Software Manual
for Assignments Completed in Math 5610

Sam Christiansen

December 16, 2016

Math 5610: Computational Linear Algebra

and Solution of Nonlinear Systems

Last revised: January 2, 2017

Contents:

1) Introduction

This software manual is a compilation of computer programming routines written for my MATH 5610 course, Computational Linear Algebra and Solution of Nonlinear Systems, which I took at Utah State University in the fall semester of 2016. All of the routines in this manual were written in Microsoft's Visual Studios, in the language of C++. The routines were written to be self-contained methods that can be imported into future projects if desired.

In order to ease the coding process when dealing with linear systems, I wrote two data structures: one to hold information for vectors and another for matrices, called MyVector and MyMatrix, respectively. A MyVector object can be created to be size n, while a MyMatrix object can be a mxn sized matrix. This allowed me to easily test linear algebra routines with matrices and vectors of varying size. The definitions of these two classes will be presented in chapter 2 of this manual, and references to these classes will be seen throughout the software manual.

Chapters 3 and 4 also contain many routines that will be referenced in many parts of this manual. Chapter 3 focuses on basic computational routines, such as finding vector and matrix norms, while chapter 4 contains basic methods for linear systems, such as routines to find the product of a vector and matrix.

The latter chapters have routines that are more complex and that solve specific types of problems. Chapter 5 covers a variety of methods to find roots of a given function. Chapter 6 dives into solving linear systems, using both direct and iterative methods. Chapter 7 then deals with methods to find eigenvalues of linear systems. Many of the algorithms for these routines were provided both during lecture time and in the course text book, *A First Course in Numerical Methods* by Uri M. Ascher and Chen Greif.

2) **Reusable Matrix and Vector Classes**

This chapter will outline the data structures that I developed and used throughout many of the routines created for this class. I have written two data structures: My Matrix and MyVector. This are written in its own file and can be copied into other projects. Both have simple constructors that use pointer arithmetic and initializes each element in the matrix / vector to zero. MyMatrix can be created for nxn and nxm matrices, and MyVector can represent a vector of size n. Both also have print functions that allows the user to easily see the elements of the matrices and vertices.

This chapter also has functions that create specific functions matrices, which allow the user to test different functions.

2.1) MyMatrix

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++
//Creating my own Matrix class
#ifndef MY_MATRIX_H
#define MY_MATRIX_H
#include <iostream>

class MyMatrix{
        double** myMatrix;                              //Dynamic double array
public:
        int rowCount;                                   //Variable to hold the number of rows
        int columnCount;                                //Variable to hold the number of columns
        MyMatrix(int size){                             //1st Constructor for square matrix
                rowCount = size;                        //Sets the row and column count to size
                columnCount = size;
                myMatrix = new double*[size];
                for (int i = 0; i < size; i++)
                        myMatrix[i] = new double[size];
                for (int i = 0; i < size; i++)
                        for (int j = 0; j < size; j++)
                                myMatrix[i][j] = 0;     //Initializes each position to 0
        }
        MyMatrix(int rows, int columns){                //2nd Constructor for non-square matrix
                rowCount = rows;                        //Sets the rowCount
                columnCount = columns;                  //Sets the columnCount
                myMatrix = new double*[rowCount];
                for (int i = 0; i < rowCount; i++)
                        myMatrix[i] = new double[columnCount];
                for (int i = 0; i < rowCount; i++)
                        for (int j = 0; j < columnCount; j++)
                                myMatrix[i][j] = 0;     //Initializes each position to 0
        }
        void print(){                                   //Function to print the matrix
                for (int i = 0; i < columnCount; i++){
                        for (int j = 0; j < rowCount; j++){
                                std::cout << myMatrix[j][i] << "  ";
                        }
                        std::cout << std::endl;
                }
                std::cout << std::endl;
        }
        double& operator() (int i, int j) {
                return myMatrix[i][j];                  //Overloading () to access elements
        }
};
#endif
```

2.2) MyVector

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++
//Creating my own Vector class
#ifndef MY_VECTOR_H
#define MY_VECTOR_H
#include <iostream>

class MyVector{
        double* myArray;                        //Dynamic Array
        int arraySize;                          //Variable for the size

public:                                         //The constructor
        MyVector(int size) : arraySize(size), myArray(new double[size]){
                for (int i = 0; i < arraySize; i++)
                        myArray[i] = 0;         //Initializes each position to 0
        }
        int size() const{                       //Function to return the size
                return arraySize;
        }
        void print(){                           //Function to print the Vector
                for (int i = 0; i < arraySize; i++){
                        std::cout << myArray[i] << std::endl;
                }
                std::cout << std::endl;
        }
        double& operator[](int i){              //Overloading [] to access elements
                if (i < 0 || i >= arraySize)
                        throw("range error");
                return myArray[i];
        }
};
#endif
```

2.3) Matrix Constructors

These are the functions to create specific matrices, which are used to test functions that deal with solving systems of linear equations. Each of the functions take the size of the matrix as a parameter and return the newly created matrix.

2.3.1) *generateSimpleMatrix*

This constructor, as suggested by the name, is meant to create a simple nxn matrix. The function sets each element of the matrix to the sum of its row number and column number. So the matrix A will have A(i,j) = i+i+j. The one exception is that A(0,0) = 1, so that the matrix can be used in a function that doesn't use pivoting.

```
//Function to create a simple matrix, with 1 parameter:
//The parameter, size, is the size of the matrix to be made
//The function returns the created matrix
MyMatrix generateSimpleMatrix(int size){
        MyMatrix A = MyMatrix(size);
        for (int i = 0; i < size; i++){
                for (int j = 0; j < size; j++){
                        A(i, j) = i + i + j;
                }
        }
        A(0, 0) = 1;
        return A;
}
```

***Sample driver code for *generateSimpleMatrix*:**

```
int main(){
	MyMatrix A = generateSimpleMatrix(5);
	A.print();

	return 0;
}
```

***Output:**

```
1 1 2 3 4
2 3 4 5 6
4 5 6 7 8
6 7 8 9 10
8 9 10 11 12

Press any key to continue . . .
```

2.3.2) *generateRandomMatrix*

This constructor is used to create a matrix filled with random values between 0 and 100. Important to note that this constructor does not create a matrix that is diagonally dominate, which is what occurs in the constructor in 2.3.3.

```
//Function to create a random, square matrix without a dominate diagonal, with 1 parameter:
//The parameter, size, is the size of the square matrix to be made
//The function returns the created matrix
MyMatrix generateRandomMatrix(int size){
	srand(time(NULL));                              //Setting seed for rand()

	MyMatrix A = MyMatrix(size);                    //Declaring new matrix
	for (int i = 0; i < size; i++){                 //Loops 'size' times
		for (int j = 0; j < size; j++){         //Loops 'size' times
			A(i, j) = (double(rand()) /     //Setting each entry to a
				double(RAND_MAX)) * 100; //random number between 0 and 100
		}
	}
	return A;                                       //Returning new matrix A
}
```

Sample driver code for *generateRandomMatrix:

```
int main(){
	int size = 5;
	MyMatrix A = generateRandomMatrix(size);
	cout << "Printing Matrix A: " << endl;
	A.print();
}
```

***Output:**

```
Printing Matrix A:
23.307 51.4969 22.1595 10.8493 93.6674
95.7671 96.176 49.7635 49.6933 42.378
13.2328 24.9855 53.6271 15.4149 18.7048
97.5249 81.0846 62.1052 98.6877 75.7286
57.0513 7.32444 28.9956 30.8359 21.7872
Press any key to continue . . .
```

2.3.3) *generateRandomSquareMatrix*

This constructor is the most used of the four constructors in this section. Each element of the nxn matrix is set to a random number between 0 and 1. Because we also want the generated matrix to be diagonally dominant, each diagonal entry A(i,i) has the product 10*n added to its value.

```
//Function to create a random, square, diagonally dominate matrix, with 1 parameter:
//The parameter, size, is the size of the matrix to be made
//The function returns the created matrix
MyMatrix generateRandomSquareMatrix(int size){
        srand(time(NULL));                                  //Setting seed for rand()

        MyMatrix A = MyMatrix(size);                        //Declaring new matrix
        for (int i = 0; i < size; i++){                     //Loops 'size' times
                for (int j = 0; j < size; j++){             //Loops 'size' times
                        A(i, j) = double(rand())            //Setting each entry to a
                                / double(RAND_MAX);         //random number between 0 and 1
                }
        }
        for (int i = 0; i < size; i++){                     //Loops 'size' times
                A(i, i) = A(i, i) + (10 * size);            //Sets each i=j entry A[i][j] to a
        }                                                   //much larger number (as directed)
        return A;                                           //Returning new matrix A
}
```

***Sample driver code for *generateRandomSquareMatrix*:**

```
int main(){
        MyMatrix A = generateRandomSquareMatrix(5);
        A.print();

        return 0;
}
```

***Output:**

```
50.0345 0.212439 0.335917 0.609943 0.504654
0.220466 50.7221 0.22251 0.347392 0.668233
0.398175 0.294656 50.3553 0.395093 0.694021
0.79223 0.264199 0.794 50.1307 0.55266
0.858943 0.117924 0.625446 0.295938 50.0661

Press any key to continue . . .
```

2.3.4) *generateRandomSymmetricSquareMatrix*

This constructor is used to create an nxn matrix which has elements with random values between 1 and 4. This function also insures that the matrix is symmetric. Note that this does not generate a diagonally dominate matrix.

```
//Function to create a random, square and symmetric matrix, with 1 parameter:
//The parameter, size, is the size of the matrix to be made
//The function returns the created matrix
MyMatrix generateRandomSymmetricSquareMatrix(int size){
        srand(time(NULL));                                  //Setting seed for rand()

        MyMatrix A = MyMatrix(size);                        //Declaring new matrix
```

```
        for (int i = 0; i < size; i++){                 //Loops 'size' times
                for (int j = i; j < size; j++){         //Loops 'size' times
                        A(i, j) = rand() % 4 + 1;       //Setting each entry to a
                                                        //random number between 1 and 4.
                        A(j, i) = A(i, j);              //This makes the matrix symmetric
                }
        }
        return A;                                       //Returning new symmetric matrix A
}
```

***Sample driver code for *generateRandomSquareMatrix*:**

```
int main(){
        MyMatrix A = generateRandomSymmetricSquareMatrix(5);
        A.print();

        return 0;
}
```

***Output:**

```
1 3 3 1 2
3 1 3 4 4
3 3 3 4 2
1 4 4 3 1
2 4 2 1 4

Press any key to continue . . .
```

2.3.5) *generateTridiagonalMatrix*

This last constructor is used to test only one function, the *solveTridiagonalMatrix* function. It creates a matrix that has values of the main diagonal elements set to -2, while the elements of the diagonal directly above and below have their values set to 1. The remainder of the values are set to zero.

```
//Function to create a Tridiagonal Matrix, as directed, with 1 parameter:
//The parameter, size, is the size of the matrix to be made
//The function returns the Tridiagonal Matrix
MyMatrix generateTridiagonalMatrix(int size){
        MyMatrix A = MyMatrix(size);

        for (int i = 0; i < size; i++){
                A(i, i) = -2.0;                         //For main diagonal entries
                if (i>0)
                        A(i - 1, i) = A(i, i - 1) = 1.0; //For lower and upper diagonal entries
        }
        return A;                                       //Returning new tridiagonal matrix A
}
```

***Sample driver code for *generateRandomSquareMatrix*:**

```
int main(){
        MyMatrix A = generateTridiagonalMatrix(5);
        A.print();

        return 0;
}
```

***Output:**

```
-2  1  0  0  0
1  -2  1  0  0
0  1  -2  1  0
0  0  1  -2  1
0  0  0  1  -2

Press any key to continue . . .
```

3) Basic Computational Routines

This chapter is dedicated to a few different types of simple computational routines. Some of the routines, such the routines to find the machine epsilon of a computer, are standalone functions, while other routines, such as the routines to find matrix and vector norms, will be used repeatedly within other routines.

3.1) Machine Precision Routines

These two routines are to be used by a user to find the precision that a computer will produce. By knowing the precision of the computer, users will be able to better understand the approximations of other routines.

3.1.1) *singlePrecision*

This routine finds the single precision of a computer by using the type float to hold the number. The number is divided by 2 repeatedly in a loop until the number's value stops changing.

```
//Sam Christiansen
//9/9/2016
//Programing Language: C++

//Function to find the single precision of a computer:
//The function first prints out the smallest number computed,
//and then prints out the power of 2
void singlePrecision(){
        float num = 1.0;                        //Number of type float for single precision
        int exp = 0;                            //Number for the exponent value
        while (num + 1 != 1){                   //Main loop
                exp--;                          //Decrement exponent value
                num /= 2;                       //Divide the number by 2
        }
        cout << "Single Precision: " <<         //Printing results
                num << "\t Exponent: " <<
                exp << endl;
}
```

***Sample driver code for *singlePrecision*:**

```
int main(){
        singlePrecision();
        return 0;
}
```

***Output:**

```
Single Precision: 5.96046448e-008  Exponent: -24

Press any key to continue . . .
```

3.1.2) *doublePrecision*

This routine finds the double precision of a computer by using the type double to hold the number. The number is divided by 2 repeatedly in a loop until the number's value stops changing.

```
//Sam Christiansen
//9/9/2016
//Programing Language: C++

//Function to find the double precision of a computer:
//The function first prints out the smallest number computed,
//and then prints out the power of 2
void doublePrecision(){
        double num = 1.0;                       //Number of type double for double precision
        int exp = 0;                            //Number for the exponent value
        while (num + 1 != 1){                   //Main loop
                exp--;                          //Decrement exponent value
                num /= 2;                       //Divide the number by 2
        }
        cout << "Double Precision: " <<         //Printing results
                num << "\t Exponent: " <<
                exp << endl;
}
```

***Sample driver code for *singlePrecision*:**

```
int main(){
        singlePrecision();
        return 0;
}
```

***Output:**

```
Double Precision: 1.11022e-016  Exponent: -53

Press any key to continue . . .
```

3.2) Vector Norms

This section contains the three most used vector norms: l_1 norm, l_2 norm, and l_∞ norm. Calculating the different norms gives us valuable information about the vectors, such as the average of the elements in the vector.

3.2.1) *l1Norm*

The l_1 norm is used if the user wants to find the average of the elements contained by a vector. The routine sums up the absolute value of each element in the vector and then divides that sum by the size of the vector. The resulting value is then returned.

```
//Sam Christiansen
//9/23/2016
//Programing Language: C++

//Function to find the l1 Norm of a vector, with 1 parameter:
//The parameter is the vector we want the l1 Norm for
//The function returns the value of the l1 Norm
double l1Norm(MyVector u){
        double num = 0;                         //Initialing variable to be returned
        for (int i = 0; i < u.size(); i++){     //Loop for size of the vector
                num += abs(u[i]);               //Summing the absolute value of each element
        }
        return num / u.size();                  //Returning the sum divided by the vector size
}
```

Sample driver code for *l1Norm

```
int main() {
        int size = 5;                                   //Size of the vector
        MyVector u(size);
        for (int i = 0; i < size; i++)                  //Setting vector u[i] value
                u[i] = (i*i) + 2;                       //to (i*i) + 2

        double num = l1Norm(u);                         //Calling l1Norm function
        cout << "l1 Norm for vector u" << "
                = {2,3,6,11,18} is " << num << endl;    //Printing results
        return 0;
}
```

***Output:**

l1 Norm for vector u={2,3,6,11,18} is 8

Press any key to continue . . .

3.2.2) *l2Norm*

The l_2 norm, also known as the Euclidian length of a vector, is used to determine the magnitude of the given vector. To find the magnitude, this routine squares the value at each element of the vector, and then sums them all together. The routine then takes the square root of this sum and returns it as the l_2 norm.

```
//Sam Christiansen
//9/23/2016
//Programing Language: C++

//Function to find the l2 Norm of a vector, with 1 parameter:
//The parameter is the vector we want the l2 Norm for
//The function returns the value of the l2 Norm
double l2Norm(MyVector u){
        double num = 0;                         //Initialing variable to be returned
        for (int i = 0; i < u.size(); i++){     //Loop for size of the vector
                num += pow(u[i], 2);            //Summing the square of each element
        }
        return sqrt(num);                       //Returning square root of the sum
}
```

Sample driver code for *l2Norm:

```
int main() {
        int size = 5;                                   //Size of the vector
        MyVector u(size);
        for (int i = 0; i < size; i++)                  //Setting vector u[i] value
                u[i] = (i*i) + 2;                       //to (i*i) + 2

        double num = l2Norm(u);                         //Calling l2Norm function
        cout << "l2 Norm for vector u" <<
                "= {2,3,6,11,18} is " << num << endl;   //Printing results
        return 0;
}
```

***Output:**

```
l2 Norm for vector u = {2,3,6,11,18} is 22.2261

Press any key to continue . . .
```

3.2.3) *lInfinityNorm*

The l_∞ norm is used to find the maximum value held within the vector. This is done by setting the value of the first element of the vector as the temporary max, and then looping through the entire vector and comparing the value of each element with the value of the temporary max variable. As the routine is looping through the vector, if one of the elements has a greater value than the temporary max, set that value as the temporary max. After the loop is finished, the routine returns the value saved in the temporary max variable.

```
//Sam Christiansen
//9/23/2016
//Programing Language: C++

//Function to find the l∞ Norm of a vector, with 1 parameter:
//The parameter is the vector we want the l∞ Norm for
//The function returns the value of the l∞ Norm
double lInfinityNorm(MyVector u){
        int max = 0;                            //Initialing variable for the vector index
        for (int i = 0; i < u.size(); i++){     //Loop for the size of the vector
                if (u[i]>u[max])                //Checking which element is larger
                        max = i;                //If the new element is larger, set max to it
        }
        return u[max];                          //Returning the max value of the vector
}
```

Sample driver code for *lInfinityNorm:

```
int main() {
        int size = 5;                                   //Size of the vector
        MyVector u(size);
        for (int i = 0; i < size; i++)                  //Setting vector u[i] value
                u[i] = (i*i) + 2;                       //to (i*i) + 2

        double num = lInfinityNorm(u);                  //Calling lInfinityNorm function
        cout << "l2 Norm for vector u" <<
                "= {2,3,6,11,18} is " << num << endl;   //Printing results
        return 0;
}
```

***Output:**

```
l∞ Norm for vector u= {2,3,6,11,18} is 18

Press any key to continue . . .
```

3.3) Matrix Norms

This section has two types of matrix norms: the 1-norm and the ∞-norm, both of which are very similar. The 1-norm looks for the max absolute row sum, while the ∞-norm finds the max absolute column sum.

3.3.1) matrix1Norm

To find the max absolute row sum, this routine has a temporary max variable and a double for loop. As the routine loops through each row, it sums the absolute value of each element into a

temporary variable (setting the temporary variable of the first row as the temporary max). After each row, the routine compares the total of that row with the value of the temporary max, and if the total of the row is greater than the temporary max, it becomes the new temporary max. At the end of looping through the matrix, the routine returns the value of the temporary max.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to find the 1-Norm of a Real Square Matrix
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, size, is the size of the matrix.
//The function returns the value of the 1-Norm
double matrix1Norm(MyMatrix A, int size){
        double max = 0;
        double temp;
        for (int i = 0; i < size; i++){
                temp = 0;
                for (int j = 0; j < size; j++){  //temp holds the sum of the absolute value
                        temp += fabs(A(i, j));   //of each element for each row
                }

                if (temp > max)                  //If temp > max, the temp value becomes the max value
                        max = temp;
        }
        //Return the max absolute row sum
        return max;
}
```

Sample driver code for *matrix1Norm:

```
int main(){
        MyMatrix A = generateSimpleMatrix(5);                    //Creating simple matrix
        double Norm = matrix1Norm(A, 5);                         //Calling the 1norm function
        cout << "The matrix 1 Norm of the matrix A = " << endl;
        A.print();                                               //Printing Matrix
        cout << "is " << Norm << endl;                           //Printing resulting 1norm
        return 0;
}
```

***Output:**

```
The matrix 1 Norm of the matrix A =
1 1 2 3 4
2 3 4 5 6
4 5 6 7 8
6 7 8 9 10
8 9 10 11 12
is 50

Press any key to continue . . .
```

3.3.2) *matrixInfinityNorm*

Finding the matrix ∞-norm is very similar to finding the 1-norm. To find the max absolute column sum, this routine has a temporary max variable and a double for loop. As the routine loops through each column, it sums the absolute value of each element into a temporary variable (setting the temporary variable of the first column as the temporary max). After each column, the routine compares the total of that column with the value of the temporary max, and if the total of the column is greater than the temporary max, it becomes the new temporary

max. At the end of looping through the matrix, the routine returns the value of the temporary max.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++
//Problem Set 4: Infinity-Norm of a Real Square Matrix

//Function to find the Infinity-Norm of a Real Square Matrix
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, size, is the size of the matrix.
//The function returns the value of the Infinity-Norm
double matrixInfinityNorm(MyMatrix A, int size){
        double max = 0;
        double temp;
        for (int i = 0; i < size; i++){
                temp = 0;
                for (int j = 0; j < size; j++){
                        temp += fabs(A(j, i));  //temp holds the sum of the absolute value
                }                               //of each element for each column
                if (temp > max)
                        max = temp;             //If temp > max, the temp value becomes the max value
        }
        return max;                             //Return the max absolute column sum
}
```

Sample driver code for *matrixInfinityNorm:

```
int main(){
        MyMatrix A = generateSimpleMatrix(5);                   //Creating simple matrix
        double Norm = matrixInfinityNorm(A, 5);                 //Calling the infinity norm function
        cout << "The matrix 1 Norm of the matrix A = " << endl;
        A.print();                                              //Printing Matrix
        cout << "is " << Norm << endl;                          //Printing resulting infinity norm
        return 0;
}
```

***Output:**

```
The matrix 1 Norm of the matrix A =
1 1 2 3 4
2 3 4 5 6
4 5 6 7 8
6 7 8 9 10
8 9 10 11 12
is 40

Press any key to continue . . .
```

3.4) Matrix Condition Numbers

The condition number of a matrix is calculated by using the norm of the matrix. So by using different matrix norms, we can will produce different results for the matrix condition number. The general form for finding the condition number for a given matrix A is multiplying the norm of A by the norm of the inverse of A. As we have functions to find the 1-norm and the ∞-norm of a matrix, two functions have been written in this section. It is worth noting that computing the condition number of a matrix is generally not suggested, because it takes a lot of work to find the inverse of a matrix and it is usually not worth the extra work. The method to invert matrices that is used in these routines is explained in section 4.2.1 of this manual.

3.4.1) *conditionNumber1Norm:*

This routine uses the 1-norm of a matrix to compute its condition number. It first finds the inverse of the given matrix A, and then finds the 1-norm of both the given matrix A and its inverse matrix. This function then multiplies the two norms together and returns the product.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to find the Condition Number of a Square Matrix
//              using the 1-Norm
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, size, is the size of the matrix.
//The function returns the value of the 1-Norm Condition Number
double conditionNumber1Norm(MyMatrix A, int size){
        MyMatrix Ainverse = invertMatrix(A, size);              //Inverting the Matrix
        return matrix1Norm(A, size)*matrix1Norm(Ainverse, size); //Returning the product of A's 1norm
}
                                                                //A's inverse's 1norm
```

Sample driver code for *conditionNumber1Norm

```
int main(){
        MyMatrix A = generateSimpleMatrix(5);                   //Creating simple matrix
        double num = conditionNumber1Norm(A,5);                 //Calling the function
        cout << "The condition number of the matrix A = " << endl;
        A.print();                                              //Printing Matrix
        cout << "(using the 1 Norm) is " << num << endl;        //Printing the condition number
        return 0;
}
```

***Output:**

```
The condition number of the matrix A =
1 1 2 3 4
2 3 4 5 6
4 5 6 7 8
6 7 8 9 10
8 9 10 11 12
(using the 1 Norm) is 0

Press any key to continue . . .
```

3.4.2) *conditionNumberInfNorm:*

This routine uses the ∞-norm of a matrix to compute its condition number. It first finds the inverse of the given matrix A, and then finds the ∞-norm of both the given matrix A and its inverse matrix. This function then multiplies the two norms together and returns the product.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to find the Condition Number of a Square Matrix
//              using the Infinity-Norm
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, size, is the size of the matrix.
//The function returns the value of the Infinity-Norm Condition Number
double conditionNumberInfNorm(MyMatrix A, int size){
        MyMatrix Ainverse = invertMatrix(A, size);              //Inverting the Matrix
```

```
        return matrixInfinityNorm(A, size) *                    //Returning the product of A's InfNorm
                matrixInfinityNorm(Ainverse, size);             //and A's inverse's InfNorm
}
```

Sample driver code for *conditionNumberInfNorm:

```
int main(){
        MyMatrix A = generateSimpleMatrix(5);                   //Creating simple matrix
        double num = conditionNumberInfNorm(A,5);               //Calling the function
        cout << "The condition number of the matrix A = " << endl;
        A.print();                                              //Printing Matrix
        cout << "(using the Infinity Norm) is " << num << endl;  //Printing the condition number
        return 0;
}
```

***Output:**

```
The condition number of the matrix A =
1 1 2 3 4
2 3 4 5 6
4 5 6 7 8
6 7 8 9 10
8 9 10 11 12
(using the 1 Norm) is 0

Press any key to continue . . .
```

4) Basic Linear Operations

This chapter contains many of the basic linear operations that are needed to solve linear systems. Most of these routines are simple addition and multiplication rules that are learned in the first course of linear algebra. These are explained in the first section of this chapter. The second section has functions that are used to produce transform matrices.

4.1) Vector and Matrix Operations

The functions in this section include many of the product that result from multiplying different types of containers together, such as multiplying a vector by a scalar or a matrix by a vector. All of these functions return the new value, vector or matrix that resulted from the operation.

4.1.1) *dotProduct*

The dot product is a function that takes two vectors, u and v, as parameters and returns number of type double. The dot product is found by looping through each element of both vectors at the same time and multiplying each *ith* element of u by the *ith* element of v. The routine then sums together each of the *i* products. After summing all of the products together, the function returns that sum as the solution of the dot product.

```
//Sam Christiansen
//9/23/2016
//Programing Language: C++

//Function to find the dot product of 2 vector, with 2 parameters:
//The first and second parameter are the vectors supplied for the dot product
//The function returns the value of the dot product
double dotProduct(MyVector u, MyVector v){
        double num = 0;                         //Initializing variable to hold the answer
        for (int i = 0; i < u.size(); i++){     //Looping for the size of the vector
                num += u[i] * v[i];             //Summing the products of u[i] and v[i]
        }
        return num;                             //Returning the result
}
```

Sample driver code for *dotProduct:

```
int main(){
        int size = 3;                           //Initializing variable for the size
        MyVector u(size);                       //Initializing the first vector
        MyVector v(size);                       //Initializing the second vector
        for (int i = 0; i < size; i++){         //Looping to set values of vectors
                u[i] = i + 2;                   //Setting vector u
                v[i] = (i + 1) * 3;             //Setting vector v
        }
        double num = dotProduct(u, v);          //Calling dot product function
        cout << "The dot product of u" <<
                "= {2,3,4} and v = {3,6,9}" <<  //Printing the results
                " is " << num << endl;
        return 0;
}
```

***Output:**

```
The dot product of u = {2,3,4} and v = {3,6,9} is 60

Press any key to continue . . .
```

4.1.2) *crossProduct*

The cross product is function that takes two vector, u and v, as parameters and returns a new vector, w, as the solution. This specific function to find the cross product is built for vectors of size 3. The process for finding each value of the vector w is seen below in the function.

```
//Sam Christiansen
//9/23/2016
//Programing Language: C++

//Function to find the cross product of 2 vector of size 3, with 2 parameters:
//The first and second parameter are the vectors supplied for the cross product
//The function returns new vector fesulting from the cross product
MyVector crossProduct(MyVector u, MyVector v){
        MyVector w(3);                          //Declaring new vector to be returned
        double num;                             //Initializing variable to hold the values
        num = (u[1] * v[2]) - (u[2] * v[1]);    //Solving for w[0]
        w[0] = num;
        num = (u[0] * v[2]) - (u[2] * v[0]);    //Solving for w[1]
        w[1] = num;
        num = (u[0] * v[1]) - (u[1] * v[0]);    //Solving for w[2]
        w[2] = num;
        return w;                               //Returning the resulting vector, w
}
```

Sample driver code for *crossProduct:

```
int main(){
        int size = 3;                           //Initializing variable for the size
        MyVector u(size);                       //Initializing the first vector
        MyVector v(size);                       //Initializing the second vector
        for (int i = 0; i < size; i++){         //Looping to set values of vectors
                u[i] = i + 2;                   //Setting vector u
                v[i] = (i + 1) * 3;             //Setting vector v
        }
        MyVector w = crossProduct(u, v);        //Calling cross product function
        cout << "The cross product of u" <<
                "= {2,3,4} and v = {3,6,9}" <<  //Printing the results
                " is w = {" << w[0] << "," <<
                w[1] << "," << w[2] << "}" << endl;

        return 0;
}
```

***Output:**

```
The cross product of u = {2,3,4} and v = {3,6,9} is w = {3,6,3}

Press any key to continue . . .
```

4.1.3) *scalarVectorProduct*

The scalar vector product is a simple routine that takes each element of the given vector and multiplies it by the given scalar value. This is done by using a for loop, looping for the size of the vector and multiplying each *ith* element by the scalar value. After the looping is completed, the new vector is returned.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to calculate the product of a scalar and vector
//The first parameter, b, is the vector
//The second parameter, c, is the scalar value
//The third parameter, size, is the size of the vector
MyVector scalarVectorProduct(MyVector b, double c, int size){
	MyVector x = MyVector(size);	//Initializing the vector to be returned
	for (int i = 0; i < size; i++)	//For loop for the size of the vector
		x[i] = b[i] * c;	//Sets the value of the product for each element
	return x;			//Returns the solved vector
}
```

Sample driver code for *scalarVectorProduct:

```
int main(){
	int size = 3;				//Initializing variable for the size
	double scalar = 6.3;			//Initializing a scalar variable
	MyVector u(size);			//Initializing the vector
	for (int i = 0; i < size; i++){		//Looping to set values of vectors
		u[i] = i + 2;			//Setting vector u
	}
	MyVector w = scalarVectorProduct(u,	//Calling the function
		scalar, size);
	cout << "The scalar vector product of u" <<
		"= {2,3,4} and scalar = 6.3" <<	//Printing the results
		" is w = {" << w[0] << "," <<
		w[1] << "," << w[2] << "}" << endl;

	return 0;
}
```

***Output:**

```
The scalar vector product of u = {2,3,4} and scalar = 6.3 is w = {12.6,18.9,25.2}

Press any key to continue . . .
```

4.1.4) *vectorVectorAddition*

The process of adding two vectors together is very straight forward: the routine loops through both vectors and adds the *ith* element of both vectors together. The same rule is used for subtraction, and because the rule is the same, I decided to combine both operations into this one routine. The first two parameters of the function are the two vector, u and v, while the third parameter is a boolean variable, isSubtraction. If this variable is set to true, then the function will subtract the second vector from the first. If the isSubtraction variable is set to false, then

the function simply adds the two vectors together. After looping through each element and either adding or subtracting each *ith* element, the function returns the new resulting vector.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to calculate adding two vectors together
//The first parameter, a, is the first vector
//The second parameter, b, is the second vector
//The third parameter, isSubraction, is a boolean to say if
//      the function should do subtraction:
//      if true: solve a-b
//      if false: solve a+b
MyVector vectorVectorAddition(MyVector a, MyVector b, bool isSubtraction){
        MyVector c = MyVector(a.size());                //Initializing vector to be returned
        if (isSubtraction){                             //If subtraction is wanted
                for (int i = 0; i < a.size(); i++)      //Loop for the size of the vectors
                        c[i] = a[i] - b[i];             //Sets value of the difference for each element
        }
        else{                                           //If subtraction is not wanted
                for (int i = 0; i < a.size(); i++)      //Loop for the size of the vectors
                        c[i] = a[i] + b[i];             //Sets value of the sum for each element
        }
        return c;                                       //Returns the solved vector
}
```

Sample driver code for *vectorVectorAddition:

```
int main(){
        int size = 3;                           //Initializing variable for the size
        MyVector u(size);                       //Initializing the first vector
        MyVector v(size);                       //Initializing the second vector
        for (int i = 0; i < size; i++){         //Looping to set values of vectors
                u[i] = i + 2;                   //Setting vector u
                v[i] = (i + 1) * 3;             //Setting vector v
        }
        MyVector w = vectorVectorAddition(u, v,  //Calling the function
                false);                         //False is set for addition
        cout << "The vector sum of u" <<
                "= {2,3,4} and v = {3,6,9}" <<   //Printing the results
                " is w = {" << w[0] << "," <<
                w[1] << "," << w[2] << "}" << endl;

        w = vectorVectorAddition(u, v,          //Calling the function
                true);                          //True is set for subtraction
        cout << "The vector difference of u" <<
                "= {2,3,4} and v = {3,6,9}" <<   //Printing the results
                " is w = {" << w[0] << "," <<
                w[1] << "," << w[2] << "}" << endl;

        return 0;
        }
```

***Output:**

```
The vector sum of u= {2,3,4} and v = {3,6,9} is w = {5,9,13}
The vector difference of u= {2,3,4} and v = {3,6,9} is w = {-1,-3,-5}

Press any key to continue . . .
```

4.1.5) *matrixVectorProduct*

The product of a matrix and a vector results in a new vector. In addition to the matrix and vector as parameters, the size of the vector is passed into the function, as it is imported to make sure that the size of the vector matches the size of the matrix. The steps for finding each *ith* element of the solution vector can be seen in the code below.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to calculate the product of a matrix and vector, with 3 parameters:
//The first parameter, A, is the matrix provided
//The second parameter, y, is the vector provided
//The third parameter, size, is the size of the vector
//The function returns the new resulting vector
MyVector matrixVectorProduct(MyMatrix A, MyVector y, int size){
	MyVector b = MyVector(size);                    //Declaring new vector to be solved and returned
	for (int i = 0; i < size; i++){                 //Will loop 'size' times
		for (int j = 0; j < size; j++){         //Will loop 'size' times
			b[i] = b[i] + A(i, j)*y[j];     //Will sum the product for each matrix row
		}                                       //i element and vector element,
	}                                               //and saving it in b[i]
	return b;                                       //Returning the solved vector b
}
```

Sample driver code for *matrixVectorProduct:

```
int main(){
	int size = 3;                           //Initializing variable for the size
	MyMatrix A(size);                       //Initializing the matrix
	for (int i = 0; i < size; i++){
		for (int j = 0; j < size; j++){
			A(i, j) = (i + j) + 3;  //Setting values for matrix A
		}
	}
	MyVector u(size);                       //Initializing the vector
	for (int i = 0; i < size; i++){         //Looping to set values of vectors
		u[i] = i + 2;                   //Setting values for vector u
	}
	MyVector w = matrixVectorProduct(A, u,  //Calling the function
		size);
	cout << "The matrix vector product of u" <<
		"= {2,3,4} and A = " << endl << endl;
	A.print();
	cout << "is w = {" << w[0] << "," <<     //Printing the results
		w[1] << "," << w[2] << "}" << endl;

	return 0;
}
```

***Output:**

```
The matrix vector product of u = {2,3,4} and A =
3 4 5
4 5 6
5 6 7
is w = {38,47,56}

Press any key to continue . . .
```

4.1.6) *scalarMatrixProduct*

Scalar matrix multiplication follows the exact same pattern that is used in the scalarVectorProduct function. The function uses a double for loop to loop each element of the given matrix and multiplies each value of the matrix by the scalar value. The function returns a matrix with the new values at each element.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to calculate the product of a scalar and matrix
//The first parameter, B, is the matrix
//The second parameter, c, is the scalar value
//The third parameter, size, is the size of the matrix
MyMatrix scalarMatrixProduct(MyMatrix A, double c, int size){
	MyMatrix B = MyMatrix(size);	//Initializing the matrix to be returned
	for (int i = 0; i < size; i++)	//Double for loop for the size of the matrix
		for (int j = 0; j < size; j++)
			B(i,j) = B(i,j) * c; //Sets the value of the product for each element
	return B;			//Returns the solved vector
}
```

Sample driver code for *scalarMatrixProduct:

```
int main(){
	int size = 3;				//Initializing variable for the size
	MyMatrix A(size);			//Initializing the matrix
	for (int i = 0; i < size; i++){
		for (int j = 0; j < size; j++){
			A(i, j) = (i + j) + 3;	//Setting values for matrix A
		}
	}
	double scalar = 6.3;			//Initializing scalar value
	MyMatrix B = scalarMatrixProduct(A,	//Calling the function
		scalar, size);
	cout << "The scalar Matrix product" <<	//Printing the results
		"of A =" << endl << endl;
	A.print();
	cout << "and scalar 6.3 is B = " << endl;
	B.print();
	return 0;
}
```

***Output:**

```
The scalar Matrix product of A =
3  4  5
4  5  6
5  6  7
and scalar 6.3 is B =
18.9  25.2  31.5
25.2  31.5  37.8
31.5  37.8  44.1

Press any key to continue . . .
```

4.1.7) *matrixMatrixProduct*

This function takes 3 parameters: the first two are the two matrices that the user wants to multiply together, and the third is the size of the matrices. It is important to note that both matrices need to be of size nxn, where n is the same value as the size parameter. This function needs a triple for loop to properly work, and the algorithm can be seen below. After going through the three for loops, the solution matrix with the new values is returned.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to calculate the product of two matrices, with 3 parameters
//The first parameter, A, is the first matrix provided
//The second parameter, B, is the second matrix provided
//The third parameter, size, is the size of the matrices
//The function returns the resulting matrix
MyMatrix matrixMatrixProduct(MyMatrix A, MyMatrix B, int size){
        MyMatrix C = MyMatrix(size);                    //Initializing matrix to be returned
        for (int i = 0; i < size; ++i){                 //Triple for loop
                for (int j = 0; j < size; ++j){
                        for (int k = 0; k < size; ++k){
                                C(i, j) = C(i,j)        //Setting result into each element
                                        + A(i, k)*B(k, j);
                        }
                }
        }
        return C;                                       //Returning the resulting matrix
}
```

Sample driver code for *matrixMatrixProduct:

```
int main(){
        int size = 3;                                   //Initializing variable for the size
        MyMatrix A(size);                               //Initializing the first matrix
        for (int i = 0; i < size; i++){
                for (int j = 0; j < size; j++){
                        A(i, j) = (i + j) + 3;          //Setting values for matrix A
                }
        }
        MyMatrix B(size);                               //Initializing the second matrix
        for (int i = 0; i < size; i++){
                for (int j = 0; j < size; j++){
                        B(i, j) = (i + j + 1) * 2;      //Setting values for matrix B
                }
        }
        MyMatrix C = matrixMatrixProduct(A,             //Calling the function
                B, size);
        cout << "The Matrix Matrix product" <<          //Printing the results
                " of A =" << endl << endl;
        A.print();
        cout << "and B = " << endl << endl;
        B.print();
        cout << "is C = " << endl << endl;
        C.print();
        return 0;
}
```

***Output:**

```
The Matrix Matrix product of A =
3 4 5
4 5 6
5 6 7
and B =
2 4 6
4 6 8
6 8 10
is C =
52 76 100
64 94 124
76 112 148

Press any key to continue . . .
```

4.2) Matrix Transformations

The routines in this section take a matrix as a parameter and returns a transformation of the given matrix. This section has 2 matrix transformations, the first of which, invertMatrix, is really expensive to compute. It is usually suggested that you don't compute the invertMatrix function because of how expensive it is.

4.2.1) *invertMatrix*

As mentioned above, this function takes a lot of work to compute. It is generally recommended that people avoid compute the inverse of a matrix at all cost, because the cost to compute the inverse isn't worth it. But if wanted, a user can use this function below to compute the inverse of a function. The function takes two parameters: first the matrix to be inverted and second the size of the matrix. After computing, the function returns the inverted matrix.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to calculate the inverse of a matrix
//The first parameter, A, is the nonsingular nxn matrix to be inverted
//The second parameter, size, is the size of the matrix
//The function returns the inverted matrix
MyMatrix invertMatrix(MyMatrix A, int size)
{
	int n = size;
	MyMatrix B = MyMatrix(size);		//Initializing matrix to be returned

	for (int i = 0; i < n; i++){
		for (int j = 0; j < n; j++)
			B(i, j) = A(i, j);	//Copy the given matrix into this matrix
	}
	float t;
	for (int i = 0; i<n; i++)
	{
		for (int j = n; j<2 * n; j++)
		{
			if (i == j - n)
				B(i, j) = 1;
			else
				B(i, j) = 0;
		}
	}
```

```
        for (int i = 0; i<n; i++)
        {
                t = B(i, i);
                for (int j = i; j<2 * n; j++)
                        B(i, j) = B(i, j) / t;
                for (int j = 0; j<n; j++)
                {
                        if (i != j)
                        {
                                t = B(j, i);
                                for (int k = 0; k<2 * n; k++)
                                        B(j, k) = B(j, k) - t*B(i, k);
                        }
                }
        }
        return B;                                   //Return the inverted matrix
}
```

Sample driver code for *invertMatrix:

```
int main(){
        int size = 3;                               //Initializing variable for the size
        MyMatrix A = GenerateRandomSquareMatrix(size);
        MyMatrix C = invertMatrix(A,                //Calling the inverting function
                 size);
        cout << "The inverse of Matrix " <<         //Printing the results
                "A =" << endl << endl;
        A.print();
        cout << "is C = " << endl << endl;
        C.print();
        return 0;
}
```

***Output:**

```
The inverse of Matrix A =
30.7306 0.99765 0.471267
0.397382 30.5639 0.772057
0.89697 0.156774 30.3266
is C =
0.0325686 -0.00106063 -0.000479105
-0.000399165 0.0327356 -0.000827183
-0.000961218 -0.000137857 0.0329928

Press any key to continue . . .
```

4.2.2) *transposeMatrix*

Computing the transpose of a matrix is not nearly as costly as computing the inverse. This routine takes a matrix, A, as the parameter and returns the newly made matrix, B. To find the transpose, one simply needs to use a double for loop to loop through each of the elements of the matrix. Then the function sets the value of B[i,j] = A[j,i]. After the double for loop is completed, the matrix B is returned.

```
//Sam Christiansen
//11/03/2016
//Programing Language: C++

//Function to calculate the transpose of a matrix
```

```
//The one parameter is the matrix desired to be transposed
//The function returns the transposed matrix
MyMatrix transposeMatrix(MyMatrix A){
        MyMatrix B = MyMatrix(A.columnCount, A.rowCount);//Creates new matrix to be returned
        for (int i = 0; i < A.columnCount; i++)          //Double for loop for size of matrix
                for (int j = 0; j < A.rowCount; j++)
                        B(i, j) = A(j, i);               //Sets B(i,j) = A(j,i)
        return B;                                        //Returns the transposed matrix
}
```

Sample driver code for *transposeMatrix:

```
int main(){
        MyMatrix B = MyMatrix(2, 3);                     //Initializing matrix B
        for (int i = 0; i < 2; i++)
                for (int j = 0; j < 3; j++)
                        B(i, j) = i + j;                 //Setting value for element of B
        MyMatrix C = transposeMatrix(B);                 //Calling the transpose function
        cout << "The transpose of matrix B = " << endl;  //Printing the results
        B.print();
        cout << "is C = " << endl;
        C.print();
        return 0;
}
```

***Output:**

```
The transpose of matrix B =
0 1
1 2
2 3
is C =
0 1 2
1 2 3

Press any key to continue . . .
```

5) Root Finding Methods

Root finding methods are functions that find a root for a given function. Since I am writing these codes in C++, the functions that I use to test these methods are not passed in as parameters, but rather the functions are just declared in the same file as the drive r code (which is seen in all of the same driver codes below). This chapter has just one section for individual methods. A second section can be added for hybrid methods, which is a way of combining individual methods for better results.

5.1) Individual Methods

Individual methods for finding the root of functions are presented in this section. Each of the methods have pros and cons, so it is beneficial to know each of the functions in order to choose the most optimal function for each case. Since many of these functions can break in certain situations, the functions and driver codes have built in try and catch statements just in case the method fails. The algorithms for these methods were found in chapter three of the course textbook, *A First Course in Numerical Methods*.

5.1.1) *bisectionMethod*

The bisection method for finding a root of a function is the most basic of the methods in this manual. The method is given a min value, a, and max value, b, and continuously cuts the interval in half and checks if the root is between the min and center point or between the center point and max point. The method keeps cutting the interval in half until a max iterator count is hit or until the error is less than the tolerance. The pro for this method is that it will work with almost all functions. The cons are that it can be slow, and if there are 2 roots in between initial min and max points, the function will not be able to find either of them.

```
//Sam Christiansen
//10/04/2016
//Programing Language: C++

//The function to implement the Bisection Method with 4 parameters:
//The first parameter, a, is the min starting point
//The second parameter, b, is the max starting point
//The third parameter, tol, is the given tolerance
//The fourth parameter, maxIter, is the max number of iterations we will allow
double bisectionMethod(double &a, double &b, double tol, int maxIter){
        double fa = f(a);                       //Initializing the value of the function at point a
        double fb = f(b);                       //Initializing the value of the function at point b

        if (fa*fb > 0)                          //If fa*fb > 0, then we want to throw this error
                throw("There is no root (or there are multiple roots) between”
                        + “a = "+to_string(a)+" and b = " +to_string(b)+"\n");
        if (fa*fb == 0){                        //if fa*fb == 0, then at least one of them is a root
                if (fa == fb == 0)              //if both == 0, then both are roots.
                        throw("Both a = "+to_string(a)+" and b = " +to_string(b) + " are roots\n");
                else if (fa == 0)
                        return a;               //return a as the root
                else
                        return b;               //return b as the root
        }
        int iter = 0;                           //Initializing the iterator
        double error = 10 * tol;                //Initializing the error
        double c, fc;                           //Declaring variables for the 'mid' value c and
```

```
                                                // the value of the function at c
                                                //Loop continues until the error > the tolerance
                                                // or until the maxIter is reached
        while (error > tol&&iter < maxIter){
                c = (a + b) / 2;                //Setting the 'mid' value
                fc = f(c);                      //Setting value of function at 'mid' value
                if (fa*fc < 0){                 //If true, let the 'mid' value be the new max, b
                        b = c;
                        fb = fc;
                }
                else{                           //Otherwise, let the 'mid' value be the new min, a
                        a = c;
                        fa = fc;
                }
                error = abs(b - a);             //Update the error
                iter++;                         //Update the iterator
        }
        if (iter == maxIter)
                throw("Reached the Maximum Iteration count before finding a root.");
                                                //Printing results
        cout << "Number of iterations: " << iter << endl;//Print value of iter
        cout << "Error: " << error << endl;     //Print value of error
        return c;                               //Return the root value
}
```

Sample driver code for *bisectionMethod:

```
//The function f that I will use to test the Bisection Method
//f(x) = x^2 - 4
double f(double a){
        return pow(a, 2) - 4;
}

int main(){
        double root;                    //Declaring variable for root
        double a = 1;                   //Initializing the starting point
        double b = 4;                   //Initializing the ending point

        try{                            //This will print out the root if no error is thrown
                root=bisectionMethod(a, b, .00000001, 100);
                cout << "There is a root at x = " + to_string(root) << endl;
        }
        catch(string e){                //If an error is thrown, the error message will be printed
                cout << e << endl;
        }

        return 0;
}
```

****Output:***

```
Number of iterations: 29
Error: 5.58794e-009
There is a root at x = 2.000000

Press any key to continue . . .
```

5.1.2) *functionalIteration*

Functional Iteration is the first iterative method for finding a root of a function in this software manual. Simply put, the method is base off the theory that if for any function f(x) there exists a function g(x) where f(x) = 0 if and only if g(x) = x, then we can converge to a root using the

iteration of $x_{k+1} = g(x_k)$. Functional Iteration does converge pretty fast, but one down side is that it is not always easy to find a g(x) that fits the criteria.

```
//Sam Christiansen
//10/04/2016
//Programing Language: C++

//The function to implement the Functional Iteration with 3 parameters:
//The first parameter, xk, is the initial guess
//The second parameter, maxIter, is the max number of iterations we will allow
//The third parameter, tol, is the given tolerance
double functionalIteration(double xk, int maxIter, double tol){
        if (g(xk) == 0)                 //If g(xk) == 0, then xk is a root
                return xk;

        int iter=0;                     //Initializing iterator
        double error=10*tol;            //Initializing the error
        double xkp1;                    //Declaring variable for x(k+1)
                                        //Loop continues until the error > the tolerance
        while (iter<maxIter&&error>tol){ //or until the maxIter is reached
                xkp1 = g(xk);           //Running the algorithm for Functional Iteration
                error = abs(xkp1 - xk); //Updating error
                xk = xkp1;              //Updating x(k) to x(k+1) value
                iter++;                 //Updating iterator
        }
        if (iter == maxIter)            //Throw an error if the max iterations was hit
                throw("Max Iterations hit, Root not found.");
                                        //Print the results
        cout << "Number of iterations: " << iter << endl;
        cout << "Error: " << error << endl;
        return xkp1;                    //Return the root value
}
```

Sample driver code for *functionalIteration:

```
//An example function f(x) that I will use to test Functional Iteration
// f(x) = cCosh(x/4) - x, where c is some constant
double f(double a, double c){
        return c*cosh(a/4) - a;
}

//The function g(x) such that f(x)=0 if and only if g(x) = x
// g(x) = 2Cosh(x/4)
double g(double a){
        return 2*cosh(a/4);
}

int main(){
        double root;                    //Declaring variable for the root
        try{                            //If the root is found, this will print out the root
                root = functionalIteration(0, 100, .001);
                cout << "Root found at " << to_string(root) << endl;
        }
        catch (string e){               //If there is an error, the error message will be printed
                cout << e << endl;
        }
        return 0;
}
```

****Output:***

```
Number of iterations: 7
Error: 0.000644645
Root found at 2.357259

Press any key to continue . . .
```

5.1.3) *newtonsMethod*

The Newton's Method for finding a root of a function is unique among the other methods in this software manual in that it relies on the derivative of the given function to find the root. The algorithm is based on the Newton iteration, which is $x_{k+1} = x_k - f(x_k) / f'(x_k)$, where the first x_k is the initial guess given as a parameter. One down for this iteration is that if $f'(x_k) = 0$, then there will be division by 0 and the function will break. So the code below has written in a try and catch statement to check if $f'(x_k) = 0$. Also, Newton's Method will only work if the initial guess x_k is close enough to the root. So there are a couple ways that will cause Newton's Method to break, but if these conditions are properly met then this method tends to converge quickly.

```
//Sam Christiansen
//10/04/2016
//Programing Language: C++

//The function to implement Newton's Method with 3 parameters:
//The first parameter, xk, is the initial guess
//The second parameter, maxIter, is the max number of iterations we will allow
//The third parameter, tol, is the given tolerance
double newtonsMethod(double xk, int maxIter, double tol){
        double xkp1;                            //Declaring variable for x(k+1)
        double fx = f(xk);                      //Initializing variable for value of f(x)
        double fPrimeX = fPrime(xk);            //Initializing variable for value of f'(x)
        int iter = 0;                           //Initializing the iterator
        double error = 10 * tol;                //Initializing the error

                                                //Loop continues until the error > the tolerance
                                                //or until the maxIter is reached
        while (iter<maxIter && error > tol){
                if (fPrimeX == 0){              //If f'(x) == 0, then we need to throw this
                                                //error to prevent division by 0:
                        throw("For iteration #" + to_string(iter) +
                                ", f'(x) = 0, so Newton's Method was unsuccesful at finding a root.");
                }
                xkp1 = xk - (fx / fPrimeX);     //Running algorithm for Newton's Method
                iter++;                         //Updating the iterator
                error = abs(xkp1 - xk);         //Updating the error
                xk = xkp1;                      //Setting x(k) value to x(k+1) for next iteration
                fx = f(xk);                     //Setting value of f(x) for next iteration
                fPrimeX = fPrime(xk);           //Setting value of f'(x) for next iteration
        }
        if (iter == maxIter)                    //Throw an error if the max iterations was hit
                throw("Max Iterations hit, Root not found.");
                                                //Printing results
        cout << "Number of Newton iterations: " << iter << endl;
        cout << "Error: " << error << endl;                                     r
        return xkp1;                            //Returning x(k+1) for root value
}
```

Sample driver code for *newtonsMethod:

```
//The function f that I will use to test Newton's Method
// f(x) = x^2 - 4
double f(double a){
        return (a*a) - 4;
}

//The derivative of f that I will use to test Newton's Method
// f'(x) = 2x
```

```
double fPrime(double a){
	return 2 * a;
}

int main(){
	double root;			//Declaring variable for the root value
	try{				//If no error occurs, this will print the root
		root = newtonsMethod(1, 25, .00001);
		cout << "Root found at x = " << to_string(root) << endl;
	}
	catch(string e){	//If an error occurs, this will print the error
		cout << e << endl;
	}
}
```

***Output:**

```
Number of Newton iterations: 5
Error: 9.29223e-008
Root found at x = 2.000000

Press any key to continue . . .
```

5.1.4) *secantMethod*

The Secant Method for finding a root of a function is similar to the Newton's Method, but instead of using the derivate of the function (as done in Newton's Method), this method uses an approximation of the derivative. This is useful if the derivate is not provided. The algorithm takes two initial guesses, x_0 and x_1, and runs them in the following secant iteration: $x_{k+1} = x_k - (f(x_k)(x_k - x_{k-1})) / (f(x_k) - f(x_{k-1}))$. Just as with Newton's Method, there is a possibility of division by 0, so there is a try catch statement to catch the error if it occurs.

```
//Sam Christiansen
//10/04/2016
//Programing Language: C++

//The function to implement the Secant Method with 4 parameters:
//The first parameter, x0, is the first initial guess
//The second parameter, x1, is the second initial guess
//The third parameter, tol, is the given tolerance
//The fourth parameter, maxIter, is the max number of iterations we will allow
double secantMethod(double x0, double x1, double tol, int maxIter){
	double fkm1 = f(x0);			//Initializing variable for the functions value at x(k-1)
	double fk = f(x1);			//Initializing variable for the functions value at x(k)
	double xkp1;				//Declaring variable for x(k+1)
	double error = 10 * tol;		//Initializing the error
	int iter = 0;				//Initializing the iterator

						//Loop continues until the error > the tolerance
						//or until the maxIter is reached
	while (iter < maxIter&&error > tol){
		if ((fk-fkm1)==0){		//If f(x(k))-f(x(k-1)) == 0, then we need to throw
						//this error to prevent division by 0:
			throw("For iteration #" + to_string(iter) +
				", f'(x) = 0, so Newton's Method was unsuccesful at finding a root.");
		}
						//Running algorithm for Secant Method
		xkp1 = x1 - ((fk)*(x1 - x0)) / (fk - fkm1);
		error = abs(xkp1 -x1);	//Updating the error
		iter++;				//Updating the iterator
		fkm1 = fk;			//Setting value of f(x(k-1)) to f(x) for the next iteration
```

```
                fk = f(xkp1);           //Setting value of f(x) to f(x(k+1)) for the next iteration
                x0 = x1;                //Setting value of x(k-1) to x(k) for the next iteration
                x1 = xkp1;              //Setting value of x(k) to x(k+1) for the next iteration
        }
                                        //Printing results
        cout << "Number of iterations: " << iter << endl;
        cout << "Error: " << error << endl;
        return x1;                      //Return x(k+1) for the root value
}
```

Sample driver code for *secantMethod:

```
//The function f that I will use to test the Secant Method
// f(x) = x^2 - 4
double f(double a){
        return (a*a) - 4;
}

//Driver code:
int main(){
        double root;            //Declaring variable for the root
        try{                    //If no error occurs, this will print out the root
                root = secantMethod(1, 10, .0000001, 100);
                cout << "Root found at " << root << endl;
        }
        catch(string e){        //If an error occurs, this will print out the error
                cout << e << endl;
        }
        return 0;
}
```

****Output:***

```
Number of iterations: 8
Error: 5.02999e-010
Root found at 2

Press any key to continue . . .
```

6) Solving Linear Systems

This chapter is dedicated to methods that can be used to solve the linear system Ax = b. This chapter is the most reliant on the other methods already presents in the previous chapters of this software manual. It is split into three sections: first, direct methods; second, iterative methods; and lastly, methods using pivoting.

In order to test each of these methods, in the driver I create a matrix A and a vector x. The vector x is then filled with 1's everywhere, and then I create a vector b and set it equal to the result of the *matrixVectorProduct* (from section 4.1.5). This creates the linear system Ax = b, and A and b are the parameters passed into the methods. When this system is solved for x, if the method worked correctly then x will be full of 1's. So seeing a vector full of 1's at the end of the output lines will signify that the method worked correctly.

6.1) Direct Methods

These direct methods are some of the more straight-forward methods for solving linear systems. Some of these functions need to be used together (such as back substitution and Gaussian Elimination), and that is why there are addition routines that combine these (such as *solveWithGEandBS*).

6.1.1) *backSubstitution*

Back substitution takes two main parameters to solve a linear system: first, an upper triangular matrix, and second, the corresponding right-hand side vector b. This function is commonly paired with Gaussian Elimination (which is explained in 6.1.3), and therefore the sample driver code and output code for this function will not presented in this section, but rather it will be presented in 6.1.4 with the *solveWithGEandBS* routine.

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++

//Function to implement the Back Substitution method with 3 paramenters:
//The first parameter, A, is an Upper Triangular Matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
MyVector backSubstitution(MyMatrix A, MyVector b, int size){
        int n = size;                                   //Set n to size
        MyVector x = MyVector(size);            //Declaring new vector to be returned
        double s;                                       //Declaring variable for the factor to
                                                        //be used in each answer

        for (int i = n - 1; i >= 0; i--){               //This will loop n times
                s = 0;                                  //Setting the factor variable to 0
                for (int j = i + 1; j < n; j++){ //This will loop j times, depending
                                                        //on the i row
                        s = s + A(i, j)*x[j];   //Updating the factor variable
                }
                x[i] = (b[i] - s) / A(i, i);    //Solving x[i] using the factor variable
        }
        return x;                                       //Return the solved vector variable
}
```

***Sample driver code for *backSubstitution* is shown in 6.1.3.**

6.1.2) *forwardSubstitution*

Forward substitution is very similar to the back substitution method. It also takes a matrix A and right-hand side vector b, but this matrix A is a lower triangular matrix. The algorithm for computing this code follows the same pattern as back substitution, but it just solves a lower triangular matrix instead of an upper triangular matrix.

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++

//Function to implement the Forward Substitution method with 3 paramenters:
//The first parameter, A, is an Lower Triangular Matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
MyVector forwardSubstitution(MyMatrix A, MyVector b, int size){
        int n = size;                           //Setting n to the size
        MyVector x = MyVector(size);            //Declaring new vector to be solved

        x[0] = b[0] / A(0, 0);                  //Solving the first element x
        for (int i = 1; i < size; i++){         //This will loop n-1 times
                x[i] = b[i];                    //Updating the new x[i] to be used below
                for (int j = 0; j < i; j++){    //This will loop j times, depending on the i row
                        x[i] = x[i] - A(i, j)*x[j];//Updating the x[i] again according to the algorithm
                }
                x[i] = x[i] / A(i, i);          //Giving x[i] its final, solved value
        }

        return x;                               //Returning the solved vector
}
```

6.1.3) *gaussianElimination*

Gaussian Elimination is the process which produces an upper triangular matrix. It takes a matrix A, a right-hand side vector b, and the size of the vector as parameters. This function does not return anything because the A and b parameters are passed in by reference, meaning that when these variables are changed in the function, the changes persist to the original variables. Since this method produces an upper triangular matrix, it is usually paired with the back substitution method. Because of this connection, the routine 6.1.4 combines this method with back substitution to show how the combination solves a linear system. Therefore, sample output for Gaussian Elimination will not be provided here, but rather in section 6.1.4.

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++
//Problem Set 3: Gaussian Elimination Method

//Function to implement the Gaussian Elimination method with 3 paramenters:
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
void gaussianElimination(MyMatrix& A, MyVector& b, int size){
        int n = size;                                   //Sets n to size
        double factor = 0;                              //Declaring factor variable
```

```
        for (int k = 0; k < n-1; k++){                  //Will loop n-1 times
                for (int i = k+1; i < n; i++){          //Will loop i times depending on the row k
                        factor = A(i, k) / A(k, k);     //Updating factor variable according to algorithm
                        for (int j = 0; j < n; j++){    //Will loop n times for each element in the row
                                A(i, j) = A(i, j) - factor*A(k, j);     //Setting each matrix element to
                        }                                                //the solved value

                        b[i] = b[i] - factor*b[k];      //Setting each vector element to the solved value
                }
        }                                               //This doesn't return anything since the matrix
}                                                       //and vector were passed by reference
```

***Sample driver code for *gaussianElimination* is shown in 6.1.3.**

6.1.4) *solveWithGEandBS*

This routine is the combination of the Gaussian Elimination and back substitution routines. It takes three parameters. First, it takes the matrix A of the linear system. Second, it takes the right-hand side vector b. Lastly, it takes an integer size, which is the length of b. At the beginning of the function, the original matrix A and vector b are printed to the console. A is printed to the console again after Gaussian Elimination is complete, and then lastly the solution vector x is printed.

```
//Sam Christiansen
//10/15/2016
//Programing Language: C++

//Function to implement the Gaussian Elimination method with 3 paramenters:
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
void SolveWithGEandBS(MyMatrix A, MyVector b, int size){
        cout << "Matrix before Solving: " << endl;
        for (int i = 0; i < size; i++){                 //Printing the starting matrix and vector
                for (int j = 0; j < size; j++){         //  before solving
                        cout << A(i, j) << " ";
                }
                cout << "\t" << b[i] << endl;
        }
        cout << endl;                                   //End of printing starting matrix

        gaussianElimination(A, b, size);                //Calling GE function
        MyVector x = backSubstitution(A, b, size);      //Calling BS function on modified A and b
        cout << "Solution: " << endl;                   //Printing out the solved vector x
        for (int i = 0; i < size; i++){
                cout << x[i] << endl;
        }
        cout << endl;                                   //End of printing solved vector
}
```

Sample driver code for *solveWithGEandBS:

```
int main(){
        int size = 3;                                   //Initializing size
```

```
        MyMatrix A = GenerateRandomSquareMatrix(size);   //Create matrix
        MyVector y = MyVector(size);                     //Creates a vector
        for (int i = 0; i < size; i++){                  //Loops for 'size' times
                y[i] = 1;                                //Sets each element in vector v to 1
        }
        MyVector b = MatrixVectorProduct(A, y, size);    //Finds b to run the test

        SolveWithGEandBS(A, b, size);                    //Calls the function
        return 0;
}
```

***Output:**

```
Matrix before Solving:
30.3598 0.883358 0.511002   31.7541
0.137822 30.9129 0.565935   31.6166
0.337809 0.332255 30.4085   31.0785

Matrix after Gaussian Elimination
30.3598 0.883358   0.511002          31.7541
0          30.9089 0.563615          31.4725
0          0                30.3969  30.3969

Solution:
1
1
1

Press any key to continue . . .
```

6.1.5) *modifiedGramSchmidtOrthogonalization*

This is orthogonalization that is required to solve a system with QR Factorization. It causes the matrix to be in an orthogonal form.

```
//Function to calculate the QR factorization of a matrix using
//      the Modified Gram-Schmidt Orthogonalization, with 4 parameters
//The first parameter, A, is the given matrix
//The second parameter, Q, is an empty matrix that is passed by reference
//The third parameter, R, is also an empty matrix passed by reference
//The fourth parameter, size, is the size of the matrix.
//The function doesn't return anything, but the Q and R matrices are passed
//      by reference, so changes made to them in this function will persist outside
//  of this function.
//Source for this code: http://www.cplusplus.com/forum/general/88888/
void modifiedGramSchmidtOrthogonalization(MyMatrix A, MyMatrix &Q, MyMatrix &R, int size){
        for (int k = 0; k < size; k++){                          //Main loop
                for (int i = 0; i < size; i++)                   //These three lines compute the norm
                        R(k, k) = R(k, k) + A(i, k) * A(i, k);   //      of A, which is then saved in
                R(k, k) = sqrt(R(k, k));                         //      R(k,k)

                for (int i = 0; i < size; i++)                   //Solves each element of one
                        Q(i, k) = A(i, k) / R(k, k);             //      column of Q

                for (int j = k + 1; j < size; j++) {
                        for (int i = 0; i < size; i++)
                                R(k, j) += Q(i, k) * A(i, j);    //Finds non-diagonal values for R
                        for (int i = 0; i < size; i++)
                                A(i, j) = A(i, j) - R(k, j) * Q(i, k);//Updates A for the next loop
                }
        }
}
```

Sample driver code for *modifiedGramSchmidtOrthogonalization:

```
int main(){
        int size = 5;
        MyMatrix A = MyMatrix(5);
        A = generateRandomSquareMatrix(5);
        cout << "Printing Matrix A: " << endl;
        A.print();
        MyMatrix Q(5);
        MyMatrix R(5);
        cout << "Computing Orthogonalization..." << endl;
        modifiedGramSchmidtOrthogonalization(A, Q, R, size);
        cout << "Printing Matrix Q: " << endl;
        Q.print();
        cout << "Printing Matrxi R: " << endl;
        R.print();
        return 0;
}
```

***Output:**

```
Printing Matrix A:
50.9471 0.110965 0.580096 0.512375 0.530015
0.0332652 50.3563 0.884335 0.321879 0.905057
0.0867641 0.39201 50.4929 0.936766 0.476547
0.410749 0.944884 0.839564 50.6425 0.338481
0.738395 0.16834 0.209571 0.93173 50.2098

Computing Orthogonalization...
Printing Matrix Q:
0.999861 -0.000864889 -0.00188289 -0.00827811 -0.0143348
0.000652845 0.999788 -0.00809318 -0.0186695 -0.0029414
0.00170279 0.00777787 0.999829 -0.0162713 -0.00363115
0.00806114 0.0187353 0.0160405 0.999498 -0.0181622
0.0144914 0.00329781 0.00387214 0.0179254 0.999721

Printing Matrxi R:
50.9542  0.154549  0.676376  0.935847 1.26168
0        50.3669   1.29279   1.28053  1.08004
0        0         50.4903   1.74898  0.667991
0        0         0         50.6083  1.2093
0        0         0         0        50.1777

Press any key to continue . . .
```

6.1.6) *solveWithQR*

This function takes what is done with the Modified Gram-Schmidt Orthogonalization function and computes the rest of the steps necessary to find the solution of the linear system. The steps are outlined on page 156 of the textbook.

```
//Function to solve a linear system, Ax = b, with QR Factorization
//      with 3 parameters
//The first parameter, A, is the given matrix
//The second parameter, b, is the right-hand side vector
//The third parameter, size, is the size of the vector
//The function returns the solved vector x.
MyVector solveWithQR(MyMatrix A, MyVector b, int size){
        MyMatrix Q(size), R(size);                              //Initializing the Q and R matrices
        modifiedGramSchmidtOrthogonalization(A, Q, R, size);    //Calling the QR factorization function
        MyMatrix transposeQ = transposeMatrix(Q);               //Transposing matrix Q
        MyVector c = matrixVectorProduct(transposeQ, b, size);  //Solving c = (Q^T)b
        return backSubstitution(R, c, size);                    //Solving and returning Rx = c
```

```
}
```

Sample driver code for *solveWithQR:

```
int main(){
	int size = 5;
	MyMatrix A = MyMatrix(size);
	A = generateRandomMatrix(size);
	MyVector x(size);
	for (int i = 0; i < size; i++)
		x[i] = 1;
	MyVector b = matrixVectorProduct(A, x, size);
	cout << "Printing Matrix A: " << endl;
	A.print();
	cout << "Printing original x: " << endl;
	x.print();
	cout << "Printing b, result of A*x: " << endl;
	b.print();
	cout << "Solving with QR Factorization: " << endl;
	x = solveWithQR(A, b, size);
	cout << "Printing resulting x: " << endl;
	x.print();
	return 0;
}
```

Output:

```
Printing Matrix A:
8.57875 70.8304 45.848 73.3207 69.7989
84.2616 57.4725 53.2334 74.1997 73.2261
80.3644 26.3131 96.2737 83.2453 20.9998
94.1008 18.5766 55.1714 41.2305 67.7511
67.4398 80.5872 4.64797 91.5098 55.1012

Printing original x:
1
1
1
1
1

Printing b, result of A*x:
268.377
342.393
307.196
276.83
299.286

Solving with QR Factorization:
Printing resulting x:
1
1
1
1
1

Press any key to continue . . .
```

6.2) Iterative Methods

Using iterative methods is another way that we can solve linear systems. It is important to note that the routines presented in section 6.1 will generally be the more efficient choice for solving

many linear systems, but these iterative methods do have their place. The routine in 6.2.1 will visit the algorithm called Jacobi Iteration, while 6.2.2 will focus on the Gauss-Seidel Iteration algorithm, both of which use the iteration $\mathbf{x}_{k+1} = \mathbf{x}_k + M^{-1}\mathbf{r}_k$ as their algorithm, where M is variable that differs for the two routines. The third and last of the iteration methods, 6.2.3, is the Conjugate Gradient method.

6.2.1) *jacobiIteration*

For Jacobi Iteration, the M of the main iteration explained above is set as D, which is the diagonal matrix of A (consisting of the diagonal elements of A). This gives the following main iteration algorithm: $\mathbf{x}_{k+1} = \mathbf{x}_k + D^{-1}\mathbf{r}_k$. This routine takes three parameters, the first two being A and b respectively from the linear equation Ax = b, and the third parameter is x0, which is the initial guess. The first thing that the function does is it creates a vector $\mathbf{x}_{k-1}$ and fills its elements with the initial guess, x0.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to solve a system of linear equations, Ax=b using
//    Jacobi Iteration with 3 parameters
//The first parameter, A, is the matrix from Ax=b
//The second parameter, b, is the right hand side vector from Ax=b
//The third parameter, x0, is a value for the initial guess
MyVector jacobiIteration(MyMatrix A, MyVector b, double x0){
        int n = b.size();                                       //Setting n to the size
        int iter = 0, maxIter = 20;                             //Initializing the iterator and maxIter
        double tol = .0000001;                                  //Initializing the tolerance
        double error = tol * 10, temp;                          //Initializing the error and temp error
        MyVector xk = MyVector(n);                              //Initializing the vector for x k
        MyVector xkm1 = MyVector(n);                            //Initializing the vector for x k-1
        for (int i = 0; i < n; i++)                             //Setting each value for the x k-1 to
                xkm1[i] = x0;                                   //      the initial guess, x0
        while (iter<maxIter&&error>tol){                        //Main while loop
                for (int i = 0; i < n; i++){                    //Main for loop, for the size of vector
                        xk[i] = b[i];
                        for (int j = 0; j < i - 1; j++){        //Loop for values below
                                                                //      the main diagonal
                                xk[i] = xk[i] - A(i, j)*xkm1[j]; //Main iteration
                        }
                        for (int j = i + 1; j < n; j++){        //Loop for the values above
                                                                //      the main diagonal
                                xk[i] = xk[i] - A(i, j)*xkm1[j]; //Main iteration
                        }
                        xk[i] = xk[i] / A(i, i);                //Last step of iteration
                        temp = abs((xkm1[i]-xk[i])/xk[i]);      //Check the error for the vector at i
                        if (temp > error)                       //If this temp error is bigger than
                                error = temp;                   //the existing error, set it as error
                        xkm1[i] = xk[i];                        //set x k-1 to x k for next iteration
                }
                iter++;                                         //Add a counter
        }
        if (iter == maxIter)                                    //Shows if the maxIter was reached
                cout << "Max Iterations of " << maxIter << " was reached." << endl;
        return xk;                                              //Return the solved vector
}
```

Sample driver code for *jacobiIteration:

```
int main(){
	int size = 5;                                    //Initializing size
	MyMatrix A = GenerateRandomSquareMatrix(size);   //Initializing matrix
	MyVector x = MyVector(size);
	for (int i = 0; i < size; i++)                   //Initializing vector
		x[i] = 1;
	MyVector b = matrixVectorProduct(A, x, size);    //Setting vector to use

	cout << "Printing matrix A: " << endl;           //Printing the matrix
	A.print();
	cout << "Printing vector b: " << endl;           //Printing b
	b.print();
	cout << "Printing original vector x: " << endl;  //Print original x
	x.print();

	x = jacobiIteration(A, b, 4);                    //Calling the function
	cout << "Printing newly solved x: " << endl;     //Printing solved vector
	x.print();
	return 0;
}
```

Output:

```
Printing matrix A:
50.5665 0.102237 0.651662 0.693228 0.810175
0.187109 50.3356 0.579302 0.745781 0.195624
0.196142 0.0549028 50.206 0.713553 0.531693
0.188147 0.484085 0.20954 50.5252 0.358165
0.757927 0.399457 0.709555 0.171606 50.9388

Printing vector b:
51.8958
51.3763
52.3561
52.8494
52.8345

Printing original vector x:
1
1
1
1
1

Max Iterations of 20 was reached.
Printing newly solved x:
0.999793
1.00183
1.01138
1.01408
1.00691

Press any key to continue . . .
```

6.2.2) *gaussSeidelIteration*

For Jacobi Iteration, the M of the main iteration explained above is set as E, which is the lower triangular matrix of A. This gives the following main iteration algorithm: $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathrm{E}^{-1}\mathbf{r}_k$.

This routine takes three parameters, the first two being A and b respectively from the linear equation Ax = b, and the third parameter is x0, which is the initial guess. The first thing that the function does is it creates a vector $\mathbf{x}_{k-1}$ and fills its elements with the initial guess, x0.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to solve a system of linear equations, Ax=b using
//    Gauss-Seidel Iteration with 3 parameters
//The first parameter, A, is the matrix from Ax=b
//The second parameter, b, is the right hand side vector from Ax=b
//The third parameter, x0, is a value for the initial guess
MyVector gaussSeidelIteration(MyMatrix A, MyVector b, double x0){
        int n = b.size();                                       //Setting n to the size
        int iter = 0, maxIter = 20;                             //Initializing the iterator and maxIter
        double tol = .0000001;                                  //Initializing the tolerance
        double error = tol * 10, temp;                          //Initializing the error and temp error
        MyVector xk = MyVector(n);                              //Initializing the vector for x k
        MyVector xkm1 = MyVector(n);                            //Initializing the vector for x k-1
        for (int i = 0; i < n; i++)                             //Setting each value for the x k-1 to
                xkm1[i] = x0;                                   //      the initial guess, x0

        while (iter<maxIter&&error>tol){                        //Main while loop
                for (int i = 0; i < n; i++){                    //Main for loop, for the size vector
                        xk[i] = b[i];
                        for (int j = 0; j < i - 1; j++){        //Loop for values below
                                                                //      the main diagonal
                                xk[i] = xk[i] - A(i, j)*xk[j];  //Main iteration
                        }
                        for (int j = i + 1; j < n; j++){        //Loop for the values above
                                                                //      the main diagonal
                                xk[i] = xk[i] - A(i, j)*xk[j];  //Main iteration
                        }
                        xk[i] = xk[i] / A(i, i);                //Last step of iteration
                        temp = abs((xkm1[i]-xk[i])/xk[i]);      //Check the error for the vector at i
                        if (temp > error)                       //If this temp error is bigger than
                                error = temp;                   //the existing error, set it as error
                        xkm1[i] = xk[i];                        //set x k-1 to x k for next iteration
                }
                iter++;                                         //Add a counter
        }
        if (iter == maxIter)                                    //Shows if the maxIter was reached
                cout << "Max Iterations of " << maxIter << " was reached." << endl;
        return xk;                                              //Return the solved vector
}
```

Sample driver code for *gaussSeidelIteration:

```
int main(){
        int size = 5;                                   //Initializing size
        MyMatrix A = GenerateRandomSquareMatrix(size);  //Initializing matrix
        MyVector x = MyVector(size);
        for (int i = 0; i < size; i++)                  //Initializing vector
                x[i] = 1;
        MyVector b = matrixVectorProduct(A, x, size);   //Setting vector to use

        cout << "Printing matrix A: " << endl;          //Printing the matrix
        A.print();
        cout << "Printing vector b: " << endl;          //Printing b
        b.print();
        cout << "Printing original vector x: " << endl; //Print original x
        x.print();
```

```
        x = gaussSeidelIteration(A, b, 4);              //Calling the function
        cout << "Printing newly solved x: " << endl;    //Printing solved vector
        x.print();
        return 0;
}
```

***Output:**

```
Printing matrix A:
50.6651 0.738212 0.18305 0.518601 0.700003
0.92233 50.3702 0.910092 0.981811 0.911985
0.915708 0.614917 50.0746 0.383831 0.165654
0.198675 0.338908 0.113804 50.8356 0.286019
0.385998 0.65157 0.188086 0.908414 50.1298

Printing vector b:
53.0878
52.7138
51.4697
53.6283
52.1934

Printing original vector x:
1
1
1
1
1

Max Iterations of 20 was reached.
Printing newly solved x:
0.999342
1.01432
1.01814
1.00718
1.00539

Press any key to continue . . .
```

6.2.3) *conjugateGradientMethod*

The Conjugate Gradient method is a method that works well with matrices that are symmetric positive definite. Just like most of the other routines, the first two parameters are the matrix A (which, in this case, must be a symmetric positive definite matrix) and right-hand side vector b. The third parameter is the initial guess, which is set in the same way as the previous two routines.

```
//Sam Christiansen
//11/27/2016
//Programing Language: C++

//Function to solve a system of linear equations, Ax=b using the
//    Conjugate Gradient Method with 3 parameters
//The first parameter, A, is the matrix from Ax=b
//The second parameter, b, is the right hand side vector from Ax=b
//The third parameter, initGuess, is a value for the initial guess
MyVector conjugateGradientMethod(MyMatrix A, MyVector b, double initGuess){
        int size = b.size();                            //Setting size
        double tol = .0000001;                          //Initializing the tolerance
```

```
        MyVector xk = MyVector(size);                   //Initializing x k for initial guess
        for (int i = 0; i < size; i++)
                xk[i] = initGuess;                      //Setting each value of x k to the initial guess
        MyVector xkp1 = MyVector(size);                 //Initializing vector for x k+1
        MyVector rk = vectorVectorAddition(b,           //Initializing the first value of r k, r0, to
                matrixVectorProduct(A, xk, size),       //      r0 = b-Ax0
                true);                                  //true is set because subtraction is wanted
        MyVector rkp1 = MyVector(size);                 //Initializing vector for r k+1
        MyVector sk = MyVector(size);                   //Initializing vector for s k
        MyVector pk = rk;                               //Initializing vector for p k and settings to r0
        MyVector pkp1 = MyVector(size);                 //Initializing vector for p k+1
        double deltaK = dotProduct(rk, rk);             //Initializing δ k, δ0 = <r0,r0>
        double deltaKp1;                                //Initializing δ k+1
        double alphaK;                                  //Initializing α k
        double bDelta = dotProduct(b, b);               //Initializing bδ, bδ = <b,b>
        double totalTol = tol*tol*bDelta;               //Setting total tolerance for main loop
        int maxIter = 100, iter = 0;                    //Initializing iterators

        while (deltaK > totalTol&&iter<maxIter){        //Main while loop
                sk = matrixVectorProduct(A, pk, size);  //Updating sk, sk = A*pk
                alphaK = deltaK / dotProduct(pk, sk);   //Updating αk, αk = δ k / <pk,sk>
                xkp1 = vectorVectorAddition(xk,         //Main Iteration, x k+1 = xk + αk*pk
                        scalarVectorProduct(pk, alphaK, size),
                        false);                         //False is set because addition is wanted
                rkp1 = vectorVectorAddition(rk,         //Updating r k+1, r k+1 = rk - αk*sk
                        scalarVectorProduct(sk, alphaK, size),
                        true);                          //True is set because subtraction is wanted
                deltaKp1 = dotProduct(rkp1, rkp1);      //Updating δ k+1, δ k+1 = <r k+1, r k+1>
                pkp1 = vectorVectorAddition(rkp1,       //Updating p k+1, p k+1 = r k+1 + (δ k+1/δ k)*pk
                        scalarVectorProduct(pk, deltaKp1 / deltaK, size),
                        false);                         //False is set because addition is wanted

                deltaK = deltaKp1;                      //Set δ k to δ k+1 for next iteration
                xk = xkp1;                              //Set x k to x k+1 for next iteration
                rk = rkp1;                              //Set r k to r k+1 for next iteration
                pk = pkp1;                              //Set p k to p k+1 for next iteration
                iter++;                                 //Update iteration counter
        }
        cout << "Num of iter: " << iter << endl;        //Prints the number of iterations needed to solve
        return xkp1;                                    //Returns the solved vector
}
```

Sample driver code for *conjugateGradientMethod:

```
int main(){
        int size = 5;                                   //Initializing size
        MyMatrix A = GenerateRandomSquareMatrix(size);  //Initializing matrix
        MyVector x = MyVector(size);
        for (int i = 0; i < size; i++)                  //Initializing vector
                x[i] = 1;
        MyVector b = matrixVectorProduct(A, x, size);   //Setting vector to use

        cout << "Printing matrix A: " << endl;          //Printing the matrix
        A.print();
        cout << "Printing vector b: " << endl;          //Printing b
        b.print();
        cout << "Printing original vector x: " << endl; //Print original x
        x.print();

        x = conjugateGradientMethod(A, b, 4);           //Calling the function
        cout << "Printing newly solved x: " << endl;    //Printing solved vector
        x.print();
        return 0;
}
```

***Output:**

```
Printing matrix A:
50.6791 0.515 0.551134 0.594928 0.472304
0.844478 50.6681 0.0982391 0.560625 0.0536515
0.239662 0.108249 50.6622 0.104862 0.315928
0.00824 0.88403 0.807337 50.6573 0.255593
0.454604 0.384198 0.00320444 0.416639 50.5204

Printing vector b:
52.2261
52.5595
52.1221
52.3343
51.6179

Printing original vector x:
1
1
1
1
1

Num of iter: 4
Printing newly solved x:
1
1
1
1
1

Press any key to continue . . .
```

6.3) Methods using Pivoting

This section uses two of the direct methods of solving linear systems and adds scaled partial pivoting to the functions. This helps tremendously with the stability of the routines, especially if a given matrix A has a zero on one of the main diagonal elements. Pivoting obviously adds more complexity to computing the solution of linear systems, but sometimes it is necessary to ensure a correct solution.

6.3.1) *LUwithScaledPartialPivoting*

LU factorization is a direct method that is not included above in section 6.1, but it is declared here with the SPP pivoting strategy. The first to parameters, A and b, are passed in by reference, so any changes in the function will persist outside of the function. The last parameter is an integer size, which is the length of the vector b.

```
//Sam Christiansen
//10/26/2016
//Programing Language: C++
//Problem Set 1: LU Factorization with Scaled Partial Pivoting

//Function to implement the LU decomposition using Scaled Partial Pivoting with 3 paramenters:
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
void LUwithScaledPartialPivoting(MyMatrix& A, MyVector& b, int size){
```

```
	int* pvt = new int[size];                                   //Pivoting vector
	double* scale = new double[size];                           //Scaling vector
	for (int i = 0; i < size; i++)                              //Initializing pivoting vector
		pvt[i] = i;

	for (int i = 0; i < size; i++){                             //Loops 'size' times
		scale[i] = 0;                                           //Initialize scale to 0
		for (int j = 0; j < size; j++){                         //Loops 'size' times
			if (fabs(scale[i]) < fabs(A(i, j)))                 //if the next position is larger,
				scale[i] = fabs(A(i, j));                       //set it for the scale value
		}
	}

	double temp;                                                //Variable to check each position
	for (int k = 0; k < size - 1; k++){                         //Main loop
		int pc = k;                                             //pivot column
		double aet = fabs(A(pvt[k], k) / scale[k]);
		for (int i = k + 1; i < size; i++){                     //Loops 'size'-1 times
			temp = fabs(A(pvt[i], k) / scale[i]);
			if (temp > aet){                                    //Comparing each position with aet
				aet = temp;                                     //If temp>aet, set aet = to temp
				pc = i;                                         //and set pivot column to i
			}
		}
		if (pc != k){                                           //If pivot column was changed, we need
			int tempNum = pvt[k];                               //to switch the pivot values
			pvt[k] = pvt[pc];
			pvt[pc] = tempNum;
		}
		for (int i = k + 1; i < size; i++){                     //Now we will shift the rows in A
			if (A(pvt[i], k) != 0){
				double factor = A(pvt[i], k) / A(pvt[k], k);
				A(pvt[i], k) = factor;
				for (int j = k + 1; j < size; j++)
					A(pvt[i], j) -= factor*A(pvt[k], j);
			}
		}
	}
	cout << "Printing after SPP: " << endl;                     //Printing to console to see result of
	A.print();                                                  //Scaled Partial Pivoting
	b.print();
                                                                //Now computing the LU Decomposition
	for (int i = 1; i < size; i++)                              //First find Ly = b
		for (int j = 0; j < i; j++)
			b[pvt[i]] -= A(pvt[i], j)*b[pvt[j]];
                                                                //Now find Ux = y
	MyVector xx = MyVector(size);                               //Creating solution vector
	for (int i = size - 1; i >= 0; i--){
		for (int j = i + 1; j < size; j++)
			b[pvt[i]] -= A(pvt[i], j)*xx[j];
		xx[i] = b[pvt[i]] / A(pvt[i], i);
	}
	cout << "Printing solution vector: " << endl;               //Printing solution to console
	xx.print();
}
```

Sample driver code for *LUwithScaledPartialPivoting:

```
int main(){
	int size = 3;                               //Creates a random symmetric matrix
	MyMatrix B = GenerateRandomSymmetricSquareMatrix(size);
	cout << "Printing Randomly Generated Square Matrix with Size " << size << ":"
		<< endl;
	B.print();                                  //Prints the new random matrix
	MyVector v = MyVector(size);                //Creates a vector
```

```
	for (int i = 0; i < size; i++){			//Loops for 'size' times
		v[i] = 1;				//Sets each element in vector v to 1
	}
	MyVector b = MatrixVectorProduct(B, v, size);//Solves Matrix Vector Product of
							//B and v
	b.print();					//Prints solution to Matrix Vector
							//product, b
	LUwithScaledPartialPivoting(B, b, size); //Solves B using our LU with SPP function
}
```

6.3.2) *GEwithScaledPartialPivoting*

Since a routine to solve a linear system with Gaussian Elimination is presented in section 6.1.4, this function needs only to compute the scaled partial pivoting on the given matrix A and vector b, and then the adjusted A and b are simply plugged into the *solveWithGEandBS* routine. This function prints the matrix A after the scaled partial pivoting is completed to show the results of the pivoting strategy.

```
//Sam Christiansen
//10/26/2016
//Programing Language: C++
//Problem Set 2: Gaussian Elimination and Back-Substitution with
//                              Scaled Partial Pivoting

//Function to implement the GE and BS using Scaled Partial Pivoting with 3 paramenters:
//The first parameter, A, is a real, nonsingular nxn matrix.
//The second parameter, b, is a right-hand-side vector.
//The third parameter, size, is the length of the square matrix A and the size of b
void GEwithScaledPartialPivoting(MyMatrix& A, MyVector& b, int size){
	int* pvt = new int[size];			//Pivoting vector
	double* scale = new double[size];		//Scaling vector
	for (int i = 0; i < size; i++)			//Initializing pivoting vector
		pvt[i] = i;

	for (int i = 0; i < size; i++){			//Loops 'size' times
		scale[i] = 0;				//Initialize scale to 0
		for (int j = 0; j < size; j++){		//Loops size times
			if (fabs(scale[i]) < fabs(A(i, j)))	//f the next position is larger
				scale[i] = fabs(A(i, j));	// set it for the scale value
		}
	}

	double temp;					//Variable to check each position
	for (int k = 0; k < size-1; k++){		//Main loop
		int pc = k;				//pivot column
		double aet = fabs(A(pvt[k], k) / scale[k]);
		for (int i = k + 1; i < size; i++){	//Loops 'size'-1 times
			temp = fabs(A(pvt[i], k) / scale[i]);
			if (temp > aet){		//Comparing each position with aet
				aet = temp;		//If temp>aet, set aet = to temp
				pc = i;			//and set pivot column to i
			}
		}
		if (pc != k){				//If pivot column was changed, we
			int tempNum = pvt[k];		//need to switch the pivot values
			pvt[k] = pvt[pc];
			pvt[pc] = tempNum;
		}
		for (int i = k + 1; i < size; i++){	//Now we will shift the rows in the matrix A
			if (A(pvt[i], k) != 0){
				double factor = A(pvt[i], k) / A(pvt[k], k);
				A(pvt[i], k) = factor;
				for (int j = k + 1; j < size; j++)
					A(pvt[i], j) -= factor*A(pvt[k], j);
```

```
                        }
                }

        }
        cout << "Printing after SPP: " << endl;          //Printing to console to see result
        A.print();
        b.print();
        SolveWithGEandBS(A, b, size);                    //Now plug the adjusted matrix GE function
}
```

Sample driver code for *GEwithScaledPartialPivoting:

```
int main(){
        int size = 3;
        MyMatrix B = GenerateRandomSymmetricSquareMatrix(size);  //Creates a random symmetric matrix
        cout << "Printing Randomly Generated Square Matrix with Size " << size << ":" << endl;
        B.print();                                       //Prints the new random matrix
        MyVector v = MyVector(size);                     //Creates a vector
        for (int i = 0; i < size; i++){                  //Loops for 'size' times
                v[i] = 1;                                //Sets each element in vector v to 1
        }
        MyVector b = MatrixVectorProduct(B, v, size);    //Solves Matrix Vector Product of B and v
        b.print();                                       //Prints solution to Matrix Vector Product, b
        GEwithScaledPartialPivoting(B, b, size);         //Solves B using our GE, BS, and SPP function
}
```

****Output:***

```
Printing Randomly Generated Square Matrix with Size 3:
3 4 4
4 4 3
4 3 4
11
11
11

Printing after SPP:
0.75 1 1.75
4 4 3
1 -1 2.75
11
11
11

Matrix after GE
0.75 1 1.75
0 -1.33333 -6.33333
0 0 11.5
11
-47.6667
79.75

Solution:
-5.26087
2.80978
6.93478

Press any key to continue . . .
```

7) Finding Eigenvalues

This chapter is dedicated to methods used to find the eigenvalues of matrices. Right now, the only type of routines in this chapter are based on the Power Method.

7.1) Power Methods

The Power Method, which is the first routine presented in this chapter, is used to find the largest eigenvalue of a matrix. The Inverse Power Method also finds a single eigenvalue of a matrix, but it finds the smallest eigenvalue. In order to check if the eigenvalue produced by these methods are valid, I used the Wolfram Alpha eigenvalue calculator online on the given matrices and checked, which results confirmed that these methods found eigenvalues of the matrices provided.

7.1.1) *powerMethod*

As stated above, the Power Method is used to find the largest eigenvalue of a given matrix. The routine takes two parameters, the first of which is the given matrix A, and the second is the initial guess vector, v0.

```
//Sam Christiansen
//12/01/2016
//Programing Language: C++

//Function to find the largest eigenvalue of a matrix
//    using the Power Method
//The first parameter, A, is the matrix
//The second parameter, v0, is the vector with the initial guess
double powerMethod(MyMatrix A, MyVector v0){
        int size = v0.size();                                   //Initializing the size
        int iter = 0, maxIter = 1000;                           //Initializing values for iterators
        double tol = .000001;                                   //Initializing value for tolerance
        double error = tol * 10;                                //Initializing the error
        double lamda0 = 0, lamda1;                              //Initializing λ0 and λ1
        MyVector vk = MyVector(size);                           //Initializing vector for vk

        while (error > tol&&iter < maxIter){                    //Main loop
                vk = MatrixVectorProduct(A, v0, size);          //Compute vk = Av0
                vk = scalarVectorMultiplication(vk,             //Compute vk = (1/||vk||)*vk
                        (1 / EuclideanLength(vk)), size);
                lamda1 = dotProduct(vk,                         //Compute λ1 = <vk^T,Avk>
                        MatrixVectorProduct(A, vk, size));
                error = abs(lamda1 - lamda0);                   //Update error
                lamda0 = lamda1;                                //Update λ0
                v0 = vk;                                        //Update v0
                iter++;                                         //Update iterator
        }
        return lamda1;                                          //Return resulting λ1
}
```

Sample driver code for *powerMethod:

```
int main(){
        int size = 3;                                   //Set size
        MyMatrix A = GenerateRandomSquareMatrix(size);  //Create matrix A
        MyVector initialGuess = MyVector(size);         //Create initial guess vector
        for (int i = 0; i < size; i++)                  //Initializing initial guess vector
```

```
                initialGuess[i] = 4;

        A.print();                                      //Printing A
        initialGuess.print();                           //Printing the initial guess vector
        double lamda = powerMethod(A, initialGuess);    //Calling the Power Method
        cout << "Solution of powerMethod: " << lamda << endl;//Printing the solution

        return 0;
}
```

***Output:**

```
30.8845 0.511948 0.431013
0.88406 30.2076 0.992248
0.837611 0.493454 30.409

4
4
4

Solution of powerMethod: 31.8613

Press any key to continue . . .
```

7.1.2) *inversePowerMethod*

As stated above, the Power Method is used to find the largest eigenvalue of a given matrix. The routine takes two parameters, the first of which is the given matrix A, and the second is the initial guess vector, v0.

```
//Sam Christiansen
//12/01/2016
//Programing Language: C++

//Function to find the smallest eigenvalue of a matrix
//    using the Inverse Power Method
//The first parameter, A, is the matrix
//The second parameter, v0, is the vector with the initial guess
double inversePowerMethod(MyMatrix A, MyVector v0){
        int size = v0.size();                           //Initializing the size
        int iter = 0, maxIter = 1000;                   //Initializing values for iterators
        double tol = .0001;                             //Initializing value for tolerance
        double error = tol * 10;                        //Initializing the error
        double lamda0 = 0, lamda1;                      //Initializing λ0 and λ1
        MyVector vk = MyVector(size);                   //Initializing vector for vk
        MyVector y = LUwithScaledPartialPivoting(A,     //Initializing y by solving Ay=v0
                v0, size);

        while (error > tol&&iter < maxIter){            //Main loop
                vk = scalarVectorMultiplication(y,      //Compute vk = (1/||vk||)*vk
                        (1 / EuclideanLength(y)), size);
                y = LUwithScaledPartialPivoting(A,      //Update y, solving Ay=vk
                        vk, size);
                lamda1 = 1/dotProduct(vk, y);           //Compute λ1 = 1/<vk^T,y>
                error = abs(lamda1 - lamda0);           //Updating error
                lamda0 = lamda1;                        //Updating λ0
                v0 = vk;                                //Updating v0
                iter++;                                 //Updating iterator
        }
        return lamda1;                                  //Return resulting λ1
}
```

Sample driver code for *inversePowerMethod:

```
int main(){
        int size = 3;                                           //Set size
        MyMatrix A = GenerateRandomSquareMatrix(size);          //Create matrix A
        MyVector initialGuess = MyVector(size);                 //Create initial guess vector
        for (int i = 0; i < size; i++)                          //Initializing initial guess vector
                initialGuess[i] = 4;

        A.print();                                              //Printing A
        initialGuess.print();                                   //Printing the initial guess vector
        double lamda = inversePowerMethod(A, initialGuess);     //Calling the Inverse Power Method
        cout << "Solution of inversePowerMethod: " << lamda << endl;//Printing the solution

        return 0;
}
```

****Output:***

```
30.8666 0.0625629 0.428816
0.169286 30.8628 0.98291
0.251961 0.0430921 30.212

4
4
4

Solution of inversePowerMethod: 30.1225

Press any key to continue . . .
```

Sources

Ascher, Uri Michael, and Chen Greif. *A First Course in Numerical Methods*. Philadelphia: Society for Industrial Mathematics, 2011. Print.