HW6 - Math5610

Andrew Sheridan

November 3, 2016

Problem 1

Cholesky Decomposition

Below is my implementation of the Cholesky Decomposition, as well as a test case and all the methods required for its execution. Using a diagonally dominant symmetric matrix, I compute the Cholesky Decomposition. I then multiply this decomposition by its transpose, which gives us our original matrix, as intended.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//main.cpp
#include "Matrix.h";
#include "Vector.h"
#include <iostream>

int main()
{
  //Problem 1: Cholesky Decomposition
  int size1 = 4;
  double** matrix1 = CreateDiagonallyDominantSymmetricMatrix(size1);
  double** cholesky1 = CholeskyDecomposition(matrix1, size1);
  if (cholesky1 != NULL) {
    std::cout << "Our diagonally dominant test matrix and the result of computing the Cholesky Decomposition of the matrix." << std::endl;
    PrintMatrix(matrix1, size1);
    PrintMatrix(cholesky1, size1);

    double** transpose = Transpose(cholesky1, size1);
    std::cout << "L Transpose" << std::endl;
    PrintMatrix(transpose, size1);
    std::cout << "Multiplying the Cholesky Decomposition by its transpose." << std::endl;
    double** result = DotProduct(cholesky1, transpose, size1, size1, size1);
    PrintMatrix(result, size1);
  }

  return 0;
}
```

```
Our diagonally dominant test matrix and the result
   of computing the Cholesky Decomposition of the matrix.
```

```
40.713      0.428471    0.690885    0.71915
0.428471    40.4911     0.780028    0.410924
0.690885    0.780028    40.5797     0.139951
0.71915     0.410924    0.139951    40.401

6.38067     0           0           0
0.0671514   6.36291     0           0
0.108278    0.121447    6.36814     0
0.112708    0.0633917   0.0188514   6.35484

L Transpose
6.38067     0.0671514   0.108278    0.112708
0           6.36291     0.121447    0.0633917
0           0           6.36814     0.0188514
0           0           0           6.35484

Multiplying the Cholesky Decomposition by its transpose.
40.713      0.428471    0.690885    0.71915
0.428471    40.4911     0.780028    0.410924
0.690885    0.780028    40.5797     0.139951
0.71915     0.410924    0.139951    40.401
```

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Matrix.h
#pragma once
#include <iostream>
#include <cmath>
#include <random>
#include "Vector.h"

///Computes the Cholesky Decomposition of an n by n matrix A
/// Returns NULL if the matrix is not SPD
double** CholeskyDecomposition(double** A, unsigned int n) {
  if (IsMatrixSymmetric(A, n) == false)
    return NULL;

  double** L = new double*[n]; //Initialize the new matrix
  for (int i = 0; i < n; i++) {
    L[i] = new double[n];
    for (int j = 0; j < n; j++) {
      L[i][j] = 0;
    }
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < (i + 1); j++) {
      double entry = 0;
      for (int k = 0; k < j; k++) {
        entry += L[i][k] * L[j][k];
      }
      double sqrtValue = A[i][i] - entry;
      if (sqrtValue < 0)
        return NULL;

```

```
      // Conditional assignment. If the entry is diagonal, assign to the sqare root of the previous value.
      // Otherwise, Do computation for a nondiagonal entry.
      L[i][j] = i == j ? std::sqrt(sqrtValue) : (1.0 / L[j][j] * (A[i][j] - entry));
    }
  }

  return L;
}

///Checks to see if nxn matrix A is symmetric
bool IsMatrixSymmetric(double** A, unsigned n) {
  for (unsigned int i = 0; i < n; i++) {
    for (unsigned int j = 0; j <= i; j++) {
      if (A[i][j] != A[j][i])
        return false;
    }
  }
  return true;
}

/// Generates a random diagonally dominant square matrix of size n.
/// n: The size of the matrix
double** CreateDiagonallyDominantSymmetricMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n]; //Must do this before our second loop, so that all rows are initialized.
  }
  for (unsigned i = 0; i < n; i++) {
    for (unsigned j = i; j < n; j++) {
      double value = dis(generator); //Assign each entry in matrix to random number between 0 and 1
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }

  for (unsigned k = 0; k < n; k++) {
    matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries
  }

  return matrix;
}

/// Returns the transpose of the n by n matrix A
double** Transpose(double** A, unsigned int n) {
  double** matrix = new double*[n];
  for (int i = 0; i < n; i++) {
    matrix[i] = new double[n];
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      matrix[j][i] = A[i][j];
    }
  }
  return matrix;
```

```
}

/// Computes the dot product of matrices A (m x n) and B (n x p)
double** DotProduct(double**A, double** B, unsigned int m, unsigned int n, unsigned int p) {
  double** matrix = new double*[m];
  double** bTranspose = Transpose(B, n);
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[p];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = DotProduct(A[i], bTranspose[j], n);
    }
  }
  return matrix;
}
```

Problem 2

Testing Matrix Using Cholesky Decomposition

To test my implementation of the Cholesky Decomposition, I attempt to execute it using matrices of various forms. The first is a diagonally dominant symmetric matrix, which we expect to succeed. Other test cases used are using a diagonally dominant asymmetric, a symmetric matrix, and an asymmetric matrix. All three of these cases fail, returning null, which is the desired result.

```cpp
//Problem 2: Testing Positive Definiteness and Symmetry with Cholesky
int size2 = 4;

//Test with a diagonally dominant symmetric matrix.
double** matrix2a = CreateDiagonallyDominantSymmetricMatrix(size2);
double** cholesky2a = CholeskyDecomposition(matrix2a, size2);
std::cout << "Diagonally Dominant Symmetric Matrix Test" << std::endl;
if (cholesky2a == NULL)
  std::cout << "This matrix is not symmetric and positive definite." << std::endl;
else
  std::cout << "This matrix is symmetric and positive definite. " << std::endl;

//Test with a diagonally dominant asymmetric matrix
double** matrix2b = CreateDiagonallyDominantMatrix(size2);
double** cholesky2b = CholeskyDecomposition(matrix2b, size2);
std::cout << "Diagonally Dominant Matrix Test" << std::endl;
if (cholesky2b == NULL)
  std::cout << "This matrix is not symmetric and positive definite." << std::endl;
else
  std::cout << "This matrix is symmetric and positive definite. " << std::endl;

//Test with a symmetric matrix
double** matrix2c = CreateSymmetricMatrix(size2);
double** cholesky2c = CholeskyDecomposition(matrix2c, size2);
std::cout << "Symmetric Matrix Test" << std::endl;
if (cholesky2c == NULL)
  std::cout << "This matrix is not symmetric and positive definite." << std::endl;
else
  std::cout << "This matrix is symmetric and positive definite. " << std::endl;

//Test with a matrix
double** matrix2d = CreateMatrix(size2);
double** cholesky2d = CholeskyDecomposition(matrix2d, size2);
std::cout << "Matrix Test" << std::endl;
if (cholesky2d == NULL)
  std::cout << "This matrix is not symmetric and positive definite." << std::endl;
else
  std::cout << "This matrix is symmetric and positive definite. " << std::endl << std::endl;
```

```
Diagonally Dominant Symmetric Matrix Test
This matrix is symmetric and positive definite.
Diagonally Dominant Matrix Test
This matrix is not symmetric and positive definite.
```

```
Symmetric Matrix Test
This matrix is not symmetric and positive definite.
Matrix Test
This matrix is not symmetric and positive definite.
```

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Matrix.h
#pragma once
#include <iostream>
#include <cmath>
#include <random>
#include "Vector.h"

///Computes the Cholesky Decomposition of an n by n matrix A
/// Returns NULL if the matrix is not SPD
double** CholeskyDecomposition(double** A, unsigned int n) {
  if (IsMatrixSymmetric(A, n) == false)
    return NULL;

  double** L = new double*[n]; //Initialize the new matrix
  for (int i = 0; i < n; i++) {
    L[i] = new double[n];
    for (int j = 0; j < n; j++) {
      L[i][j] = 0;
    }
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < (i + 1); j++) {
      double entry = 0;
      for (int k = 0; k < j; k++) {
        entry += L[i][k] * L[j][k];
      }
      double sqrtValue = A[i][i] - entry;
      if (sqrtValue < 0)
        return NULL;

      // Conditional assignment. If the entry is diagonal, assign to ↩
          the sqare root of the previous value.
      // Otherwise, Do computation for a nondiagonal entry.
      L[i][j] = i == j ? std::sqrt(sqrtValue) : (1.0 / L[j][j] * (A[i↩
          ][j] - entry));
    }
  }

  return L;
}

///Checks to see if nxn matrix A is symmetric
bool IsMatrixSymmetric(double** A, unsigned n) {
  for (unsigned int i = 0; i < n; i++) {
    for (unsigned int j = 0; j <= i; j++) {
      if (A[i][j] != A[j][i])
        return false;
    }
  }
  return true;
}

```

```cpp
/// Generates a random diagonally dominant square matrix of size n.
/// n: The size of the matrix
double** CreateDiagonallyDominantSymmetricMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n]; //Must do this before our second loop, so that all rows are initialized.
  }
  for (unsigned i = 0; i < n; i++) {
    for (unsigned j = i; j < n; j++) {
      double value = dis(generator); //Assign each entry in matrix to random number between 0 and 1
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }

  for (unsigned k = 0; k < n; k++) {
    matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries
  }

  return matrix;
}

/// Generates a random  square matrix of size n.
// n: The size of the matrix
double** CreateMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = dis(generator); //Assign each entry in matrix to random number between 0 and 1
    }
  }

  return matrix;
}


/// Generates a random diagonally dominant square matrix of size n.
// n: The size of the matrix
double** CreateDiagonallyDominantMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = dis(generator); //Assign each entry in matrix to random number between 0 and 1
```

```
    }
  }

  for (unsigned k = 0; k < n; k++) {
    matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries
  }

  return matrix;
}

/// Generates a random diagonally dominant square matrix of size n.
/// n: The size of the matrix
double** CreateDiagonallyDominantSymmetricMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n]; //Must do this before our second loop, so that all rows are initialized.
  }
  for (unsigned i = 0; i < n; i++) {
    for (unsigned j = i; j < n; j++) {
      double value = dis(generator); //Assign each entry in matrix to random number between 0 and 1
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }

  for (unsigned k = 0; k < n; k++) {
    matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries
  }

  return matrix;
}

/// Generates a symmetric square matrix of size n.
// n: The size of the matrix
double** CreateSymmetricMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n]; //Must do this before our second loop, so that all rows are initialized.
  }
  for (unsigned i = 0; i < n; i++) {
    for (unsigned j = i; j < n; j++) {
      double value = dis(generator); //Assign each entry in matrix to random number between 0 and 1
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }

  return matrix;
}
```

Problem 3

1-Norm of Real Square Matrix

Below is all code necessary to compute the 1-Norm of a square n by n matrix A. I created a 4 by 4 matrix and manually augmented one of the entries to assure that the column it belongs to is returned as the maximum. After the code is the result of running the main function, and it is easy to see that the result is the column sum of largest magnitude.

```
//Andrew Sheridan
//Math 5610
//Written in C++

//Matrix.h
#pragma once
#include <iostream>
#include <cmath>
#include "Vector.h"

/// Generates a random  square matrix of size n.
// n: The size of the matrix
double** CreateMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = dis(generator); //Assign each entry in matrix to random number between 0 and 1
    }
  }

  return matrix;
}

///Computes the 1-norm of an n by n matrix A
double OneNorm(double** A, unsigned int n) {
  double columnMax = 0;
  for (unsigned int i = 0; i < n; i++) {
    double columnSum = 0;
    for (unsigned int j = 0; j < n; j++) {
      columnSum += std::abs(A[i][j]);
    }
    if (columnSum > columnMax)
      columnMax = columnSum;
  }
  return columnMax;
}

///Outputs an nxn matrix to the console
void PrintMatrix(double** matrix, unsigned size) {
  for (unsigned i = 0; i < size; i++) {
    for (unsigned j = 0; j < size; j++) {
      std::cout << std::setw(13) << std::left << matrix[i][j];
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
```

```
}
```

```
//Andrew Sheridan
//Math 5610
//Written in C++

//Matrix.h
#include "Matrix.h";
#include "Vector.h"
#include <iostream>

int main()
{
  //Problem 4: Infinity-Norm
  int size4 = 4;
  double** matrix4 = CreateMatrix(size4);
  matrix4[size4 - 2][size4 - 1] *= 10; //Augmentation of single entry.
  double result4 = InfinityNorm(matrix4, size4);

  std::cout << "Our test matrix and the result of computing the Infinity-Norm of the matrix." << std::endl;
  PrintMatrix(matrix4, size4);
  std::cout << result4 << std::endl << std::endl;

  return 0;
}
```

```
Our test matrix and the result of computing the 1-Norm of the matrix.
0.712955     0.428471     0.690885     0.71915
0.491119     0.780028     0.410924     0.579694
0.139951     0.401018     0.627317     3.24151
0.244759     0.694755     0.593902     0.631792

4.40979
```

Problem 4

Infinity-Norm of Real Square Matrix

Below is all code necessary to compute the Infinity-Norm of a square n by n matrix A. I created a 4 by 4 matrix and manually augmented one of the entries to assure that the row it belongs to is returned as the maximum. After the code is the result of running the main function, and it is easy to see that the result is the row sum of largest magnitude.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++

//Matrix.h
#pragma once
#include <iostream>
#include <cmath>
#include <random>
#include "Vector.h"

///Computes the inifinity norm of an n by n matrix A
double InfinityNorm(double** A, unsigned int n) {
  double rowMax = 0;
  for (unsigned int j = 0; j < n; j++) {
    double rowSum = 0;
    for (unsigned int i = 0; i < n; i++) {
      rowSum += std::abs(A[i][j]);
    }
    if (rowSum > rowMax)
      rowMax = rowSum;
  }
  return rowMax;
}

/// Generates a random  square matrix of size n.
// n: The size of the matrix
double** CreateMatrix(unsigned n) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = dis(generator); //Assign each entry in matrix to random number between 0 and 1
    }
  }

  return matrix;
}

///Outputs an nxn matrix to the console
void PrintMatrix(double** matrix, unsigned size) {
  for (unsigned i = 0; i < size; i++) {
    for (unsigned j = 0; j < size; j++) {
      std::cout << std::setw(13) << std::left << matrix[i][j];
    }
    std::cout << std::endl;
  }
```

```
  std::cout << std::endl;
}
```

```
#include "Matrix.h";
#include "Vector.h"
#include <iostream>

int main()
{
  //Problem 4: Infinity-Norm
  int size4 = 4;
  double** matrix4 = CreateMatrix(size4);
  matrix4[size4 - 2][size4 - 1] *= 10; //Augmentation of single entry.
  double result4 = InfinityNorm(matrix4, size4);

  std::cout << "Our test matrix and the result of computing the ↩
      Infinity-Norm of the matrix." << std::endl;
  PrintMatrix(matrix4, size4);
  std::cout << result4 << std::endl << std::endl;

  return 0;
}
```

```
Our test matrix and the result of computing the Infinity-Norm of the matrix.
0.712955     0.428471     0.690885     0.71915
0.491119     0.780028     0.410924     0.579694
0.139951     0.401018     0.627317     3.24151
0.244759     0.694755     0.593902     0.631792

5.17215
```

Problem 5

Estimating Condition Number

Below is my implementation of estimating condition number. To estimate it, we compute the inverse of a matrix A, then take the norm of both A and A inverse, then multiply the two values. This is tested below with a 5x5 matrix.

```
//Problem 5: Condition Number
int size5 = 5;
double** matrix5 = CreateMatrixWithRange(size5, 10000, -10000);
double result5 = ConditionNumber(matrix5, size5);

std::cout << "Our test matrix and the result of computing condition ↩
    number." << std::endl;
PrintMatrix(matrix5, size5);
std::cout << "Condition Number: " << result5 << std::endl << std::endl↩
    ;
```

```
Our test matrix and the result of computing condition number.
4259.11     -1430.58    3817.7      4383.01     -177.621
5600.56     -1781.51    1593.89     -7200.98    -1979.65
2546.34     -3516.98    -5104.81    3895.1      1878.05
2635.84     -1194.86    -8325.47    4246.6      -1442.73
-4044.39    -158.305    4805.93     -2845.42    -1655.8

Condition Number: 14.2361
```

```
/// Generates a random  square matrix of size n.
// n: The size of the matrix
double** CreateMatrixWithRange(unsigned n, int maxValue, int minValue)↩
    {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(minValue, maxValue); //↩
      Desired distribution

  double** matrix;
  matrix = new double *[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = dis(generator); //Assign each entry in matrix to ↩
          random number between 0 and 1
    }
  }

  return matrix;
}

///Creates the identity matrix of size n
double** CreateIdentityMatrix(unsigned n) {
  double** matrix = new double*[n];
  for (unsigned i = 0; i < n; i++) {
    matrix[i] = new double[n];
    for (unsigned j = 0; j < n; j++) {
      matrix[i][j] = 0;
    }
```

```
      matrix[i][i] = 1;
    }
  return matrix;
}

///Computes the inifinity norm of an n by n matrix A
double InfinityNorm(double** A, unsigned int n) {
  double rowMax = 0;
  for (unsigned int j = 0; j < n; j++) {
    double rowSum = 0;
    for (unsigned int i = 0; i < n; i++) {
      rowSum += std::abs(A[i][j]);
    }
    if (rowSum > rowMax)
      rowMax = rowSum;
  }
  return rowMax;
}

//Computes the inverse of n by n matrix A
double** Inverse(double** A, unsigned int n) {
  double** matrix = CreateIdentityMatrix(n);
  double ratio, a;
  int i, j, k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      if (i != j) {
        ratio = A[j][i] / A[i][i];
        for (k = 0; k < n; k++) {
          A[j][k] -= ratio * A[i][k];
        }
        for (k = 0; k < n; k++) {
          matrix[j][k] -= ratio * matrix[i][k];
        }
      }
    }
  }
  for (i = 0; i < n; i++) {
    a = A[i][i];
    for (j = 0; j < n; j++) {
      matrix[i][j] /= a;
    }
  }
  return matrix;
}

//Estimates the condition number of n by n matrix A
double ConditionNumber(double** A, unsigned int n) {
  double** aCopy = CopyMatrix(A, n);
  double** inverse = Inverse(aCopy, n);

  double aNorm = InfinityNorm(A, n);
  double inverseNorm = InfinityNorm(inverse, n);

  return aNorm * inverseNorm;
}
```

Problem 6

Work Needed for Condition Number Estimation

Similar to a problem from the previous assignment, I test my method with matrices ranging from size 5x5 to 160x160. The operations required in each phase of the computation are summed and returned, which we output to the console, as shown below.

```
//Problem 6: Condition Number Operation Counts
for (int size6 = 5; size6 <= 160; size6 *= 2) {
  double** matrix6 = CreateMatrix(size6);
  double conditionNumber = ConditionNumber(matrix6, size6);
}
```

```
The operations required to estimate CN for size 5: 330
The operations required to estimate CN for size 10: 2310
The operations required to estimate CN for size 20: 17220
The operations required to estimate CN for size 40: 132840
The operations required to estimate CN for size 80: 1043280
The operations required to estimate CN for size 160: 8268960
```

```
//Matrix.h

///Computes the infinity norm of an n by n matrix A
double InfinityNormOperations(double** A, unsigned int n, long& ↩
    counter) {
  double rowMax = 0;
  for (unsigned int j = 0; j < n; j++) {
    double rowSum = 0;
    for (unsigned int i = 0; i < n; i++) {
      counter++;
      rowSum += std::abs(A[i][j]);
    }
    if (rowSum > rowMax)
      rowMax = rowSum;
    counter++;
  }
  return rowMax;
}

//Computes the inverse of n by n matrix A, incrementing the passed in ↩
    counter
double** InverseOperations(double** A, unsigned int n, long& counter) ↩
    {
  double** matrix = CreateIdentityMatrix(n);
  double ratio, a;
  int i, j, k;
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      counter++;
      if (i != j) {
        ratio = A[j][i] / A[i][i];
        counter++;
        for (k = 0; k < n; k++) {
          A[j][k] -= ratio * A[i][k];
          counter++;
        }
        for (k = 0; k < n; k++) {
```

```
          matrix[j][k] -= ratio * matrix[i][k];
          counter++;
        }
      }
    }
  }
  for (i = 0; i < n; i++) {
    a = A[i][i];
    for (j = 0; j < n; j++) {
      counter++;
      matrix[i][j] /= a;
    }
  }
  return matrix;
}

//Estimates the condition number of n by n matrix A
double ConditionNumber(double** A, unsigned int n) {
  long counter = 0;
  double** aCopy = CopyMatrix(A, n);
  double** inverse = InverseOperations(aCopy, n, counter);

  double aNorm = InfinityNormOperations(A, n, counter);
  double inverseNorm = InfinityNormOperations(inverse, n, counter);

  std::cout << "The operations required to estimate CN for size " << n
      << ": " << counter << std::endl;
  return aNorm * inverseNorm;
}
```

Problem 7

Solving Tridiagonal Systems

To ensure the success of our specialized tridiagonal algorithms, I have implemented this system twice. First, with the specialized algorithms and an n by 3 matrix, and second, with a n by n matrix and pre-existing algorithms. For simplicity, the right hand side is a vector of ones. After performing Gaussian Elimination and back substitution on each matrix with its right hand side, we get the same resulting vector x.

```
//Andrew Sheridan
//Math 5610
//Written in C++

//main.cpp
#include "Matrix.h";
#include "Vector.h"
#include <iostream>

int main()
{
  //Problem 7: Tridiagonal Matrices
  int size7 = 5;

  std::cout << "TESTING MINIFIED TRIDIAGONALS" << std::endl;
  std::cout << "Tridiagonal matrix before gaussian elimination" << std::endl;
  double** matrix7 = CreateMinifiedTridiagonal(size7);
  double** copy7 = CopyMatrix(matrix7, size7);
  double* vector7 = CreateOnesVector(size7);
  PrintMatrix(matrix7, size7, 3);
  PrintVector(vector7, size7);

  std::cout << "Tridiagonal matrix after gaussian elimination" << std::endl;
  TridiagonalElimination(copy7, vector7, size7);
  PrintMatrix(copy7, size7, 3);
  PrintVector(vector7, size7);

  std::cout << "Solution after back substitution" << std::endl;
  double* result7 = TridiagonalBackSubstitution(copy7, vector7, size7);
  PrintVector(result7, size7);

  std::cout << "COMPARING TO FULL TRIDIAGONALS MATRIX OPERATIONS" << std::endl;
  std::cout << "Tridiagonal matrix before gaussian elimination" << std::endl;
  double** matrix7b = CreateTridiagonalMatrix(size7);
  PrintMatrix(matrix7b, size7);
  double* vector7b = CreateOnesVector(size7);

  std::cout << "Tridiagonal matrix after gaussian elimination" << std::endl;
  GaussianElimination(matrix7b, vector7b, size7);
  PrintMatrix(matrix7b, size7);
  PrintVector(vector7b, size7);
  double* result7b = BackSubstitution(matrix7b, vector7b, size7);

  std::cout << "Solution without minification." << std::endl;
  PrintVector(result7b, size7);

  return 0;
```

```
}
```

```
TESTING MINIFIED TRIDIAGONALS
Tridiagonal matrix before gaussian elimination
0             -2            1
1             -2            1
1             -2            1
1             -2            1
1             -2            0

1             1             1             1             1

Tridiagonal matrix after gaussian elimination
0             -2            1
0             -1.5          1
0             -1.33333      1
0             -1.25         1
0             -1.2          0

1             1.5           2             2.5           3

Solution after back substitution
-2.5          -4            -4.5          -4            -2.5

COMPARING TO FULL TRIDIAGONALS MATRIX OPERATIONS
Tridiagonal matrix before gaussian elimination
-2            1             0             0             0
1             -2            1             0             0
0             1             -2            1             0
0             0             1             -2            1
0             0             0             1             -2

Tridiagonal matrix after gaussian elimination
-2            1             0             0             0
0             -1.5          1             0             0
0             0             -1.33333      1             0
0             0             0             -1.25         1
0             0             0             0             -1.2

1             1.5           2             2.5           3

Solution without minification.
-2.5          -4            -4.5          -4            -2.5
```

```
//Matrix.h

///Creates an n by n tridiagonal matrix
double** CreateTridiagonalMatrix(unsigned n) {
  double** newMatrix = new double*[n];
  for (int i = 0; i < n; i++) {
    newMatrix[i] = new double[n];
```

```
    for (int j = 0; j < n; j++) {
      newMatrix[i][j] = 0;
    }
  }
  for (int i = 0; i < n - 1; i++) {
    newMatrix[i][i] = -2;
    newMatrix[i][i + 1] = 1;
    newMatrix[i + 1][i] = 1;
  }
  newMatrix[n - 1][n - 1] = -2;

  return newMatrix;
}

///Creates and n by 3 tridiagonal matrix
double** CreateMinifiedTridiagonal(unsigned n) {
  double** newMatrix = new double*[n];
  newMatrix[0] = new double[3];
  newMatrix[0][0] = 0;
  newMatrix[0][1] = -2;
  newMatrix[0][2] = 1;
  for (int i = 1; i < n - 1; i++) {
    newMatrix[i] = new double[3];
    newMatrix[i][0] = 1;
    newMatrix[i][1] = -2;
    newMatrix[i][2] = 1;
  }
  newMatrix[n - 1] = new double[3];
  newMatrix[n - 1][0] = 1;
  newMatrix[n - 1][1] = -2;
  newMatrix[n - 1][2] = 0;
  return newMatrix;
}

///Does Gaussian Elimination on a minified tridiagonal matrix and right-hand-side b
void TridiagonalElimination(double** A, double* b, unsigned int n) {
  for (unsigned int i = 0; i < n - 1; i++) {
    double factor = A[i + 1][0] / A[i][1];
    A[i + 1][0] = 0;
    A[i + 1][1] -= (A[i][2] * factor);
    b[i + 1] -= b[i] * factor;
  }
}

///Does Back Substitution on a minified tridiagonal matrix and right-hand-side b
double* TridiagonalBackSubstitution(double** A, double* b, unsigned int n) {
  double* x = new double[n];
  x[n - 1] = b[n - 1] / A[n - 1][1];
  for (int k = n-2; k >= 0; k--) {
    x[k] = b[k];
    x[k] -= A[k][2] * x[k + 1];
    x[k] /= A[k][1];
  }
  return x;
}
```