HW4 - Math5610

Andrew Sheridan

October 2016

Problem 1

Back Substitution Algorithm

In this problem there are a number of algorithms used to set up the problem for testing, as well as creating the vectors and matrices. They will be reused throughout the assignment, and I will reference them in other problems as appropriate. My Back Substitution algorithm takes a square matrix A, a vector b, and an integer representing the size of these entities. Pages 1-4 contain the majority of the header file, which contains the Back Substitution algorithm and all the additional functions. Page 5 contains my Main function, as well as the result in the form of console output.

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

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

/// Solves an nxn set of linear equations using back substitution
// A: The nxn upper-triangular coefficient matrix
// b: Right-Hand-Side
// n: The size of the matrices
double* BackSubstitution(double **A, double *b, unsigned n) {

  double* x;
  x = new double[n];
  try {
    x[n - 1] = b[n - 1] / A[n - 1][n - 1];
    for (int k = n - 2; k >= 0; k--) {
      x[k] = b[k];
      for (int i = k + 1; i < n; i++) {
        x[k] -= A[k][i] * x[i];
      }
      x[k] /= A[k][k];
    }
```

```cpp
  }
  catch (std::exception& e)
  {
    std::cout << "These matrices are not the correct size." << std::endl;
    return new double[n];
  }
  return x;
}

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

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

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

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

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

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

```
  }

  return matrix;
}

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

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

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

  return matrix;
}

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

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

  return vector;
}

///Multiplies an nxn matrix A by the vector x
double* VectorMatrixMultiply(double** A, double* x, unsigned n) {
  double* result = new double[n];
  try {
    for (unsigned i = 0; i < n; i++) {
      result[i] = 0;
```

```
      for (unsigned j = 0; j < n; j++) {
        result[i] += A[i][j] * x[j];
      }
    }
    return result;
  }
  catch (std::exception& e) {
    std::cout << "These matrices are not the correct size." << std::endl;
  }
}

///Creates a vector of all ones of size n
double* CreateOnesVector(unsigned n) {
  double* vector = new double[n];
  for (unsigned i = 0; i < n; i++) {
    vector[i] = 1;
  }
  return vector;
}

///Outputs an nxn matrix to the console
void PrintMatrix(double** matrix, unsigned size) {
  for (unsigned i = 0; i < size; i++) {
    for (unsigned j = 0; j < size; j++) {
      std::cout << std::setw(10) << std::left << matrix[i][j];
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}

///Outputs a size n vector to the console
void PrintVector(double* vector, unsigned size) {
  for (unsigned i = 0; i < size; i++) {
    std::cout << std::setw(10) << std::left << vector[i];
  }
  std::cout << std::endl << std::endl;
}

///Outputs an augmented coefficient matrix to the console
void PrintAugmentedMatrix(double** matrix, double* vector, unsigned size) {
  for (unsigned i = 0; i < size; i++) {
    for (unsigned j = 0; j < size; j++) {
      std::cout << std::setw(10) << std::left << matrix[i][j];
    }
    std::cout << "| " << vector[i] << std::endl;
  }
  std::cout << std::endl;
}

///Returns a copy of the input nxn matrix
double** CopyMatrix(double** matrix, unsigned size) {
  double** result = new double*[size];
  for (unsigned i = 0; i < size; i++) {
    result[i] = new double[size];
```

```
    for (unsigned j = 0; j < size; j++) {
      result[i][j] = matrix[i][j];
    }
  }
  return result;
}

///Returns a copy of the input vector
double* CopyVector(double* vector, unsigned size) {
  double* result = new double[size];
  for (unsigned i = 0; i < size; i++) {
    result[i] = vector[i];
  }
  return result;
}
```

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

//Main.cpp
#include<iostream>
#include "Matrix.h"

int main(void) {
  const unsigned size = 3;

  //Problem 1
  double** matrix = CreateUpperTriangularMatrix(size);
  double* vector = CreateOnesVector(size);
  vector = VectorMatrixMultiply(matrix, vector, size);
  double** matrixCopy = CopyMatrix(matrix, size);
  double* resultVector = BackSubstitution(matrixCopy, vector, size);

  std::cout << "Problem 1: Back Substitution" << std::endl;
  std::cout << "Upper triangular matrix" << std::endl;
  PrintMatrix(matrix, size);
  std::cout << "Test vector " << std::endl;
  PrintVector(vector, size);
  std::cout << "Result of back substitition " << std::endl;
  PrintVector(resultVector, size);
}
```

```
Problem 1: Back Substitution
Upper triangular matrix
30.713    0.428471  0.690885
0         30.7192   0.491119
0         0         30.78

Test vector
31.8323   31.2103   30.78
Result of back substitition
1         1         1
```

Problem 2

Forward Substitution Algorithm

This problem contains additional functions from the Matrix.h header file, which can all be found in Problem 1. I have included the Forward Substitution algorithm below, as well as the portion of my Main function which implements it. The console output for the test is also included.

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

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

 /// Solves a set of linear equations using forward substitution
 //a: the nxn lower-triangular coefficient matrix
 //b: right-hand-side
 //n: the size of the matrices
double* ForwardSubstitution(double** A, double* b, unsigned n) {
  double* x;
  x = new double[n];
  try {
    x[0] = b[0];
    for (int k = 0; k < n; k++) {
      x[k] = b[k];
      for(int j = 0; j < k; j++){
        x[k] = x[k] - A[k][j] * x[j];
      }
      x[k] = x[k] / A[k][k];
    }
  }
  catch (std::exception& e)
  {
    std::cout << "These matrices are not the correct size." << std::endl;
    return new double[n];
  }

  return x;
}
```

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

//Main.cpp
#include<iostream>
#include "Matrix.h"

int main(void) {
  const unsigned size = 3;

  //Problem 2
  double** matrix = CreateLowerTriangularMatrix(size);
  double* vector = CreateOnesVector(size);
  vector = VectorMatrixMultiply(matrix, vector, size);
  double** matrixCopy = CopyMatrix(matrix, size);
  double* resultVector = ForwardSubstitution(matrixCopy, vector, size);

  std::cout << "Problem 2: Forward Substitution" << std::endl;
  std::cout << "Lower triangular matrix" << std::endl;
  PrintMatrix(matrix, size);
  std::cout << "Test vector " << std::endl;
  PrintVector(vector, size);
  std::cout << "Result of forward substition " << std::endl;
  PrintVector(resultVector, size);

  return 0;
}
```

```
Problem 2: Forward Substitution
Lower triangular matrix
30.713    0         0
0.428471  30.6909   0
0.71915   0.491119  30.78

Test vector
30.713    31.1194   31.9903

Result of forward substition
1         1         1
```

Problem 3

Gaussian Elimination

Below is my implementation of Gaussian Elimination is contained below. The extra functions called in Main.cpp are not included in this problem, but can be found with problem 1.

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

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

/// Reduces an nxn matrix A and right-hand-side b to upper-triangular ↩
    form using Gaussian Elimination
// A: The nxn coefficient matrix
// b: Right-Hand-Side
// n: The size of the matrices
double** GaussianElimination(double** A, double* b, unsigned n) {
  try {
    for (int k = 0; k < n; k++) {
      for (int i = k+1; i < n; i++) {
        double factor = A[i][k] / A[k][k];
        for (int j = 0; j < n; j++) {
          A[i][j] = A[i][j] - factor*A[k][j];
        }
        b[i] = b[i] - factor*b[k];
      }
    }
  }
  catch (std::exception& e)
  {
    std::cout << "These matrices are not the correct size." << std::↩
        endl;
  }

  return A;
}
```

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

//Main.cpp
#include<iostream>

#include "Matrix.h"

```

```cpp
int main(void) {
  const unsigned size = 3;

  //Problem 3
  double** matrix = CreateMatrix(size);
  double* vector = CreateOnesVector(size);
  vector = VectorMatrixMultiply(matrix, vector, size);
  double** matrixCopy = CopyMatrix(matrix, size);
  double* resultVector = CopyVector(vector, size);
  double** resultMatrix = GaussianElimination(matrixCopy, resultVector
      , size);

  std::cout << "Problem 3: Gaussian Elimination" << std::endl;
  std::cout << "Test Matrix" << std::endl;
  PrintMatrix(matrix, size);
  std::cout << "Test vector " << std::endl;
  PrintVector(vector, size);
  std::cout << "Result of gaussian elimination " << std::endl;
  PrintAugmentedMatrix(resultMatrix, resultVector, size);

  return 0;
}
```

```
Problem 3: Gaussian Elimination
Test Matrix
30.713    0.428471  0.690885
0.71915   30.4911   0.780028
0.410924  0.579694  30.14

Test vector
31.8323   31.9903   31.1306

Result of gaussian elimination
30.713    0.428471  0.690885  | 31.8323
0         30.4811   0.763851  | 31.2449
0         0         30.1163   | 30.1163
```

Problem 4

Gaussian Elimination and Back Substitution

Thankfully this problem is simple now that we've created the formulas for Gaussian Elimination and Back Substitution. The extra functions used to set up the testing of the problem are contained in Problem 1.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++https://www.sharelatex.com/project/580106d11c10bbf058bb84e1

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

/// Solves an nxn set of linear equations using back substitution
// A: The nxn upper-triangular coefficient matrix
// b: Right-Hand-Side
// n: The size of the matrices
double* BackSubstitution(double **A, double *b, unsigned n) {

  double* x;
  x = new double[n];
  try {
    x[n - 1] = b[n - 1] / A[n - 1][n - 1];
    for (int k = n - 2; k >= 0; k--) {
      x[k] = b[k];
      for (int i = k + 1; i < n; i++) {
        x[k] -= A[k][i] * x[i];
      }
      x[k] /= A[k][k];
    }
  }
  catch (std::exception& e)
  {
    std::cout << "These matrices are not the correct size." << std::endl;
    return new double[n];
  }

  return x;
}

/// Reduces an nxn matrix A and right-hand-side b to upper-triangular form using Gaussian Elimination
// A: The nxn coefficient matrix
// b: Right-Hand-Side
// n: The size of the matrices
double** GaussianElimination(double** A, double* b, unsigned n) {
  try {
    for (int k = 0; k < n; k++) {
```

```
      for (int i = k+1; i < n; i++) {
        double factor = A[i][k] / A[k][k];
        for (int j = 0; j < n; j++) {
          A[i][j] = A[i][j] - factor*A[k][j];
        }
        b[i] = b[i] - factor*b[k];
      }
    }
  }
  catch (std::exception& e)
  {
    std::cout << "These matrices are not the correct size." << std::endl;
  }

  return A;
}

}
```

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

//Main.cpp
#include<iostream>
#include "Matrix.h"

int main(void) {
  const unsigned size = 3;

  //Problem 4
  double** matrix = CreateMatrix(size);
  double* vector = CreateOnesVector(size);
  vector = VectorMatrixMultiply(matrix, vector, size);
  double** matrixCopy = CopyMatrix(matrix, size);
  double* resultVector = CopyVector(vector, size);
  double** resultMatrix = GaussianElimination(matrixCopy, resultVector, size);

  double* finalVector = BackSubstitution(resultMatrix, resultVector, size);

  std::cout << "Problem 4: Gaussian Elimination w/ Back Substitution" << std::endl;
  std::cout << "Test Matrix" << std::endl;
  PrintMatrix(matrix, size);
  std::cout << "Test vector " << std::endl;
  PrintVector(vector, size);
  std::cout << "Result of gaussian elimination and back substitution" << std::endl;
  PrintAugmentedMatrix(resultMatrix, finalVector, size);

  return 0;
}
```

```
Problem 4: Gaussian Elimination w/ Back Substitution
Test Matrix
30.713    0.428471  0.690885
0.71915   30.4911   0.780028
0.410924  0.579694  30.14

Test vector
31.8323   31.9903   31.1306

Result of gaussian elimination and back substitution
30.713    0.428471  0.690885  | 1
0         30.4811   0.763851  | 1
0         0         30.1163   | 1
```

Problem 5

Testing our Methods

I've included the basic algorithms used in the problem below. The algorithm was tested on $n \times n$ matrices of sizes $n = 10, 20, 40, 80$, and 160. Each of them give the result of vectors full of ones. Even for the 160x160 matrix, the result was a vector of 160 ones.

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

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

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

  return matrix;
}

///Multiplies an nxn matrix A by the vector x
double* VectorMatrixMultiply(double** A, double* x, unsigned n) {
  double* result = new double[n];
  try {
    for (unsigned i = 0; i < n; i++) {
      result[i] = 0;
      for (unsigned j = 0; j < n; j++) {
        result[i] += A[i][j] * x[j];
      }
    }
    return result;
  }
  catch (std::exception& e) {
    std::cout << "These matrices are not the correct size." << std::endl;
  }
}

///Creates a vector of all ones of size n
double* CreateOnesVector(unsigned n) {
  double* vector = new double[n];
  for (unsigned i = 0; i < n; i++) {
    vector[i] = 1;
```

```
  }
  return vector;
}
```

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

//Main.cpp
#include<iostream>
#include "Matrix.h"

int main(void) {
  const unsigned size = 3;

  //Problem 5
  double** matrix = CreateMatrix(size);
  double* vector = new double[size];
  vector[0] = 1;
  vector[1] = 2;
  vector[2] = 3;

  double* fivePartOne = VectorMatrixMultiply(matrix, vector, size);
  std::cout << "Problem 5: Testing" << std::endl;
  std::cout << "5.1" << std::endl;
  PrintMatrix(matrix, size);
  PrintVector(vector, size);
  PrintVector(fivePartOne, size);

  double* onesVector = CreateOnesVector(size);
  double* fivePartThree = VectorMatrixMultiply(matrix, onesVector, size);

  std::cout << "5.3" << std::endl;
  PrintVector(onesVector, size);
  PrintVector(fivePartThree, size);

  std::cout << "5.4" << std::endl;
  for (int i = 10; i *= 2; i <= 160) {
    matrix = CreateMatrix(i);
    vector = CreateOnesVector(i);
    vector = VectorMatrixMultiply(matrix, vector, i);
    matrixCopy = CopyMatrix(matrix, i);
    resultMatrix = GaussianElimination(matrixCopy, vector, i);
    resultVector = BackSubstitution(resultMatrix, vector, i);

    std::cout << "Problem 5: Testing" << std::endl;
    std::cout << "Start matrix" << std::endl;
    PrintMatrix(matrix, i);
    std::cout << "Test vector " << std::endl;
    PrintVector(vector, i);
    std::cout << "Result of GE and Back Substitution " << std::endl;
    PrintVector(resultVector, i);
  }

  return 0;
```

```
}
```

```
Problem 5: Testing
------------5.1-------------
30.713    0.428471  0.690885
0.71915   30.4911   0.780028
0.410924  0.579694  30.14

1         2         3

33.6426   64.0415   91.9902

------------5.3-------------
1         1         1

31.8323   31.9903   31.1306

------------5.4-------------
A Shizload of ones
```

Problem 6

5.1 Our starting system of equations is as follows:

$$x_1 - x_2 + 3x_3 = 2$$

$$x_1 + x_2 = 4$$

$$3c_1 - 2x_2 + x_3 = 1$$

The matrix form will look like

$$\left(\begin{array}{ccc|c} 1 & -1 & 3 & 2 \\ 1 & 1 & 0 & 4 \\ 3 & -2 & 1 & 1 \end{array}\right) \tag{1}$$

We will then perform one step of reduction, where we subtract a factor of row 1 from rows 2 and 3.

$$\left(\begin{array}{ccc|c} 1 & -1 & 3 & 2 \\ 0 & 2 & -3 & 2 \\ 0 & 1 & -8 & -5 \end{array}\right) \tag{2}$$

Then, for sake of computational simplicity, I'm going to interchange rows 2 and 3.

$$\left(\begin{array}{ccc|c} 1 & -1 & 3 & 2 \\ 0 & 1 & -8 & -5 \\ 0 & 2 & -3 & 2 \end{array}\right) \tag{3}$$

$$\left(\begin{array}{ccc|c} 1 & -1 & 3 & 2 \\ 0 & 1 & -8 & -5 \\ 0 & 2 & -3 & 2 \end{array}\right) \tag{4}$$

Now we subtract a multiple of row 2 from row 3 to get

$$\left(\begin{array}{ccc|c} 1 & -1 & 3 & 2 \\ 0 & 1 & -8 & -5 \\ 0 & 0 & 13 & 12 \end{array}\right) \tag{5}$$

Now things get a little more complicated, because we need to subtract multiples of row 3 from rows 1 and 2. If we perform this subtraction, we get

$$\left(\begin{array}{ccc|c} 1 & -1 & 0 & \frac{-10}{13} \\ 0 & 1 & 0 & \frac{21}{13} \\ 0 & 0 & 13 & 12 \end{array}\right) \tag{6}$$

Finally, we subtract a factor of row 2 from row 1, then simplify the matrix so our diagonal entries are zero. This results in

$$\left(\begin{array}{ccc|c} 1 & 1 & 0 & \frac{21}{13} \\ 0 & 1 & 0 & \frac{31}{13} \\ 0 & 0 & 1 & \frac{12}{13} \end{array}\right) \tag{7}$$

Problem 7

5.4

Did not finish.

Problem 8

4.1 Our objective is to show that if the rows in an n by n matrix A sum to zero, then the A is singular. We can sum the rows of a matrix by using the ones vector 1_n, where n is the number of entries in the vector. $A1_n$ will sum the rows a give the zero vector, 0_n.

$$A1_n = 0$$

Therefore, 1_n is part of the null space of A. This tells us that the rank of our matrix A is less than n, and therefore our matrix is singular.

Problem 9

4.2

If $Ax = \lambda x$ and A is non-singular, then A^{-1} has eigenvalue $\frac{1}{\lambda}$.

If we start with our matrix and vector Ax, we know that $Ax = \lambda x$. If we then multiply both sides by the inverse matrix A^{-1}, we get $AA^{-1}x = \lambda A^{-1}x$. AA^{-1} simplifies to the identity matrix I, and this results in $Ix = \lambda A^{-1}x$, which is the same as $x = \lambda A^{-1}x$. We can then divide both side by lambda, and have the result $x\lambda^{-1} = A^{-1}x$, which shows that if our initial conditions hold, then A^{-1} has eigenvalue $\frac{1}{\lambda}$.