Software Manual for Math 5610

Andrew Sheridan

December 16th, 2016 *

*Last Revised: December 9, 2016

Contents

1 Introduction

The purpose of this document is twofold. The primary purpose is educational. Creating functional and well-structured code is a difficult task. In most computational course work, once an assignment is finished, the code quickly becomes lost and forgotten, never to be opened or utilized again. Knowing that I would eventually create this document drove me to write better code, so that I wouldn't need to remind myself of how it works before adding it to the document.

Its second reason is professional, or rather, pre-professional. It is an attempt to internalize the principles of well-documented and well-maintained code, so that in my professional career I will develop code which can be picked up and understood by co-workers easily.

I was once told that the key to success in Software Development is to make yourself dispensable. Creating code which is difficult to understand and maintain will make employers depend on your knowledge of the code, and create a poor product. Chances are you'll get stuck at the same job working on the same project for many years, making your knowledge specific and stale. Writing code which is readable and maintainable makes it easy for others to pick up where you left off, so you can move on to bigger and better things.

The main body of this document will be the documentation of code which has been developed over the course of the semester. The sections of said body will rely heavily upon the structures contained in the Appendix. For example, matrix operations such as Gaussian Elimination and Back Substitution will be explored in the main body, while the Matrix structure itself is contained in the Appendix.

The first section of this document contains routines dealing with error and precision in regards to machine computing. Topics such as relative and absolute error, vector and matrix norms, and the rounding unit will be discussed. Basic routines relating to all these topics will be given.

Section two will explore nonlinear equations of one variable. In particular, it contains various root-finding methods, such as Bisection and the Newton Method.

The next section discusses methods for solving linear systems of equations. It is broken into subsections, such as Direct Methods (Gaussian Elimination, LU Factorization) and Iterative Methods (Jacobi Iteration, Conjugate Gradient Method).

TODO: Summary of the last section

TODO: Intro Conclusion

2 Precision and Error

2.1 Error

2.1.1 Absolute Error

2.1.2 Relative Error

2.1.3 Absolute Error in Complex Numbers

2.1.4 Relative Error in Complex Numbers

2.2 Rounding

2.3 Machine Precision

2.3.1 Single Precision

2.3.2 Double Precision

2.4 Vector Norms

2.4.1 One Norm/Manhattan Norm

2.4.2 Infinity Norm

2.4.3 L2 Norm

2.5 Matrix Norms

2.5.1 One Norm

2.5.2 Infinity Norm

3 Nonlinear Equations In One Variable

TODO: Section Header

3.1 Bisection

Description:

Input:

1.
2.
3.
4.

Output:

Code Written:

```
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
```

```
Console Output
--------------
```

3.2 Fixed Iteration

Description:

Input:

1.
2.
3.
4.

Output:

Code Written:

Usage Sample:

```
//Written by Andrew Sheridan in C++
```

```
Console Output
--------------
```

3.3 Newton Method

Description:

Input:

1.
2.
3.
4.

Output:

Code Written:

```
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
```

```
Console Output
--------------
```

3.4 Secant Method

Description:

Input:

1.
2.
3.
4.

Output:

Code Written:

```
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
```

```
Console Output
--------------
```

3.5 Hybrid Method

Description:

Input:

1.
2.
3.
4.

Output:

Code Written:

Usage Sample:

```
//Written by Andrew Sheridan in C++
```

```
Console Output
--------------
```

4 Matrix Operations

4.1 Direct Methods

Todo: Write up description for this section

4.1.1 Back Substitution

Description: Solves an upper triangular system of equations using back substitution.

Input:

1. Matrix A (Upper-triangular matrix. Note: Will not be reduced.)
2. Vector b (Right-Hand-Side vector.)

Output: Vector (the solution to the system of equations)

Returns NULL if the Matrix and Vector are of incorrect size.

Code Written:

```
/// Solves a set of linear equations using back substitution
/// Does not reduce matrix A
//A: The matrix to be reduced
// b: Right-Hand-Side
Vector BackSubstitution(Matrix A, Vector b) {
  if (A.GetRows() != b.GetSize()) return NULL;

  Vector x(b.GetSize());

  for(int i = b.GetSize() - 1; i >= 0; i--)
  {
    x[i] = b[i];
    for (int j = i + 1; j < b.GetSize(); j++) {
      x[i] -= A[i][j] * x[j];
    }
    x[i] /= A[i][i];
  }
  return x;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
int size = 4;
Matrix matrix = MatrixFactory::Instance()->UpperTriangular(size, size)
    ;

Vector vector = Vector(size);
vector.InitializeAllOnes();
vector = matrix * vector;
Vector resultVector = BackSubstitution(matrix, vector);

std::cout << "Upper triangular matrix" << std::endl;
matrix.Print();
std::cout << "Test vector " << std::endl;
vector.Print();
```

```
std::cout << "Result of back substition " << std::endl;
resultVector.Print();
```

```
Console Output
--------------
Upper triangular matrix
40.713        0.428471      0.690885      0.71915
0             40.4911       0.780028      0.410924
0             0             40.5797       0.139951
0             0             0             40.401

Test vector
42.5515       41.6821       40.7196       40.401

Result of back substition
1             1             1             1
```

4.1.2 Forward Substitution

Description: Solves a lower-triangular set of linear equations using forward substitution.

Input:

1. Matrix A (lower triangular system of equations)
2. Vector b (right-hand-side)

Output: Vector (solution to the system of equations)

Returns NULL if the Matrix and Vector are of incorrect size.

Code Written:

```
/// Solves a set of linear equations using forward substitution
/// Does not reduce matrix A
//A: The lower-triangular matrix
//b: right-hand-side
Vector ForwardSubstitution(Matrix A, Vector b) {
  if (A.GetRows() != b.GetSize()) return NULL;

  Vector x(b.GetSize());

  x[0] = b[0];
  for (unsigned i = 0; i < A.GetRows(); i++) {
    x[i] = b[i];
    for (unsigned j = 0; j < i; j++) {
      x[i] = x[i] - (A[i][j] * x[j]);
    }
    x[i] = x[i] / A[i][i];
  }

  return x;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
int size = 4;
Matrix matrix = MatrixFactory::Instance()->LowerTriangular(size, size)↩
    ;

Vector vector = Vector(size);
vector.InitializeAllOnes();
vector = matrix * vector;
Vector resultVector = ForwardSubstitution(matrix, vector);

std::cout << "Upper triangular matrix" << std::endl;
matrix.Print();
std::cout << "Test vector " << std::endl;
vector.Print();
std::cout << "Result of back substitution " << std::endl;
resultVector.Print();
```

```
Console Output
--------------

Upper triangular matrix
40.713      0           0           0
0.428471    40.6909     0           0
0.71915     0.491119    40.78       0
0.410924    0.579694    0.139951    40.401

Test vector
40.713      41.1194     41.9903     41.5316

Result of back substition
1           1           1           1
```

4.1.3 Gaussian Elimination

Description: Reduces a matrix to its upper-triangular form using Gaussian Elimination.

Input:

1. Matrix A (matrix to be reduced to upper-triangular form)
2. Vector b (right-hand-side, will be reduced along with the matrix)

Output: void

Code Written:

```
/// Reduces a matrix right-hand-side b to upper-triangular form using Gaussian Elimination
//A: The matrix to be reduced
// b: Right-Hand-Side
void GaussianElimination(Matrix& A, Vector& b) {
  for (unsigned k = 0; k < A.GetRows(); k++) {
    for (unsigned i = k + 1; i < A.GetRows(); i++) {
      double factor = A[i][k] / A[k][k];
      for (unsigned j = 0; j < A.GetColumns(); j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      A[i][k] = 0;
      b[i] = b[i] - factor*b[k];
    }
  }
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
int size = 4;

  Matrix matrix = MatrixFactory::Instance() -> DiagonallyDominant(size, size);
  Vector vector(size);
  vector.InitializeAllOnes();
  vector = matrix * vector;
  Matrix reducedMatrix = matrix;
  Vector resultVector = vector;
  GaussianElimination(reducedMatrix, resultVector);

  std::cout << "Gaussian Elimination" << std::endl;
  std::cout << "Test Matrix" << std::endl;
  matrix.Print();
  std::cout << "Test vector " << std::endl;
  vector.Print();
  std::cout << "Result of gaussian elimination " << std::endl;
  reducedMatrix.PrintAugmented(resultVector);
```

```
Console Output
--------------
Gaussian Elimination
Test Matrix
```

```
40.713      0.428471    0.690885    0.71915
0.491119    40.78       0.410924    0.579694
0.139951    0.401018    40.6273     0.324151
0.244759    0.694755    0.593902    40.6318

Test vector
42.5515     42.2618     41.4924     42.1652

Result of gaussian elimination
40.713      0.428471    0.690885    0.71915     | 42.5515
0           40.7749     0.40259     0.571019    | 41.7485
0           0           40.621      0.316084    | 40.9371
0           0           0           40.6132     | 40.6132
```

4.1.4 LU Factorization

Description: Finds the LU Factorization of a system of linear equations. L is lower-triangular, U, upper-triangular.

Input:

1. Matrix A (Coefficient Matrix)
2. Vector b (Right-Hand-Side)

Output: Matrix*
The first entry in this array is L, the second entry, U.

Code Written:

```
/// Finds the LU factorization of matrix A.
/// RHS b will be modified
// A: The nxn coefficient matrix
// b: Right-Hand-Side
Matrix* LUFactorization(Matrix A, Vector& b) {
  if (A.GetRows() != b.GetSize()) return NULL;

  Matrix L(A.GetRows(), A.GetColumns());
  L.InitializeIdentityMatrix();

  for (int k = 0; k < A.GetRows(); k++) {
    for (int i = k + 1; i < A.GetRows(); i++) {
      double factor = A[i][k] / A[k][k];
      L[i][k] = factor;
      for (int j = 0; j < A.GetColumns(); j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      b[i] = b[i] - factor*b[k];
    }
  }

  Matrix* LU = new Matrix[2]{ L, A };
  return LU;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
int size = 4;
Matrix matrix = MatrixFactory::Instance()->DiagonallyDominant(size, size);
Vector vector(size);
vector = matrix * vector;
Matrix* LU = LUFactorization(matrix, vector);
Matrix L = LU[0];
Matrix U = LU[1];

std::cout << "LU Factorization" << std::endl;
std::cout << "Starting system: " << std::endl;
matrix.Print();
std::cout << "Result of Scaled LU Factorization " << std::endl;
U.Print();
```

```
L.Print();
```

```
Console Output
--------------
LU Factorization
Starting system:
40.713        0.428471      0.690885      0.71915
0.491119      40.78         0.410924      0.579694
0.139951      0.401018      40.6273       0.324151
0.244759      0.694755      0.593902      40.6318

Result of Scaled LU Factorization
40.713        0.428471      0.690885      0.71915
0             40.7749       0.40259       0.571019
0             -5.55112e-17 40.621        0.316084
0             7.9659e-19    0             40.6132

1             0             0             0
0.012063      1             0             0
0.0034375     0.0097988     1             0
0.00601183    0.0169756     0.0143501     1
```

4.1.5 Cholesky Decomposition

Description: Computes the Cholesky Decomposition of a symmetric-positive-definite matrix.

Input:

1. Matrix A (Symmetric Positive Definite Matrix)

Output: Matrix
Returns NULL if the matrix is not SPD.

Code Written:

```cpp
///Computes the Cholesky Decomposition of an n by n matrix A
/// Returns NULL if the matrix is not SPD
Matrix CholeskyDecomposition(Matrix& A) {
  if (A.IsSymmetric() == false)
    return NULL;

  Matrix L(A.GetRows(), A.GetColumns()); //Initialize the new matrix

  for (unsigned i = 0; i < A.GetRows(); i++) {
    for (unsigned j = 0; j < (i + 1); j++) {
      double entry = 0;
      for (unsigned 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;
}
```

Usage Sample:

```cpp
//Written by Andrew Sheridan in C++
int size1 = 4;
Matrix matrix1 = MatrixFactory::Instance()->SPD(size1);
Matrix cholesky1 = CholeskyDecomposition(matrix1);
if (cholesky1 != NULL) {
  std::cout << "Our diagonally dominant test matrix and the result of computing the Cholesky Decomposition of the matrix." << std::endl;
  matrix1.Print();
  cholesky1.Print();

  Matrix transpose = cholesky1.Transpose();
  std::cout << "L Transpose" << std::endl;
  transpose.Print();
```

```
  std::cout << "Multiplying the Cholesky Decomposition by its ↩
      transpose." << std::endl;
  Matrix result = cholesky1 * transpose;
  result.Print();
}
```

```
Console Output
--------------
Our diagonally dominant test matrix and the result of computing the Cholesky Decomposition of the matr
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
```

4.2 Pivoting Strategies

There will be systems of equations whose pivots will cause the system to yield poor results. To deal with this we can use pivoting strategies to use rows or columns whose pivots are well conditioned for solving the system. In this subsection I have modified some existing direct methods to use pivoting.

4.2.1 Scaled Gaussian Elimination

Description: This is Gaussian Elimination with Scaled Partial Pivoting. When pivots cause the normal Gaussian Elimination routine to give poor results, this algorithm should be used.

Input:

1. Matrix A
2. Vector b

Output: Matrix
This is the upper triangular matrix which has been reduced via Gaussian Elimination. Will return null if the matrix and vector are of incorrect size.

Code Written:

```
/// Reduces an matrix A and right-hand-side b to upper-triangular form using Gaussian Elimination
// A: The coefficient matrix
// b: Right-Hand-Side
Matrix GaussianEliminationWithScaledPivoting(Matrix A, Vector& b) {
  if (A.GetRows() != b.GetSize()) return NULL;

  int n = b.GetSize();

  for (int k = 0; k < n; k++) {
    double* ratios = new double[n - k]; // New vector of size n - k to store the ratios
    for (int i = k; i < n; i++) {
      double rowMax = A[i].FindMaxMagnitudeStartingAt(k);
      ratios[i - k] = rowMax / A[i][k];
    }
    int newPivot = FindMaxIndex(ratios, n - k) + k; //Find the best row for this iteration

    Vector temp = A[k]; //
    A[k] = A[newPivot]; // Switch the current row with the best row for this iteration
    A[newPivot] = temp; //

    double tempEntry = b[k];
    b[k] = b[newPivot];
    b[newPivot] = tempEntry;

    for (int i = k + 1; i < n; i++) {
      double factor = A[i][k] / A[k][k];
      for (int j = 0; j < n; j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      b[i] = b[i] - (factor*b[k]);
    }
  }
```

```
  return A;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
int size = 4;

Matrix matrix = MatrixFactory::Instance() -> Random(size, size);
Vector vector(size);
vector.InitializeAllOnes();
vector = matrix * vector;
Vector resultVector = vector;
Matrix reducedMatrix = GaussianEliminationWithScaledPivoting(↩
    reducedMatrix, resultVector);

std::cout << "Gaussian Elimination With Scaled Pivoting" << std::endl;
std::cout << "Test Matrix" << std::endl;
matrix.Print();
std::cout << "Test vector " << std::endl;
vector.Print();
std::cout << "Result of gaussian elimination " << std::endl;
reducedMatrix.PrintAugmented(resultVector);
```

```
Console Output
--------------
Gaussian Elimination With Scaled Pivoting
Test Matrix
0.712955      0.428471      0.690885      0.71915
0.491119      0.780028      0.410924      0.579694
0.139951      0.401018      0.627317      0.324151
0.244759      0.694755      0.593902      0.631792

Test vector
2.55146       2.26177       1.49244       2.16521

Result of gaussian elimination
0.139951      0.401018      0.627317      0.324151      | 1
0             -0.00658268   -0.50321      0.0648859     | 1
0             0             120.911       -16.8459      | 1
0             0             0             -0.309529     | 1
```

4.2.2 Scaled LU Factorization

Description: This is a modified version of the LU Factorization method, which uses pivoting to ensure that poor pivots are not chosen.

Input:

1. Matrix A
2. Vector b

Output: Matrix*
An array of matrices. The first entry is the lower-diagonal matrix L, the second entry, the upper-diagonal matrix U.

Code Written:

```
/// Finds the LU Factorization of matrix A with RHS b. Returns a pair of matrices. The first is L, the second, U.
// A: The nxn coefficient matrix
// b: Right-Hand-Side
// n: The size of the matrices
Matrix* ScaledLUFactorization(Matrix A, Vector b) {
  if (A.GetRows() != b.GetSize()) return NULL;
  Matrix L(A.GetRows(), A.GetColumns());
  L.InitializeIdentityMatrix();

  for (unsigned k = 0; k < A.GetRows(); k++) {
    Vector ratios(A.GetRows() - k); // New vector of size A.GetRows() - k to store the ratios
    for (unsigned i = k; i < A.GetRows(); i++) {
      double rowMax = A[i].FindMaxMagnitudeStartingAt(k);
      ratios[i - k] = rowMax / A[i][k];
    }
    unsigned newPivot = ratios.FindMaxIndex() + k; //Find the best row for this iteration

    Vector temp = A[k]; //
    A[k] = A[newPivot];  // Switch the current row with the best row for this iteration
    A[newPivot] = temp;  //

    double tempEntry = b[k];
    b[k] = b[newPivot];
    b[newPivot] = tempEntry;

    for (unsigned i = k + 1; i < A.GetRows(); i++) {
      double factor = A[i][k] / A[k][k];
      L[i][k] = factor;
      for (unsigned j = k + 1; j < A.GetColumns(); j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      A[i][k] = 0;
      b[i] = b[i] - factor*b[k];
    }
  }

  return new Matrix[2]{ L, A };
}
```

Usage Sample:

```cpp
//Written by Andrew Sheridan in C++
int size = 4;

Matrix matrix = MatrixFactory::Instance() -> Random(size, size);
Vector vector(size);
vector.InitializeAllOnes();
Vector resultVector = matrix * vector;
Matrix* LU = ScaledLUFactorization(matrix, resultVector);
Matrix L = LU[0];
Matrix U = LU[1];

std::cout << "LU Factorization with pivoting" << std::endl;
std::cout << "Test Matrix" << std::endl;
matrix.Print();
std::cout << "Test vector " << std::endl;
vector.Print();
std::cout << "Lower-Diagonal Matrix L " << std::endl;
L.Print();
std::cout << "Upper-Diagonal Matrix U " << std::endl;
U.Print();
```

```
Console Output
--------------
Test Matrix
0.712955     0.428471     0.690885     0.71915
0.491119     0.780028     0.410924     0.579694
0.139951     0.401018     0.627317     0.324151
0.244759     0.694755     0.593902     0.631792

Test vector
1            1            1            1

Lower-Diagonal Matrix L
1            0            0            0
3.50923      1            0            0
5.09433      245.257      1            0
1.7489       95.2855      0.381754     1

Upper-Diagonal Matrix U
0.139951     0.401018     0.627317     0.324151
0            -0.00658268  -0.50321     0.0648859
0            0            120.911      -16.8459
0            0            0            -0.309529
```

4.3 Linear Least Squares Problems

This section deals with the algorithms used to solve Least Squares Problems. That is, solutions for the algebraic problem

$$min_x||b - Ax||_2$$

This is particularly useful in problems dealing with data fitting.

4.3.1 Least Squares

Description: This algorithm solves the equation

$$min_x||b - Ax||_2$$

for x using Cholesky Decomposition followed by forward and back substitution. Note that within this method we call three other methods, which are CholeskyDecomposition, ForwardSubstitution, and BackSubstitution. The details for these methods can be found earlier in this section.

Input:

1. Matrix A (system of equations)
2. Vector B (right-hand-side)

Output: Vector (solution to the system of equations)

Code Written:

```
/// A Least Squares algorithm via Normal Equations
/// Requires a matrix A and a vector b
Vector LeastSquares(Matrix A, Vector b) {
  Matrix AT = A.Transpose();
  Matrix B = AT * A;
  Vector y = AT * b;

  std::cout << "B: " << std::endl;
  B.Print();
  std::cout << "Y: " << std::endl;
  y.Print();

  Matrix G = CholeskyDecomposition(B);
  Vector z = ForwardSubstitution(G, y);
  Vector x = BackSubstitution(G.Transpose(), z);

  return x;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
matrix[0][0] = 1;
matrix[0][1] = 0;
matrix[0][2] = 1;
matrix[1][0] = 2;
matrix[1][1] = 3;
```

```
matrix[1][2] = 5;
matrix[2][0] = 5;
matrix[2][1] = 3;
matrix[2][2] = -2;
matrix[3][0] = 3;
matrix[3][1] = 5;
matrix[3][2] = 4;
matrix[4][0] = -1;
matrix[4][1] = 6;
matrix[4][2] = 3;
Vector vector = Vector(5);
vector[0] = 4;
vector[1] = -2;
vector[2] = 5;
vector[3] = -2;
vector[4] = 1;

Vector result = LeastSquares(matrix, vector);
result.Print();
```

```
Console Output
--------------
B:
40          30          10
30          79          47
10          47          55

Y:
18          5           -21

0.347226    0.399004    -0.785917
```

4.3.2 Gram Schmidt

Description: Computes the QR factorization of a matrix.

Input:

1. Matrix A

Output: Matrix* This is an array of Matrices with two entries. The first is Q, the second, R.

Code Written:

```
///Computes the QR factorization of Matrix A
///Returns a pair of matrices in an array. The first is Q, the second, R.
Matrix* GramSchmidt(Matrix A) {
  if (A.GetRows() != A.GetColumns()) return NULL;

  Matrix r(A.GetRows(), A.GetColumns());
  Matrix q(A.GetRows(), A.GetColumns());

  for (int k = 0; k < A.GetRows(); k++) {
    r[k][k] = 0;
    for (int i = 0; i < A.GetRows(); i++)
      r[k][k] = r[k][k] + A[i][k] * A[i][k];

    r[k][k] = sqrt(r[k][k]);

    for (int i = 0; i < A.GetRows(); i++)
      q[i][k] = A[i][k] / r[k][k];

    for (int j = k + 1; j < A.GetColumns(); j++) {
      r[k][j] = 0;
      for (int i = 0; i < A.GetRows(); i++)
        r[k][j] += q[i][k] * A[i][j];

      for (int i = 0; i < A.GetRows(); i++)
        A[i][j] = A[i][j] - r[k][j] * q[i][k];
    }
  }
  Matrix* QR = new Matrix[2];
  QR[0] = q;
  QR[1] = r;
  return QR;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
Matrix matrix = MatrixFactory::Instance()->DiagonallyDominant(5, 5);
Matrix* QR = GramSchmidt(matrix);
Matrix Q = QR[0];
Matrix R = QR[1];

std::cout << "Starting Matrix: " << std::endl;
matrix.Print();
std::cout << "Q: " << std::endl;
Q.Print();
std::cout << "R: " << std::endl;
```

```
R.Print();

std::cout << "Q * R" << std::endl;
Matrix TestResult = Q * R;
TestResult.Print();

Matrix Norm = matrix - TestResult;
double normValue = Norm.OneNorm();
std::cout << "||A - (Q * R)|| : " << normValue << std::endl;
```

```
Console Output
--------------
Starting Matrix:
50.713      0.428471    0.690885    0.71915     0.491119
0.780028    50.4109     0.579694    0.139951    0.401018
0.627317    0.324151    50.2448     0.694755    0.593902
0.631792    0.440257    0.0837265   50.7123     0.427863
0.29778     0.492085    0.740296    0.357729    50.4172

Q:
0.99971     -0.0156152  -0.0123673  -0.0123431  -0.00545983
0.0153768   0.999777    -0.00647483 -0.00868187 -0.00956081
0.0123664   0.00613281  0.999798    -0.00149318 -0.0145344
0.0124546   0.00843422  0.00118936  0.999863    -0.00674982
0.00587018  0.00962131  0.0144128   0.00657804  0.999811

R:
50.7276     1.21588     1.32633     1.36339     0.805775
0           50.4034     0.884747    0.564112    0.88559
0           0           50.2331     0.750286    1.31227
0           0           0           50.6966     0.749021
0           0           0           0           50.3896

Q * R
50.713      0.428471    0.690885    0.71915     0.491119
0.780028    50.4109     0.579694    0.139951    0.401018
0.627317    0.324151    50.2448     0.694755    0.593902
0.631792    0.440257    0.0837265   50.7123     0.427863
0.29778     0.492085    0.740296    0.357729    50.4172

||A - (Q * R)|| : 7.21645e-15
```

4.4 Iterative Methods

This section will approach the problem of solving linear systems of equation via iterative methods. This is beneficial because iterative methods can be less expensive, particularly when dealing with sparse systems of equations.

4.4.1 Jacobi Iteration

Description: Solves a system of equations using Jacobi Iteration

Input:

1. Matrix A
2. Vector x0 (initial guess)
3. Vector b (Right-Hand-Side)
4. int maxIterations
5. double tolerance

Output: Vector

Code Written:

```
///Solves the system of equations using Jacobi Iteration
///A: The Matrix
///x0: The initial guess
///b: The Right-Hand-Side
Vector JacobiIteration(Matrix A, Vector x0, Vector b, int ↩
    maxIterations, double tolerance) {
  int iterations = 0;
  int n = A.GetRows();
  Vector newX(x0);
  double error = 10 * tolerance;
  while(iterations < maxIterations && tolerance < error){
    Vector oldX(newX);
    for (int i = 0; i < n;  i++) {
      newX[i] = b[i];
      for (int j = 0; j < i; j++) {
        newX[i] = newX[i] - A[i][j] * oldX[j];
      }
      for (int j = i + 1; j < n; j++) {
        newX[i] = newX[i] - A[i][j] * oldX[j];
      }
      newX[i] = newX[i] / A[i][i];
      error = (oldX - newX).L2Norm();
      iterations++;
    }
  }
  return newX;
}
```

Usage Sample:

```cpp
//Written by Andrew Sheridan in C++
Matrix m1(3);
m1[0][0] = 7;
m1[0][1] = 3;
m1[0][2] = 1;
m1[1][0] = -3;
m1[1][1] = 10;
m1[1][2] = 2;
m1[2][0] = 1;
m1[2][1] = 7;
m1[2][2] = -15;

Vector b1(3);
b1[0] = 3;
b1[1] = 4;
b1[2] = 2;

Vector x1(3);

int maxIter = 10000;
double tolerance = 0.00001;

std::cout << "The results of Jacobi Iteration with max iterations of "
    << maxIter << " and tolerance of" << tolerance << std::endl;
Vector result1 = JacobiIteration(m1, x1, b1, maxIter, tolerance);
result1.Print();
```

```
Console Output
--------------
The results of Jacobi Iteration with max iterations of 10000 and tolerance of 1e-05
0.223242     0.448775     0.0909775
```

4.4.2 Conjugate Gradient Method

Description: Uses the iterative method of the Conjugate Gradient Method to solve system of linear equations starting with an initial guess.

Input:

1. Matrix A
2. Vector b (Right-Hand-Side)
3. Vector x0 (initial guess)
4. double tolerance

Output: Vector (the solution to the system of equations)

Code Written:

```
///Solves a matrix and RHS using Conjugate Gradient Method
///A : The matrix to be solved
///b : The RHS vector
///x0 : The initial guess vector
///tol : The tolerance of our method
Vector ConjugateGradient(Matrix A, Vector b, Vector x0, double tol) {
  Vector rk = b - (A * x0);
  double dk = rk * rk;
  double bd = b * b;
  int k = 0;
  Vector pk = rk;
  Vector xk = x0;
  while (dk > tol * tol * bd) {
    Vector sk = A * pk;
    double ak = dk / (pk * sk);
    Vector xkp1 = xk + (ak * pk);
    Vector rkp1 = rk - (ak * sk);
    double dkp1 = rkp1 * rkp1;
    Vector pkp1 = rkp1 + ((dkp1 / dk) * pk);
    k++;
    //All values have been computed, set all kth values to equal the k+1 value in preparation for next iteration
    xk = xkp1;
    rk = rkp1;
    pk = pkp1;
    dk = dkp1;
  }
  return xk;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++
Matrix m5(3);//Initialize 3x3 matrix
m5[0][0] = 7; //Assign values
m5[0][1] = 3;
m5[0][2] = 1;
m5[1][0] = 3;
m5[1][1] = 10;
```

```
m5[1][2] = 2;
m5[2][0] = 1;
m5[2][1] = 2;
m5[2][2] = 15;

Vector b5(3);
b5[0] = 28;
b5[1] = 31;
b5[2] = 22;

Vector x5(3);

Vector result5 = ConjugateGradient(m5, b5, x5, 0.000001);
std::cout << "Result of conjugate gradient method: " << std::endl;
result5.Print();
```

```
Console Output
------
Result of conjugate gradient method:--------
3           2           1
```

4.5 Eigenvalues and Singular Values

This section contains methods which aid in the approximation of Eigenvalues and Singular values.

4.5.1 PowerMethod

Description: Finds an approximation of the largest eigenvalue of a matrix.

Input:

1. Matrix A (the Matrix whose eigenvalue will be approximated)
2. Vector x0 (the initial guess)
3. double tol (the tolerance of the algorithm)
4. int maxIter (the maximum number of iterations)

Output: double (the approximation of the largest eigenvalue)

Code Written:

```
///Finds an approximation of the largest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the ↩
    method
double PowerMethod(Matrix A, Vector x0, double tol, int maxIter) {
  double error = 10 * tol;
  int k = 0;
  Vector y = A * x0;
  Vector xk = x0;
  double lambda_k = 0;

  while (error > tol && k < maxIter) {
    Vector xkp1 = y / y.L2Norm();
    y = A * xkp1;
    double lambda_kp1 = xkp1 * y;
    error = abs(lambda_kp1 - lambda_k);

    lambda_k = lambda_kp1;
    k++;
  }

  return lambda_k;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++

  Matrix A(2); //New 2x2 matrix
  A[0][0] = 2; //Initialize entries
  A[0][1] = -12;
  A[1][0] = 1;
  A[1][1] = -5;
```

```cpp

Vector x_0(2); //New vector size 2
x_0[0] = 1; //Initialize entries
x_0[1] = 1;

//Find the largest eigenvalue of A
double lambda = PowerMethod(A, x_0, 0.0001, 1000);
std::cout << "Lambda: " << lambda << std::endl;
```

```
Console Output
--------------
Lambda: -2.00005
```

4.5.2 InversePowerMethod

Description: Approximates the smallest eigenvalue of a matrix.

Input: Finds an approximation of the smallest Eigenvalue of a matrix

1. Matrix A (square matrix whose smallest eigenvalue will be approximated)
2. Vector x0 (initial guess vector)
3. double tol (error tolerance)
4. int maxIter (maximum number of iterations)

Output: double (approximation of the smallest eigenvalue)

Code Written:

```
///Finds an approximation of the smallest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the ↩
    method
double InversePowerMethod(Matrix A, Vector x0, double tol, int maxIter↩
    ) {

  int k = 0;
  Matrix U = A;
  Matrix L = LUFactorization(U, x0);// L is returned, U is modified to↩
       be upper triangular
  Vector y = BackSubstitution(U, x0);// Solve for y by doing back ↩
      substitution.
  double lambda_x = 0;
  while (error > tol && k < maxIter) {
    Vector x = y / y.L2Norm();
    y = SolveSystem(A, x); //Does Gaussian Elimination then Back ↩
        Substitution
    double lambda_xp1 = x * y;
    error = abs(lambda_xp1 - lambda_x);

    lambda_x = lambda_xp1;
    k++;
  }

  output.close();
  return lambda_x;
}
```

Usage Sample:

```
//Written by Andrew Sheridan in C++

  Matrix A(2);
  A[0][0] = 2;
  A[0][1] = -12;
  A[1][0] = 1;
  A[1][1] = -5;
```

```cpp

  Vector x_0(2);
  x_0[0] = 1;
  x_0[1] = 1;

  double lambda = InversePowerMethod(A, x_0, 0.000001, 1000);
  std::cout << "Lambda: " << lambda << std::endl;
```

```
Console Output
--------------
Lambda: -0.999999
```

5 Citations

6 Appendices

6.1 Appendix B: Vector Class

```
#pragma once
//Andrew Sheridan
//Math 5610
//Written in C++
//Vector.h

class Vector {
public:
  //Initialization and Destruction
  Vector();
  Vector(unsigned n);
  Vector(const Vector &v);
  Vector(double* v, unsigned size);
  Vector operator=(const Vector& v);
  ~Vector();

  void InitializeRandomEntries();
  void InitializeAllOnes();

  //Overloaded Operators
  double& operator[] (unsigned x) { return entries[x]; }
  friend double operator*( Vector& a, Vector& b);
  friend Vector operator*(Vector& a, double constant);
  friend Vector operator*(double constant, Vector& a);
  friend Vector operator+(Vector& a, Vector& b);
  friend Vector operator-(Vector& a, Vector& b);
  friend Vector operator/(Vector& a, double constant);
  friend Vector operator+(Vector& a, Vector& b);

  //Basic Algorithms
  double FindMaxMagnitudeStartingAt(unsigned start);
  unsigned FindMaxIndex();
  double L2Norm();

  //Accessing Data
  void Print();
  unsigned GetSize() { return size; }
  void SetSize(unsigned newSize) { size = newSize; }

protected:
  unsigned size;

private:
  double* entries; //The stored values of the vector
};
```

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

#pragma once
#include "Vector.h"
#include <random>
#include <iostream>
#include <iomanip>

///Default constructor
Vector::Vector() :size(0){ }

/// Initializes the entries to an empty array of size n
Vector::Vector(unsigned n) : size(n){
  entries = new double[size];
  for (unsigned i = 0; i < n; i++) {
    entries[i] = 0;
  }
}

///Copy Constructor
Vector::Vector(const Vector &v) : size(v.size) {
  entries = new double[size];
  for (unsigned i = 0; i < size; i++) {
    entries[i] = v.entries[i];
  }
}

///Copy Constructor
///v: an array of doubles
///n: the size of v
Vector::Vector(double* v, unsigned n) :size(n) {
  entries = new double[size];
  for (unsigned i = 0; i < size; i++) {
    entries[i] = v[i];
  }
}

/// Copy Assignment Operator
Vector Vector::operator= (const Vector& v) {
  size = v.size;
  entries = new double[v.size];
  for (unsigned i = 0; i < v.size; i++) {
    entries[i] = v.entries[i];
  }
  return *this;
}

///Destructor
Vector::~Vector()
{
}

///Inner Product
double operator* (Vector& a, Vector& b) {
  if (a.size != b.size)
    return NULL;
  double sum = 0;
  for (int i = 0; i < a.size; i++) {
    sum += a[i] * b[i];
  }
```

```
  return sum;
}

///Multiply each entry by a constant
Vector operator* (Vector& a, double constant) {
  Vector newVector(a.size);
  for (int i = 0; i < a.size; i++) {
    newVector[i] = a[i] * constant;
  }
  return newVector;
}

///Multiply each entry by a constant
Vector operator* (double constant, Vector& a) {
  Vector newVector(a.GetSize());
  for (int i = 0; i < a.GetSize(); i++) {
    newVector[i] = a[i] * constant;
  }
  return newVector;
}

///Subtracts the elements of vector a from the elements of vector b
Vector operator- (Vector& a, Vector& b) {
  if (a.size != b.size) return a;
  Vector newVector(a.size);
  for (int i = 0; i < a.size; i++) {
    newVector[i] = a[i] - b[i];
  }
  return newVector;
}

///Divides the entries of the vector by a constant
Vector operator/ (Vector& a, double constant) {
  Vector newVector(a.GetSize());
  for (int i = 0; i < a.GetSize(); i++) {
    newVector[i] = a[i] / constant;
  }
  return newVector;
}

///Returns a new vector where its entries are the sum of corresponding↩
     entries in a and b
Vector operator+ (Vector& a, Vector& b) {
  if (a.size != b.size) {
    std::cout << "Vectors are not same size. Returning first vector." ↩
        << std::endl;
    return a;
  }
  Vector newVector(a.size);
  for (int i = 0; i < a.size; i++) {
    newVector[i] = a[i] + b[i];
  }
  return newVector;
}

/// Initializes the entries to values between 0 and 1
void Vector::InitializeRandomEntries() {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↩
      distribution

  for (unsigned i = 0; i < size; i++) {
    entries[i] = dis(generator); //Assign each entry in entries to ↩
        random number between 0 and 1
```

```
  }
}

/// Initializes the entries to 1
void Vector::InitializeAllOnes() {
  for (unsigned i = 0; i < size; i++) {
    entries[i] = 1; //Assign each entry in entries to 1
  }
}

///Finds the entry with the largest magnitude, starting with entry "↩
    start".
double Vector::FindMaxMagnitudeStartingAt(unsigned start) {
  double max = 0;
  for (unsigned i = start; i < size; i++) {
    double value = std::abs(entries[i]);
    if (value > max) max = value;
  }
  return max;
}

///Finds the index of the value with the largest magnitude
unsigned Vector::FindMaxIndex() {
  double max = 0;
  unsigned index = -1;
  for (unsigned i = 0; i < size; i++) {
    double value = std::abs(entries[i]);
    if (value > max) {
      max = value;
      index = i;
    }
  }
  return index;
}

///Computes the L2 norm of the vector
double Vector::L2Norm() {
  double sum = 0;
  for (int i = 0; i < size; i++) {
    sum += (entries[i] * entries[i]);
  }
  return std::sqrt(sum);
}

///Outputs the vector's entries to the console
void Vector::Print() {
  for (unsigned i = 0; i < size; i++) {
    std::cout << std::setw(13) << std::left << entries[i];
  }
  std::cout << std::endl << std::endl;
}
```

6.2 Appendix C: Matrix Class

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
//Matrix.h

#pragma once
#include "Vector.h"

class Matrix{
public:
  //Initialization and Deconstruction
  Matrix() = default;
  Matrix(unsigned size);
  Matrix(unsigned rowCount, unsigned columnCount);
  Matrix(const Matrix &m);
  Matrix operator =(const Matrix& m);
  ~Matrix();

  void InitializeIdentityMatrix();
  void InitializeRandom();
  void InitializeRange(double minValue, double maxValue);
  void InitializeDiagonallyDominant();

  //Overloaded Operators
  Vector &operator[] (unsigned row) { return entries[row]; }
  friend bool operator == (const Matrix& A, const Matrix& B);
  friend bool operator != (const Matrix& A, const Matrix& B);
  friend Vector operator * (const Matrix& A, Vector& x);
  friend Vector operator / (const Matrix& A, Vector& x);
  friend Matrix operator * (Matrix A, Matrix B);
  friend Matrix operator - (Matrix A, Matrix B);

  //Basic Algorithms
  bool IsSymmetric();
  Matrix Transpose();
  double OneNorm();
  double InfinityNorm();

  //Getters and Setters
  unsigned GetRows() { return rows; }
  unsigned GetColumns() { return columns; }
  void SetRows(unsigned r) { rows = r; }
  void SetColumns(unsigned c) { columns = c; }

  //Output
  void Print();
  void PrintAugmented(Vector v);

private:
  Vector* entries; //The entries of the matrix

  unsigned rows;
  unsigned columns;
};
```

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

#include "Matrix.h"
#include "Vector.h"
#include <random>
#include <iostream>
#include <iomanip>


#pragma region Constructors and Initialization
///Initializes an nxn matrix of size "size". All entries are zeroes.
Matrix::Matrix(unsigned size) : rows(size), columns(size)
{
  entries = new Vector[size];
  for (unsigned i = 0; i < size; i++) {
    entries[i] = Vector(size);
    for (unsigned j = 0; j < size; j++) {
      entries[i][j] = 0;
    }
  }
}

///Initializes an nxm matrix. All entries are zeroes.
Matrix::Matrix(unsigned rowCount, unsigned columnCount) : rows(↩
    rowCount), columns(columnCount) {
  entries = new Vector[rowCount];
  for (unsigned i = 0; i < rowCount; i++) {
    entries[i] = Vector(columnCount);
    for (unsigned j = 0; j < columnCount; j++) {
      entries[i][j] = 0;
    }
  }
}

///Copy Constructor
Matrix::Matrix(const Matrix &m) : rows(m.rows), columns(m.columns) {
  entries = new Vector[rows];
  for (unsigned int i = 0; i < rows; i++) {
    entries[i] = Vector(columns);
    for (unsigned int j = 0; j < columns; j++) {
      entries[i][j] = m.entries[i][j];
    }
  }
}

//Deconstructor
Matrix::~Matrix() {
}

///Assignment operator overload
Matrix Matrix::operator=(const Matrix& m) {
  this->rows = m.rows;
  this->columns = m.columns;
  this->entries = new Vector[m.rows];
  for (int i = 0; i < m.rows; i++) {
    this->entries[i] = m.entries[i];
  }

  return *this;
}
```

```
///Initializes the matrix to have ones on the main diagonal
void Matrix::InitializeIdentityMatrix() {
  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = 0; j < i; j++) {
      entries[i][j] = 0;
    }
    entries[i][i] = 1;
    for (unsigned j = i + 1; j < rows; j++) {
      entries[i][j] = 0;
    }
  }
}

///Sets the values of all entries between 0 and 1
void Matrix::InitializeRandom() {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↩
      distribution

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

///Sets the values of all entries between minValue and maxValue
void Matrix::InitializeRange(double minValue, double maxValue) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(minValue, maxValue); //↩
      Desired distribution

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

/// Sets the values of the entries to be diagonally dominant
void Matrix::InitializeDiagonallyDominant() {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↩
      distribution

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

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

#pragma endregion

#pragma region Printing
```

```cpp
///Outputs matrix to the console
void Matrix::Print() {
  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = 0; j < columns; j++) {
      std::cout << std::setw(13) << std::left << entries[i][j];
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}

///Outputs an augmented coefficient matrix to the console
void Matrix::PrintAugmented(Vector v) {
  if (v.GetSize()  != rows) {
    std::cout << "Vector and Matrix do not match sizes" << std::endl;
    return;
  }

  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = 0; j < columns; j++) {
      std::cout << std::setw(13) << std::left << entries[i][j];
    }
    std::cout << "| " << v[i] << std::endl;
  }
  std::cout << std::endl;
}
#pragma endregion

#pragma region Comparison Operations
///Compares two matrices. Returns true if all corresponding entries ↩
    are equal, and false if any are not equal.
bool operator == (const Matrix& A, const Matrix& B) {
  if (A.columns != B.columns) return false;
  if (A.rows != B.rows) return false;

  for (unsigned i = 0; i < A.rows; i++) {
    for (unsigned j = 0; j < A.columns; j++) {
      if (A.entries[i][j] != B.entries[i][j]) {
        return false;
      }
    }
  }
  return true;
}

///Compares two matrices. Returns true if any corresponding entries ↩
    are not equal, and false if all are equal.
bool operator != (const Matrix& A, const Matrix& B) {
  if (A.columns != B.columns) return true;
  if (A.rows != B.rows) return true;

  for (unsigned i = 0; i < A.rows; i++) {
    for (unsigned j = 0; j < A.columns; j++) {
      if (A.entries[i][j] != B.entries[i][j]) {
        return true;
      }
    }
  }
  return false;
}

///Checks to see if matrix is symmetric
bool Matrix::IsSymmetric() {
  for (unsigned int i = 0; i < rows; i++) {
```

```cpp
    for (unsigned int j = 0; j <= i; j++) {
      if (entries[i][j] != entries[j][i])
        return false;
    }
  }
  return true;
}

#pragma endregion


#pragma region operations
/// Returns the transpose of the n by n matrix A
Matrix Matrix::Transpose() {
  Matrix matrix(columns, rows);

  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = 0; j < columns; j++) {
      matrix[j][i] = entries[i][j];
    }
  }
  return matrix;
}

///Multiplies an nxn matrix A by the vector x
Vector operator *(const Matrix& A, Vector& x) {
  if (A.columns != x.GetSize()) return NULL;

  Vector result(A.rows);
  for (unsigned i = 0; i < A.rows; i++) {
    result[i] = 0;
    for (unsigned j = 0; j < A.columns; j++) {
      result[i] += A.entries[i][j] * x[j];
    }
  }
  return result;
}

///Divides an nxn matrix A by the vector x
Vector operator /(const Matrix& A, Vector& x) {
  if (A.columns != x.GetSize()) return NULL;

  Vector result(A.rows);
  for (unsigned i = 0; i < A.rows; i++) {
    result[i] = 0;
    for (unsigned j = 0; j < A.columns; j++) {
      result[i] += A.entries[i][j] / x[j];
    }
  }
  return result;
}


///Multiplies an matrix A by matrix B
Matrix operator* (Matrix A, Matrix B) {
  if (A.columns != B.rows) throw "Incompatible sizes";

  Matrix matrix(A.rows, B.columns);
  Matrix bTranspose = B.Transpose();

  //std::cout << "Starting multiplication. A:" << std::endl;
  /*A.Print();
  std::cout << "B: " << std::endl;
  B.Print();
```

```
  std::cout << "Bt: " << std::endl;
  bTranspose.Print();*/

  for (unsigned i = 0; i < A.rows; i++) {
    for (unsigned j = 0; j < B.columns; j++) {
      //matrix[i][j] = DotProduct(A[i], bTranspose[j], A.columns);
      matrix[i][j] = A[i] * bTranspose[j];
    }
  }
  return matrix;
}

///Multiplies an matrix A by matrix B
Matrix operator- (Matrix A, Matrix B) {
  if (A.columns != B.columns && A.rows != B.rows) throw "Incompatible sizes";
  Matrix matrix(A.rows, B.columns);
  for (unsigned i = 0; i < A.rows; i++) {
    for (unsigned j = 0; j < A.columns; j++) {
      matrix[i][j] = A[i][j] - B[i][j];
    }
  }
  return matrix;
}

///Computes the 1-norm of the matrix
double Matrix::OneNorm() {
  double columnMax = 0;
  for (unsigned i = 0; i < rows; i++) {
    double columnSum = 0;
    for (unsigned j = 0; j < columns; j++) {
      columnSum += std::abs(entries[i][j]);
    }
    if (columnSum > columnMax)
      columnMax = columnSum;
  }
  return columnMax;
}

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

#pragma endregion
```

6.3 Appendix D: Matrix Factory

This component is used to create matrices of various types, such as symmetric or diagonally dominant matrices. It was created in order to simplify the matrix.cpp file, and to keep the initialization logic separate from the operation logic.

6.3.1 Usage:

Because this component is a singleton, the syntax to use it is more complex, but useful because a new matrix can be created in a single line of your main code. If we wanted to create a new identity matrix, the code would look like this.

```
Matrix A = MatrixFactory::Instance() -> Identity(rowCount, columnCount);
```

This way, there need not be any declaration or construction of an instance of the Matrix Factory. We can just ask the singleton for an instance, and use that instance to product the specified matrix, all in one line.

6.3.2 Header File

```
//Andrew Sheridan
//Math 5610
//Written in C++
//MatrixFactory.h
#pragma once
#include "Matrix.h"

///A singleton which creates new matrices
class MatrixFactory {
public:
  static MatrixFactory* Instance();
  Matrix Identity(unsigned rows, unsigned columns);
  Matrix Ones(unsigned rows, unsigned columns);
  Matrix UpperTriangular(unsigned rows, unsigned columns);
  Matrix LowerTriangular(unsigned rows, unsigned columns);
  Matrix Random(unsigned rows, unsigned columns);
  Matrix RandomRange(unsigned rows, unsigned columns, double min, double max);
  Matrix DiagonallyDominant(unsigned rows, unsigned columns);
  Matrix Symmetric(unsigned size);
  Matrix SPD(unsigned size);

private:
  MatrixFactory() {};
  MatrixFactory(MatrixFactory const&) {};
  MatrixFactory& operator=(MatrixFactory const&) {};
  static MatrixFactory* m_instance;
};
```

6.3.3 Code Written

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

#include "MatrixFactory.h"
#include <random>

MatrixFactory* MatrixFactory::m_instance = nullptr;

///Returns the instance of the MatrixFactory singleton
MatrixFactory* MatrixFactory::Instance() {
  if (!m_instance)
    m_instance = new MatrixFactory();

  return m_instance;
}

///Creates an identity matrix
Matrix MatrixFactory::Identity(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeIdentityMatrix();
  return m;
}

///Creates a matrix where every entry is a 1
Matrix MatrixFactory::Ones(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < columns; j++) {
      m[i][j] = 1;
    }
  }
  return m;
}

///Creates a matrix where every entry is a value between 0 and 1
Matrix MatrixFactory::Random(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeRandom();
  return m;
}

///Creates a matrix where every entry is a value between minValue and maxValue
Matrix MatrixFactory::RandomRange(unsigned rows, unsigned columns, double minValue, double maxValue) {
  Matrix m(rows, columns);
  m.InitializeRange(minValue, maxValue);
  return m;
}

///Creates an upper-triangular matrix
Matrix MatrixFactory::UpperTriangular(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

```

```
  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = i; j < columns; j++) {
      m[i][j] = dis(generator); //Assign entry in matrix to random number between 0 and 1
    }
    m[i][i] += 10 * rows; //Add 10*n to all diagonal entries
  }

  return m;
}

///Creates an upper-triangular matrix
Matrix MatrixFactory::LowerTriangular(unsigned rows, unsigned columns){
  Matrix m(rows, columns);
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  for (unsigned i = 0; i < rows; i++) {
    for (unsigned j = 0; j <= i; j++) {
      m[i][j] = dis(generator); //Assign entry in matrix to random number between 0 and 1
    }
    m[i][i] += 10 * rows; //Add 10*n to all diagonal entries
  }

  return m;
}

///Creates a Diagonally Dominant matrix
Matrix MatrixFactory::DiagonallyDominant(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeDiagonallyDominant();
  return m;
}

///Creates a symmetric matrix, where all entries have values between 0 and 1
Matrix MatrixFactory::Symmetric(unsigned size) {
  Matrix m(size);
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

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

  return m;
}

///Creates a symmetric positive definite matrix
Matrix MatrixFactory::SPD(unsigned size) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

```

```
  Matrix matrix(size);

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

  return matrix;
}
```

6.4 Appendix E: Complex Numbers

TODO

6.4.1 Usage:

Because this component is a singleton, the syntax to use it is more complex, but useful because a new matrix can be created in a single line of your main code. If we wanted to create a new identity matrix, the code would look like this.

```
Matrix A = MatrixFactory::Instance() -> Identity(rowCount, columnCount↩
    );
```

This way, there need not be any declaration or construction of an instance of the Matrix Factory. We can just ask the singleton for an instance, and use that instance to product the specified matrix, all in one line.

6.4.2 Header File

```
#pragma once
#include<cmath>

class Complex
{
public:
  double real;
  double imaginary;

  Complex(double newReal, double newImaginary)
  {
    real = newReal;
    imaginary = newImaginary;
  }

  Complex operator+(Complex c) {
    double realResult = real + c.real;
    double imaginaryResult = imaginary + c.imaginary;
    Complex result(realResult, imaginaryResult);
    return result;
  }

  Complex operator-(Complex c) {
    double realResult = real - c.real;
    double imaginaryResult = imaginary - c.imaginary;
    Complex result(realResult, imaginaryResult);
    return result;
  }

  Complex operator*(Complex c) {
    double realResult = real * c.real - (imaginary * c.imaginary);
    double imaginaryResult = real * c.imaginary + c.real * imaginary;
    Complex result(realResult, imaginaryResult);
    return result;
  }

  Complex operator/(Complex c) {
    double realNumerator = real*c.real - imaginary*c.imaginary;
    double imaginaryNumerator = imaginary*c.real - real*c.imaginary;
    double denominator = (c.real*c.real) - (c.imaginary*c.imaginary);
```

```
    return Complex(realNumerator / denominator, imaginaryNumerator / denominator);
  }
};

double complexAbsolute(Complex a) {
  return std::sqrt(std::pow(a.real, 2) + std::pow(a.imaginary, 2));
}
```