1 Objectives

- To experience creating an abstract data type (ADT)
- To implement an ADT in C++, using the operator overloading facility of the C++ language
- To learn how to turn objects of a class into "function objects"
- To learn how to convert objects of a class to objects of unrelated types (e.g., `Foo` to `bool`)

2 Background

A *data type* represents a set of data values sharing common properties. An abstract data type (ADT) specifies a set of operations on a *data type*, independent of how the data values are actually represented or how the operations are implemented.

Classic ADTs representing mathematical entities such as rational number and complex number ADTs support many arithmetic, relational and other operations, making them ideal data types for operator overloading.

Since there is no shortage of code for C++ classes representing rational and complex numbers, assignments designed to provide practice with operator overloading tend to get a bit creative with their choice of *data types*; ideally, a *data type* that is not as ubiquitous as rational and complex number ADTs, but one that lends itself to operator overloading just as good.

3 Mat2x2 ADT

Mat2x2 is an ADT that encapsulates a "2×2 matrix", a simple mathematical entity of the form $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, where the numbers a, b, c, and d represent the "value" of a 2×2 matrix data type.

For example, the numbers 4, 8, 7, and 3 together can represent the value of the 2×2 matrix $\begin{pmatrix} 4 & 8 \\ 7 & 3 \end{pmatrix}$.

3.1 Notation

$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $xI = \begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix} = Ix$, x a real number, a multiplicative scalar

$B = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$, $J = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $xJ = \begin{pmatrix} x & x \\ x & x \end{pmatrix} = Jx$, x a real number, an additive scalar

3.2 Operations

Scalar Multiplication $Ax = xA = xIA = \begin{pmatrix} ax & bx \\ cx & dx \end{pmatrix}$

Scalar Addition
$$x + A = xJ + A = \begin{pmatrix} x+a & x+b \\ x+c & x+d \end{pmatrix}, \quad A + x = x + A$$

Scalar Subtraction
$$x - A = xJ - A = \begin{pmatrix} x-a & x-b \\ x-c & x-d \end{pmatrix}, \quad A - x = -(x - A)$$

addition $A + B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix}$

subtraction $A - B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} - \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a-a' & b-b' \\ c-c' & d-d' \end{pmatrix}$

multiplication $AB = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} aa'+bc' & ab'+bd' \\ a'c+c'd & b'c+dd' \end{pmatrix}$

determinant $\det(A) = \det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$

inverse $A^{-1} = \text{inv}(A) = \text{inv} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \frac{1}{\det(A)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}, \qquad \det(A) \neq 0$

division $A/B = AB^{-1}, \qquad \det B \neq 0$

Norm $\|A\| = \text{norm}(A) = \sqrt{a^2 + b^2 + c^2 + d^2}$

Relational Equality $A = B$ iff $\|A - B\| \leq \epsilon$.

The symbol ϵ denotes a tolerance, a small positive amount the value $\|A - B\|$ can change and still be acceptable that A is equal to B.

Relational Inequality $A < B$ if $\neg(A = B)$ and $\|A\| < \|B\|$

where $\neg$ denotes the negation operator.

Recall that the definitions of the operators $<$ and $=$ on objects X and Y are sufficient for deriving the definitions of the other four relational operators $>$, $\geq$, $\neq$, and $\leq$:

- $X > Y \equiv Y < X$
- $X \geq Y \equiv \neg(X < Y)$
- $X \neq Y \equiv \neg(X = Y)$
- $X \leq Y \equiv X < Y$ or $X = Y$

4 Your Task

Implement the Mat2x2 ADT above using a C++ class, called `Mat2x2`, that uses one of the representations listed in section 5 to store the underlying data as `private` member(s).

The public interface of your `Mat2x2` class must include the following member functions:

1. Constructors:

```
explicit Mat2x2(double = 0, double = 0, double = 0, double = 0);
```

```
Mat2x2(const array<double, 4> &);          // using std::array;
```

```
Mat2x2(const array<array<double, 2>, 2>&); // using std::array;
```

```
Mat2x2(const initializer_list<double>);    // using std::initializer_list;
```

2. Defaulted copy/move constructors, copy/move assignment operators, and destructor.
3. `norm()`, returns the norm of the calling object
4. `inverse()`, returns the inverse of the calling object
5. `det()`, returns the determinant of the calling object

4.1 Member Operator Overload Functions

6. Compound assignment operator overloads.

   `Mat2x2 op Mat2x2` `x += y`, `x -= y`, `x *= y`, `x /= y`

   `Mat2x2 op double` `x += k`, `x -= k`, `x *= k`, `x /= k`

7. Unary operators `++x`, `x++`, `+x`, `--x`, `x--`, `-x`, where `x` is a `Mat2x2` object.
8. An overloaded XOR `operator^` such that `x^k` returns the `Mat2x2` object resulting from raising `x` to the power `k` (an integer). It does not modify `x`.
9. Subscript operators `[ ]`, both `const` and non-`const` overloads. If subscript is invalid, must throw: `invalid_argument("index out of bounds")`

   These operator overloads provide direct access to the underlying data members, effectively eliminating the need for `friend` functions.

10. `operator bool() const` Returns whether or not the invoking object has inverse. For example, if `x` is a `Mat2x2` object, it returns `true` if $|\texttt{x.det()}| > \epsilon$, and returns `false` otherwise.

4.2 Function objects

Overloading the function call operator, these overloads effectively turn `Mat2x2` objects into functions; hence the name "function object."

11. `double operator()()const` Returns the `norm` of the invoking `Mat2x2` object. For example, if `x` is a `Mat2x2` object, then `x()` returns `x.norm()`.

12. `double& operator()(size_t r, size_t c)`

    Returns an lvalue reference to the entry at row `r` and column `c`.

    Since it would be more intuitive for humans to start row and column indexing at 1, this function must ensure that the values of `r` and `c` are 1-based.

    For example, if `x` stores the matrix $\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$, then `x(i, j)` should return a reference to x_{ij}, $i = 1, 2$, $j = 1, 2$.

    If `r` or `c` is invalid, then this function must throw an exception as shown below:

```
if (r < 1 || r > 2) throw invalid_argument("row index out of bounds");
if (c < 1 || c > 2) throw invalid_argument("column index out of bounds");
```

4.3 `static` Members

```
   static double tolerance; // initial value = 1.0E-6
   static void setTolerance(double tol);
   static double getTolerance();
```

See Relational Equality in section 3.2 where "`tolerance`" is defined.

4.4 Non-Member Operator Overload Functions

1. Overloaded extraction operator `>>` for reading `Mat2x2` values
2. Overloaded insertion operator `<<` for writing `Mat2x2` values
3. Basic arithmetic operators. None modifies it operands. Not all can be implemented as members (which ones?). For consistency, all are typically implemented as free functions.

   `Mat2x2 op Mat2x2` `x + y`, `x - y`, `x * y`, `x / y`

   `Mat2x2 op double` `x + k`, `x - k`, `x * k`, `x / k`

   `double op Mat2x2` `k + y`, `k - y`, `k * y`, `k / y`

4. Relational operators. None modifies it operands. For consistency, all are typically implemented as free functions.

   `Mat2x2 op Mat2x2` `x < y`, `x <= y`, `x > y`, `x >= y`, `x == y`, `x != y`

5 Alternative Representations for a 2×2 Matrix

Applying the OOP's principle of information hiding, an implementation of an ADT can optimize both representation of the data type and implementation of the operations without affecting the client code.

However, since an ADT does not depend on the representation of the data type it specifies, the first order of business for any implementation of an ADT is to define concrete representation of the data type so that the operations on the type can be implemented.

The examples below show three common representations of a 2×2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$:

Eaxmple 1: Viewing the 2×2 matrix as four individual elements a, b, c, and d:

```
double a; // top-left
double b; // top-right
double c; // bottom-left
double d; // bottom-right
```

For the benefit of human readers of the code, such representation must explicitly document the correspondence between the matrix elements and the variable `a`, `b`, `c`, and `d`.

Eaxmple 2: Viewing the 2×2 matrix as a sequence (y_0, y_1, y_2, y_3)

For storing and managing any sequence of `const` size, modern C++ provides a smart array class template in the `<array>` header file:

```
std::array <double, 4> y; // an array of 4 doubles
```

std::array provides a rich set of useful methods as well as supporting the familiar array notation `y[0]`, `y[1]`, `y[2]`, and `y[3]`.

This representation too must document the correspondence between the matrix elements and the variable `y[0]`, `y[1]`, `y[2]`, and `y[3]`.

Eaxmple 3: Viewing the 2×2 matrix as $\left(\begin{pmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{pmatrix}\right)$, or as $\left(\begin{pmatrix} x_{00} & x_{01} \end{pmatrix} \begin{pmatrix} x_{10} & x_{11} \end{pmatrix}\right)$, as an array of two rows, each in turn an array of two elements.

```
std::array< std::array <double, 2>, 2> x;
// x is an array of size 2;
// each element of x is an array of 2 doubles;
```

Note that `x` is a `std::array` of `std::array`s, representing a two-dimensional table or matrix of `double`s.

Overloading the subscript operator `[]`, `std::array` provides the familiar array notation. Specifically, the two elements (the rows) `x[0]` and `x[1]` of the 2D-array `x` are each of type `std::array<double, 2>`, and as such, `x[0]`'s two elements on the first row are `x[0][0]`, and `x[0][1]`, and `x[1]`'s two elements on the second row are `x[1][0]`, and `x[1][1]`.

5.1 What About Multi-Dimensional Raw Arrays?

C++ does not provide a special multi-dimensional *raw array type*. Instead, C++ provides only one-dimensional array type, but it allows the array elements themselves to be "one-dimensional arrays".

For example, denoting a two-dimensional 5×3 array of 5 rows and 3 columns, the declaration `double A[5][3]` means that `A` is a one-dimensional array of 5 elements, each of which, in turn, is an array of 3 `double`s. Specifically,

`A[0]` points to an array of 3 elements `A[0][0]`, `A[0][1]`, `A[0][2]` on row 1,
`A[1]` points to an array of 3 elements `A[1][0]`, `A[1][1]`, `A[1][2]` on row 2, and in general,
`A[r]` points to an array of 3 elements `A[r][0]`, `A[r][1]`, `A[r][2]` on row `r+1`.

Similarly, the declaration `double B[3][4][5]` declares `B` as an array of size 3 whose elements are each arrays of size 4 whose elements are each arrays of 5 `double`s.

Question 1

Write a declaration for a function named `process2D` that is to receive and process a raw array of 7 raw arrays, each of 5 `doubles`.

Question 2

Write a declaration for a function named `process3D` that is to receive and process a three-dimensional $5 \times 3 \times 7$ array of integers.

5.2 Prefer `std::array`s to Raw Arrays

Given `std::array<T,n>` container, a smart class template modeling an array of fixed size `n` and elements of type `T`, there is really no good reason to use raw arrays in C++.

The justification for this is that a `std::array<T,n>` object

- stores and can tell the size of the array,
- provides bounds checking facilities,
- can be passed as an argument to a function without decaying to a pointer,
- can be used by any algorithm designed to work with C++ container classes,
- provides the familiar raw array notation (by overloading the `[]` operator),
- and more.

6 Deliverables

1. Header files: `Mat2x2.h`
2. Implementation files: `Mat2x2.cpp`, `Mat2x2_test_driver.cpp`
3. A `README.txt` text file (as described in the course outline).

7 Grading scheme

Functionality	• Correctness of execution of your program, • Proper implementation of all specified requirements, • Efficiency	60%
OOP style	• Encapsulating only the necessary data inside your objects, • Information hiding, • Proper use of C++ constructs and facilities. • No global variables • No use of the operator delete. • No C-style memory functions such as malloc, alloc, realloc, free, etc.	20%
Documentation	• Description of purpose of program, • Javadoc comment style for all methods and fields, • Comments for non-trivial code segments	10%
Presentation	• Format, clarity, completeness of output, • User friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing	5%

8 Sample Test Driver

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cassert>
#include "Mat2x2.h"
using std::cout;
using std::cin;
using std::endl;

/*
Tests class Mat2x2 . Specifically, tests constructors, compound assignment
operator overloads, basic arithmetic operator overloads, unary +, unary -,
pre/post-increment/decrement, subscripts, function objects,
input/output operators, and relational operators.
@return 0 to indicate success.
*/

int main()
{
   const Mat2x2  ZERO;
   // must not compile, because zero is const
   //ZERO[1] = 0;
   //ZERO[2] = 0;
   //ZERO[3] = 0;
   //ZERO[4] = 0;
   const Mat2x2  IDENTITY(1, 0, 0, 1);

   // ctor that takes an std::initializer_list<double>
   Mat2x2 a = { 11, 22, 33, 44, 55, 66.5 }; // notice intentional too many initializers
   cout << "a = " << a << endl;
   assert(a == Mat2x2(11, 22, 33, 44));

   Mat2x2 b{ 111,222.7,333 };
   cout << "b = " << b << endl;
   assert(b == Mat2x2(111, 222.7, 333, 0));

   Mat2x2 c{ 1234 };
   cout << "c = " << c << endl;
   assert(c == Mat2x2(1234, 0, 0, 0));

   // a conversion constructor
   Mat2x2 d(1234);       // int -> Mat2x2 // [1234, 0, 0, 0]
   cout << "d = " << d << endl;
   assert(d == Mat2x2(1234, 0, 0, 0));
```

```
45
46      Mat2x2  e;                              // default ctor
47      cout << "e = " << e << endl;            // cout << Mat2x2
48      assert(e == ZERO);                      // Mat2x2  == Mat2x2
49
50      Mat2x2  f(2);                           // normal ctor with 1 arg
51      cout << "f = " << f << endl;
52      assert(f == Mat2x2(2, 0, 0, 0));
53
54      Mat2x2  g(2, 3);                        // normal ctor with 2 args
55      cout << "g = " << g << endl;
56      assert(g == Mat2x2(2, 3, 0, 0));
57
58      Mat2x2  h(2, 3, 8);                     // normal ctor with 3 args
59      cout << "h = " << h << endl;
60      assert(h == Mat2x2(2, 3, 8, 0));
61
62      Mat2x2  m1(2.5, 3.6, 8.7, 5.8);                 // normal ctor with 4 args
63      Mat2x2   m1_inverse = m1.inverse();             // inverse, copy ctor
64
65      Mat2x2  m1_inverse_times_m1 = m1_inverse * m1; // Mat2x2  * Mat2x2
66      assert(m1_inverse_times_m1 == IDENTITY);        // invariant, must hold
67
68      Mat2x2  m1_times_m1_inverse = m1 * m1_inverse;
69      assert(m1_times_m1_inverse == IDENTITY);        // invariant, must hold
70
71      assert(+m1 == -(-m1));                          // +Mat2x2 , -Mat2x2
72      Mat2x2  t1 = m1;
73      ++m1;                                           // ++Mat2x2
74      assert(m1 == t1 + 1);
75      --m1;                                           // --Mat2x2
76      assert(m1 == t1);
77
78      Mat2x2  m1_post_inc = m1++;                     // Mat2x2 ++
79      assert(m1_post_inc == t1);
80      assert(m1 == t1 + 1);
81
82      Mat2x2  m1_post_dec = m1--;                     // Mat2x2 --
83      assert(m1_post_dec == t1 + 1);
84      assert(m1 == t1);
85
86      cout << "\n";
87      h += Mat2x2(0, 0, 0, 5);                  // Mat2x2  += Mat2x2
88      Mat2x2  m2 = h + 1.0;                     // Mat2x2  = Mat2x2 + int
89      assert(m2 == Mat2x2(3, 4, 9, 6));
90      cout << "m2 = " << m2 << endl;
```

```
91
92      m2 = 1 + h;                             // Mat2x2  = double + Mat2x2;
93      assert(m2 == Mat2x2(3, 4, 9, 6));
94
95      Mat2x2  m3 = m2 - 1.0;                  // Mat2x2  = Mat2x2 - double
96      assert(m3 == h);
97      cout << "m3 = " << m3 << endl;
98
99      Mat2x2  m4 = 1.0 - m3;                  // Mat2x2  = double - Mat2x2
100     cout << "m4 = " << m4 << endl;
101     assert(m4 == Mat2x2(-1, -2, -7, -4));
102
103
104     Mat2x2  m5 = m4 * 2.0;                  // Mat2x2  = Mat2x2 * double
105     cout << "m5 = " << m5 << endl;
106     assert(m5 == Mat2x2(-2, -4, -14, -8));
107
108     Mat2x2  m6 = -1 * m5;                   // Mat2x2  = double * Mat2x2
109     cout << "m6 = " << m6 << endl;
110     assert(m6 == Mat2x2(2, 4, 14, 8));
111     assert(m6 / -1.0 == m5);                // Mat2x2  = Mat2x2 / double
112     assert(1 / m6 == 1 * m6.inverse());     // double / Mat2x2, inverse
113     assert(-1.0 * m4 * 2.0 == m6);          // double * Mat2x2 * double
114
115     Mat2x2  m7 = m1++;                      //Mat2x2 ++
116     cout << "m1 = " << m1 << endl;
117     cout << "m7 = " << m7 << endl;
118     assert(m7 == m1 - Mat2x2(1, 1, 1, 1)); // Mat2x2  - Mat2x2
119
120     Mat2x2  m8 = --m1;                      // --Mat2x2
121     cout << "m1 = " << m1 << endl;
122     cout << "m8 = " << m8 << endl;
123     assert(m8 == m1);
124
125     m8--;                                   // Mat2x2--
126     cout << "m8 = " << m8 << endl;
127     assert(m1 == 1 + m8);                   // double + Mat2x2
128     assert(m1 - 1 == m8);
129     assert(-m1 + 1 == -m8);
130     assert(2 * m1 == m8 + m1 + 1);
131     assert(m1 * m1 == m1 * (1 + m8));
132
133     Mat2x2  m9(123, 6, 6, 4567.89);
134     cout << "m9 = " << m9 << endl;
```

```
135
136     // subscripts (non-const)
137     m9[0] = 3;
138     m9[1] = 1;
139     m9[2] = 7;
140     m9[3] = 4;
141     cout << "m9 = " << m9 << endl;
142     assert(m9 == Mat2x2(3, 1, 7, 4));
143
144     // relational operators
145     double smallTol = Mat2x2::getTolerance() / 10.0;
146     Mat2x2  m9Neighbor(3 - smallTol, 1 + smallTol, 7 - smallTol, 4 + smallTol);
147     assert(m9 == m9Neighbor);
148
149     double tol = Mat2x2::getTolerance();
150     assert(m9 == m9);
151     assert(m9 == (m9 + 0.1 * tol));
152     assert(m9 == (m9 + 0.2 * tol));
153     assert(m9 == (m9 + 0.3 * tol));
154     assert(m9 == (m9 + 0.4 * tol));
155     assert(m9 == (m9 + 0.5 * tol));
156     assert(m9 != (m9 + 0.6 * tol));
157     assert(m9 != (m9 + tol));
158
159     assert(m9 < (m9 + 0.001));
160     assert(m9 <= (m9 + 0.001));
161     assert((m9 + 0.001) <= (m9 + 0.001));
162
163     assert((m9 + 0.001) > m9);
164     assert((m9 + 0.001) >= m9);
165     assert((m9 + 0.001) >= (m9 + 0.001));
166
167     // compound operators
168
169     m9 += m9;
170     cout << "m9 = " << m9 << endl;
171     assert(m9 == 2 * Mat2x2(3, 1, 7, 4));
172
173     Mat2x2  m10;
174     m10 += (m9 / 2);
175     cout << "m10 = " << m10 << endl;
176     assert(m10 == Mat2x2(3, 1, 7, 4));
177
178     m10 *= 2;
179     cout << "m10 = " << m10 << endl;
180     assert(m10 == m9);
```

```
181
182     m10 /= 2;
183     cout << "m10 = " << m10 << endl;
184     assert(m10 == m9 / 2);
185
186     m10 += 10;
187     cout << "m10 = " << m10 << endl;
188     assert(m10 == (m9 + 20) / 2);
189
190     m10 -= 10;
191     cout << "m10 = " << m10 << endl;
192     assert(m10 == 0.5 * m9);
193
194     // ctor that takes a std::array<double, 4>
195     std::array<double, 4> arr = { 2, 0, 0, 2 };
196     Mat2x2 m11{ arr };
197     cout << "m11 = " << m11 << endl;
198
199     // ctor that takes a std::array< std::array <double, 2>, 2>
200     std::array <double, 2> row1{ 2, 0 };
201     std::array <double, 2> row2{ 0, 2 };
202     std::array< std::array <double, 2>, 2> mat{ row1, row2 };
203     Mat2x2 m12{ mat };
204     cout << "m12 = " << m12 << endl;
205     assert(m12 == arr);
206
207     // multiplications
208     Mat2x2 i{ 1,2,3,4 };
209     Mat2x2 j{ 2,0,1,2 };
210     assert((i * j) == Mat2x2(4, 4, 10, 8));
211     assert((j * i) == Mat2x2(2, 4, 7, 10));
212
213     // inverse operation
214     Mat2x2 k{ 4,7,2,6 };
215     if (k) // this is how if(cin) works!
216     {
217        cout << "k is invertible\n";
218        Mat2x2 m4aInv1 = k.inverse();
219        Mat2x2 m4aInv2 = k ^ (-1); // operator ^ overload
220        assert(m4aInv1 == m4aInv2);
221     }
222     else
223     {
224        cout << "k is NOT invertible\n";
225     }
```

```
226
227     Mat2x2 p{ k * k * k * k * k };
228     Mat2x2 q{ k ^ (5) };
229     assert(p == q);
230
231     Mat2x2 x = Mat2x2{ 4,7,2,6 }.inverse();
232     Mat2x2 y{ x * x * x * x * x };
233     Mat2x2 z{ q ^ (-1) };
234     assert(y == z);
235
236     // test function objects (that is, function call operators)
237     assert(k() == k.norm());
238
239     cout << "k = " << k << endl;
240     k(1, 1) = 10;
241     k(1, 2) = 20;
242     k(2, 1) = 30;
243     k(2, 2) = 40;
244     cout << "k = " << k << endl;
245
246     try
247     {
248        k(3, 1) = 40;
249     }
250     catch (std::invalid_argument& ia)
251     {
252        cout << "Problem:\n" << ia.what() << endl;
253     }
254
255     try
256     {
257        k(1, 3) = 40;
258     }
259     catch (std::invalid_argument& ia)
260     {
261        cout << "Problem: " << ia.what() << endl;
262     }
263
264     //testing operator>>
265     cout << "Please enter the numbers 1, 2, 3, 4.5, in that order\n\n";
266     Mat2x2  input;
267     cin >> input;
268     cout << "input = " << input << endl;
```

```
269
270     Mat2x2  diff = input - Mat2x2(1, 2, 3, 4.5);
271     assert(diff.norm() <= tol);    // absolute value
272     assert(diff() <= tol);             // function object
273
274     cout << "Test completed successfully!" << endl;
275
276     return 0;
277 }
```

8.1 Output

```
1  a = [11.00, 22.00, 33.00, 44.00]
2  b = [111.00, 222.70, 333.00, 0.00]
3  c = [1234.00, 0.00, 0.00, 0.00]
4  d = [1234.00, 0.00, 0.00, 0.00]
5  e = [0.00, 0.00, 0.00, 0.00]
6  f = [2.00, 0.00, 0.00, 0.00]
7  g = [2.00, 3.00, 0.00, 0.00]
8  h = [2.00, 3.00, 8.00, 0.00]
9
10 m2 = [3.00, 4.00, 9.00, 6.00]
11 m3 = [2.00, 3.00, 8.00, 5.00]
12 m4 = [-1.00, -2.00, -7.00, -4.00]
13 m5 = [-2.00, -4.00, -14.00, -8.00]
14 m6 = [2.00, 4.00, 14.00, 8.00]
15 m1 = [3.50, 4.60, 9.70, 6.80]
16 m7 = [2.50, 3.60, 8.70, 5.80]
17 m1 = [2.50, 3.60, 8.70, 5.80]
18 m8 = [2.50, 3.60, 8.70, 5.80]
19 m8 = [1.50, 2.60, 7.70, 4.80]
20 m9 = [123.00, 6.00, 6.00, 4567.89]
21 m9 = [3.00, 1.00, 7.00, 4.00]
22 m9 = [6.00, 2.00, 14.00, 8.00]
23 m10 = [3.00, 1.00, 7.00, 4.00]
24 m10 = [6.00, 2.00, 14.00, 8.00]
25 m10 = [3.00, 1.00, 7.00, 4.00]
26 m10 = [13.00, 11.00, 17.00, 14.00]
27 m10 = [3.00, 1.00, 7.00, 4.00]
28 m11 = [2.00, 0.00, 0.00, 2.00]
29
30 m12 = [2.00, 0.00, 0.00, 2.00]
31 k is NOT invertible
32 k = [4.00, 7.00, 2.00, 6.00]
33 k = [10.00, 20.00, 30.00, 40.00]
```

```
Problem: row index out of bounds
Problem: column index out of bounds
Please enter the numbers 1, 2, 3, 4.5, in that order

1 2 3 4.5
input = [1.00, 2.00, 3.00, 4.50]
Test completed successfully!
```