```
1  /*
2   * Matrix.h
3   *
4   *  Created on: 21.12.2017
5   *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
6   */
7
8  // Header file for class Matrix - provides support for arbitrarily sized m-by-n matrices
9
10 // Declares class data type called Matrix.
11
12 // Include header guards to avoid multiple attempts at defining class.
13
14 #ifndef MATRIX_H_
15 #define MATRIX_H_
16
17 #include <iostream>
18
19 using namespace std;
20
21 class Matrix {
22
23 // Part 1.
24
25 protected:
26         int noOfRows; //stores the number of rows
27         int noOfCols; //stores the number of columns
28         double* data; //stores the address to the 1-D array of the matrix entries arranged column-
   wise
29
30         // getter
31         int GetIndex (const int& rowIdx, const int& columnIdx) const; // determines the position
   (index) along 'data' of a matrix entry, in the row and column specified by 'rowIdx' and
   'columnIdx', respectively.
32
33 public:
34
35 // 1. ConstructorsTesting
36
37         // constructors
38         Matrix (); // default constructor -  must be defined as other constructors have been
   defined here
39         Matrix (const int& noOfRows_rhs, const int& noOfCols_rhs); // standard custom constructor
40         Matrix (const Matrix& input); // default copy constructor
41
42         // named static constructors:
43         static Matrix Zeros(const int& noOfRows_rhs, const int& noOfCols_rhs);
44         static Matrix Ones (const int& noOfRows_rhs, const int& noOfCols_rhs);
45
46         // default destructor
47         virtual ~Matrix ();
48
49         // printing with ostream
50         friend ostream& operator<< (ostream& out, const Matrix& rhs);
51
52 // 2. AssignmentTesting
53
54         //default assignment operator
55         Matrix& operator= (const Matrix& rhs);
56
57 // 3. ToeplitzTesting & ToeplitzTestingHelper
58
59         static Matrix Print (const double* const data_rhs, const int& noOfRows_rhs, const int&
   noOfCols_rhs);
60         static Matrix Toeplitz (const double* const column, const int& noOfRows_rhs, const double*
   const row, const int& noOfCols_rhs); //creates a new Toeplitz matrix
61
```

```
62  // 4. TransposeTesting
63
64          static Matrix Transpose (const Matrix& input);  //static function - creates a new transpose
    matrix
65          void Transpose ();  //non-static function - converts matrix into tranpose of itself
66
67  // 5. MultiplicationTesting
68
69          friend Matrix operator* (const Matrix& lhs, const Matrix& rhs); //non-member multiplication
    function - creates new matrix which is result of lhs * rhs
70          Matrix& operator*= (const Matrix& rhs); // member self-modifying multiplication function -
    converts input (lhs) matrix into result of lhs * rhs
71
72  // 6. RowColumnExchangeTesting
73
74          Matrix& ExchangeRows (const int& row1, const int& row2); //swaps row1 and row2, for all
    columns in matrix
75          Matrix& ExchangeRows (const int& row1, const int& row2, const int& col1, const int&
    col2); //swaps row1 and row2, but only between col1 and col2
76          Matrix& ExchangeColumns (const int& col1, const int& col2); //swaps col1 and col2, for all
    rows in matrix
77          Matrix& ExchangeColumns (const int& col1, const int& col2, const int& row1, const int&
    row2); //swaps col1 and col2, but only between row1 and row2
78
79
80  // 7. OtherTesting
81
82          void Zeros (); // (sets every entry of matrix to zero)
83          void Ones (); // (sets every entry of matrix to one)
84          int GetNoOfRows () const; // (find out number of rows in matrix)
85          int GetNoOfColumns () const; // (find out number of columns in matrix)
86          double GetEntry (const int& rowIdx, const int& columnIdx) const; // (find out value of
    particular entry (i,j) in Matrix, starting indexing at (0,0) in upper left corner of matrix)
87
88  // Part 2.
89
90          static Matrix Test (const double* data_rhs, const int& noOfRows_rhs, const int&
    noOfCols_rhs); //for constructing specified test matrices for testing in Parts 2 & 3
91          static Matrix PrintMATLAB (const Matrix& rhs); //Specifically to print matrices in default
    MATLAB style to fixed 4.d.p precision for Parts 2 & 3
92
93  // Part 3.
94
95  // 8. Forward Substitution - Algorithm 3.1
96
97          virtual Matrix ForwardSub (const Matrix& b) const;
98                  //Invoking matrix L is a lower triangular m-by-m square matrix (from SquareMatrix
    class function).
99                  //Input b is a m-by-1 column vector.
100                 //Returns a m-by-1 column vector y, such that Ly=b.
101
102 // 9. Backward Substitution - Algorithm 3.2
103
104         virtual Matrix BackwardSub (const Matrix& y) const;
105                 //Invoking matrix U is an upper triangular m-by-m square matrix (from SquareMatrix
    class function).
106                 //Input y is a m-by-1 column vector.
107                 //Returns a m-by-1 column vector x, such that Ux=y.
108
109 // 10. Solving a system of linear equations in which the number of equations is equal to the number
    of unknowns, i.e. a uniquely determined solution exists.
110
111         Matrix SolveUnique (const Matrix& L, const Matrix& U, const Matrix& P, Matrix& b) const;
112                 //Function is invoked by a m-by-m square matrix A (from SquareMatrix class
    function), which represents the matrix of coefficients
113                 //Input matrices L, U and P are produced in class SquareMatrix from LU
    decomposition of A using Algorithm 2.2. All 3 are m-by-m square matrices.
```

```
114                 //Input b is a m-by-1 column vector.
115                 //Returns a m-by-1 column vector x, such that Ax=b.
116
117     // 11. Solving an overdetermined system of linear equations in which the number of equations is
        greater than the number of unknowns, i.e. a least squares solution is produced.
118
119             Matrix LeastSquaresHelper (const Matrix& b, const string& output) const;
120                 //Function is invoked by a non-square m-by-n matrix A, which represents the matrix
        of coefficients
121                 //Input b is a m-by-1 column vector.
122                 //Returns n-by-n square matrix A'*A if output = 'A'*A', or n-by-1 column vector
        A'*b if output = 'A'*b'.
123
124     };
125
126     #endif /* MATRIX_H_ */
```

```cpp
/*
 * Matrix.cpp
 *
 *  Created on: 21.12.17
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Source file for class Matrix, to define implementations for functions declared in Matrix.h

#include "Matrix.h"
#include <iostream>
#include <cstdlib>
#include <string>
#include <cmath>  //for abs function
#include <iomanip> //for printing precision

using namespace std;

// Part 1.

//getter - to specify the position (index) along 'data' of a matrix entry in the row and column specified by 'rowIdx' and 'columnIdx', respectively
//assuming that the matrix indexing starts at (0,0) and the array indexing starts at (0).
int Matrix::GetIndex (const int& rowIdx, const int& columnIdx) const {
	int index = rowIdx + columnIdx*noOfRows;
	return index;
}

// 1. ConstructorsTesting

//default constructor
Matrix::Matrix () {
	noOfRows = 0;
	noOfCols = 0;
	data = 0;
}

//standard custom constructor
Matrix::Matrix (const int& noOfRows_rhs, const int& noOfCols_rhs) {
	// first check that both inputs are greater than zero
	if (noOfRows_rhs <= 0 or noOfCols_rhs <= 0) {
		cerr << endl;
		cerr << "Error:  In standard custom constructor ..." << endl;
		cerr << "        Cannot create a matrix with a non-positive dimension." << endl;
		cerr << "        Exiting program ... " << endl;
		cerr << endl;
		exit(1);
	}
	//assign dimension values
	noOfRows = noOfRows_rhs;
	noOfCols = noOfCols_rhs;
	//allocate new memory
	int product = noOfRows*noOfCols;
	data = new double [product];
	//then assign '0' to each value (this is done linearly here, rather than column-wise, as all values are the same)
	for (int i = 0; i < product; ++i) {
		data[i] = 0;
	}
}

// default copy constructor
Matrix::Matrix (const Matrix& rhs) {

	//assign dimensions
	this->noOfRows = rhs.noOfRows;
	this->noOfCols = rhs.noOfCols;
```

```
66
67          // allocate the memory of appropriate size
68          int product = noOfRows*noOfCols;
69          this->data = new double[product];
70
71          // ensure that a 'deep', rather than a 'shallow', copy is done
72          for (int i = 0; i < product; ++i) {
73                  this->data[i] = rhs.data[i];
74          }
75  }
76
77  // static Zeros constructor
78  Matrix Matrix::Zeros(const int& noOfRows_rhs, const int& noOfCols_rhs) {
79          // There is no vector object attached to a static member function
80
81          // Therefore need to create Matrix object of zeros using standard custom constructor
82          Matrix Zeros (noOfRows_rhs, noOfCols_rhs);
83
84          // return the object
85          return Zeros;
86  }
87
88  // static Ones constructor
89  Matrix Matrix::Ones(const int& noOfRows_rhs, const int& noOfCols_rhs) {
90
91          //Create Matrix object of zeros using standard custom constructor
92          Matrix Ones (noOfRows_rhs, noOfCols_rhs);
93
94          //Then assign '1' to each value (this is done linearly here, rather than column-wise, as
    all values are the same)
95          int product = noOfRows_rhs*noOfCols_rhs;
96          for (int i = 0; i < product; ++i) {
97                  Ones.data[i] = 1;
98          }
99
100         //Return the object
101         return Ones;
102 }
103
104
105 //destructor
106 Matrix::~Matrix () {
107         //dynamically delete all current instances of member variable data
108         delete[] data;
109 }
110
111 //printer (remembering that the matrix has been stored columnwise in data)
112 ostream& operator<< (ostream& out, const Matrix& rhs)  {
113         //loop through the row indices
114         for (int i = 0; i < rhs.noOfRows; ++i) {
115         out << "\t"; // leave a tab (8 spaces) at the beginning of the row
116                 //loop through the column indices
117                 for (int j = 0; j < rhs.noOfCols; ++j) {
118                         int index = i + j*rhs.noOfRows;         // GetIndex (i,j) cannot be used
    here as not declared in this scope
119                         double element = rhs.data[index];
120                         double tolerance = 1e-14;
121                         if (abs(element) < tolerance) { //set to zero if magnitude of element is
    less than tolerance;
122                                 element = 0;
123                         }
124                         if (element >= 0) {
125                                 out << " " << element; // leave an extra space if element is
    positive, to ensure alignment of matrix entries in console out
126                         }
127                         else {
128                                 out << element;
```

```
129                         }
130                         out << "         "; // leave 9 spaces between columns (tab + 1 doesn't
    work ...)
131                 }
132                 out << endl; // go to next line at end of row
133         }
134
135         return out;
136 }
137
138 // 2. AssignmentTesting
139
140 // default assignment operator
141 Matrix& Matrix::operator= (const Matrix& rhs) {
142         // check the dimensions of the rhs input
143         if (this->noOfRows != rhs.noOfRows and this->noOfCols!= rhs.noOfCols) {
144                 cerr << "Error:  Assignment operation failed ..." << endl;
145                 cerr << "        rhs input has incompatible number of rows and columns." << endl;
146                 cerr << endl;
147         }
148         else if (this->noOfRows != rhs.noOfRows) {
149                 cerr << "Error:  Assignment operation failed ..." << endl;
150                 cerr << "        rhs input has incompatible number of rows." << endl;
151                 cerr << endl;
152         }
153         else if (this->noOfCols!= rhs.noOfCols) {
154                 cerr << "Error:  Assignment operation failed..." << endl;
155                 cerr << "        rhs input has incompatible number of columns." << endl;
156                 cerr << endl;
157         }
158         //guard against copy into self
159         else if (this == &rhs) {
160                 cerr << "Error:  Assignment operation failed..." << endl;
161                 cerr << "        copy into self attempted." << endl;
162                 cerr << endl;
163         }
164         else {
165                 // assign the data of rhs to lhs
166                 int product = noOfRows*noOfCols;
167                 for (int i = 0; i < product; ++i) {
168                         this->data[i] = rhs.data[i];
169                 }
170         }
171         // return the reference to self
172         return *this;
173 }
174
175 // 3. ToeplitzTesting & ToeplitzTestingHelper
176
177 //print function
178 Matrix Matrix::Print (const double* const data_rhs, const int& noOfRows_rhs, const int&
    noOfCols_rhs) {
179
180         //Create Matrix object of zeros using standard custom constructor
181         Matrix Print (noOfRows_rhs, noOfCols_rhs);
182
183         //copy across the content of data_rhs
184         int product = Print.noOfRows*Print.noOfCols;
185         for (int i = 0; i < product; ++i) {
186                         Print.data[i] = data_rhs[i];
187                 }
188
189         //loop through the row indices of print
190         for (int i = 0; i < Print.noOfRows; ++i) {
191         cout << "\t"; // leave a tab (8 spaces) at the beginning of the row
192                 //loop through the column indices of print
193                 for (int j = 0; j < Print.noOfCols; ++j) {
```

```
194                         int index = i + j*Print.noOfRows;       // GetIndex(i,j) cannot be user
    here without object
195                         double element = data_rhs[index];
196                         if (element >= 0) {
197                                 cout << " " << element; // leave an extra space if element is
    positive, to ensure alignment of matrix entries in console output
198                         }
199                         else {
200                                 cout << element;
201                         }
202                         cout << "         "; // leave 9 spaces between columns (tab + 1 doesn't
    work ...)
203                 }
204                 cout << endl; // go to next line at end of row
205         }
206
207         return Print;
208 }
209
210 //generate Toeplitz function
211 Matrix Matrix::Toeplitz (const double* const column, const int& noOfRows_rhs, const double* const
    row, const int& noOfCols_rhs) {
212
213         //Create Matrix object of zeros using standard custom constructor
214         Matrix Toeplitz (noOfRows_rhs, noOfCols_rhs);
215
216         //copy across the content of data_rhs
217         int product = Toeplitz.noOfRows*Toeplitz.noOfCols;
218
219         //Display a warning message if first element of column and first element of row are not
    equal
220         if (column[0] != row[0]) {
221                 cout << "Warning: In Toeplitz constructor, first element of input column does not
    match" << endl;
222                 cout << "         first element of input row! Column wins diagonal conflict." <<
    endl;
223                 cout << endl;
224         }
225
226         //To construct Toeplitz matrix based on columns and row, compressed into a 1D array
    columnwise
227         //loop through the rows indices
228         for (int i = 0 ; i <Toeplitz.noOfRows; i++) {
229                 //loop through the column indices
230                 for (int j = 0 ; j <Toeplitz.noOfCols; j++) {
231                         int index = (i + j*Toeplitz.noOfRows);  // GetIndex(i,j) cannot be user
    here without object
232                         //for diagonal of matrix or below the diagonal
233                         if (i>=j){
234                                 Toeplitz.data[index]=column[i-j];
235                         }
236                         //above this diagonal line
237                         else if (j>i){
238                                 Toeplitz.data[index]=row[j-i];
239                         }
240                 }
241         }
242
243         // return the object
244         return Toeplitz;
245 }
246
247 // 4. TransposeTesting
248
249 //static transpose function
250 Matrix Matrix::Transpose(const Matrix& rhs) {
251         //create an instance of Matrix object called Transpose
```

```
252         Matrix Transpose(rhs);  //calls default copy constructor to replicate rhs
253
254         //invoke the non-static transpose function (see below)
255         Transpose.Transpose();
256
257         //return the object
258         return Transpose;
259
260 }
261
262 //non-static transpose function
263 void Matrix::Transpose () {
264         //swap the number of columns and rows
265         int tmp = 0;
266         tmp = noOfRows;
267         noOfRows = noOfCols;
268         noOfCols = tmp;
269
270         // allocate the memory for temporary holder for data
271         int product = noOfRows*noOfCols;
272         double *tmp_data = new double[product];
273
274         //re-allocate members of transposed data as a tranpose of original data
275         int k = 0; //use this as a counter to index original data
276         //loop through the row indices of tranposed data
277         for (int i = 0; i < noOfRows; ++i) {
278                 //looping through the column indices of transposed data
279                 for (int j = 0; j < noOfCols; ++j) {
280                         int index = GetIndex(i,j); //to index tranposed data
281                         tmp_data[index] = data[k]; //appropriate transformation to switch columns
    and rows (hold temporarily in tmp_data)
282                         k += 1; //increment the counter
283                         }
284                 }
285
286         // then copy across the content from tmp_data to data in a linear fashion
287         for (int i = 0; i < product; ++i) {
288                         data[i] = tmp_data[i];
289         }
290
291         //delete dynamically allocated memory for tmp_data
292         delete [] tmp_data;
293
294         //empty return as void function
295         return;
296 }
297
298 // 5. MultiplicationTesting
299
300 //non-member multiplication function - creates new matrix
301 Matrix operator* (const Matrix& lhs, const Matrix& rhs) {
302
303         Matrix result(lhs); //calls default copy constructor
304         result *= rhs;      //calls self-modifying member multiplication function (see below)
305         return result;      //returns object
306
307 }
308
309 //member multiplication function
310 Matrix& Matrix::operator*= (const Matrix& rhs) {
311
312         //check that number of cols of LHS is equal to number of rows of RHS
313         if (this->noOfCols != rhs.noOfRows) {
314                 cerr << endl;
315                 cerr << "Error:  In member multiplication function ... " << endl;
316                 cerr << "        Inner matrix dimensions of rhs and lhs do not agree." << endl;
317                 cerr << "        Unmultiplied lhs matrix has been returned." << endl;
```

```
318                 cerr << endl;
319                 Matrix error;
320                 return *this;
321         }
322         // The returned matrix will have the same number of rows as LHS matrix
323         // But the returned matrix will have the same number of columns as the RHS matrix
324         // Therefore amend number of columns of returned matrix if need be
325         if (this->noOfCols != rhs.noOfCols) {
326                 int noOfCols_old = this->noOfCols;
327                 int product_old = this->noOfRows*this->noOfCols;
328                 this->noOfCols = rhs.noOfCols;
329                 int product_new = this->noOfRows*this->noOfCols;
330                 if (noOfCols_old < this->noOfCols) {
331                         //extend data by allocating extra columns and padding with
    zeros:
332                         //dynamically allocate memory for array of zeros of appropriate length
333                         double *tmp = new double[product_new];
334                         //initialise these elements all to zero
335                         for (int i = 0; i < product_new; ++i) {
336                                 tmp[i] = 0;
337                         }
338                         //then copy across the content from data to temp
339                         for (int i = 0; i < product_old; ++i) {
340                                 tmp[i] = this->data[i];
341                         }
342                         //reassign data
343                         this->data = tmp;
344                 }
345         }
346
347         // allocate the memory for temporary holder for data
348         int product = this->noOfRows*this->noOfCols;
349         double *tmp_data = new double[product];
350
351         // looping through the row indices of lhs
352         for (int i = 0; i < this->noOfRows; ++i) {
353                 //looping through the column indices of lhs
354                 for (int j = 0; j < this->noOfCols; ++j) {
355                         double tmp = 0; //to store temporary element value of tmp_data in
356                         //looping through the rows of the rhs matrix
357                         for (int k = 0; k < rhs.noOfRows; ++k) {
358                                 int index = GetIndex(i,j); //use to index tmp_data to temporarily
    store result in
359                                 //tmp_data[index] = dot product of (lhs row i) and (rhs col
    j)
360                                 int index_LHS = i + k*this->noOfRows; // use to index lhs matrix
    for calculation
361                                 int index_RHS = k + j*rhs.noOfRows; //use to index rhs matrix for
    calculation
362                                 tmp += this->data[index_LHS] * rhs.data[index_RHS];  //increment
    tmp by this result
363                                 tmp_data[index] = tmp;
364                         }
365                 }
366         }
367
368         // then copy across the content from tmp_data to data
369         for (int i = 0; i < product; ++i) {
370                         this->data[i] = tmp_data[i];
371         }
372
373         //delete dynamically allocated memory for tmp_data
374         delete [] tmp_data;
375
376         // return the reference to self
377         return *this;
378 }
```

```
379
380 // 6. RowColumnExchangeTesting
381
382 //ExchangeRows function (overloaded with two input parameter options)
383
384 //swaps row1 and row2, across all columns
385 Matrix& Matrix::ExchangeRows (const int& row1, const int& row2) {
386
387         //looping through all columns
388         for (int j = 0; j < noOfCols; ++j) {
389                 int index_row1 = GetIndex(row1,j);    //get index for element in row1
390                 double tmp = data[index_row1];        //store in tmp variable
391                 int index_row2 = GetIndex(row2,j);    //get index for element in row2
392                 data[index_row1] = data[index_row2];  //swap the elements
393                 data[index_row2] = tmp;               //swap the elements
394         }
395
396         // return the reference to self
397         return *this;
398 }
399
400 //swaps row1 and row2, but only between col1 and col2
401 Matrix& Matrix::ExchangeRows (const int& row1, const int& row2, const int& col1, const int& col2) {
402
403         //looping through columns from col1 to col2
404         for (int j = col1; j <= col2; ++j) {
405                 int index_row1 = GetIndex(row1,j);    //get index for element in row1
406                 double tmp = data[index_row1];        //store in tmp variable
407                 int index_row2 = GetIndex(row2,j);    //get index for element in row2
408                 data[index_row1] = data[index_row2];  //swap the elements
409                 data[index_row2] = tmp;               //swap the elements
410         }
411
412         // return the reference to self
413         return *this;
414 }
415
416 //ExchangeColumns function (overloaded with two input parameter options)
417
418 //swaps col1 and col2, across all rows
419 Matrix& Matrix::ExchangeColumns (const int& col1, const int& col2) {
420
421         //looping through all rows
422         for (int i = 0; i < noOfRows; ++i) {
423                 int index_col1 = GetIndex(i,col1);    //get index for element in col1
424                 double tmp = data[index_col1];        //store in tmp variable
425                 int index_col2 = GetIndex(i,col2);    //get index for element in col2
426                 data[index_col1] = data[index_col2];  //swap the elements
427                 data[index_col2] = tmp;               //swap the elements
428         }
429
430         // return the reference to self
431         return *this;
432
433 }
434
435 //swaps col1 and col2, but only between row1 and row2
436 Matrix& Matrix::ExchangeColumns (const int& col1, const int& col2, const int& row1, const int& row2)
    {
437
438         //looping through rows between row1 and row2
439         for (int i = row1; i <= row2; ++i) {
440                 int index_col1 = GetIndex(i,col1);    //get index for element in col1
441                 double tmp = data[index_col1];        //store in tmp variable
442                 int index_col2 = GetIndex(i,col2);    //get index for element in col2
443                 data[index_col1] = data[index_col2];  //swap the elements
444                 data[index_col2] = tmp;               //swap the elements
```

```
445         }
446
447         // return the reference to self
448         return *this;
449
450 }
451
452 // 7. OtherTesting
453
454 //Zeros (sets every entry to zero)
455 void Matrix::Zeros () {
456         //Assign 0 to every element in data, in a linear fashion
457         int product = noOfRows*noOfCols;
458         for (int i = 0; i < product; ++i) {
459                 data[i] = 0;
460         }
461
462         //empty return as void function
463         return;
464 }
465
466 //Ones (sets every entry to one)
467 void Matrix::Ones () {
468         //Assign 1 to every element in data, in a linear fashion
469         int product = noOfRows*noOfCols;
470         for (int i = 0; i < product; ++i) {
471                 data[i] = 1;
472         }
473
474         //empty return as void function
475         return;
476 }
477
478 //GetNoOfRows (find out number of rows)
479 int Matrix::GetNoOfRows () const {
480         return noOfRows;
481 }
482
483 //GetNoOfColumns (find out number of columns)
484 int Matrix::GetNoOfColumns () const {
485         return noOfCols;
486 }
487
488 //GetEntry (find out value of particular entry (i,j) in Matrix, starting indexing at (0,0) in upper
    left corner of matrix)
489 double Matrix::GetEntry (const int& rowIdx, const int& columnIdx) const {
490         int index= GetIndex (rowIdx, columnIdx);
491         return data[index];
492 }
493
494 // Part 2.
495
496 //To construct user specified matrices for testing functions
497 Matrix Matrix::Test (const double* data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs) {
498
499         // create an empty Matrix object
500         Matrix Test;
501
502         // then set up the dimensions
503         Test.noOfRows = noOfRows_rhs;
504         Test.noOfCols = noOfCols_rhs;
505
506         // allocate the memory
507         int product = noOfRows_rhs*noOfCols_rhs;
508         Test.data = new double[product];
509
510         //then copy across the content from data_rhs to Test.data
```

```
511         for (int i = 0; i < product; ++i) {
512                 Test.data[i] = data_rhs[i];
513         }
514
515         // return the object
516         return Test;
517 }
518
519 //Print function to print matrices in default MATLAB style to fixed 4.d.p precision
520 Matrix Matrix::PrintMATLAB (const Matrix& rhs) {
521         //Need to create an empty SquareMatrix object
522         Matrix print;
523
524         // then set up the dimensions
525         print.noOfRows = rhs.noOfRows;
526         print.noOfCols = rhs.noOfCols;
527
528         // allocate the memory
529         int product = print.noOfRows*print.noOfCols;
530         print.data = new double[product];
531
532         //copy across the content of data_rhs
533         for (int i = 0; i < product; ++i) {
534                         print.data[i] = rhs.data[i];
535                 }
536
537         //set to display numbers to fixed 4.d.p.
538         cout << fixed << setprecision(4);
539
540         //loop through the row indices of print
541         for (int i = 0; i < print.noOfRows; ++i) {
542         cout << "\t"; // leave a tab (8 spaces) at the beginning of the row
543                 //loop through the column indices of print
544                 for (int j = 0; j < print.noOfCols; ++j) {
545                         int index = i + j*print.noOfRows;       // GetIndex(i,j) cannot be user
    here without object
546                         double element = print.data[index];
547                         double tolerance = 1e-14;
548                         if (abs(element) < tolerance) {
549                                 element = 0;  //set to zero if magnitude of element is less than
    tolerance;
550                         }
551                         if (element == 0) {
552                                 cout << "      0";  //leave 6 spaces to ensure alignment
553                         }
554                         else if (element >= 0) {
555                                 cout << " " << element; // leave an extra space if element is
    positive, to ensure alignment of matrix entries in console output
556                         }
557                         else {
558                                 cout << element;
559                         }
560                         cout << "         "; // leave 9 spaces between columns (tab + 1 doesn't
    work ...)
561                 }
562                 cout << endl; // go to next line at end of row
563         }
564         cout << endl;
565
566         cout.unsetf(ios::floatfield);       // unset floatfield from 'fixed' to 'not set'
567         cout << setprecision(6);            // set precision back to default value of 6.d.p
568
569         return print;
570 }
571
572 // Part 3.
573
```

```
574  // 8. Forward Substitution - Algorithm 3.1 - implemented here so can access and manipulate column
     vector b
575
576  Matrix Matrix::ForwardSub (const Matrix& b) const {
577
578          //construct a m-by-1 column vector to hold return result y in
579          int dim = this->noOfCols;
580          Matrix y = Matrix(dim,1);
581
582          //Expanding from the top row of L:
583          //1. Calculate first entry of y
584          y.data[0] = b.data[0]/this->data[0];
585
586          //2. Calculate subsequent entries of y
587          for (int k= 1; k <dim; ++k) {
588                  double tmp = 0;
589                  for (int i= 0; i <k; ++i) {
590                          int index_ki = GetIndex(k,i);
591                          tmp += (this->data[index_ki]*y.data[i]);
592                  }
593                  int index_kk = GetIndex(k,k);
594                  y.data[k] =  (b.data[k] - tmp)/this->data[index_kk];
595
596          }
597
598          return y;
599
600  }
601
602  // 9. Backward Substitution - Algorithm 3.2 - implemented here so can access and manipulate column
     vector y
603
604  Matrix Matrix::BackwardSub (const Matrix& y) const {
605
606          //construct a m-by-1 column vector to hold return result x in
607          int dim = this->noOfCols;
608          Matrix x = Matrix(dim,1);
609
610          //Expanding from the bottom row of U:
611          //1. Calculate first entry of x
612          x.data[dim-1] = y.data[dim-1]/this->data[dim*dim-1];
613
614          //2. Calculate subsequent entries of x
615          for (int k= dim-2; k >=0; --k) {
616                  //cout << k << endl;
617                  double tmp = 0;
618                  for (int i= k+1; i <dim; ++i) {
619                          int index_ki = GetIndex(k,i);
620                          tmp += (this->data[index_ki]*x.data[i]);
621                  }
622                  int index_kk = GetIndex(k,k);
623                  x.data[k] =  (y.data[k] - tmp)/this->data[index_kk];
624          }
625
626          return x;
627
628  }
629
630  // 10. Solving a system of linear equations in which the number of equations is equal to the number
     of unknowns, i.e. a unique determined solution exists.
631
632  Matrix Matrix::SolveUnique (const Matrix& L, const Matrix& U, const Matrix& P, Matrix& b) const {
633
634          //multiply P by b
635          b = P*b;                 //NB assignment operator throws an error here if number of columns
     in b and P do not match
636
```

```
637         //solve for y in Ly=b using Algorithm 3.1
638         Matrix y = L.ForwardSub(b);
639 
640         //solve for x in Ux=y using Algorithm 3.2
641         Matrix x = U.BackwardSub(y);
642 
643         return x;
644 }
645 
646 // 11. Solving an overdetermined system of linear equations in which the number of equations is
    greater than the number of unknowns, i.e. a least squares solution is produced.
647 
648 Matrix Matrix::LeastSquaresHelper (const Matrix& b, const string& output) const {
649 
650         //check that number of rows in A is bigger than the number of columns in A
651         if (this->noOfRows == this->noOfCols and output == "A'*A") {
652                 cout << "Warning: In function LeastSquaresHelper ..." << endl;
653                 cout << "         Matrix A is square." << endl;
654                 cout << "         An exact solution, not a least squares solution, will be
    provided." << endl;
655                 cout << "         Consider using SolveUnique instead." << endl;
656                 cout << endl;
657         }
658         // check that m > n - error #1
659         if (this->noOfRows < this->noOfCols) {
660                 cerr << endl;
661                 cerr << "Error:  In function LeastSquaresHelper ... " << endl;
662                 cerr << "        In matrix A, the number of rows is less than the number of
    columns." << endl;
663                 cerr << endl;
664                 Matrix error;
665                 return error;
666         }
667         //check that b is of appropriate length -  error #2
668         if (this->noOfRows != b.noOfRows) {
669                 cerr << "Error:  In function LeastSquaresHelper ... " << endl;
670                 cerr << "        The number of rows in b does not equal the number of rows in A."
    << endl;
671                 cerr << endl;
672                 Matrix error;
673                 return error;
674         }
675 
676         //produce ATA
677         Matrix A (*this);
678         Matrix AT = Matrix::Transpose(A);
679         Matrix ATA = AT*A;
680 
681         //produce ATb
682         Matrix ATb = AT*b;
683 
684         //return ATA or ATb dependent upon output string
685         if (output == "A'*A") {
686                 cout << "Number of rows in A'*A = " << ATA.GetNoOfRows()  << endl;
687                 cout << "Number of columns in A'*A = " << ATA.GetNoOfColumns() << endl;
688                 cout << endl;
689                 return ATA;
690         }
691         if (output == "A'*b") {
692                 return ATb;
693         }
694 }
```

```
1  /*
2   * SquareMatrix.h
3   *
4   *  Created on: 21.12.2017
5   *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
6   */
7
8  // Header file for class SquareMatrix - extends class Matrix to provide specialised support for
   square m-by-m matrices.
9
10 // Declares class data type called SquareMatrix.
11
12 // Include header guards to avoid multiple attempts at defining class.
13
14 #ifndef SQUAREMATRIX_H_
15 #define SQUAREMATRIX_H_
16
17 #include "Matrix.h"
18
19 class SquareMatrix : public Matrix { // class SquareMatrix inherits publicly from class Matrix
20
21 private:
22
23         void checkSquare () const;  //to check if user is attempting to make square copy of
   unsquare matrix
24
25 public:
26
27 // Part 2.
28
29 // 1. ConstructorsTesting
30
31         //constructors
32         SquareMatrix (); //default constructor
33         SquareMatrix (const int& dim); // standard custom constructor
34         SquareMatrix (const SquareMatrix& input); // default copy constructor #1  //returns
   SquareMatrix copy of SquareMatrix
35         SquareMatrix (const Matrix& input); // default copy constructor #2        //returns
   SquareMatrix copy of Matrix
36
37         // named static constructors:
38         static SquareMatrix Zeros (const int& dim);
39         static SquareMatrix Ones (const int& dim);
40         static SquareMatrix Eye (const int& dim);
41         static SquareMatrix Test (const double* data_rhs, const int& dim); //for constructing
   specified test matrices for testing
42
43         // default destructor
44         ~SquareMatrix ();
45
46 // 2. ToeplitzTesting & ToeplitzTestingHelper
47
48         static SquareMatrix Toeplitz (const double* const row, const int& dim); //creates a new
   Toeplitz matrix when only first row is specified
49         static SquareMatrix Toeplitz (const double* const column, const double* const row, const
   int& dim); //creates a new Toeplitz matrix when both first column and first row are specified
50
51 // 3. TransposeTesting
52
53         //no new function declarations required.
54
55 // 4. TriangularExtractionTesting
56
57         //extract upper triangular part of square matrix
58         SquareMatrix TriUpper () const;  // non-static member function
59
60         //extract lower triangular part of square matrix
```

```
61          SquareMatrix TriLower () const;  // non-static member function
62
63  // 5. LUDecompositionTesting - Algorithm 2.1 - Gaussian Elimination without Pivoting
64
65          SquareMatrix LUDecompositionOne (const char& output) const;
66                  //Takes a non-singular square matrix A (i.e. with non-zero determinant) as input
    (i.e. invoking square matrix).
67                  //Returns lower triangular matrix L if output = 'L', or upper triangular matrix U
    if output = 'U', such that A=LU.
68
69  // 6. LUDecompositionTesting - Algorithm 2.2 - Gaussian Elimination with Partial Pivoting
70
71          SquareMatrix LUDecompositionTwo (const char& output) const;
72                  //Takes a non-singular square matrix A (i.e. with non-zero determinant) as input
    (i.e. invoking square matrix).
73                  //Returns lower triangular matrix L if output = 'L', or upper triangular matrix U
    if output = 'U', or permutation matrix P if output = 'P', such that PA=LU.
74
75  // Part 3.
76
77  // 7. Forward Substitution - Algorithm 3.1
78
79          Matrix ForwardSub (const Matrix& b) const;
80                  //Invoking matrix L is a lower triangular m-by-m square matrix.
81                  //Input b is a m-by-1 column vector.
82                  //Returns a m-by-1 column vector y, such that Ly=b (via a call to the Matrix class
    function).
83
84  // 8. Backward Substitution - Algorithm 3.2
85
86          Matrix BackwardSub (const Matrix& y) const;
87                  //Invoking matrix U is an upper triangular m-by-m square matrix.
88                  //Input y is a m-by-1 column vector.
89                  //Returns a m-by-1 column vector x, such that Ux=y (via a call to the Matrix class
    function).
90
91  // 9. Solving a system of linear equations in which the number of equations is equal to the number
    of unknowns, i.e. a uniquely determined solution exists.
92
93          Matrix SolveUnique (Matrix& b) const;
94                  //Function is invoked by a m-by-m square matrix A, which represents the matrix of
    coefficients
95                  //Input b is a m-by-1 column vector.
96                  //Returns a m-by-1 column vector x, such that Ax=b (via a call to the Matrix class
    function).
97
98  };
99
100 #endif /* SQUAREMATRIX_H_ */
```

```cpp
/*
 * SquareMatrix.cpp
 *
 *  Created on: 21.12.17
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Source file for class SquareMatrix, to define implementations for functions declared in SquareMatrix.h

#include "SquareMatrix.h"
#include <cstdlib>
#include <cmath>  //for abs function

using namespace std;

// Part 2.

// 1. ConstructorsTesting

//default constructor
SquareMatrix::SquareMatrix () : Matrix () {
}

//standard custom constructor
SquareMatrix::SquareMatrix (const int& dim) : Matrix(dim,dim) {
}

//default copy constructor #1
SquareMatrix::SquareMatrix (const SquareMatrix& input) : Matrix(input) {   //returns SquareMatrix copy of SquareMatrix
}

//default copy constructor #2
SquareMatrix::SquareMatrix (const Matrix& input) : Matrix(input) {   //can return SquareMatrix copy of Matrix if Matrix is square
        //error check to see if attempting to return square copy of non-square matrix ...
        checkSquare();
}

void SquareMatrix::checkSquare () const {
        if (this->noOfRows != this->noOfCols) {
                cerr << endl;
                cerr << "Error:  In default copy constructor #2 ..." << endl;
                cerr << "       Cannot create a square matrix with unequal dimensions." << endl;
                cerr << "        Exiting program ... " << endl;
                cerr << endl;
                exit(1);
        }
}

//static zeros constructor
SquareMatrix SquareMatrix::Zeros(const int& dim) {

        //return an instance of type SquareMatrix invoked by static zeros function in class Matrix of appropriate dimensions
        return Matrix::Zeros(dim,dim);
}

//static ones constructor
SquareMatrix SquareMatrix::Ones(const int& dim) {

        //return an instance of type SquareMatrix invoked by static ones function in class Matrix of appropriate dimensions
        return Matrix::Ones(dim,dim);
}

```

```
63  //static identity matrix constructor
64  SquareMatrix SquareMatrix::Eye(const int& dim) {
65
66          // ceate a SquareMatrix object of zeros using the standard custom constructor
67          SquareMatrix Eye(dim);
68
69          //loop through the row indices of Eye
70          for (int i = 0; i < dim; ++i) {
71                  //looping through the column indices of Eye
72                  for (int j = 0; j < dim; ++j) {
73                          int index = i + j*dim; //to index Eye data   (cannot use GetIndex(i,j) here)
74                          if (i==j) {
75                                  Eye.data[index] = 1; //assign '1' to entries on main diagonal
76                          }
77                          else {
78                                  Eye.data[index] = 0; //assign '0' to other entries
79                          }
80                  }
81          }
82
83          // return the object
84          return Eye;
85  }
86
87  //for constructing specified test square matrices for general testing
88  SquareMatrix SquareMatrix::Test (const double* data_rhs, const int& dim) {
89
90          return Matrix::Test (data_rhs, dim, dim);
91  }
92
93  //destructor
94  SquareMatrix::~SquareMatrix () {
95          //will always invoke base class destructor, therefore do not delete dynamic memory here
96  }
97
98  // 2. ToeplitzTesting & ToeplitzTestingHelper
99
100 //creates a new Toeplitz square matrix when only first row is specified
101 SquareMatrix SquareMatrix::Toeplitz (const double* const row, const int& dim) {
102
103         return Matrix::Toeplitz (row, dim, row, dim);
104 }
105
106 //creates a new Toeplitz square matrix when both first column and first row are specified
107 SquareMatrix SquareMatrix::Toeplitz (const double* const column, const double* const row, const int&
    dim) {
108
109         return Matrix::Toeplitz (column, dim, row, dim);
110 }
111
112 // 3. TransposeTesting
113
114 //no new function declarations required.
115
116 // 4. TriangularExtractionTesting
117
118 //extract upper triangular part of square matrix, converts all values below the main diagonal to
    zero, non-static member version
119 SquareMatrix SquareMatrix::TriUpper () const {
120
121         //initialise result with invoking matrix using default copy constructor
122         SquareMatrix result(*this);
123
124         int dim = this->noOfRows;
125
126         // looping through the row indices of square matrix
127         for (int i = 0; i < dim; ++i) {
```

```
128                 //looping through the column indices of square matrix
129                 for (int j = 0; j < dim; ++j) {
130                         int index= GetIndex (i, j);  //to index entries
131                         if (i>j) {
132                                 result.data[index] = 0;    //set values below the main diagonal to
    zero
133                         }
134                 }
135         }
136
137         return result;
138 }
139
140 //extract lower triangular part of square matirx, converts all values above the main diagonal to
    zero, non-static member version
141 SquareMatrix SquareMatrix::TriLower () const {
142
143         //initialise result with invoking matrix using default copy constructor
144         SquareMatrix result(*this);
145
146         int dim = this->noOfRows;
147
148         // looping through the row indices of square matrix
149         for (int i = 0; i < dim; ++i) {
150                 //looping through the column indices of square matrix
151                 for (int j = 0; j < dim; ++j) {
152                         int index= GetIndex (i, j);     //to index entries
153                         if (i<j) {
154                                 result.data[index] = 0; //set values above the main diagonal to zero
155                         }
156                 }
157         }
158
159         return result;
160 }
161
162 // 5. LUDecompositionTesting - Algorithm 2.1 - Gaussian Elimination without Pivoting
163
164 //Non-Static version read only function
165 SquareMatrix SquareMatrix::LUDecompositionOne (const char& output) const {
166
167         //initialise U with invoking matrix using default copy constructor
168         SquareMatrix U(*this);
169
170         //initialise L with an identity matrix of appropriate dimension
171         int dim = this->noOfRows;
172         SquareMatrix L = SquareMatrix::Eye(dim);
173
174         //looping through the column indices of L and U
175         for (int k = 0; k < dim-1; ++k) {
176                 //looping through the row indices of L and U
177                 for (int j = k+1; j < dim; ++j) {
178
179                         int index_jk = GetIndex(j,k);   //to index jk entries
180                         int index_kk = GetIndex(k,k);   //to index kk entries
181
182                         //to check if diagonal entries in U (aka pivots) are zero
183                         if (U.data[index_kk] == 0) {
184                                 cerr << endl;
185                                 cerr << "Error:  In Algorithm 2.1 - Gaussian Elimination without
    Pivoting ..." << endl;
186                                 cerr << "\tA zero pivot value has occurred." <<
    endl;
187                                 cerr << "\tAlgorithm failed." << endl;
188                                 cerr << endl;
189                                 SquareMatrix error;   //use the default constructor to return the
    minimal version of SquareMatrix
```

```
190                                 return error;
191                         }
192
193                         //if not then proceed with algorithm  (NB - no tmp data holders required here)
194                         else {
195                                 //to alter L_jk value
196                                 L.data[index_jk] = U.data[index_jk]/U.data[index_kk];
197
198                                 //to alter U values in row j from columns k to dim
199                                 for (int i = k; i < dim; ++i) {
200                                         int index_ji = GetIndex(j,i);   //to index j,k:m entries
201                                         int index_ki = GetIndex(k,i);   //to index k,k:m entries
202                                         U.data[index_ji] -=  (L.data[index_jk]*U.data[index_ki]);
203                                 }
204                         }
205                 }
206         }
207
208         if (output == 'L') {
209                 return L;
210         }
211         if (output == 'U') {
212                 return U;
213         }
214
215 }
216
217 // 6. LUDecompositionTesting - Algorithm 2.2 - Gaussian Elimination with Partial Pivoting
218
219 //Non-static read only function
220 SquareMatrix SquareMatrix::LUDecompositionTwo (const char& output) const {
221
222         //initialise U with invoking matrix using default copy constructor
223         SquareMatrix U(*this);
224
225         //initialise L and P with an identity matrix of appropriate dimension
226         int dim = this->noOfRows;
227         SquareMatrix L = SquareMatrix::Eye(dim);
228         SquareMatrix P = SquareMatrix::Eye(dim);
229
230         //looping through the column indices of L, U and P
231         for (int k = 0; k < dim-1; ++k) {
232
233                 // select i>=k to maximise magnitude of U_ik  (i.e. for column k, determine entry, on or below the diagonal of U, with the largest absolute value)
234                 int i = 0;
235                 double max = 0;
236                 for (int b = k; b < dim; ++b) {                 //use b instead of i here to avoid confusion
237                         int index_bk = GetIndex(b,k);           //to index bk entries
238                         double tmp = abs(U.data[index_bk]);
239                         if (tmp > max) {
240                                 max = tmp;
241                                 i = b;
242                         }
243                 }
244
245                 // U_ik is now the new pivot in column k
246                 // Swap the row (k) containing the diagonal of the column (the default pivot) with the row (i) containing the new pivot, in all 3 matrices:
247
248                 //interchange U_(k,k:m) and U_(i,k:m) (i.e. swap row k with row i in U matrix, but only between columns k to m)
249                 U.ExchangeRows(k, i, k, dim-1);
250
```

```
251                 //interchange L_(k,1:k-1) and L_(i,1:k-1) (i.e. swap row k with row i in L matrix,
    but only between columns 1 to k-1)
252                 L.ExchangeRows(k, i, 0, k-1);
253
254                 //Interchange P_(k,:) and P (i,:) (i.e. swap row k with row i in P matrix, across
    all columns)
255                 P.ExchangeRows(k, i);
256
257                 //looping through the row indices of L and U
258                 for (int j = k+1; j < dim; ++j) {
259
260                         int index_jk = GetIndex(j,k);   //to index jk entries
261                         int index_kk = GetIndex(k,k);   //to index kk entries
262
263                         //to check if diagonal entries in U (aka pivots) are zero
264                         if (U.data[index_kk] == 0) {
265                                 cerr << endl;
266                                 cerr << "Error:  In Algorithm 2.2 - Gaussian Elimination with
    Partial Pivoting ..." << endl;
267                                 cerr << "\tA zero pivot value has occurred." << endl;
268                                 cerr << "\tAlgorithm failed." << endl;
269                                 cerr << endl;
270                                 SquareMatrix error;   //use the default constructor to return the
    minimal version of SquareMatrix
271                                 return error;
272                         }
273
274                         //if not then proceed with algorithm  (NB - no tmp data holders required
    here)
275                         else {
276                                 //to alter L_jk value
277                                 L.data[index_jk] = U.data[index_jk]/U.data[index_kk];
278
279                                 //to alter U values in row j from columns k to dim  (swapped i for
    a here to avoid ambiguity)
280                                 for (int a = k; a < dim; ++a) {
281                                         int index_ja = GetIndex(j,a);   //to index j,k:m entries
282                                         int index_ka = GetIndex(k,a);   //to index k,k:m
    entries
283                                         U.data[index_ja] -=  (L.data[index_jk]*U.data[index_ka]);
284                                 }
285                         }
286                 }
287         }
288
289         if (output == 'L') {
290                 return L;
291         }
292         if (output == 'U') {
293                 return U;
294         }
295         if (output == 'P') {
296                 return P;
297         }
298 }
299
300 // Part 3.
301
302 // 7. Forward Substitution - Algorithm 3.1
303
304 Matrix SquareMatrix::ForwardSub (const Matrix& b) const {
305
306         //Check that invoking matrix is lower rectangular - error #1
307         int dim = this->noOfCols;
308         for (int i = 0; i < dim; ++i) {
309                 for (int j = 0; j < dim; ++j) {
310                         int index= GetIndex (i, j);
```

```
311                         if (i<j) {
312                                 double tolerance = 1e-14;
313                                 if (abs(this->data[index]) > tolerance ) {
314                                         cerr << endl;
315                                         cerr << "Error:  In Algorithm 3.1 - Forward
    Substitution..." << endl;
316                                         cerr << "\tMatrix L is not lower rectangular." << endl;
317                                         cerr << endl;
318                                         Matrix error;   //use the default constructor to return the
    minimal version of Matrix
319                                         return error;
320                                 }
321                         }
322                 }
323         }
324
325         //Check that matrix b is of appropriate length - error #2
326         int rows_b = b.GetNoOfRows();
327         if (this->noOfRows != rows_b) {
328                 cerr << endl;
329                 cerr << "Error:  In Algorithm 3.1 - Forward Substitution..." << endl;
330                 cerr << "\tNumber of rows in b does not match dimensions of L." << endl;
331                 cerr << endl;
332                 Matrix error;   //use the default constructor to return the minimal version of
    Matrix
333                 return error;
334         }
335
336         //check that none of the diagonal entries in L are zero - error #3
337         for (int i = 0; i < dim; ++i) {
338                 for (int j = 0; j < dim; ++j) {
339                         int index= GetIndex (i, j);
340                         if (i==j) {
341                                 double tolerance = 1e-14;
342                                 if (abs(this->data[index]) < tolerance ) {
343                                         cerr << endl;
344                                         cerr << "Error:  In Algorithm 3.1 - Forward
    Substitution..." << endl;
345                                         cerr << "\tMatrix L cannot have a zero valued diagonal
    entry." << endl;
346                                         cerr << endl;
347                                         Matrix error;   //use the default constructor to return the
    minimal version of Matrix
348                                         return error;
349                                 }
350                         }
351                 }
352         }
353
354         //once error checks are complete refer function to class Matrix for non-square matrix
    arithmetic and to access manipulate protected values in b
355         return Matrix::ForwardSub(b);
356
357 }
358
359 // 8. Backward Substitution - Algorithm 3.2
360
361 Matrix SquareMatrix::BackwardSub (const Matrix& y) const {
362
363         //Check that invoking matrix is upper rectangular - error #1
364         int dim = this->noOfCols;
365         for (int i = 0; i < dim; ++i) {
366                 for (int j = 0; j < dim; ++j) {
367                         int index= GetIndex (i, j);
368                         if (i>j) {
369                                 double tolerance = 1e-14;
370                                 if (abs(this->data[index]) > tolerance ) {
```

```
371                                             cerr << endl;
372                                             cerr << "Error:  In Algorithm 3.2 - Backward
    Substitution..." << endl;
373                                             cerr << "\tMatrix U is not upper rectangular." << endl;
374                                             cerr << endl;
375                                             Matrix error;   //use the default constructor to return the
    minimal version of Matrix
376                                             return error;
377                                     }
378                             }
379                     }
380             }
381
382             //Check that matrix b is of appropriate length - error #2
383             int rows_y = y.GetNoOfRows();
384             if (this->noOfRows != rows_y) {
385                     cerr << endl;
386                     cerr << "Error:  In Algorithm 3.2 - Backward Substitution..." << endl;
387                     cerr << "\tNumber of rows in y does not match dimensions of U." << endl;
388                     cerr << endl;
389                     Matrix error;   //use the default constructor to return the minimal version of
    Matrix
390                     return error;
391             }
392
393             //check that none of the diagonal entries in U are zero - error #3
394             for (int i = 0; i < dim; ++i) {
395                     for (int j = 0; j < dim; ++j) {
396                             int index= GetIndex (i, j);
397                             if (i==j) {
398                                     double tolerance = 1e-14;
399                                     if (abs(this->data[index]) < tolerance ) {
400                                             cerr << endl;
401                                             cerr << "Error:  In Algorithm 3.2 - Backward
    Substitution..." << endl;
402                                             cerr << "\tMatrix U cannot have a zero valued diagonal
    entry." << endl;
403                                             cerr << endl;
404                                             Matrix error;   //use the default constructor to return the
    minimal version of Matrix
405                                             return error;
406                                     }
407                             }
408                     }
409             }
410
411             //once error checks are complete refer function to class Matrix for non-square matrix
    arithmetic and to access protected values in y
412             return Matrix::BackwardSub(y);
413
414     }
415
416     // 9. Solving a system of linear equations in which the number of equations is equal to the number
    of unknowns, i.e. a uniquely determined solution exists.
417
418     Matrix SquareMatrix::SolveUnique (Matrix& b) const {
419
420             //Check that matrix b is of appropriate length - error #1
421             int rows_b = b.GetNoOfRows();
422             if (this->noOfRows != rows_b) {
423                     cerr << endl;
424                     cerr << "Error:  In function SolveUnique..." << endl;
425                     cerr << "\tNumber of rows in b does not match dimensions of A." << endl;
426                     cerr << endl;
427                     Matrix error;   //use the default constructor to return the minimal version of
    Matrix
428                     return error;
```

```
429         }
430
431         //decompose A into L, U and P matrices
432         SquareMatrix L = this->LUDecompositionTwo('L');
433         //check that LU decomposition has occured succesfully - error #2
434         if (L.noOfRows == 0) {
435                 //error message from LUDecompositionTwo will be displayed
436                 Matrix error;
437                 return error;
438         }
439         SquareMatrix U = this->LUDecompositionTwo('U');
440         SquareMatrix P = this->LUDecompositionTwo('P');
441
442         //pass these matrices to class Matrix for non-square matrix arithmetic and to access and
    manipulate protected values in b
443         return Matrix::SolveUnique(L, U , P, b);
444 }
```

```cpp
        /*
 * SquareMatrixTest.cpp
 *
 *  Created on: 21.12.2017
 *      Adapted from code provided by: Gary Hui Zhang (gary.zhang@ucl.ac.uk)
 */

// A program to demonstrate and test the Matrix class and the SquareMatrix class

#include "Matrix.h"
#include "SquareMatrix.h"
#include <iostream>
#include <cstdlib>  //need this for 'system' functions


using namespace std;

void ConstructorsTesting () {
        cout << "Testing the SquareMatrix constructors:" << endl;
        cout << endl;

        cout << "Case 1: Creating a square matrix of zeros with the standard constructor:" << endl;
        {
                SquareMatrix matrix(4);  //4 is dim in example.m
                cout << matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Creating a square matrix of zeros with the static Zeros function: " <<
endl;
        {
                SquareMatrix matrix = SquareMatrix::Zeros(4);
                cout << matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Creating a square matrix of ones with the static Ones function: " << endl;
        {
                SquareMatrix matrix1 = SquareMatrix::Ones(4);
                cout << matrix1 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Creating an identity matrix with the static Eye function:" << endl;
        {
                SquareMatrix matrix2 = SquareMatrix::Eye(4);
                cout << matrix2 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 5: Copying an identity matrix with a copy constructor:" << endl;
        {
                SquareMatrix matrix2 = SquareMatrix::Eye(4);
                cout << endl;
                cout << "The input matrix = " << endl;
                cout << matrix2 << endl;
                SquareMatrix copy_matrix2 (matrix2);
                cout << "The copy = " << endl;
                cout << copy_matrix2 << endl;
```

```
67                  cout << "Press return to continue ..." << flush;
68                  system("read");
69                  cout << endl;
70          }
71
72          cout << "Case 6: Constructing a user defined matrix with the test constructor:" << endl;
73          {
74                  double vector[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
75                  SquareMatrix matrix = SquareMatrix::Test(vector, 4);
76                  cout << matrix << endl;
77                  cout << "Press return to continue ..." << flush;
78                  system("read");
79                  cout << endl;
80          }
81
82          return;
83  }
84
85  //'first row only' overloaded variety
86  void ToeplitzTestingHelper (const double* const row, const int& dim, const double* const expected) {
87          cout << "The 1st row = " << endl;
88          Matrix::Print(row, 1, dim);
89          cout << endl;
90          cout << "The matrix created by the toeplitz function in MATLAB = " << endl;
91          Matrix::Print(expected, dim, dim);
92          cout << endl;
93          SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, dim);
94          cout << "The matrix created by SquareMatrix::Toeplitz = " << endl;
95          cout << matrix3 << endl;
96
97          return;
98  }
99
100 //'first column and first row' overloaded variety
101 void ToeplitzTestingHelper (const double* const column, const double* const row, const int& dim,
    const double* const expected) {
102         cout << "The 1st column = " << endl;
103         Matrix::Print(column, dim, 1);
104         cout << endl;
105         cout << "The 1st row = " << endl;
106         Matrix::Print(row, 1, dim);
107         cout << endl;
108         cout << "The matrix created by the toeplitz function in MATLAB = " << endl;
109         Matrix::Print(expected, dim, dim);
110         cout << endl;
111         SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, dim);
112         cout << "The matrix created by SquareMatrix::Toeplitz = " << endl;
113         cout << matrix4 << endl;
114
115         return;
116 }
117
118 void ToeplitzTesting () {
119         cout << "Testing the static functions SquareMatrix::Toeplitz:" << endl;
120         cout << endl;
121
122         cout << "Case 1: When only the first row is specified:" << endl;
123         cout << endl;
124
125         {
126                 double row[4] = {4, 3, 2, 1};  //aka vec1
127                 // matrix should be stored, in 1-D, column-wise
128                 //  4     3     2     1
129                 //  3     4     3     2
130                 //  2     3     4     3
131                 //  1     2     3     4
132                 double expected[16] = {4, 3, 2, 1, 3, 4, 3, 2, 2, 3, 4, 3, 1, 2, 3, 4};
```

```
133                     ToeplitzTestingHelper(row, 4, expected);
134                     cout << "Press return to continue ..." << flush;
135                     system("read");
136                     cout << endl;
137             }
138
139             cout << "Case 2: When both the first column and the first row are specified, and the" <<
    endl;
140             cout << "        first elements are equal:" << endl;
141             cout << endl;
142
143             {
144                     double column[4] = {2, 1, 0, -1};
145                     double row[4] = {2, 3, 4, 5};
146                     // matrix should be stored, in 1-D, column-wise
147                     //  2     3     4     5
148                     //  1     2     3     4
149                     //  0     1     2     3
150                     // -1     0     1     2
151                     double expected[16] = {2, 1, 0, -1, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3, 2};
152                     ToeplitzTestingHelper(column, row, 4, expected);
153                     cout << "Press return to continue ..." << flush;
154                     system("read");
155                     cout << endl;
156             }
157
158             cout << "Case 3: When both the first column and the first row are specified, and the" <<
    endl;
159             cout << "        first elements of each are not equal:" << endl;
160             cout << endl;
161
162             {
163                     double column[4] = {1, 2, 3, 4}; //aka vec2
164                     double row[4] = {4, 3, 2, 1};    //aka vec1
165                     // matrix should be stored, in 1-D, column-wise
166                     //  1     3     2     1
167                     //  2     1     3     2
168                     //  3     2     1     3
169                     //  4     3     2     1
170                     double expected[16] = {1, 2, 3, 4, 3, 1, 2, 3, 2, 3, 1, 2, 1, 2, 3, 1};
171                     ToeplitzTestingHelper(column, row, 4, expected);
172                     cout << "Press return to continue ..." << flush;
173                     system("read");
174                     cout << endl;
175             }
176
177             return;
178     }
179
180     void TransposeTesting () {
181             cout << "Testing the class Matrix Transpose functions for SquareMatrix objects:" << endl;
182             cout << endl;
183
184             cout << "Case 1: the non-static self-modifying Transpose function:" << endl;
185             cout << endl;
186
187             {
188                     // the same matrix as in ToeplitzTesting case 3
189                     double column[4] = {1, 2, 3, 4};  //aka vec2
190                     double row[4] = {4, 3, 2, 1};     //aka vec1
191                     SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
192                     cout << "The original Matrix = " << endl;
193                     cout << matrix4 << endl;
194                     matrix4.Transpose();
195                     SquareMatrix matrix5 = matrix4;
196                     cout << "The transposed version = " << endl;
197                     cout << matrix5 << endl;
```

```
198                 cout << "Press return to continue ..." << flush;
199                 system("read");
200                 cout << endl;
201         }
202
203         cout << "Case 2: the static Transpose function:" << endl;
204         cout << endl;
205
206         {
207                 // the same matrix as in ToeplitzTesting case 3
208                 double column[4] = {1, 2, 3, 4}; //aka vec2
209                 double row[4] = {4, 3, 2, 1};    //aka vec1
210                 SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
211                 cout << "The original Matrix = " << endl;
212                 cout << matrix4 << endl;
213                 SquareMatrix matrix5 = Matrix::Transpose(matrix4);
214                 cout << "The transposed version = " << endl;
215                 cout << matrix5 << endl;
216                 cout << "Press return to continue ..." << flush;
217                 system("read");
218                 cout << endl;
219         }
220
221         return;
222
223 }
224
225 void TriangularExtractionTesting () {
226         cout << "Testing the triangular extraction functions of class SquareMatrix:" << endl;
227         cout << endl;
228
229         cout << "Case 1: the upper triangular extraction non-static function:" << endl;
230
231         {
232                 // the same matrix as in ToeplitzTesting case 3
233                 double column[4] = {1, 2, 3, 4}; //aka vec2
234                 double row[4] = {4, 3, 2, 1};    //aka vec1
235                 cout << endl;
236                 SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
237                 cout << "The original square matrix = " << endl;
238                 cout << matrix4 << endl;
239                 SquareMatrix matrix6 = matrix4.TriUpper();
240                 cout << "The square matrix after upper triangular extraction = " << endl;
241                 cout << matrix6 << endl;
242                 cout << "Press return to continue ..." << flush;
243                 system("read");
244                 cout << endl;
245         }
246
247         cout << "Case 2: the lower triangular extraction non-static function:" << endl;
248
249         {
250                 // the same matrix as in ToeplitzTesting case 3
251                 double column[4] = {1, 2, 3, 4}; //aka vec2
252                 double row[4] = {4, 3, 2, 1};    //aka vec1
253                 cout << endl;
254                 SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
255                 cout << "The original square matrix = " << endl;
256                 cout << matrix4 << endl;
257                 SquareMatrix matrix7 = matrix4.TriLower();
258                 cout << "The square matrix after lower triangular extraction = " << endl;
259                 cout << matrix7 << endl;
260                 cout << "Press return to continue ..." << flush;
261                 system("read");
262                 cout << endl;
263         }
264
```

```
265         return;
266
267 }
268
269 void LUDecompositionTestingOne() {
270         cout << "Testing the LU decomposition of a square matrix using Algorithm 2.1, Gaussian" <<
    endl;
271         cout << "elimination without pivoting:" << endl;
272         cout << endl;
273
274         cout << "Case 1: Matrix A is non-singular and all entries on main diagonal are non-zero:"
    << endl;
275         cout << endl;
276
277         {
278                 // the same matrix as in ToeplitzTesting case 1
279                 double row[4] = {4, 3, 2, 1};  //aka vec1
280                 SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, 4);
281                 cout << "The input Matrix (A) = " << endl;
282                 cout << matrix3 << endl;
283                 SquareMatrix lmatrix1 = matrix3.LUDecompositionOne('L');
284                 cout << "The L Matrix = " << endl;
285                 Matrix::PrintMATLAB(lmatrix1);
286                 SquareMatrix umatrix1 = matrix3.LUDecompositionOne('U');
287                 cout << "The U Matrix = " << endl;
288                 Matrix::PrintMATLAB(umatrix1);
289                 cout << "Checking that (L*U = A):" << endl;
290                 cout << lmatrix1 * umatrix1 << endl;
291                 cout << "Press return to continue ..." << flush;
292                 system("read");
293                 cout << endl;
294         }
295
296         cout << "Case 2: Matrix A is non-singular and 2nd, 3rd and 4th entries on main diagonal" <<
    endl;
297         cout << "        are all zero:" << endl;
298         cout << endl;
299
300         {
301                 double data[16] = {6, 3, 6, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
302                 SquareMatrix matrix = SquareMatrix::Test(data, 4);
303                 cout << "The input Matrix (A) = " << endl;
304                 cout << matrix << endl;
305                 SquareMatrix lmatrix = matrix.LUDecompositionOne('L');
306                 cout << "The L Matrix = " << endl;
307                 Matrix::PrintMATLAB(lmatrix);
308                 SquareMatrix umatrix = matrix.LUDecompositionOne('U');
309                 cout << "The U Matrix = " << endl;
310                 Matrix::PrintMATLAB(umatrix);
311                 cout << "Checking that (L*U = A):" << endl;
312                 cout << lmatrix * umatrix << endl;
313                 cout << "Press return to continue ..." << flush;
314                 system("read");
315                 cout << endl;
316         }
317
318         cout << "Case 3: Matrix A is non-singular and only 1st entry on main diagonal is zero:" <<
    endl;
319         cout << endl;
320
321         {
322                 double data[16] = {0, 3, 6, 7, 9, 2, 3, 8, 7, 1, 2, 9, 4, 2, 1, 5};
323                 SquareMatrix matrix = SquareMatrix::Test(data, 4);
324                 cout << "The input Matrix (A) = " << endl;
325                 cout << matrix << endl;
326                 cout << "Attempting to decompose into L and U matrices ... " << endl;
327                 SquareMatrix lmatrix1 = matrix.LUDecompositionOne('L');
```

```
328                     cout << "Press return to continue ..." << flush;
329                     system("read");
330                     cout << endl;
331             }
332
333             return;
334
335     }
336
337     void LUDecompositionTestingTwo() {
338             cout << "Testing the LU decomposition of a square matrix using Algorithm 2.2, Gaussian" <<
    endl;
339             cout << "elimination with partial pivoting:" << endl;
340             cout << endl;
341
342             cout << "Case 1: Matrix A is non-singular and magnitude of each entry on main diagonal is"
    << endl;
343             cout << "        maximal within its column. Partial pivoting is not required: " << endl;
344             cout << endl;
345
346             {
347                     // the same matrix as in ToeplitzTesting case 1
348                     double row[4] = {4, 3, 2, 1};  //aka vec1
349                     SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, 4);
350                     cout << "The input Matrix (A) = " << endl;
351                     cout << matrix3 << endl;
352                     SquareMatrix lmatrix1 = matrix3.LUDecompositionTwo('L');
353                     cout << "The L Matrix = " << endl;
354                     Matrix::PrintMATLAB(lmatrix1);
355                     SquareMatrix umatrix1 = matrix3.LUDecompositionTwo('U');
356                     cout << "The U Matrix = " << endl;
357                     Matrix::PrintMATLAB(umatrix1);
358                     SquareMatrix pmatrix1 = matrix3.LUDecompositionTwo('P');
359                     cout << "The P Matrix = " << endl;
360                     cout << pmatrix1 << endl;
361                     cout << "Checking that (L*U = P*A): " << endl;
362                     cout << endl;
363                     cout << "L*U = " << endl;
364                     cout << lmatrix1 * umatrix1 << endl;
365                     cout << "P*A = " << endl;
366                     cout << pmatrix1 * matrix3 << endl;
367                     cout << "Press return to continue ..." << flush;
368                     system("read");
369                     cout << endl;
370             }
371
372             cout << "Case 2: Matrix A is non-singular and magnitude of each entry on main diagonal is"
    << endl;
373             cout << "        not maximal within its column. Partial pivoting is required: " << endl;
374             cout << endl;
375
376             {
377                     // the same matrix as in ToeplitzTesting case 3
378                     double column[4] = {1, 2, 3, 4}; //aka vec2
379                     double row[4] = {4, 3, 2, 1};    //aka vec1
380                     SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
381                     cout << "The input Matrix (A) = " << endl;
382                     cout << matrix4 << endl;
383                     SquareMatrix lmatrix2 = matrix4.LUDecompositionTwo('L');
384                     cout << "The L Matrix = " << endl;
385                     Matrix::PrintMATLAB(lmatrix2);
386                     SquareMatrix umatrix2 = matrix4.LUDecompositionTwo('U');
387                     cout << "The U Matrix = " << endl;
388                     Matrix::PrintMATLAB(umatrix2);
389                     SquareMatrix pmatrix2 = matrix4.LUDecompositionTwo('P');
390                     cout << "The P Matrix = " << endl;
391                     cout << pmatrix2 << endl;
```

```
392                     cout << "Checking that (L*U = P*A): " << endl;
393                     cout << endl;
394                     cout << "L*U = " << endl;
395                     cout << lmatrix2 * umatrix2 << endl;
396                     cout << "P*A = " << endl;
397                     cout << pmatrix2 * matrix4 << endl;
398                     cout << "Press return to continue ..." << flush;
399                     system("read");
400                     cout << endl;
401             }
402
403             cout << "Case 3: Matrix A is non-singular and all diagonal entries are zero:" << endl;
404             cout << endl;
405
406             {
407                     double data[16] = {0, 3, 6, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
408                     SquareMatrix matrix = SquareMatrix::Test(data, 4);;
409                     cout << "The input Matrix (A) = " << endl;
410                     cout << matrix << endl;
411                     SquareMatrix lmatrix = matrix.LUDecompositionTwo('L');
412                     cout << "The L Matrix = " << endl;
413                     Matrix::PrintMATLAB(lmatrix);
414                     SquareMatrix umatrix = matrix.LUDecompositionTwo('U');
415                     cout << "The U Matrix = " << endl;
416                     Matrix::PrintMATLAB(umatrix);
417                     SquareMatrix pmatrix = matrix.LUDecompositionTwo('P');
418                     cout << "The P Matrix = " << endl;
419                     cout << pmatrix << endl;
420                     cout << "Checking that (L*U = P*A): " << endl;
421                     cout << endl;
422                     cout << "L*U = " << endl;
423                     cout << lmatrix * umatrix << endl;
424                     cout << "P*A = " << endl;
425                     cout << pmatrix * matrix << endl;
426                     cout << "Press return to continue ..." << flush;
427                     system("read");
428                     cout << endl;
429             }
430
431             cout << "Case 4: Matrix A is non-singular, all diagonal entries are zero, and first three"
    << endl;
432             cout << "        entries of first column are zero:" << endl;
433             cout << endl;
434
435             {
436                     double data[16] = {0, 0, 0, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
437                     SquareMatrix matrix = SquareMatrix::Test(data, 4);;
438                     cout << "The input Matrix (A) = " << endl;
439                     cout << matrix << endl;
440                     SquareMatrix lmatrix = matrix.LUDecompositionTwo('L');
441                     cout << "The L Matrix = " << endl;
442                     Matrix::PrintMATLAB(lmatrix);
443                     SquareMatrix umatrix = matrix.LUDecompositionTwo('U');
444                     cout << "The U Matrix = " << endl;
445                     Matrix::PrintMATLAB(umatrix);
446                     SquareMatrix pmatrix = matrix.LUDecompositionTwo('P');
447                     cout << "The P Matrix = " << endl;
448                     cout << pmatrix << endl;
449                     cout << "Checking that (L*U = P*A): " << endl;
450                     cout << endl;
451                     cout << "L*U = " << endl;
452                     cout << lmatrix * umatrix << endl;
453                     cout << "P*A = " << endl;
454                     cout << pmatrix * matrix << endl;
455                     cout << "Press return to continue ..." << flush;
456                     system("read");
457                     cout << endl;
```

```
458         }
459
460         cout << "Case 5: Matrix A is singular, all diagonal entries are zero, and all entries" <<
    endl;
461         cout << "        of first column are zero:" << endl;
462         cout << endl;
463
464         {
465                 double data[16] = {0, 0, 0, 0, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
466                 SquareMatrix matrix = SquareMatrix::Test(data, 4);
467                 cout << "The input Matrix (A) = " << endl;
468                 cout << matrix << endl;
469                 cout << "Attempting to decompose into L, U and P matrices ... " << endl;
470                 SquareMatrix umatrix = matrix.LUDecompositionTwo('L');
471                 cout << "Press return to continue ..." << flush;
472                 system("read");
473                 cout << endl;
474         }
475
476         return;
477 }
478
479 void ForwardSubstitution () {
480
481         cout << "Testing Algorithm 3.1 - Forward Substitution. To solve for y in L*y = b:" << endl;
482         cout << endl;
483
484         cout << "Case 1: Successful Forward Substitution: " << endl;
485         cout << endl;
486
487         {
488                 double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, -3};
489                 SquareMatrix L = SquareMatrix::Test(matrix, 3);
490                 cout << "The lower triangular matrix, L = " << endl;
491                 cout << L << endl;
492                 double vector[3] = {16, 43, 57};
493                 Matrix b = Matrix::Test(vector, 3, 1);
494                 cout << "The column vector, b  = " << endl;
495                 cout << b << endl;
496                 cout << "The result of the forward substitution, the column vector y = " <<
    endl;
497                 Matrix y = L.ForwardSub(b);
498                 cout << y << endl;
499                 cout << "Checking that (L*y = b): " << endl;
500                 cout << L * y << endl;
501                 cout << "Press return to continue ..." << flush;
502                 system("read");
503                 cout << endl;
504         }
505
506         cout << "Case 2: Testing error message #1:" << endl;
507         cout << endl;
508
509         {
510                 double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 1, -3};
511                 SquareMatrix L = SquareMatrix::Test(matrix, 3);
512                 cout << "The input matrix, 'L' = " << endl;
513                 cout << L << endl;
514                 double vector[3] = {16, 43, 57};
515                 Matrix b = Matrix::Test(vector, 3, 1);
516                 cout << "The column vector, b  = " << endl;
517                 cout << b << endl;
518                 cout << "Attempting forward substitution ... " << endl;
519                 Matrix y = L.ForwardSub(b);
520                 cout << "Press return to continue ..." << flush;
521                 system("read");
522                 cout << endl;
```

```
523         }
524
525         cout << "Case 3: Testing error message #2:" << endl;
526         cout << endl;
527
528         {
529                 double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, -3};
530                 SquareMatrix L = SquareMatrix::Test(matrix, 3);
531                 cout << "The lower triangular matrix, L = " << endl;
532                 cout << L << endl;
533                 double vector[4] = {16, 43, 57, 29};
534                 Matrix b = Matrix::Test(vector, 4, 1);
535                 cout << "The column vector, b  = " << endl;
536                 cout << b << endl;
537                 cout << "Attempting forward substitution ... " << endl;
538                 Matrix y = L.ForwardSub(b);
539                 cout << "Press return to continue ..." << flush;
540                 system("read");
541                 cout << endl;
542         }
543
544         cout << "Case 4: Testing error message #3:" << endl;
545         cout << endl;
546
547         {
548                 double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, 0};
549                 SquareMatrix L = SquareMatrix::Test(matrix, 3);
550                 cout << "The lower triangular matrix, L = " << endl;
551                 cout << L << endl;
552                 double vector[3] = {16, 43, 57};
553                 Matrix b = Matrix::Test(vector, 3, 1);
554                 cout << "The column vector, b  = " << endl;
555                 cout << b << endl;
556                 cout << "Attempting forward substitution ... " << endl;
557                 Matrix y = L.ForwardSub(b);
558                 cout << "Press return to continue ..." << flush;
559                 system("read");
560                 cout << endl;
561         }
562
563         return;
564
565 }
566
567 void BackwardSubstitution () {
568
569         cout << "Testing Algorithm 3.2 - Backward Substitution. To solve for x in U*x = y:" << endl;
570         cout << endl;
571
572         cout << "Case 1: Successful Backward Substitution: " << endl;
573         cout << endl;
574
575         {
576                 double matrix[9] = {1, 0, 0, 2, 1, 0, 3, 1, 1};
577                 SquareMatrix U = SquareMatrix::Test(matrix, 3);
578                 cout << "The upper triangular matrix, U = " << endl;
579                 cout << U << endl;
580                 double vector[3] = {16, 5, 4};
581                 Matrix y = Matrix::Test(vector, 3, 1);
582                 cout << "The column vector, y  = " << endl;
583                 cout << y << endl;
584                 cout << "The result of the backward substitution, the column vector x = " << endl;
585                 Matrix x = U.BackwardSub(y);
586                 cout << x << endl;
587                 cout << "Checking that (U*x = y): " << endl;
588                 cout << U * x << endl;
589                 cout << "Press return to continue ..." << flush;
```

```
590                     system("read");
591                     cout << endl;
592             }
593
594             cout << "Case 2: Testing error message #1:" << endl;
595             cout << endl;
596
597             {
598                     double matrix[9] = {1, 0, 1, 2, 1, 0, 3, 1, 1};
599                     SquareMatrix U = SquareMatrix::Test(matrix, 3);
600                     cout << "The input matrix, 'U' = " << endl;
601                     cout << U << endl;
602                     double vector[3] = {16, 43, 57};
603                     Matrix y = Matrix::Test(vector, 3, 1);
604                     cout << "The column vector, y  = " << endl;
605                     cout << y << endl;
606                     cout << "Attempting backward substitution ... " << endl;
607                     Matrix x = U.BackwardSub(y);
608                     cout << "Press return to continue ..." << flush;
609                     system("read");
610                     cout << endl;
611             }
612
613             cout << "Case 3: Testing error message #2:" << endl;
614             cout << endl;
615
616             {
617                     double matrix[9] = {1, 0, 0, 2, 1, 0, 3, 1, 1};
618                     SquareMatrix U = SquareMatrix::Test(matrix, 3);
619                     cout << "The upper triangular matrix, U = " << endl;
620                     cout << U << endl;
621                     double vector[4] = {16, 43, 57, 29};
622                     Matrix y = Matrix::Test(vector, 4, 1);
623                     cout << "The column vector, y  = " << endl;
624                     cout << y << endl;
625                     cout << "Attempting backward substitution ... " << endl;
626                     Matrix x = U.BackwardSub(y);
627                     cout << "Press return to continue ..." << flush;
628                     system("read");
629                     cout << endl;
630             }
631
632             cout << "Case 4: Testing error message #3:" << endl;
633             cout << endl;
634
635             {
636                     double matrix[9] = {0, 0, 0, 2, 1, 0, 3, 1, 1};
637                     SquareMatrix U = SquareMatrix::Test(matrix, 3);
638                     cout << "The upper triangular matrix, U = " << endl;
639                     cout << U << endl;
640                     double vector[3] = {16, 43, 57};
641                     Matrix y = Matrix::Test(vector, 3, 1);
642                     cout << "The column vector, y  = " << endl;
643                     cout << y << endl;
644                     cout << "Attempting backward substitution ... " << endl;
645                     Matrix x = U.BackwardSub(y);
646                     cout << "Press return to continue ..." << flush;
647                     system("read");
648                     cout << endl;
649             }
650
651             return;
652
653     }
654
655     void UniqueSolution() {
656
```

```
657         cout << "Solving a system of linear equations where a unique solution exists:" << endl;
658         cout << endl;
659 
660         cout << "Case 1: Successful solution for a 4 x 4 matrix:" << endl;
661         cout << endl;
662 
663         {
664                 double matrix[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
665                 SquareMatrix A = SquareMatrix::Test(matrix, 4);
666                 cout << "The matrix of coefficients, A = " << endl;
667                 cout << A << endl;
668                 double vector[4] = {1, 0, 0, 0};
669                 Matrix b = Matrix::Test(vector, 4, 1);
670                 cout << "The column vector, b  = " << endl;
671                 cout << b << endl;
672                 Matrix x = A.SolveUnique(b);
673                 cout << "The result of solving for x in (A*x = b), the column vector x  = " <<
    endl;
674                 Matrix::PrintMATLAB(x);
675                 cout << "Checking that (A*x = b): " << endl;
676                 cout << A * x << endl;
677                 cout << "Press return to continue ..." << flush;
678                 system("read");
679                 cout << endl;
680         }
681 
682         cout << "Case 2: Successful solution for a 5 x 5 matrix:" << endl;
683         cout << endl;
684 
685         {
686                 double matrix[25] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8, 4, 6, 8, 2, 0,
    9, 7, 3, 4};
687                 SquareMatrix A = SquareMatrix::Test(matrix, 5);
688                 cout << "The matrix of coefficients, A = " << endl;
689                 cout << A << endl;
690                 double vector[5] = {6, 5, 3, 1, 4};
691                 Matrix b = Matrix::Test(vector, 5, 1);
692                 cout << "The column vector, b  = " << endl;
693                 cout << b << endl;
694                 Matrix x = A.SolveUnique(b);
695                 cout << "The result of solving for x in (A*x = b), the column vector x  = " <<
    endl;
696                 Matrix::PrintMATLAB(x);
697                 cout << "Checking that (A*x = b): " << endl;
698                 cout << A * x << endl;
699                 cout << "Press return to continue ..." << flush;
700                 system("read");
701                 cout << endl;
702         }
703 
704         cout << "Case 3: Error message testing:" << endl;
705         cout << endl;
706 
707         {
708                 double matrix[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
709                 SquareMatrix A = SquareMatrix::Test(matrix, 4);
710                 cout << "The matrix of coefficients, A = " << endl;
711                 cout << A << endl;
712                 double vector[5] = {1, 0, 0, 0, 2};
713                 Matrix b = Matrix::Test(vector, 5, 1);
714                 cout << "The column vector, b  = " << endl;
715                 cout << b << endl;
716                 cout << "Attempting to solve for x in (A*x = b) ... " << endl;
717                 Matrix x = A.SolveUnique(b);
718                 cout << "Press return to continue ..." << flush;
719                 system("read");
720                 cout << endl;
```

```
721         }
722
723         cout << "Case 4: Error message testing:" << endl;
724         cout << endl;
725
726         {
727                 double data[16] = {0, 0, 0, 0, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
728                 SquareMatrix A = SquareMatrix::Test(data, 4);
729                 cout << "When the matrix of coefficients is singular. A = " << endl;
730                 cout << A << endl;
731                 double vector[4] = {1, 0, 0, 0};
732                 Matrix b = Matrix::Test(vector, 4, 1);
733                 cout << "The column vector, b  = " << endl;
734                 cout << b << endl;
735                 cout << "Attempting to solve for x in (A*x = b) ... " << endl;
736                 Matrix x = A.SolveUnique(b);
737                 cout << "Press return to continue ..." << flush;
738                 system("read");
739                 cout << endl;
740         }
741
742         return;
743 }
744
745 void LeastSquareSolution() {
746
747         cout << "Solving an overdetermined system of linear equations for a least squares" << endl;
748         cout << "solution:" << endl;
749         cout << endl;
750
751         cout << "Case 1: Successful solution for a 4 x 2 matrix:" << endl;
752         cout << endl;
753
754         {
755                 double matrix[8] = {4, 3, 2, 1, 3, 4, 3, 2};
756                 Matrix A = Matrix::Test(matrix, 4, 2);
757                 cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
758                 cout << A << endl;
759                 double vector1[4] = {1, 2, 3, 4};
760                 Matrix b = Matrix::Test(vector1, 4, 1);
761                 cout << "The m-by-1 column vector, b  = " << endl;
762                 cout << b << endl;
763                 SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
764                 cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
765                 Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
766                 cout << ATA << endl;
767                 cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
768                 cout << ATb << endl;
769                 Matrix x = ATA.SolveUnique(ATb);
770                 cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x
    = " << endl;
771                 Matrix::PrintMATLAB(x);
772                 cout << "Checking against the solution given in MATLAB for x = A\\b : " << endl;
773                 double vector2[2] = {-1.1724, 1.7241};
774                 Matrix expected = Matrix::Test(vector2, 2, 1);
775                 Matrix::PrintMATLAB(expected);
776                 cout << "Press return to continue ..." << flush;
777                 system("read");
778                 cout << endl;
779         }
780
781         cout << "Case 2: Successful solution for a 5 x 4 matrix:" << endl;
782         cout << endl;
783
784         {
785                 double matrix[20] = {4, 3, 2, 1, 3, 4, 3, 2, 2, 3, 4, 3, 1, 2, 3, 4, 4, 3, 2, 1};
786                 Matrix A = Matrix::Test(matrix, 5, 4);
```

```
787                 cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
788                 cout << A << endl;
789                 double vector1[5] = {1, 2, 3, 4, 5};
790                 Matrix b = Matrix::Test(vector1, 5, 1);
791                 cout << "The m-by-1 column vector, b  = " << endl;
792                 cout << b << endl;
793                 SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
794                 cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
795                 Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
796                 cout << ATA << endl;
797                 cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
798                 cout << ATb << endl;
799                 Matrix x = ATA.SolveUnique(ATb);
800                 cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x
    = " << endl;
801                 Matrix::PrintMATLAB(x);
802                 cout << "Checking against the solution given in MATLAB for x = A\\b : " << endl;
803                 double vector2[4] = {-2.7453, 7.5652, -2.9689, -1.2236};
804                 Matrix expected = Matrix::Test(vector2, 4, 1);
805                 Matrix::PrintMATLAB(expected);
806                 cout << "Press return to continue ..." << flush;
807                 system("read");
808                 cout << endl;
809         }
810
811         cout << "Case 3: Testing warning message:" << endl;
812         cout << endl;
813
814         {
815
816                 double column[4] = {1, 2, 3, 4};
817                 SquareMatrix A = SquareMatrix::Toeplitz(column,4);
818                 cout << "When the matrix of coefficients is square. A = " << endl;
819                 cout << A << endl;
820                 double vector1[4] = {1, 2, 3, 4};
821                 Matrix b = Matrix::Test(vector1, 4, 1);
822                 cout << "The m-by-1 column vector, b  = " << endl;
823                 cout << b << endl;
824                 SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
825                 cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
826                 Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
827                 cout << ATA << endl;
828                 cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
829                 cout << ATb << endl;
830                 Matrix x = ATA.SolveUnique(ATb);
831                 cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x
    = " << endl;
832                 Matrix::PrintMATLAB(x);
833                 cout << "Checking that (A*x = b): " << endl;
834                 cout << A * x << endl;
835                 cout << "Press return to continue ..." << flush;
836                 system("read");
837                 cout << endl;
838         }
839
840         cout << "Case 4: Testing error message #1:" << endl;
841         cout << endl;
842
843         {
844                 double matrix[8] = {1, 1, 1, 1, 1, 2, 3, 4};
845                 Matrix A = Matrix::Test(matrix, 2, 4);
846                 cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
847                 cout << A << endl;
848                 double vector1[2] = {6, 5};
849                 Matrix b = Matrix::Test(vector1, 2, 1);
850                 cout << "The m-by-1 column vector, b  = " << endl;
851                 cout << b << endl;
```

```
852                     cout << "Attempting to initiate a least squares solution ... " << endl;
853                     SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
854                     cout << "Press return to continue ..." << flush;
855                     system("read");
856                     cout << endl;
857             }
858
859             cout << "Case 5:Testing error message #2:" << endl;
860             cout << endl;
861
862             {
863                     double matrix[8] = {1, 1, 1, 1, 1, 2, 3, 4};
864                     Matrix A = Matrix::Test(matrix, 4, 2);
865                     cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
866                     cout << A << endl;
867                     double vector1[5] = {6, 5, 7, 10, 11};
868                     Matrix b = Matrix::Test(vector1, 5, 1);
869                     cout << "The p-by-1 column vector, b  = " << endl;
870                     cout << b << endl;
871                     cout << "Attempting to initiate a least squares solution ... " << endl;
872                     cout << endl;
873                     SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
874                     cout << "Press return to continue ..." << flush;
875                     system("read");
876                     cout << endl;
877             }
878
879
880             return;
881     }
882
883     int main () {
884
885             for (;;) {
886                     cout << endl;
887                     cout << "Choose to test one of the following:" << endl;
888                     cout << endl;
889                     cout << "  Enter \'A\' for the Ones, Eye and other Constructors Testing" << endl;
890                     cout << "  Enter \'B\' for the Toeplitz function Testing" << endl;
891                     cout << "  Enter \'C\' for the Transpose function Testing" << endl;
892                     cout << "  Enter \'D\' for the Triangular Extraction Testing" << endl;
893                     cout << "  Enter \'E\' for the LU Decomposition Testing - Algorithm 2.1" << endl;
894                     cout << "  Enter \'F\' for the LU Decomposition Testing - Algorithm 2.2" << endl;
895                     cout << "  Enter \'G\' for the Forward Substitution Testing - Algorithm 3.1" <<
    endl;
896                     cout << "  Enter \'H\' for the Backward Substitution Testing - Algorithm 3.2" <<
    endl;
897                     cout << "  Enter \'I\' for Solving Systems of Linear Equations in which the number
    of" << endl;
898                     cout << "             equations is equal to the number of unknowns" << endl;
899                     cout << "  Enter \'J\' for Solving Systems of Linear Equations in which the number
    of" << endl;
900                     cout << "             equations is greater than the number of unknowns" << endl;
901                     cout << endl;
902                     cout << ">> ";
903                     char choice;
904                     cin >> choice;
905                     cout << endl;
906                     switch (choice) {
907                             case 'A':
908                             case 'a':       ConstructorsTesting();
909                                             break;
910                             case 'B':
911                             case 'b':       ToeplitzTesting();
912                                             break;
913                             case 'C':
914                             case 'c':       TransposeTesting();
```

```
915                                             break;
916                     case 'D':
917                     case 'd':       TriangularExtractionTesting();
918                                             break;
919                     case 'E':
920                     case 'e':       LUDecompositionTestingOne();
921                                             break;
922                     case 'F':
923                     case 'f':       LUDecompositionTestingTwo();
924                                             break;
925                     case 'G':
926                     case 'g':       ForwardSubstitution();
927                                             break;
928                     case 'H':
929                     case 'h':       BackwardSubstitution();
930                                             break;
931                     case 'I':
932                     case 'i':       UniqueSolution();
933                                             break;
934                     case 'J':
935                     case 'j':       LeastSquareSolution();
936                                             break;
937             }
938             cout << "Enter \'0\' to exit or \'1\' to choose another test" << endl;
939             cout << endl;
940             cout << ">> ";
941             cin >> choice;
942             if (choice == '0') {
943                     cout << endl;
944                     cout << "Goodbye!" << endl;
945                     cout << endl;
946                     return 0;
947             }
948     }
949
950 }
```