

Software Manual for Math 5610

Andrew Sheridan

December 16th, 2016 *

*Last Revised: December 9, 2016

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 4 |
| 2 | Precision and Error | 5 |
| 2.1 | Error | 5 |
| 2.1.1 | Absolute Error | 5 |
| 2.1.2 | Relative Error | 5 |
| 2.1.3 | Absolute Error in Complex Numbers | 5 |
| 2.1.4 | Relative Error in Complex Numbers | 5 |
| 2.2 | Rounding | 6 |
| 2.3 | Machine Precision | 7 |
| 2.3.1 | Single Precision | 7 |
| 2.3.2 | Double Precision | 7 |
| 2.4 | Vector Norms | 8 |
| 2.4.1 | One Norm/Manhattan Norm | 8 |
| 2.4.2 | Infinity Norm | 8 |
| 2.4.3 | L2 Norm | 8 |
| 2.5 | Matrix Norms | 8 |
| 2.5.1 | One Norm | 8 |
| 2.5.2 | Infinity Norm | 8 |
| 3 | Nonlinear Equations In One Variable | 9 |
| 3.1 | Bisection | 9 |
| 3.2 | Fixed Iteration | 10 |
| 3.3 | Newton Method | 11 |
| 3.4 | Secant Method | 12 |
| 3.5 | Hybrid Method | 13 |
| 4 | Matrix Operations | 14 |
| 4.1 | Direct Methods | 14 |
| 4.1.1 | Back Substitution | 14 |
| 4.1.2 | Forward Substitution | 16 |
| 4.1.3 | Gaussian Elimination | 18 |
| 4.1.4 | LU Factorization | 20 |
| 4.1.5 | Cholesky Decomposition | 22 |
| 4.2 | Pivoting Strategies | 24 |
| 4.2.1 | Scaled Gaussian Elimination | 24 |
| 4.2.2 | Scaled LU Factorization | 26 |
| 4.3 | Linear Least Squares Problems | 28 |
| 4.3.1 | Least Squares | 28 |
| 4.3.2 | Gram Schmidt | 30 |
| 4.4 | Iterative Methods | 32 |
| 4.4.1 | Jacobi Iteration | 32 |
| 4.4.2 | Conjugate Gradient Method | 34 |
| 4.5 | Eigenvalues and Singular Values | 36 |
| 4.5.1 | PowerMethod | 36 |
| 4.5.2 | InversePowerMethod | 38 |
| 5 | Citations | 40 |

| | | |
|----------|---------------------------------------|-----------|
| 6 | Appendices | 41 |
| 6.1 | Appendix B: Vector Class | 41 |
| 6.2 | Appendix C: Matrix Class | 45 |
| 6.3 | Appendix D: Matrix Factory | 51 |
| 6.3.1 | Usage: | 51 |
| 6.3.2 | Header File | 51 |
| 6.3.3 | Code Written | 52 |
| 6.4 | Appendix E: Complex Numbers | 55 |
| 6.4.1 | Usage: | 55 |
| 6.4.2 | Header File | 55 |

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:

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

Console Output

3.2 Fixed Iteration

Description:

Input:

- 1.
- 2.
- 3.
- 4.

Output:

Code Written:

Usage Sample:

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

Console Output

3.3 Newton Method

Description:

Input:

- 1.
- 2.
- 3.
- 4.

Output:

Code Written:

Usage Sample:

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

Console Output

3.4 Secant Method

Description:

Input:

- 1.
- 2.
- 3.
- 4.

Output:

Code Written:

Usage Sample:

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

Console Output

3.5 Hybrid Method

Description:

Input:

- 1.
- 2.
- 3.
- 4.

Output:

Code Written:

Usage Sample:

```
1 //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:

```
1  /// Solves a set of linear equations using back substitution
2  /// Does not reduce matrix A
3  /// A: The matrix to be reduced
4  /// b: Right-Hand-Side
5  Vector BackSubstitution(Matrix A, Vector b) {
6      if (A.GetRows() != b.GetSize()) return NULL;
7
8      Vector x(b.GetSize());
9
10     for(int i = b.GetSize() - 1; i >= 0; i--)
11     {
12         x[i] = b[i];
13         for (int j = i + 1; j < b.GetSize(); j++) {
14             x[i] -= A[i][j] * x[j];
15         }
16         x[i] /= A[i][i];
17     }
18     return x;
19 }
```

Usage Sample:

```
1  //Written by Andrew Sheridan in C++
2  int size = 4;
3  Matrix matrix = MatrixFactory::Instance()->UpperTriangular(size, size)↵
4      ;
5  Vector vector = Vector(size);
6  vector.InitializeAllOnes();
7  vector = matrix * vector;
8  Vector resultVector = BackSubstitution(matrix, vector);
9
10 std::cout << "Upper triangular matrix" << std::endl;
11 matrix.Print();
12 std::cout << "Test vector " << std::endl;
13 vector.Print();
```

```
14 std::cout << "Result of back substitution " << std::endl;
15 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 substitution

| | | | |
|---|---|---|---|
| 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:

```
1  /// Solves a set of linear equations using forward substitution
2  /// Does not reduce matrix A
3  ///A: The lower-triangular matrix
4  ///b: right-hand-side
5  Vector ForwardSubstitution(Matrix A, Vector b) {
6      if (A.GetRows() != b.GetSize()) return NULL;
7
8      Vector x(b.GetSize());
9
10     x[0] = b[0];
11     for (unsigned i = 0; i < A.GetRows(); i++) {
12         x[i] = b[i];
13         for (unsigned j = 0; j < i; j++) {
14             x[i] = x[i] - (A[i][j] * x[j]);
15         }
16         x[i] = x[i] / A[i][i];
17     }
18
19     return x;
20 }
```

Usage Sample:

```
1  //Written by Andrew Sheridan in C++
2  int size = 4;
3  Matrix matrix = MatrixFactory::Instance()->LowerTriangular(size, size)↵
4      ;
5  Vector vector = Vector(size);
6  vector.InitializeAllOnes();
7  vector = matrix * vector;
8  Vector resultVector = ForwardSubstitution(matrix, vector);
9
10 std::cout << "Upper triangular matrix" << std::endl;
11 matrix.Print();
12 std::cout << "Test vector " << std::endl;
13 vector.Print();
14 std::cout << "Result of back substitution " << std::endl;
15 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 substitution

| | | | |
|---|---|---|---|
| 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:

```
1  /// Reduces a matrix right-hand-side b to upper-triangular form using ↔
    Gaussian Elimination
2  //A: The matrix to be reduced
3  // b: Right-Hand-Side
4  void GaussianElimination(Matrix& A, Vector& b) {
5      for (unsigned k = 0; k < A.GetRows(); k++) {
6          for (unsigned i = k + 1; i < A.GetRows(); i++) {
7              double factor = A[i][k] / A[k][k];
8              for (unsigned j = 0; j < A.GetColumns(); j++) {
9                  A[i][j] = A[i][j] - factor*A[k][j];
10             }
11             A[i][k] = 0;
12             b[i] = b[i] - factor*b[k];
13         }
14     }
15 }
```

Usage Sample:

```
1  //Written by Andrew Sheridan in C++
2  int size = 4;
3
4  Matrix matrix = MatrixFactory::Instance() -> DiagonallyDominant(size↔
    , size);
5  Vector vector(size);
6  vector.InitializeAllOnes();
7  vector = matrix * vector;
8  Matrix reducedMatrix = matrix;
9  Vector resultVector = vector;
10 GaussianElimination(reducedMatrix, resultVector);
11
12 std::cout << "Gaussian Elimination" << std::endl;
13 std::cout << "Test Matrix" << std::endl;
14 matrix.Print();
15 std::cout << "Test vector " << std::endl;
16 vector.Print();
17 std::cout << "Result of gaussian elimination " << std::endl;
18 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:

```
1  /// Finds the LU factorization of matrix A.
2  /// RHS b will be modified
3  /// A: The nxn coefficient matrix
4  /// b: Right-Hand-Side
5  Matrix* LUFactorization(Matrix A, Vector& b) {
6      if (A.GetRows() != b.GetSize()) return NULL;
7
8      Matrix L(A.GetRows(), A.GetColumns());
9      L.InitializeIdentityMatrix();
10
11     for (int k = 0; k < A.GetRows(); k++) {
12         for (int i = k + 1; i < A.GetRows(); i++) {
13             double factor = A[i][k] / A[k][k];
14             L[i][k] = factor;
15             for (int j = 0; j < A.GetColumns(); j++) {
16                 A[i][j] = A[i][j] - factor*A[k][j];
17             }
18             b[i] = b[i] - factor*b[k];
19         }
20     }
21
22     Matrix* LU = new Matrix[2]{ L, A };
23     return LU;
24 }
```

Usage Sample:

```
1  //Written by Andrew Sheridan in C++
2  int size = 4;
3  Matrix matrix = MatrixFactory::Instance()->DiagonallyDominant(size, ←
    size);
4  Vector vector(size);
5  vector = matrix * vector;
6  Matrix* LU = LUFactorization(matrix, vector);
7  Matrix L = LU[0];
8  Matrix U = LU[1];
9
10 std::cout << "LU Factorization" << std::endl;
11 std::cout << "Starting system: " << std::endl;
12 matrix.Print();
13 std::cout << "Result of Scaled LU Factorization " << std::endl;
14 U.Print();
```

15 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:

```
1  ///Computes the Cholesky Decomposition of an n by n matrix A
2  /// Returns NULL if the matrix is not SPD
3  Matrix CholeskyDecomposition(Matrix& A) {
4      if (A.IsSymmetric() == false)
5          return NULL;
6
7      Matrix L(A.GetRows(), A.GetColumns()); //Initialize the new matrix
8
9      for (unsigned i = 0; i < A.GetRows(); i++) {
10         for (unsigned j = 0; j < (i + 1); j++) {
11             double entry = 0;
12             for (unsigned k = 0; k < j; k++) {
13                 entry += L[i][k] * L[j][k];
14             }
15             double sqrtValue = A[i][i] - entry;
16             if (sqrtValue < 0)
17                 return NULL;
18
19             // Conditional assignment. If the entry is diagonal, assign to ←
20             // the square root of the previous value.
21             // Otherwise, Do computation for a nondiagonal entry.
22             L[i][j] = i == j ? std::sqrt(sqrtValue) : (1.0 / L[j][j] * (A[i]←
23                 ][j] - entry));
24         }
25     }
26     return L;
27 }
```

Usage Sample:

```
1  ///Written by Andrew Sheridan in C++
2  int size1 = 4;
3  Matrix matrix1 = MatrixFactory::Instance()->SPD(size1);
4  Matrix cholesky1 = CholeskyDecomposition(matrix1);
5  if (cholesky1 != NULL) {
6      std::cout << "Our diagonally dominant test matrix and the result of ←
7      computing the Cholesky Decomposition of the matrix." << std::←
8      endl;
9      matrix1.Print();
10     cholesky1.Print();
11
12     Matrix transpose = cholesky1.Transpose();
13     std::cout << "L Transpose" << std::endl;
14     transpose.Print();
15 }
```

```

13  std::cout << "Multiplying the Cholesky Decomposition by its ↵
    transpose." << std::endl;
14  Matrix result = cholesky1 * transpose;
15  result.Print();
16 }

```

Console Output

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 |

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:

```
1  /// Reduces an matrix A and right-hand-side b to upper-triangular form↔
    using Gaussian Elimination
2  // A: The coefficient matrix
3  // b: Right-Hand-Side
4  Matrix GaussianEliminationWithScaledPivoting(Matrix A, Vector& b) {
5      if (A.GetRows() != b.GetSize()) return NULL;
6
7      int n = b.GetSize();
8
9      for (int k = 0; k < n; k++) {
10         double* ratios = new double[n - k]; // New vector of size n - k to↔
            store the ratios
11         for (int i = k; i < n; i++) {
12             double rowMax = A[i].FindMaxMagnitudeStartingAt(k);
13             ratios[i - k] = rowMax / A[i][k];
14         }
15         int newPivot = FindMaxIndex(ratios, n - k) + k; //Find the best ↔
            row for this iteration
16
17         Vector temp = A[k]; //
18         A[k] = A[newPivot]; // Switch the current row with the best row ↔
            for this iteration
19         A[newPivot] = temp; //
20
21         double tempEntry = b[k];
22         b[k] = b[newPivot];
23         b[newPivot] = tempEntry;
24
25         for (int i = k + 1; i < n; i++) {
26             double factor = A[i][k] / A[k][k];
27             for (int j = 0; j < n; j++) {
28                 A[i][j] = A[i][j] - factor*A[k][j];
29             }
30             b[i] = b[i] - (factor*b[k]);
31         }
32     }
```



```

33     return A;
34 }

```

Usage Sample:

```

1  //Written by Andrew Sheridan in C++
2  int size = 4;
3
4  Matrix matrix = MatrixFactory::Instance() -> Random(size, size);
5  Vector vector(size);
6  vector.InitializeAllOnes();
7  vector = matrix * vector;
8  Vector resultVector = vector;
9  Matrix reducedMatrix = GaussianEliminationWithScaledPivoting(←
    reducedMatrix, resultVector);
10
11 std::cout << "Gaussian Elimination With Scaled Pivoting" << std::endl;
12 std::cout << "Test Matrix" << std::endl;
13 matrix.Print();
14 std::cout << "Test vector " << std::endl;
15 vector.Print();
16 std::cout << "Result of gaussian elimination " << std::endl;
17 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:

```
1  /// Finds the LU Factorization of matrix A with RHS b. Returns a pair ↔
   of matrices. The first is L, the second, U.
2  // A: The nxn coefficient matrix
3  // b: Right-Hand-Side
4  // n: The size of the matrices
5  Matrix* ScaledLUFactorization(Matrix A, Vector b) {
6      if (A.GetRows() != b.GetSize()) return NULL;
7      Matrix L(A.GetRows(), A.GetColumns());
8      L.InitializeIdentityMatrix();
9
10     for (unsigned k = 0; k < A.GetRows(); k++) {
11         Vector ratios(A.GetRows() - k); // New vector of size A.GetRows() ↔
            - k to store the ratios
12         for (unsigned i = k; i < A.GetRows(); i++) {
13             double rowMax = A[i].FindMaxMagnitudeStartingAt(k);
14             ratios[i - k] = rowMax / A[i][k];
15         }
16         unsigned newPivot = ratios.FindMaxIndex() + k; //Find the best row↔
            for this iteration
17
18         Vector temp = A[k]; //
19         A[k] = A[newPivot]; // Switch the current row with the best row ↔
            for this iteration
20         A[newPivot] = temp; //
21
22         double tempEntry = b[k];
23         b[k] = b[newPivot];
24         b[newPivot] = tempEntry;
25
26         for (unsigned i = k + 1; i < A.GetRows(); i++) {
27             double factor = A[i][k] / A[k][k];
28             L[i][k] = factor;
29             for (unsigned j = k + 1; j < A.GetColumns(); j++) {
30                 A[i][j] = A[i][j] - factor*A[k][j];
31             }
32             A[i][k] = 0;
33             b[i] = b[i] - factor*b[k];
34         }
35     }
36
37     return new Matrix[2]{ L, A };
38 }
```

Usage Sample:

```
1 //Written by Andrew Sheridan in C++
2 int size = 4;
3
4 Matrix matrix = MatrixFactory::Instance() -> Random(size, size);
5 Vector vector(size);
6 vector.InitializeAllOnes();
7 Vector resultVector = matrix * vector;
8 Matrix* LU = ScaledLUFactorization(matrix, resultVector);
9 Matrix L = LU[0];
10 Matrix U = LU[1];
11
12 std::cout << "LU Factorization with pivoting" << std::endl;
13 std::cout << "Test Matrix" << std::endl;
14 matrix.Print();
15 std::cout << "Test vector " << std::endl;
16 vector.Print();
17 std::cout << "Lower-Diagonal Matrix L " << std::endl;
18 L.Print();
19 std::cout << "Upper-Diagonal Matrix U " << std::endl;
20 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:

```
1  /// A Least Squares algorithm via Normal Equations
2  /// Requires a matrix A and a vector b
3  Vector LeastSquares(Matrix A, Vector b) {
4      Matrix AT = A.Transpose();
5      Matrix B = AT * A;
6      Vector y = AT * b;
7
8      std::cout << "B: " << std::endl;
9      B.Print();
10     std::cout << "Y: " << std::endl;
11     y.Print();
12
13     Matrix G = CholeskyDecomposition(B);
14     Vector z = ForwardSubstitution(G, y);
15     Vector x = BackSubstitution(G.Transpose(), z);
16
17     return x;
18 }
```

Usage Sample:

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

```

7 matrix[1][2] = 5;
8 matrix[2][0] = 5;
9 matrix[2][1] = 3;
10 matrix[2][2] = -2;
11 matrix[3][0] = 3;
12 matrix[3][1] = 5;
13 matrix[3][2] = 4;
14 matrix[4][0] = -1;
15 matrix[4][1] = 6;
16 matrix[4][2] = 3;
17 Vector vector = Vector(5);
18 vector[0] = 4;
19 vector[1] = -2;
20 vector[2] = 5;
21 vector[3] = -2;
22 vector[4] = 1;
23
24 Vector result = LeastSquares(matrix, vector);
25 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:

```
1  ///Computes the QR factorization of Matrix A
2  ///Returns a pair of matrices in an array. The first is Q, the second,↵
   R.
3  Matrix* GramSchmidt(Matrix A) {
4      if (A.GetRows() != A.GetColumns()) return NULL;
5
6      Matrix r(A.GetRows(), A.GetColumns());
7      Matrix q(A.GetRows(), A.GetColumns());
8
9      for (int k = 0; k < A.GetRows(); k++) {
10         r[k][k] = 0;
11         for (int i = 0; i < A.GetRows(); i++)
12             r[k][k] = r[k][k] + A[i][k] * A[i][k];
13
14         r[k][k] = sqrt(r[k][k]);
15
16         for (int i = 0; i < A.GetRows(); i++)
17             q[i][k] = A[i][k] / r[k][k];
18
19         for (int j = k + 1; j < A.GetColumns(); j++) {
20             r[k][j] = 0;
21             for (int i = 0; i < A.GetRows(); i++)
22                 r[k][j] += q[i][k] * A[i][j];
23
24             for (int i = 0; i < A.GetRows(); i++)
25                 A[i][j] = A[i][j] - r[k][j] * q[i][k];
26         }
27     }
28     Matrix* QR = new Matrix[2];
29     QR[0] = q;
30     QR[1] = r;
31     return QR;
32 }
```

Usage Sample:

```
1  ///Written by Andrew Sheridan in C++
2  Matrix matrix = MatrixFactory::Instance()->DiagonallyDominant(5, 5);
3  Matrix* QR = GramSchmidt(matrix);
4  Matrix Q = QR[0];
5  Matrix R = QR[1];
6
7  std::cout << "Starting Matrix: " << std::endl;
8  matrix.Print();
9  std::cout << "Q: " << std::endl;
10 Q.Print();
11 std::cout << "R: " << std::endl;
```

```

12 R.Print();
13
14 std::cout << "Q * R" << std::endl;
15 Matrix TestResult = Q * R;
16 TestResult.Print();
17
18 Matrix Norm = matrix - TestResult;
19 double normValue = Norm.OneNorm();
20 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:

```
1  ///Solves the system of equations using Jacobi Iteration
2  ///A: The Matrix
3  ///x0: The initial guess
4  ///b: The Right-Hand-Side
5  Vector JacobiIteration(Matrix A, Vector x0, Vector b, int ←
    maxIterations, double tolerance) {
6      int iterations = 0;
7      int n = A.GetRows();
8      Vector newX(x0);
9      double error = 10 * tolerance;
10     while(iterations < maxIterations && tolerance < error){
11         Vector oldX(newX);
12         for (int i = 0; i < n; i++) {
13             newX[i] = b[i];
14             for (int j = 0; j < i; j++) {
15                 newX[i] = newX[i] - A[i][j] * oldX[j];
16             }
17             for (int j = i + 1; j < n; j++) {
18                 newX[i] = newX[i] - A[i][j] * oldX[j];
19             }
20             newX[i] = newX[i] / A[i][i];
21             error = (oldX - newX).L2Norm();
22             iterations++;
23         }
24     }
25     return newX;
26 }
```

Usage Sample:

```

1 //Written by Andrew Sheridan in C++
2 Matrix m1(3);
3 m1[0][0] = 7;
4 m1[0][1] = 3;
5 m1[0][2] = 1;
6 m1[1][0] = -3;
7 m1[1][1] = 10;
8 m1[1][2] = 2;
9 m1[2][0] = 1;
10 m1[2][1] = 7;
11 m1[2][2] = -15;
12
13 Vector b1(3);
14 b1[0] = 3;
15 b1[1] = 4;
16 b1[2] = 2;
17
18 Vector x1(3);
19
20 int maxIter = 10000;
21 double tolerance = 0.00001;
22
23 std::cout << "The results of Jacobi Iteration with max iterations of " <<
    << maxIter << " and tolerance of " << tolerance << std::endl;
24 Vector result1 = JacobiIteration(m1, x1, b1, maxIter, tolerance);
25 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:

```
1  ///Solves a matrix and RHS using Conjugate Gradient Method
2  ///A : The matrix to be solved
3  ///b : The RHS vector
4  ///x0 : The initial guess vector
5  ///tol : The tolerance of our method
6  Vector ConjugateGradient(Matrix A, Vector b, Vector x0, double tol) {
7      Vector rk = b - (A * x0);
8      double dk = rk * rk;
9      double bd = b * b;
10     int k = 0;
11     Vector pk = rk;
12     Vector xk = x0;
13     while (dk > tol * tol * bd) {
14         Vector sk = A * pk;
15         double ak = dk / (pk * sk);
16         Vector xkp1 = xk + (ak * pk);
17         Vector rkp1 = rk - (ak * sk);
18         double dkp1 = rkp1 * rkp1;
19         Vector pkp1 = rkp1 + ((dkp1 / dk) * pk);
20         k++;
21         //All values have been computed, set all kth values to equal the k←
           +1 value in preparation for next iteration
22         xk = xkp1;
23         rk = rkp1;
24         pk = pkp1;
25         dk = dkp1;
26     }
27     return xk;
28 }
```

Usage Sample:

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

```

8 m5[1][2] = 2;
9 m5[2][0] = 1;
10 m5[2][1] = 2;
11 m5[2][2] = 15;
12
13 Vector b5(3);
14 b5[0] = 28;
15 b5[1] = 31;
16 b5[2] = 22;
17
18 Vector x5(3);
19
20 Vector result5 = ConjugateGradient(m5, b5, x5, 0.000001);
21 std::cout << "Result of conjugate gradient method: " << std::endl;
22 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:

```
1  ///Finds an approximation of the largest Eigenvalue of a matrix
2  ///A : The square matrix
3  ///x0 : The initial guess vector
4  ///tol : The tolerance of the algorithm
5  ///maxIter : The maximum number of iterations to be executed by the ←
   method
6  double PowerMethod(Matrix A, Vector x0, double tol, int maxIter) {
7      double error = 10 * tol;
8      int k = 0;
9      Vector y = A * x0;
10     Vector xk = x0;
11     double lambda_k = 0;
12
13     while (error > tol && k < maxIter) {
14         Vector xkp1 = y / y.L2Norm();
15         y = A * xkp1;
16         double lambda_kp1 = xkp1 * y;
17         error = abs(lambda_kp1 - lambda_k);
18
19         lambda_k = lambda_kp1;
20         k++;
21     }
22
23     return lambda_k;
24 }
```

Usage Sample:

```
1  ///Written by Andrew Sheridan in C++
2
3  Matrix A(2); //New 2x2 matrix
4  A[0][0] = 2; //Initialize entries
5  A[0][1] = -12;
6  A[1][0] = 1;
7  A[1][1] = -5;
```

```
8
9  Vector x_0(2); //New vector size 2
10 x_0[0] = 1; //Initialize entries
11 x_0[1] = 1;
12
13 //Find the largest eigenvalue of A
14 double lambda = PowerMethod(A, x_0, 0.0001, 1000);
15 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:

```
1  ///Finds an approximation of the smallest Eigenvalue of a matrix
2  ///A : The square matrix
3  ///x0 : The initial guess vector
4  ///tol : The tolerance of the algorithm
5  ///maxIter : The maximum number of iterations to be executed by the ↵
   method
6  double InversePowerMethod(Matrix A, Vector x0, double tol, int maxIter↵
   ) {
7
8      int k = 0;
9      Matrix U = A;
10     Matrix L = LUFactorization(U, x0); // L is returned, U is modified to↵
        be upper triangular
11     Vector y = BackSubstitution(U, x0); // Solve for y by doing back ↵
        substitution.
12     double lambda_x = 0;
13     while (error > tol && k < maxIter) {
14         Vector x = y / y.L2Norm();
15         y = SolveSystem(A, x); //Does Gaussian Elimination then Back ↵
            Substitution
16         double lambda_xp1 = x * y;
17         error = abs(lambda_xp1 - lambda_x);
18
19         lambda_x = lambda_xp1;
20         k++;
21     }
22
23     output.close();
24     return lambda_x;
25 }
```

Usage Sample:

```
1  ///Written by Andrew Sheridan in C++
2
3  Matrix A(2);
4  A[0][0] = 2;
5  A[0][1] = -12;
6  A[1][0] = 1;
7  A[1][1] = -5;
```

```
8
9  Vector x_0(2);
10 x_0[0] = 1;
11 x_0[1] = 1;
12
13 double lambda = InversePowerMethod(A, x_0, 0.000001, 1000);
14 std::cout << "Lambda: " << lambda << std::endl;
```

Console Output

Lambda: -0.999999

5 Citations

6 Appendices

6.1 Appendix B: Vector Class

```
1 #pragma once
2 //Andrew Sheridan
3 //Math 5610
4 //Written in C++
5 //Vector.h
6
7 class Vector {
8 public:
9     //Initialization and Destruction
10    Vector();
11    Vector(unsigned n);
12    Vector(const Vector &v);
13    Vector(double* v, unsigned size);
14    Vector operator=(const Vector& v);
15    ~Vector();
16
17    void InitializeRandomEntries();
18    void InitializeAllOnes();
19
20    //Overloaded Operators
21    double& operator[] (unsigned x) { return entries[x]; }
22    friend double operator*( Vector& a, Vector& b);
23    friend Vector operator*(Vector& a, double constant);
24    friend Vector operator*(double constant, Vector& a);
25    friend Vector operator+(Vector& a, Vector& b);
26    friend Vector operator-(Vector& a, Vector& b);
27    friend Vector operator/(Vector& a, double constant);
28    friend Vector operator+(Vector& a, Vector& b);
29
30    //Basic Algorithms
31    double FindMaxMagnitudeStartingAt(unsigned start);
32    unsigned FindMaxIndex();
33    double L2Norm();
34
35    //Accessing Data
36    void Print();
37    unsigned GetSize() { return size; }
38    void SetSize(unsigned newSize) { size = newSize; }
39
40 protected:
41     unsigned size;
42
43 private:
44     double* entries; //The stored values of the vector
45 };
```

```

1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4 //Vector.cpp
5
6 #pragma once
7 #include "Vector.h"
8 #include <random>
9 #include <iostream>
10 #include <iomanip>
11
12 ///Default constructor
13 Vector::Vector() :size(0){ }
14
15 /// Initializes the entries to an empty array of size n
16 Vector::Vector(unsigned n) : size(n){
17     entries = new double[size];
18     for (unsigned i = 0; i < n; i++) {
19         entries[i] = 0;
20     }
21 }
22
23 ///Copy Constructor
24 Vector::Vector(const Vector &v) : size(v.size) {
25     entries = new double[size];
26     for (unsigned i = 0; i < size; i++) {
27         entries[i] = v.entries[i];
28     }
29 }
30
31 ///Copy Constructor
32 ///v: an array of doubles
33 ///n: the size of v
34 Vector::Vector(double* v, unsigned n) :size(n) {
35     entries = new double[size];
36     for (unsigned i = 0; i < size; i++) {
37         entries[i] = v[i];
38     }
39 }
40
41 /// Copy Assignment Operator
42 Vector Vector::operator= (const Vector& v) {
43     size = v.size;
44     entries = new double[v.size];
45     for (unsigned i = 0; i < v.size; i++) {
46         entries[i] = v.entries[i];
47     }
48     return *this;
49 }
50
51 ///Destructor
52 Vector::~Vector()
53 {
54 }
55
56 ///Inner Product
57 double operator* (Vector& a, Vector& b) {
58     if (a.size != b.size)
59         return NULL;
60     double sum = 0;
61     for (int i = 0; i < a.size; i++) {
62         sum += a[i] * b[i];
63     }

```

```

64     return sum;
65 }
66
67 ///Multiply each entry by a constant
68 Vector operator* (Vector& a, double constant) {
69     Vector newVector(a.size);
70     for (int i = 0; i < a.size; i++) {
71         newVector[i] = a[i] * constant;
72     }
73     return newVector;
74 }
75
76 ///Multiply each entry by a constant
77 Vector operator* (double constant, Vector& a) {
78     Vector newVector(a.GetSize());
79     for (int i = 0; i < a.GetSize(); i++) {
80         newVector[i] = a[i] * constant;
81     }
82     return newVector;
83 }
84
85 ///Subtracts the elements of vector a from the elements of vector b
86 Vector operator- (Vector& a, Vector& b) {
87     if (a.size != b.size) return a;
88     Vector newVector(a.size);
89     for (int i = 0; i < a.size; i++) {
90         newVector[i] = a[i] - b[i];
91     }
92     return newVector;
93 }
94
95 ///Divides the entries of the vector by a constant
96 Vector operator/ (Vector& a, double constant) {
97     Vector newVector(a.GetSize());
98     for (int i = 0; i < a.GetSize(); i++) {
99         newVector[i] = a[i] / constant;
100     }
101     return newVector;
102 }
103
104 ///Returns a new vector where its entries are the sum of corresponding↔
    entries in a and b
105 Vector operator+ (Vector& a, Vector& b) {
106     if (a.size != b.size) {
107         std::cout << "Vectors are not same size. Returning first vector." ↔
108             << std::endl;
109         return a;
110     }
111     Vector newVector(a.size);
112     for (int i = 0; i < a.size; i++) {
113         newVector[i] = a[i] + b[i];
114     }
115     return newVector;
116 }
117
118 /// Initializes the entries to values between 0 and 1
119 void Vector::InitializeRandomEntries() {
120     std::mt19937 generator(123); //Random number generator
121     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↔
        distribution
122     for (unsigned i = 0; i < size; i++) {
123         entries[i] = dis(generator); //Assign each entry in entries to ↔
            random number between 0 and 1

```

```

124     }
125 }
126
127 /// Initializes the entries to 1
128 void Vector::InitializeAllOnes() {
129     for (unsigned i = 0; i < size; i++) {
130         entries[i] = 1; //Assign each entry in entries to 1
131     }
132 }
133
134 ///Finds the entry with the largest magnitude, starting with entry "↔
135     start".
136 double Vector::FindMaxMagnitudeStartingAt(unsigned start) {
137     double max = 0;
138     for (unsigned i = start; i < size; i++) {
139         double value = std::abs(entries[i]);
140         if (value > max) max = value;
141     }
142     return max;
143 }
144
145 ///Finds the index of the value with the largest magnitude
146 unsigned Vector::FindMaxIndex() {
147     double max = 0;
148     unsigned index = -1;
149     for (unsigned i = 0; i < size; i++) {
150         double value = std::abs(entries[i]);
151         if (value > max) {
152             max = value;
153             index = i;
154         }
155     }
156     return index;
157 }
158
159 ///Computes the L2 norm of the vector
160 double Vector::L2Norm() {
161     double sum = 0;
162     for (int i = 0; i < size; i++) {
163         sum += (entries[i] * entries[i]);
164     }
165     return std::sqrt(sum);
166 }
167
168 ///Outputs the vector's entries to the console
169 void Vector::Print() {
170     for (unsigned i = 0; i < size; i++) {
171         std::cout << std::setw(13) << std::left << entries[i];
172     }
173     std::cout << std::endl << std::endl;
174 }

```

6.2 Appendix C: Matrix Class

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4 //Matrix.h
5
6 #pragma once
7 #include "Vector.h"
8
9 class Matrix{
10 public:
11     //Initialization and Deconstruction
12     Matrix() = default;
13     Matrix(unsigned size);
14     Matrix(unsigned rowCount, unsigned columnCount);
15     Matrix(const Matrix &m);
16     Matrix operator = (const Matrix& m);
17     ~Matrix();
18
19     void InitializeIdentityMatrix();
20     void InitializeRandom();
21     void InitializeRange(double minValue, double maxValue);
22     void InitializeDiagonallyDominant();
23
24     //Overloaded Operators
25     Vector &operator [] (unsigned row) { return entries[row]; }
26     friend bool operator == (const Matrix& A, const Matrix& B);
27     friend bool operator != (const Matrix& A, const Matrix& B);
28     friend Vector operator * (const Matrix& A, Vector& x);
29     friend Vector operator / (const Matrix& A, Vector& x);
30     friend Matrix operator * (Matrix A, Matrix B);
31     friend Matrix operator - (Matrix A, Matrix B);
32
33     //Basic Algorithms
34     bool IsSymmetric();
35     Matrix Transpose();
36     double OneNorm();
37     double InfinityNorm();
38
39     //Getters and Setters
40     unsigned GetRows() { return rows; }
41     unsigned GetColumns() { return columns; }
42     void SetRows(unsigned r) { rows = r; }
43     void SetColumns(unsigned c) { columns = c; }
44
45     //Output
46     void Print();
47     void PrintAugmented(Vector v);
48
49 private:
50     Vector* entries; //The entries of the matrix
51
52     unsigned rows;
53     unsigned columns;
54 };
```

```

1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4 //Matrix.cpp
5
6 #include "Matrix.h"
7 #include "Vector.h"
8 #include <random>
9 #include <iostream>
10 #include <iomanip>
11
12
13 #pragma region Constructors and Initialization
14 ///Initializes an nxn matrix of size "size". All entries are zeroes.
15 Matrix::Matrix(unsigned size) : rows(size), columns(size)
16 {
17     entries = new Vector[size];
18     for (unsigned i = 0; i < size; i++) {
19         entries[i] = Vector(size);
20         for (unsigned j = 0; j < size; j++) {
21             entries[i][j] = 0;
22         }
23     }
24 }
25
26 ///Initializes an nxm matrix. All entries are zeroes.
27 Matrix::Matrix(unsigned rowCount, unsigned columnCount) : rows(↵
    rowCount), columns(columnCount) {
28     entries = new Vector[rowCount];
29     for (unsigned i = 0; i < rowCount; i++) {
30         entries[i] = Vector(columnCount);
31         for (unsigned j = 0; j < columnCount; j++) {
32             entries[i][j] = 0;
33         }
34     }
35 }
36
37 ///Copy Constructor
38 Matrix::Matrix(const Matrix &m) : rows(m.rows), columns(m.columns) {
39     entries = new Vector[rows];
40     for (unsigned int i = 0; i < rows; i++) {
41         entries[i] = Vector(columns);
42         for (unsigned int j = 0; j < columns; j++) {
43             entries[i][j] = m.entries[i][j];
44         }
45     }
46 }
47
48 ///Destructor
49 Matrix::~Matrix() {
50 }
51
52 ///Assignment operator overload
53 Matrix Matrix::operator=(const Matrix& m) {
54     this->rows = m.rows;
55     this->columns = m.columns;
56     this->entries = new Vector[m.rows];
57     for (int i = 0; i < m.rows; i++) {
58         this->entries[i] = m.entries[i];
59     }
60
61     return *this;
62 }

```

```

63
64
65 ///Initializes the matrix to have ones on the main diagonal
66 void Matrix::InitializeIdentityMatrix() {
67     for (unsigned i = 0; i < rows; i++) {
68         for (unsigned j = 0; j < i; j++) {
69             entries[i][j] = 0;
70         }
71         entries[i][i] = 1;
72         for (unsigned j = i + 1; j < rows; j++) {
73             entries[i][j] = 0;
74         }
75     }
76 }
77
78 ///Sets the values of all entries between 0 and 1
79 void Matrix::InitializeRandom() {
80     std::mt19937 generator(123); //Random number generator
81     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↔
        distribution
82
83     for (unsigned i = 0; i < columns; i++) {
84         for (unsigned j = 0; j < rows; j++) {
85             entries[i][j] = dis(generator); //Assign each entry in matrix to↔
                random number between 0 and 1
86         }
87     }
88 }
89
90 ///Sets the values of all entries between minValue and maxValue
91 void Matrix::InitializeRange(double minValue, double maxValue) {
92     std::mt19937 generator(123); //Random number generator
93     std::uniform_real_distribution<double> dis(minValue, maxValue); //↔
        Desired distribution
94
95     for (unsigned i = 0; i < columns; i++) {
96         for (unsigned j = 0; j < rows; j++) {
97             entries[i][j] = dis(generator); //Assign each entry in matrix to↔
                random number between 0 and 1
98         }
99     }
100 }
101
102 /// Sets the values of the entries to be diagonally dominant
103 void Matrix::InitializeDiagonallyDominant() {
104     std::mt19937 generator(123); //Random number generator
105     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↔
        distribution
106
107     for (unsigned i = 0; i < rows; i++) {
108         for (unsigned j = 0; j < columns; j++) {
109             entries[i][j] = dis(generator); //Assign each entry in matrix to↔
                random number between 0 and 1
110         }
111     }
112
113     for (unsigned k = 0; k < rows; k++) {
114         entries[k][k] += 10 * rows; //Add 10*n to all diagonal entries
115     }
116 }
117
118 #pragma endregion
119
120 #pragma region Printing

```

```

121 ///Outputs matrix to the console
122 void Matrix::Print() {
123     for (unsigned i = 0; i < rows; i++) {
124         for (unsigned j = 0; j < columns; j++) {
125             std::cout << std::setw(13) << std::left << entries[i][j];
126         }
127         std::cout << std::endl;
128     }
129     std::cout << std::endl;
130 }
131
132 ///Outputs an augmented coefficient matrix to the console
133 void Matrix::PrintAugmented(Vector v) {
134     if (v.GetSize() != rows) {
135         std::cout << "Vector and Matrix do not match sizes" << std::endl;
136         return;
137     }
138
139     for (unsigned i = 0; i < rows; i++) {
140         for (unsigned j = 0; j < columns; j++) {
141             std::cout << std::setw(13) << std::left << entries[i][j];
142         }
143         std::cout << " | " << v[i] << std::endl;
144     }
145     std::cout << std::endl;
146 }
147 #pragma endregion
148
149 #pragma region Comparison Operations
150 ///Compares two matrices. Returns true if all corresponding entries ←
151   are equal, and false if any are not equal.
152 bool operator == (const Matrix& A, const Matrix& B) {
153     if (A.columns != B.columns) return false;
154     if (A.rows != B.rows) return false;
155
156     for (unsigned i = 0; i < A.rows; i++) {
157         for (unsigned j = 0; j < A.columns; j++) {
158             if (A.entries[i][j] != B.entries[i][j]) {
159                 return false;
160             }
161         }
162     }
163     return true;
164 }
165
166 ///Compares two matrices. Returns true if any corresponding entries ←
167   are not equal, and false if all are equal.
168 bool operator != (const Matrix& A, const Matrix& B) {
169     if (A.columns != B.columns) return true;
170     if (A.rows != B.rows) return true;
171
172     for (unsigned i = 0; i < A.rows; i++) {
173         for (unsigned j = 0; j < A.columns; j++) {
174             if (A.entries[i][j] != B.entries[i][j]) {
175                 return true;
176             }
177         }
178     }
179     return false;
180 }
181
182 ///Checks to see if matrix is symmetric
183 bool Matrix::IsSymmetric() {
184     for (unsigned int i = 0; i < rows; i++) {

```



```

183     for (unsigned int j = 0; j <= i; j++) {
184         if (entries[i][j] != entries[j][i])
185             return false;
186     }
187 }
188 return true;
189 }
190
191 #pragma endregion
192
193
194 #pragma region operations
195 /// Returns the transpose of the n by n matrix A
196 Matrix Matrix::Transpose() {
197     Matrix matrix(columns, rows);
198
199     for (unsigned i = 0; i < rows; i++) {
200         for (unsigned j = 0; j < columns; j++) {
201             matrix[j][i] = entries[i][j];
202         }
203     }
204     return matrix;
205 }
206
207 ///Multiplies an nxn matrix A by the vector x
208 Vector operator *(const Matrix& A, Vector& x) {
209     if (A.columns != x.GetSize()) return NULL;
210
211     Vector result(A.rows);
212     for (unsigned i = 0; i < A.rows; i++) {
213         result[i] = 0;
214         for (unsigned j = 0; j < A.columns; j++) {
215             result[i] += A.entries[i][j] * x[j];
216         }
217     }
218     return result;
219 }
220
221 ///Divides an nxn matrix A by the vector x
222 Vector operator /(const Matrix& A, Vector& x) {
223     if (A.columns != x.GetSize()) return NULL;
224
225     Vector result(A.rows);
226     for (unsigned i = 0; i < A.rows; i++) {
227         result[i] = 0;
228         for (unsigned j = 0; j < A.columns; j++) {
229             result[i] += A.entries[i][j] / x[j];
230         }
231     }
232     return result;
233 }
234
235
236 ///Multiplies an matrix A by matrix B
237 Matrix operator* (Matrix A, Matrix B) {
238     if (A.columns != B.rows) throw "Incompatible sizes";
239
240     Matrix matrix(A.rows, B.columns);
241     Matrix bTranspose = B.Transpose();
242
243     //std::cout << "Starting multiplication. A:" << std::endl;
244     /*A.Print();
245     std::cout << "B: " << std::endl;
246     B.Print();

```

```

247     std::cout << "Bt: " << std::endl;
248     bTranspose.Print();*/
249
250     for (unsigned i = 0; i < A.rows; i++) {
251         for (unsigned j = 0; j < B.columns; j++) {
252             //matrix[i][j] = DotProduct(A[i], bTranspose[j], A.columns);
253             matrix[i][j] = A[i] * bTranspose[j];
254         }
255     }
256     return matrix;
257 }
258
259 ///Multiplies an matrix A by matrix B
260 Matrix operator- (Matrix A, Matrix B) {
261     if (A.columns != B.columns && A.rows != B.rows) throw "Incompatible ↔
        sizes";
262     Matrix matrix(A.rows, B.columns);
263     for (unsigned i = 0; i < A.rows; i++) {
264         for (unsigned j = 0; j < A.columns; j++) {
265             matrix[i][j] = A[i][j] - B[i][j];
266         }
267     }
268     return matrix;
269 }
270
271 ///Computes the 1-norm of the matrix
272 double Matrix::OneNorm() {
273     double columnMax = 0;
274     for (unsigned i = 0; i < rows; i++) {
275         double columnSum = 0;
276         for (unsigned j = 0; j < columns; j++) {
277             columnSum += std::abs(entries[i][j]);
278         }
279         if (columnSum > columnMax)
280             columnMax = columnSum;
281     }
282     return columnMax;
283 }
284
285 ///Computes the infinity norm of an n by n matrix A
286 double Matrix::InfinityNorm() {
287     double rowMax = 0;
288     for (unsigned j = 0; j < columns; j++) {
289         double rowSum = 0;
290         for (unsigned i = 0; i < rows; i++) {
291             rowSum += std::abs(entries[i][j]);
292         }
293         if (rowSum > rowMax)
294             rowMax = rowSum;
295     }
296     return rowMax;
297 }
298
299 #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.

```
1 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

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4 //MatrixFactory.h
5 #pragma once
6 #include "Matrix.h"
7
8 ///A singleton which creates new matrices
9 class MatrixFactory {
10 public:
11     static MatrixFactory* Instance();
12     Matrix Identity(unsigned rows, unsigned columns);
13     Matrix Ones(unsigned rows, unsigned columns);
14     Matrix UpperTriangular(unsigned rows, unsigned columns);
15     Matrix LowerTriangular(unsigned rows, unsigned columns);
16     Matrix Random(unsigned rows, unsigned columns);
17     Matrix RandomRange(unsigned rows, unsigned columns, double min, double max);
18     Matrix DiagonallyDominant(unsigned rows, unsigned columns);
19     Matrix Symmetric(unsigned size);
20     Matrix SPD(unsigned size);
21
22 private:
23     MatrixFactory() {};
24     MatrixFactory(MatrixFactory const&) {};
25     MatrixFactory& operator=(MatrixFactory const&) {};
26     static MatrixFactory* m_instance;
27 };
```

6.3.3 Code Written

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4 //MatrixFactory.cpp
5
6 #include "MatrixFactory.h"
7 #include <random>
8
9 MatrixFactory* MatrixFactory::m_instance = nullptr;
10
11 ///Returns the instance of the MatrixFactory singleton
12 MatrixFactory* MatrixFactory::Instance() {
13     if (!m_instance)
14         m_instance = new MatrixFactory();
15
16     return m_instance;
17 }
18
19 ///Creates an identity matrix
20 Matrix MatrixFactory::Identity(unsigned rows, unsigned columns) {
21     Matrix m(rows, columns);
22     m.InitializeIdentityMatrix();
23     return m;
24 }
25
26 ///Creates a matrix where every entry is a 1
27 Matrix MatrixFactory::Ones(unsigned rows, unsigned columns) {
28     Matrix m(rows, columns);
29     for (int i = 0; i < rows; i++) {
30         for (int j = 0; j < columns; j++) {
31             m[i][j] = 1;
32         }
33     }
34     return m;
35 }
36
37 ///Creates a matrix where every entry is a value between 0 and 1
38 Matrix MatrixFactory::Random(unsigned rows, unsigned columns) {
39     Matrix m(rows, columns);
40     m.InitializeRandom();
41     return m;
42 }
43
44 ///Creates a matrix where every entry is a value between minValue and ↵
45     maxValue
46 Matrix MatrixFactory::RandomRange(unsigned rows, unsigned columns, ↵
47     double minValue, double maxValue) {
48     Matrix m(rows, columns);
49     m.InitializeRange(minValue, maxValue);
50     return m;
51 }
52
53 ///Creates an upper-triangular matrix
54 Matrix MatrixFactory::UpperTriangular(unsigned rows, unsigned columns)↵
55     {
56     Matrix m(rows, columns);
57     std::mt19937 generator(123); //Random number generator
58     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↵
59     distribution
60 }
```

```

57     for (unsigned i = 0; i < rows; i++) {
58         for (unsigned j = i; j < columns; j++) {
59             m[i][j] = dis(generator); //Assign entry in matrix to random ←
                    number between 0 and 1
60         }
61         m[i][i] += 10 * rows; //Add 10*n to all diagonal entries
62     }
63
64     return m;
65 }
66
67 //Creates an upper-triangular matrix
68 Matrix MatrixFactory::LowerTriangular(unsigned rows, unsigned columns)←
    {
69     Matrix m(rows, columns);
70     std::mt19937 generator(123); //Random number generator
71     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ←
                    distribution
72
73     for (unsigned i = 0; i < rows; i++) {
74         for (unsigned j = 0; j <= i; j++) {
75             m[i][j] = dis(generator); //Assign entry in matrix to random ←
                    number between 0 and 1
76         }
77         m[i][i] += 10 * rows; //Add 10*n to all diagonal entries
78     }
79
80     return m;
81 }
82
83 //Creates a Diagonally Dominant matrix
84 Matrix MatrixFactory::DiagonallyDominant(unsigned rows, unsigned ←
    columns) {
85     Matrix m(rows, columns);
86     m.InitializeDiagonallyDominant();
87     return m;
88 }
89
90 //Creates a symmetric matrix, where all entries have values between 0←
    and 1
91 Matrix MatrixFactory::Symmetric(unsigned size) {
92     Matrix m(size);
93     std::mt19937 generator(123); //Random number generator
94     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ←
                    distribution
95
96     for (unsigned i = 0; i < size; i++) {
97         for (unsigned j = i; j < size; j++) {
98             double value = dis(generator); //Assign each entry in matrix to ←
                    random number between 0 and 1
99             m[i][j] = value;
100            m[j][i] = value;
101        }
102    }
103
104    return m;
105 }
106
107 //Creates a symmetric positive definite matrix
108 Matrix MatrixFactory::SPD(unsigned size) {
109     std::mt19937 generator(123); //Random number generator
110     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ←
                    distribution

```

```

112 Matrix matrix(size);
113
114 for (unsigned i = 0; i < size; i++) {
115     for (unsigned j = i; j < size; j++) {
116         double value = dis(generator); //Assign each entry in matrix to ↵
            random number between 0 and 1
117         matrix[i][j] = value;
118         matrix[j][i] = value;
119     }
120 }
121
122 for (unsigned k = 0; k < size; k++) {
123     matrix[k][k] += 10 * size; //Add 10*n to all diagonal entries
124 }
125
126 return matrix;
127 }

```

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.

```
1 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

```
1 #pragma once
2 #include <cmath>
3
4 class Complex
5 {
6 public:
7     double real;
8     double imaginary;
9
10    Complex(double newReal, double newImaginary)
11    {
12        real = newReal;
13        imaginary = newImaginary;
14    }
15
16    Complex operator+(Complex c) {
17        double realResult = real + c.real;
18        double imaginaryResult = imaginary + c.imaginary;
19        Complex result(realResult, imaginaryResult);
20        return result;
21    }
22
23    Complex operator-(Complex c) {
24        double realResult = real - c.real;
25        double imaginaryResult = imaginary - c.imaginary;
26        Complex result(realResult, imaginaryResult);
27        return result;
28    }
29
30    Complex operator*(Complex c) {
31        double realResult = real * c.real - (imaginary * c.imaginary);
32        double imaginaryResult = real * c.imaginary + c.real * imaginary;
33        Complex result(realResult, imaginaryResult);
34        return result;
35    }
36
37    Complex operator/(Complex c) {
38        double realNumerator = real*c.real - imaginary*c.imaginary;
39        double imaginaryNumerator = imaginary*c.real - real*c.imaginary;
40        double denominator = (c.real*c.real) - (c.imaginary*c.imaginary);
```

```
41
42     return Complex(realNumerator / denominator, imaginaryNumerator / ↵
43         denominator);
44 };
45
46 double complexAbsolute(Complex a) {
47     return std::sqrt(std::pow(a.real, 2) + std::pow(a.imaginary, 2));
48 }
```
