```
1  #include <vector>
2  #include <iostream>
3  #include <iomanip>
4  #include <ios>
5  #include <algorithm>
6  #include <cmath>
7  #include <cassert>
8
9  #include "mmatrix.h"
10 #include "mvector.h"
11
12 using namespace std;
13
14 MVector operator*(const MMatrix& A, const MVector& v)
15 {
16     // check compatibility
17     if (A.Cols() != v.size())
18     {
19         cout << "wrong dims" << endl;
20         exception e;
21         throw e;
22     }
23     // initialise output vector
24     int vout_size = A.Rows();
25     MVector vout = MVector(vout_size);
26
27     for (int i = 0; i < A.Rows(); i++)
28     {
29         // sum across j
30         double sum = 0;
31         for (int j = 0; j < A.Cols(); j++)
32         {
33             sum += A(i, j)*v[j];
34         }
35         vout[i] = sum;
36     }
37     return vout;
38 }
39
40 MMatrix operator*(const MMatrix& A, const MMatrix& B)
41 {
42     // compatibility check
43     assert(A.nCols == B.nRows);
44
45     // construct the return matrix of correct size
46     MMatrix C(A.nRows, B.nCols);
47
48     // compute all C(i, j)
49     for (int i = 0; i < C.nRows; i++)
50     {
51         for (int j = 0; j < C.nCols; j++)
52         {
53             double sum = 0;
54             for (int k = 0; k < A.nCols; k++)
55             {
56                 sum += A(i, k)*B(k, j);
57             }
```

```
            C(i, j) = sum;
        }
    }
    return C;
}

MMatrix MMatrix::transpose() const
{
    MMatrix C(nCols, nRows);
    for (int i = 0; i < nRows; i++)
    {
        for (int j = 0; j < nCols; j++)
        {
            // hacky. i and j are rows and colums respectively of A, opposite for C
            C(j, i) = A[j + i*nCols];
        }
    }
    return C;
}

void MMatrix::initialize_normal()
{
    for (vector<double>::iterator term = A.begin(); term != A.end(); term++)
    {
        // set every entry to output of a zero mean, variance 1/m gaussian
        *term = rand_normal()/sqrt(nCols);
    }

}

ostream& operator<<(ostream& os, const MMatrix& A)
{
    for (int i = 0; i < A.Rows(); i++)
    {
        os << "|  ";
        for (int j = 0; j < A.Cols(); j++)
        {
            os << setw(3) << A(i, j) << "  ";
        }
        os << "|\n";
    }
    return os;
}
```

mvector.cpp

```
#include <vector>
#include <iostream>
#include <iomanip>
#include <algorithm>
#include <cmath>
#include <iterator>
#include <cassert>
#include <ctime>

#include "mvector.h"

using namespace std;

// constructors in header

```

```
MVector operator*(const double& lhs, const MVector& rhs)
{
    MVector temp = rhs;
    for (int i = 0; i < temp.size(); i++)
    {
        temp[i] *= lhs;
    }
    return temp;
}

MVector operator*(const MVector& lhs, const double& rhs)
{
    return rhs*lhs;
}

MVector operator+(const MVector& lhs, const MVector& rhs)
{
    if (lhs.size() != rhs.size())
    {
        exception e;
        throw e;
    }
    MVector temp(lhs.size());
    for (int i = 0; i < temp.size(); i++)
    {
        temp[i] = lhs[i] + rhs[i];
    }
    return temp;
}

MVector operator-(const MVector& lhs, const MVector& rhs)
{
    return lhs+(-1*rhs);
}

// alternative overload such that given MVector v, -v is allowed
MVector MVector::operator-()
{
    MVector vec(v.size());
    for (int i = 0; i < vec.size(); i++)
    {
        vec[i] = -v[i];
    }
    return vec;
}

MVector operator/(const MVector& lhs, const double& rhs)
{
    return (1/rhs)*lhs;
}

bool operator==(const MVector& lhs, const MVector& rhs)
{
    assert(lhs.size() == rhs.size());
    for (int i = 0; i < lhs.size(); i++)
    {
        if (lhs[i] != rhs[i])
        {
            return false;
        }
```

```
76      }
77      return true;
78
79  }
80
81  ostream& operator<<(ostream& os, const MVector& v)
82  {
83    int n = v.size();
84    os << "(";
85    for (int i = 0; i < n-1; i++)
86      {
87          os << setw(10) << v[i] << ", ";
88      }
89      os << v[n-1];
90    os << ")";
91    return os;
92  }
93
94  double MVector::LInfNorm() const
95  {
96      double maxAbs = 0;
97      size_t s = size();
98      for (int i=0; i<s; i++)
99      {
100         maxAbs = max(abs(v[i]), maxAbs);
101     }
102     return maxAbs;
103 }
104
105 double MVector::L2Norm() const
106 {
107     double sum = 0;
108     for (int i = 0; i < v.size(); i++)
109     {
110         sum += v[i]*v[i];
111     }
112     return sqrt(sum);
113 }
114
115 double dot(const MVector& lhs, const MVector& rhs)
116 {
117     if (lhs.size() != rhs.size())
118     {
119         throw invalid_argument("error: dot product not defined for vectors of non-
    equal dimension");
120     }
121     double sum = 0;
122     for (int i = 0; i < lhs.size(); i++)
123     {
124         sum += lhs[i]*rhs[i];
125     }
126     return sum;
127 }
128
129 bool abscmp(const double& a, const double& b)
130 {
131     return abs(a) < abs(b);
132 }
133
134 void MVector::threshold(int k)
```

```
{
    vector<double> v_copy = v;
    // sort a copy by absolute value, then identify the k-th largest value
    sort(v_copy.begin(), v_copy.end(), abscmp);
    // this picks out the k-th largest value
    double bound = v_copy[v_copy.size() - k];

    // iterate through the MVector contents performing the thresholding
    for (vector<double>::iterator term = v.begin(); term != v.end(); term++)
    {
        if (abscmp(*term, bound))
        {
            *term = 0;
        }
    }
}

void MVector::initialize_normal(int k)
{
    for (vector<double>::iterator term = v.begin(); term != v.end(); term++)
    {
        *term = rand_normal();
    }

    // complete copy of threshold
    vector<double> v_copy = v;
    sort(v_copy.begin(), v_copy.end(), abscmp);
    double bound = v_copy[v_copy.size() - k];
    for (vector<double>::iterator term = v.begin(); term != v.end(); term++)
    {
        if (abscmp(*term, bound))
        {
            *term = 0;
        }
    }
}

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);
}
```

A.2 Header Files

mmatrix.h

```
#ifndef MMATRIX_H // the 'include guard'
#define MMATRIX_H

#include <vector>
#include <iostream>
#include "mvector.h"
```

```

using namespace std;

// Class that represents a mathematical matrix
class MMatrix
{
public:
  // constructors
  MMatrix() : nRows(0), nCols(0) {}
  MMatrix(int n, int m, double x = 0) : nRows(n), nCols(m), A(n * m, x) {}

  // set all matrix entries equal to a double
  MMatrix &operator=(double x)
  {
    for (unsigned i = 0; i < nRows * nCols; i++) A[i] = x;
    return *this;
  }

  // access element, indexed by (row, column) [rvalue]
  double operator()(int i, int j) const
  {
    return A[j + i * nCols];
  }

  // access element, indexed by (row, column) [lvalue]
  double &operator()(int i, int j)
  {
    return A[j + i * nCols];
  }

  friend MVector operator*(const MMatrix& A, const MVector& v);
  friend MMatrix operator*(const MMatrix& A, const MMatrix& B);
  MMatrix transpose() const;

  void initialize_normal();

  friend ostream& operator<<(ostream& os, const MMatrix& A);

  // size of matrix
  int Rows() const { return nRows; }
  int Cols() const { return nCols; }

private:
  unsigned int nRows, nCols;
  vector<double> A;
};


#endif
```

mvector.h

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

#include <vector>
#include <cstdlib>
#include <cmath>
#include <iostream>

using namespace std;
```

```
10
11 // Class that represents a mathematical vector
12 class MVector
13 {
14 public:
15   // constructors
16   MVector() {}
17   MVector(int n) : v(n) {}
18   MVector(int n, double x) : v(n, x) {}
19   MVector(initializer_list<double> l) : v(l) {}
20
21   // access element (lvalue) (see example sheet 5, q5.6)
22   double &operator[](int index)
23   {
24     return v[index];
25   }
26
27   // access element (rvalue) (see example sheet 5, q5.7)
28   double operator[](int index) const {
29     return v[index];
30   }
31
32   // operator overloads
33   MVector operator-();
34
35   friend MVector operator*(const double& lhs, const MVector& rhs);
36   friend MVector operator*(const MVector& lhs, const double& rhs);
37   friend MVector operator+(const MVector& lhs, const MVector& rhs);
38   friend MVector operator-(const MVector& lhs, const MVector& rhs);
39   friend MVector operator/(const MVector& lhs, const double& rhs);
40   friend bool operator==(const MVector& lhs, const MVector& rhs);
41
42   friend ostream& operator<<(ostream& os, const MVector& v);
43
44   // get size
45   int size() const { return v.size(); } // number of elements
46
47   //vector norms
48   double LInfNorm() const;
49   double L2Norm() const;
50
51   // threshold
52   void threshold(int);
53   void initialize_normal(int);
54
55
56 private:
57   vector<double> v;
58 };
59
60 // dot product
61 double dot(const MVector& lhs, const MVector& rhs);
62
63 // comparison for sorting
64 bool abscmp(const double&, const double&);
65
66 double rand_normal();
67
68
69
```

```
#endif
```

B Octave Results for Steepest Descent

Replacing the third row of A with $[1.8, -2]$.

```
octave:30> A(3,:) = [1.8, -2]
A =
    1.0000   2.0000
    2.0000   1.0000
    1.8000  -2.0000

octave:31> [U,Z,V] = svd(A)
U =
    0.737098   0.045736   0.674236
    0.563941   0.508121  -0.650987
    -0.372367   0.860070   0.348743

Z =
Diagonal Matrix
    3.0285         0
         0    2.8405
         0         0

V =
    0.3945   0.9189
    0.9189  -0.3945

octave:35> x = [   0.87027, 2.07243]'
x =
    0.8703
    2.0724

octave:36> (b-A*x)'*U
ans =
    7.0920e-06  -2.0202e-06   7.3933e+00
```

Replacing the third row of A with $[-2, -2]$

```
octave:37> A(3,:) = [-2, -2]
A =
    1   2
    2   1
    -2  -2

octave:38> x = [-4.70588, 6.29412]
x =
    -4.7059   6.2941

octave:39> x = [-4.70588, 6.29412]'
x =
    -4.7059
    6.2941

octave:40> [U,Z,V] = svd(A)
U =
    -5.1450e-01   7.0711e-01   4.8507e-01
    -5.1450e-01  -7.0711e-01   4.8507e-01
```

```
    6.8599e-01   -1.8397e-16    7.2761e-01

Z =
Diagonal Matrix
    4.1231         0
         0    1.0000
         0         0

V =
    -0.7071   -0.7071
    -0.7071    0.7071

octave:41> (b-A*x)'*U
ans =
    1.3720e-05   -7.4695e-16    4.3656e+00
```