

# 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.

---

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Matrix.h
6 #pragma once
7 #include <iostream>
8 #include <iomanip>
9 #include <cmath>
10 #include <random>
11
12 /// Solves an nxn set of linear equations using back substitution
13 // A: The nxn upper-triangular coefficient matrix
14 // b: Right-Hand-Side
15 // n: The size of the matrices
16 double* BackSubstitution(double **A, double *b, unsigned n) {
17
18     double* x;
19     x = new double[n];
20     try {
21         x[n - 1] = b[n - 1] / A[n - 1][n - 1];
22         for (int k = n - 2; k >= 0; k--) {
23             x[k] = b[k];
24             for (int i = k + 1; i < n; i++) {
25                 x[k] -= A[k][i] * x[i];
26             }
27             x[k] /= A[k][k];
28         }
29     }
```

```

29     }
30     catch (std::exception& e)
31     {
32         std::cout << "These matrices are not the correct size." << std::endl;
33         return new double[n];
34     }
35     return x;
36 }
37
38 // Generates a random square matrix of size n
39 // n: The size of the matrix
40 double** CreateMatrix(unsigned n) {
41     std::mt19937 generator(123); //Random number generator
42     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↵
43         distribution
44     double** matrix;
45     matrix = new double *[n];
46     for (unsigned i = 0; i < n; i++) {
47         matrix[i] = new double [n];
48         for (unsigned j = 0; j < n; j++) {
49             matrix[i][j] = dis(generator); //Assign each entry in matrix to ↵
50                 random number between 0 and 1
51         }
52     }
53     for (unsigned k = 0; k < n; k++) {
54         matrix[k][k] += 10*n; //Add 10*n to all diagonal entries
55     }
56     return matrix;
57 }
58
59 // Generates a random upper triangular square matrix of size n
60 // n: The size of the matrix
61 double** CreateUpperTriangularMatrix(unsigned n) {
62     std::mt19937 generator(123); //Random number generator
63     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↵
64         distribution
65     double** matrix;
66     matrix = new double *[n];
67     for (unsigned i = 0; i < n; i++) {
68         matrix[i] = new double [n];
69         for (unsigned j = 0; j < n; j++) {
70             if (j < i) {
71                 matrix[i][j] = 0;
72             }
73             else {
74                 matrix[i][j] = dis(generator); //Assign entry in matrix to ↵
75                     random number between 0 and 1
76             }
77         }
78     }
79     for (unsigned k = 0; k < n; k++) {
80         matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries

```

```

81     }
82
83     return matrix;
84 }
85
86 // Generates a random lower triangular square matrix of size n
87 // n: The size of the matrix
88 double** CreateLowerTriangularMatrix(unsigned n) {
89     std::mt19937 generator(123); //Random number generator
90     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↵
91         distribution
92
93     double** matrix;
94     matrix = new double *[n];
95     for (unsigned i = 0; i < n; i++) {
96         matrix[i] = new double[n];
97         for (unsigned j = 0; j < n; j++) {
98             if (i < j) {
99                 matrix[i][j] = 0;
100             }
101             else {
102                 matrix[i][j] = dis(generator); //Assign entry in matrix to ↵
103                     random number between 0 and 1
104             }
105         }
106     }
107
108     for (unsigned k = 0; k < n; k++) {
109         matrix[k][k] += 10 * n; //Add 10*n to all diagonal entries
110     }
111
112     return matrix;
113 }
114
115 // Generates a random vector of size n
116 // n: The size of the vector
117 double* CreateVector(unsigned n) {
118     std::mt19937 generator(123); //Random number generator
119     std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ↵
120         distribution
121
122     double* vector;
123     vector = new double [n];
124     for (unsigned i = 0; i < n; i++) {
125         vector[i] = dis(generator); //Assign each entry in vector to ↵
126             random number between 0 and 1
127     }
128
129     return vector;
130 }
131
132 //Multiplies an nxn matrix A by the vector x
133 double* VectorMatrixMultiply(double** A, double* x, unsigned n) {
134     double* result = new double[n];
135     try {
136         for (unsigned i = 0; i < n; i++) {
137             result[i] = 0;

```

```

134         for (unsigned j = 0; j < n; j++) {
135             result[i] += A[i][j] * x[j];
136         }
137     }
138     return result;
139 }
140 catch (std::exception& e) {
141     std::cout << "These matrices are not the correct size." << std::endl;
142 }
143 }
144
145 //Creates a vector of all ones of size n
146 double* CreateOnesVector(unsigned n) {
147     double* vector = new double[n];
148     for (unsigned i = 0; i < n; i++) {
149         vector[i] = 1;
150     }
151     return vector;
152 }
153
154 //Outputs an nxn matrix to the console
155 void PrintMatrix(double** matrix, unsigned size) {
156     for (unsigned i = 0; i < size; i++) {
157         for (unsigned j = 0; j < size; j++) {
158             std::cout << std::setw(10) << std::left << matrix[i][j];
159         }
160         std::cout << std::endl;
161     }
162     std::cout << std::endl;
163 }
164
165 //Outputs a size n vector to the console
166 void PrintVector(double* vector, unsigned size) {
167     for (unsigned i = 0; i < size; i++) {
168         std::cout << std::setw(10) << std::left << vector[i];
169     }
170     std::cout << std::endl << std::endl;
171 }
172
173 //Outputs an augmented coefficient matrix to the console
174 void PrintAugmentedMatrix(double** matrix, double* vector, unsigned size) {
175     for (unsigned i = 0; i < size; i++) {
176         for (unsigned j = 0; j < size; j++) {
177             std::cout << std::setw(10) << std::left << matrix[i][j];
178         }
179         std::cout << " | " << vector[i] << std::endl;
180     }
181     std::cout << std::endl;
182 }
183
184 //Returns a copy of the input nxn matrix
185 double** CopyMatrix(double** matrix, unsigned size) {
186     double** result = new double*[size];
187     for (unsigned i = 0; i < size; i++) {
188         result[i] = new double[size];

```

```

189     for (unsigned j = 0; j < size; j++) {
190         result[i][j] = matrix[i][j];
191     }
192 }
193 return result;
194 }
195
196 //Returns a copy of the input vector
197 double* CopyVector(double* vector, unsigned size) {
198     double* result = new double[size];
199     for (unsigned i = 0; i < size; i++) {
200         result[i] = vector[i];
201     }
202     return result;
203 }

```

---

```

1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Main.cpp
6 #include<iostream>
7 #include "Matrix.h"
8
9 int main(void) {
10     const unsigned size = 3;
11
12     //Problem 1
13     double** matrix = CreateUpperTriangularMatrix(size);
14     double* vector = CreateOnesVector(size);
15     vector = VectorMatrixMultiply(matrix, vector, size);
16     double** matrixCopy = CopyMatrix(matrix, size);
17     double* resultVector = BackSubstitution(matrixCopy, vector, size);
18
19     std::cout << "Problem 1: Back Substitution" << std::endl;
20     std::cout << "Upper triangular matrix" << std::endl;
21     PrintMatrix(matrix, size);
22     std::cout << "Test vector " << std::endl;
23     PrintVector(vector, size);
24     std::cout << "Result of back substitution " << std::endl;
25     PrintVector(resultVector, size);
26 }

```

---

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 substitution

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.

---

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Matrix.h
6 #pragma once
7 #include <iostream>
8 #include <iomanip>
9 #include <cmath>
10 #include <random>
11
12 /// Solves a set of linear equations using forward substitution
13 //a: the nxn lower-triangular coefficient matrix
14 //b: right-hand-side
15 //n: the size of the matrices
16 double* ForwardSubstitution(double** A, double* b, unsigned n) {
17     double* x;
18     x = new double[n];
19     try {
20         x[0] = b[0];
21         for (int k = 0; k < n; k++) {
22             x[k] = b[k];
23             for (int j = 0; j < k; j++){
24                 x[k] = x[k] - A[k][j] * x[j];
25             }
26             x[k] = x[k] / A[k][k];
27         }
28     }
29     catch (std::exception& e)
30     {
31         std::cout << "These matrices are not the correct size." << std::endl;
32         return new double[n];
33     }
34
35     return x;
36 }
```

---

---

```

1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Main.cpp
6 #include<iostream>
7 #include "Matrix.h"
8
9 int main(void) {
10     const unsigned size = 3;
11
12     //Problem 2
13     double** matrix = CreateLowerTriangularMatrix(size);
14     double* vector = CreateOnesVector(size);
15     vector = VectorMatrixMultiply(matrix, vector, size);
16     double** matrixCopy = CopyMatrix(matrix, size);
17     double* resultVector = ForwardSubstitution(matrixCopy, vector, size)↵
18         ;
19     std::cout << "Problem 2: Forward Substitution" << std::endl;
20     std::cout << "Lower triangular matrix" << std::endl;
21     PrintMatrix(matrix, size);
22     std::cout << "Test vector " << std::endl;
23     PrintVector(vector, size);
24     std::cout << "Result of forward substitution " << std::endl;
25     PrintVector(resultVector, size);
26
27     return 0;
28 }

```

---

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 substitution

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.

---

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Matrix.h
6 #pragma once
7 #include <iostream>
8 #include <iomanip>
9 #include <cmath>
10 #include <random>
11
12 /// Reduces an nxn matrix A and right-hand-side b to upper-triangular ←
13   form using Gaussian Elimination
14 // A: The nxn coefficient matrix
15 // b: Right-Hand-Side
16 // n: The size of the matrices
17 double** GaussianElimination(double** A, double* b, unsigned n) {
18     try {
19         for (int k = 0; k < n; k++) {
20             for (int i = k+1; i < n; i++) {
21                 double factor = A[i][k] / A[k][k];
22                 for (int j = 0; j < n; j++) {
23                     A[i][j] = A[i][j] - factor*A[k][j];
24                 }
25                 b[i] = b[i] - factor*b[k];
26             }
27         }
28     } catch (std::exception& e)
29     {
30         std::cout << "These matrices are not the correct size." << std::endl;
31     }
32
33     return A;
34 }
```

---

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Main.cpp
6 #include <iostream>
7
8 #include "Matrix.h"
9
```

---



```

10 int main(void) {
11     const unsigned size = 3;
12
13     //Problem 3
14     double** matrix = CreateMatrix(size);
15     double* vector = CreateOnesVector(size);
16     vector = VectorMatrixMultiply(matrix, vector, size);
17     double** matrixCopy = CopyMatrix(matrix, size);
18     double* resultVector = CopyVector(vector, size);
19     double** resultMatrix = GaussianElimination(matrixCopy, resultVector↵
        , size);
20
21     std::cout << "Problem 3: Gaussian Elimination" << std::endl;
22     std::cout << "Test Matrix" << std::endl;
23     PrintMatrix(matrix, size);
24     std::cout << "Test vector " << std::endl;
25     PrintVector(vector, size);
26     std::cout << "Result of gaussian elimination " << std::endl;
27     PrintAugmentedMatrix(resultMatrix, resultVector, size);
28
29     return 0;
30 }

```

---

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.

---

```
1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++https://www.sharelatex.com/project/580106↵
   d11c10bbf058bb84e1
4
5 //Matrix.h
6 #pragma once
7 #include <iostream>
8 #include <iomanip>
9 #include <cmath>
10 #include <random>
11
12 /// Solves an nxn set of linear equations using back substitution
13 // A: The nxn upper-triangular coefficient matrix
14 // b: Right-Hand-Side
15 // n: The size of the matrices
16 double* BackSubstitution(double **A, double *b, unsigned n) {
17
18     double* x;
19     x = new double[n];
20     try {
21         x[n - 1] = b[n - 1] / A[n - 1][n - 1];
22         for (int k = n - 2; k >= 0; k--) {
23             x[k] = b[k];
24             for (int i = k + 1; i < n; i++) {
25                 x[k] -= A[k][i] * x[i];
26             }
27             x[k] /= A[k][k];
28         }
29     }
30     catch (std::exception& e)
31     {
32         std::cout << "These matrices are not the correct size." << std::↵
           endl;
33         return new double[n];
34     }
35
36     return x;
37 }
38
39 /// Reduces an nxn matrix A and right-hand-side b to upper-triangular ↵
   form using Gaussian Elimination
40 // A: The nxn coefficient matrix
41 // b: Right-Hand-Side
42 // n: The size of the matrices
43 double** GaussianElimination(double** A, double* b, unsigned n) {
44     try {
45         for (int k = 0; k < n; k++) {
```

```

46         for (int i = k+1; i < n; i++) {
47             double factor = A[i][k] / A[k][k];
48             for (int j = 0; j < n; j++) {
49                 A[i][j] = A[i][j] - factor*A[k][j];
50             }
51             b[i] = b[i] - factor*b[k];
52         }
53     }
54 }
55 catch (std::exception& e)
56 {
57     std::cout << "These matrices are not the correct size." << std::endl;
58 }
59
60 return A;
61 }
62
63 }

```

---

---

```

1 //Andrew Sheridan
2 //Math 5610
3 //Written in C++
4
5 //Main.cpp
6 #include<iostream>
7 #include "Matrix.h"
8
9 int main(void) {
10     const unsigned size = 3;
11
12     //Problem 4
13     double** matrix = CreateMatrix(size);
14     double* vector = CreateOnesVector(size);
15     vector = VectorMatrixMultiply(matrix, vector, size);
16     double** matrixCopy = CopyMatrix(matrix, size);
17     double* resultVector = CopyVector(vector, size);
18     double** resultMatrix = GaussianElimination(matrixCopy, resultVector, ←
19         , size);
20     double* finalVector = BackSubstitution(resultMatrix, resultVector, ←
21         size);
22     std::cout << "Problem 4: Gaussian Elimination w/ Back Substitution" <←
23         << std::endl;
24     std::cout << "Test Matrix" << std::endl;
25     PrintMatrix(matrix, size);
26     std::cout << "Test vector " << std::endl;
27     PrintVector(vector, size);
28     std::cout << "Result of gaussian elimination and back substitution" <←
29         << std::endl;
30     PrintAugmentedMatrix(resultMatrix, finalVector, size);
31     return 0;
32 }

```

---

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  $160 \times 160$  matrix, the result was a vector of 160 ones.

---

```
1  /// Generates a random square matrix of size n
2  // n: The size of the matrix
3  double** CreateMatrix(unsigned n) {
4      std::mt19937 generator(123); //Random number generator
5      std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired ←
        distribution
6
7      double** matrix;
8      matrix = new double *[n];
9      for (unsigned i = 0; i < n; i++) {
10         matrix[i] = new double [n];
11         for (unsigned j = 0; j < n; j++) {
12             matrix[i][j] = dis(generator); //Assign each entry in matrix to ←
                random number between 0 and 1
13         }
14     }
15
16     for (unsigned k = 0; k < n; k++) {
17         matrix[k][k] += 10*n; //Add 10*n to all diagonal entries
18     }
19
20     return matrix;
21 }
22
23 ///Multiplies an nxn matrix A by the vector x
24 double* VectorMatrixMultiply(double** A, double* x, unsigned n) {
25     double* result = new double[n];
26     try {
27         for (unsigned i = 0; i < n; i++) {
28             result[i] = 0;
29             for (unsigned j = 0; j < n; j++) {
30                 result[i] += A[i][j] * x[j];
31             }
32         }
33         return result;
34     }
35     catch (std::exception& e) {
36         std::cout << "These matrices are not the correct size." << std::←
            endl;
37     }
38 }
39
40 ///Creates a vector of all ones of size n
41 double* CreateOnesVector(unsigned n) {
42     double* vector = new double[n];
43     for (unsigned i = 0; i < n; i++) {
44         vector[i] = 1;
```

```

45     }
46     return vector;
47 }

```

---

```

1  //Andrew Sheridan
2  //Math 5610
3  //Written in C++
4
5  //Main.cpp
6  #include<iostream>
7  #include "Matrix.h"
8
9  int main(void) {
10     const unsigned size = 3;
11
12     //Problem 5
13     double** matrix = CreateMatrix(size);
14     double* vector = new double[size];
15     vector[0] = 1;
16     vector[1] = 2;
17     vector[2] = 3;
18
19     double* fivePartOne = VectorMatrixMultiply(matrix, vector, size);
20     std::cout << "Problem 5: Testing" << std::endl;
21     std::cout << "5.1" << std::endl;
22     PrintMatrix(matrix, size);
23     PrintVector(vector, size);
24     PrintVector(fivePartOne, size);
25
26     double* onesVector = CreateOnesVector(size);
27     double* fivePartThree = VectorMatrixMultiply(matrix, onesVector, ←
        size);
28
29     std::cout << "5.3" << std::endl;
30     PrintVector(onesVector, size);
31     PrintVector(fivePartThree, size);
32
33     std::cout << "5.4" << std::endl;
34     for (int i = 10; i *= 2; i <= 160) {
35         matrix = CreateMatrix(i);
36         vector = CreateOnesVector(i);
37         vector = VectorMatrixMultiply(matrix, vector, i);
38         matrixCopy = CopyMatrix(matrix, i);
39         resultMatrix = GaussianElimination(matrixCopy, vector, i);
40         resultVector = BackSubstitution(resultMatrix, vector, i);
41
42         std::cout << "Problem 5: Testing" << std::endl;
43         std::cout << "Start matrix" << std::endl;
44         PrintMatrix(matrix, i);
45         std::cout << "Test vector " << std::endl;
46         PrintVector(vector, i);
47         std::cout << "Result of GE and Back Substitution " << std::endl;
48         PrintVector(resultVector, i);
49     }
50
51     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:

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

$$x_1 + x_2 = 4$$

$$3x_1 - 2x_2 + x_3 = 1$$

The matrix form will look like

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

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

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

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

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

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

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

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

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



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

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

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

## 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 and 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  $\frac{1}{\lambda}$ .