Engr 101 Project 1: Analyzing a Structure

Project Due: Thursday Jan 30, 2020 at 11:59pm

Engineering intersections: Civil Engineering/Mechanical Engineering/Computer Science

Implementing this project provides an opportunity to work with variables, matrices, indexing, functions, and unit testing in MATLAB.

The autograded portion of the final submission is worth 100 points, and the style and comments grade is worth 10 points.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Project Roadmap

This is a big picture view of how to complete this project. It's like a table of contents. You can find many more details throughout the rest of the spec.

- **Read the Background, Engineering Concepts, and Computing Concepts Sections**
  Many of the other sections refer back to these.

- **Download all the starter files from the Google Drive**

- **Task 1: Implement the `criticalLoad` function**
  We recommend you start with this function first. Once you've finished, go ahead and submit to the autograder - it will give you feedback on this task, even if you haven't started the others.
  Submit: `criticalLoad.m`

- **Task 2: Implement the `actualLoad` function and write unit tests**
  This function deals with selecting submatrices using indexing expressions. For this task, you are also required to write and submit test cases to the autograder!
  Submit: `actualLoad.m`
  Submit: `TestActualLoad.m`

- **Task 3: Implement the `additionalPallets` function**
  This function involves vectorized computations and functions.
  Submit: `additionalPallets.m`

- **Task 4: Implement the `parkingRevenue` function**
  Use indexing expressions to select elements on the edge of a matrix.
  Submit: `parkingRevenue.m`

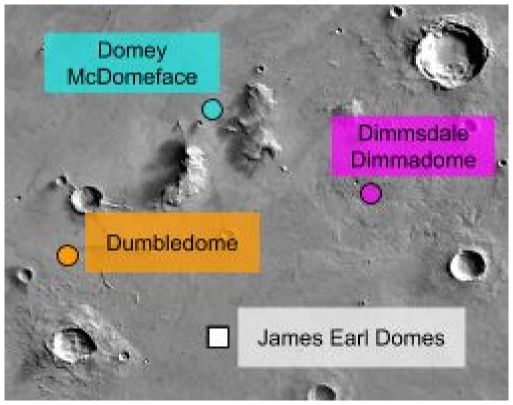
- **Check that your functions are sufficiently and correctly commented**

Purpose

The primary purpose of this project is to get hands-on experience working with matrices in MATLAB and manipulating data to do some kind of analysis. The secondary purpose is to reinforce why we write our own functions: because the function needs to be evaluated many times for different input values and because it can be used just like a built-in function.

Introduction

It's been two years since we established our settlement on Proxima b. We have built several domed structures to protect citizens from the harsh atmosphere, but space within the domes is extremely limited. One proposal for more efficiently using this space is to build underground structures below existing buildings to store materials and park our fleet of autonomous rovers.



Several domed settlements already exist. Underground structures are an efficient way to use the limited space available within the domes.

A crucial engineering challenge for underground structures is ensuring they are able to support the weight of the buildings above them. The settlers are able to manufacture metal support poles, which can be used to help hold everything up. For this project, we make a simplifying assumption that the load on the structure will be carried by four support poles placed in each quadrant of a rectangular structure (more details below).

Your Job

Your team is working to analyze the stability of an underground structure based on possible values of several design parameters (e.g. the nature of the support poles, the distribution of weight above the structure, etc.). You will implement the calculations involved in these analyses as a library of MATLAB functions. You will also use unit tests to ensure functions are working correctly.

Engineering Concepts

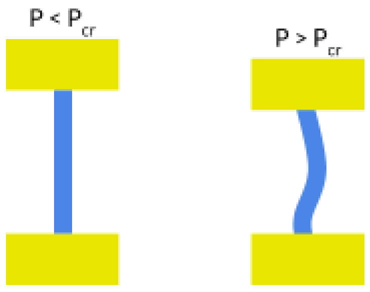
Critical Load

Euler’s critical load is the maximum load a structural column can bear while staying straight. It is particularly relevant for narrow columns, such as the support poles we plan to use on Proxima b. If the load on a column exceeds the critical load, the column is at risk of buckling (i.e. bending under pressure) which can lead to all sorts of structural problems. The critical load depends on four properties of the column. These properties are:

- **E** - Modulus of elasticity of the column material
- **I** - Minimum area moment of inertia across the cross section of the column
- **K** - Column effective length factor
- **L** - Length of the column

With these four properties, the critical load is computed as follows:

$$P_{cr} = \frac{\pi^2 EI}{(KL)^2}$$



Computing Concepts

Unit Testing

Unit tests are used to check whether a specific part of a program behaves correctly in a particular case. A common pattern is to verify that an **individual function** produces the **correct output for a given input**.

While it is possible to unit test functions by hand (e.g. from the command window), it is best to write a script containing a set of automated unit tests, which can be run all at once to test several cases quickly (and potentially re-run when the code is changed/updated).

A useful tool for automated unit testing is `assert`, which can be used to verify that a function's output matches what is expected. The `assert` function takes an expected condition as an input, and then it checks to see if this condition is met. If the condition is met, the program continues, but if the condition is not met, the assert function fails and aborts the program (i.e. the test script). This is a good thing! It's how we know something went wrong with the test and we need to debug our function implementation.

Here's an example unit test we might write for the built-in `max` function.

```
input = [1, 6, 2, 3, 5];

result = max(input);
expectedResult = 6;

assert(almostEqual(result, expectedResult)); % This fails if result doesn't match!

% We only get to this point if the assert didn't fail and crash the program
disp('max: TEST PASSED');
```

If the actual output from `max` doesn't match the expected result of 6, the test will end with an assertion failure.

The `almostEqual` function is not a part of MATLAB, but is provided to you with the starter code for this project. Make sure you have it in your current folder alongside the rest of your code. This function returns a true value if the inputs are equal within a tolerance of 0.01. This prevents our tests from failing if issues with roundoff error make a result not *exactly* match the expected value. (Recall that computers have limited numeric precision.)

How we grade your tests

Let's take Task 2 as an example. (You might want to read that task first.) In that task, you will write a function called `actualLoad`. You must also write unit tests for `actualLoad`, which use `assert` to check that the function produces the right output for each input tested. These unit tests will go in the `TestActualLoad.m` script, which you will submit to the autograder for grading.

Before running your test script, we perform a validity check on your tests. We make sure your test script doesn't report a bug when paired with a correct implementation of `actualLoad` (i.e. a false positive bug detection). Rather, when run on a correct implementation, all your `asserts` must pass. Your tests must be valid in order to earn points for catching bugs (see below).

On the autograder, we have a collection of buggy implementations of `actualLoad`. We run your `TestActualLoad.m` script against each of these and see if it can detect that they are incorrect.

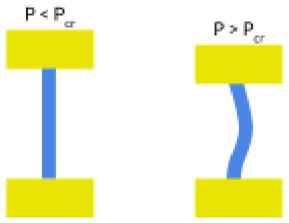
Your test suite is considered to "catch" the bug if any of your individual tests fail for a buggy implementation. (Remember, failure is good because it means an `assert` in the script detected an output mismatch for one of the test cases.)

For example, we might have a buggy version of the `actualLoad` function that works correctly for most matrices, but gives the wrong answer if the maximum load is in quadrant 1. If your `TestActualLoad.m` script contains a unit test that has the maximum load in quadrant 1, then the output of the buggy function will not match the expected output. `assert` will fail and your test catches the bug! Nice!

NOTE: For task 2, you are required to write/submit unit tests for the `actualLoad` function.

Task 1: Critical Load of Support Poles

As described in the Engineering Concepts section, the critical load is the maximum load which a column (in our case a metal support pole) can bear without buckling. You need to write a function to compute the critical load for a support pole based on several parameters describing the pole's size and the materials used to construct it.



Remember that the equation for critical load is:

$$P_{cr} = \frac{\pi^2 EI}{(KL)^2}$$

Function definition and description

The function header and comments are provided for you in a file called `criticalLoad.m`.

```
function [P_cr] = criticalLoad(E, I, K, L)
%criticalLoad computes Euler’s critical load for a column with
%  structural parameters: E, I, K, and L.
%     E: Modulus of elasticity of the column material
%     I: Minimum moment of inertia across cross section of the column
%     K: Column effective length factor
%     L: Length of column
%
%     P_cr: Euler’s critical load of column
```

Write your implementation of the function in this file.

Your function should be vectorized, so that the function works for vector parameters as well as it would for scalars. Used this way, the function could calculate a vector of critical load parameters for a vector of `E` values, a vector of `I` values, etc. In your implementation, this effectively just requires that you remember to use the dot versions of the arithmetic operators (e.g. `.*`).

In MATLAB, you can use the built-in variable `pi` to get the value of π.

Testing your `criticalLoad` function

To test your `criticalLoad` function, use the testing script provided with the starter files.

`TestCriticalLoad.m`:

```
E = 3.8;
I = 2.1;
K = 1.3;
L = 2.4;

P_cr = criticalLoad(E,I,K,L);
display(P_cr);
```

If your `criticalLoad` function is working correctly, you should get the following output when running the test script:

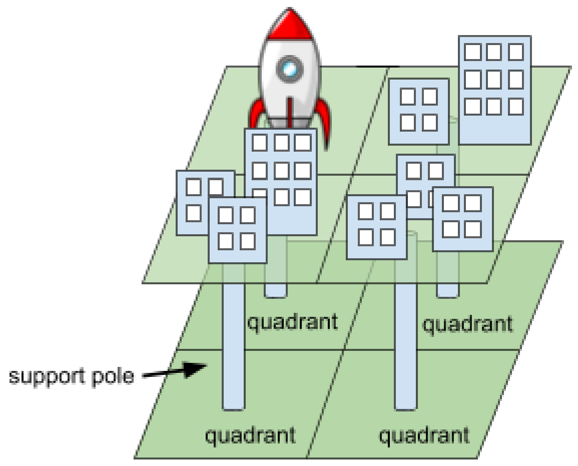
Command Window:

```
>> TestCriticalLoad

P_cr =

    8.0908
```

We recommend you test this function, but you do not have to turn in any of your tests for this task.

Task 2: Computing the Actual Load

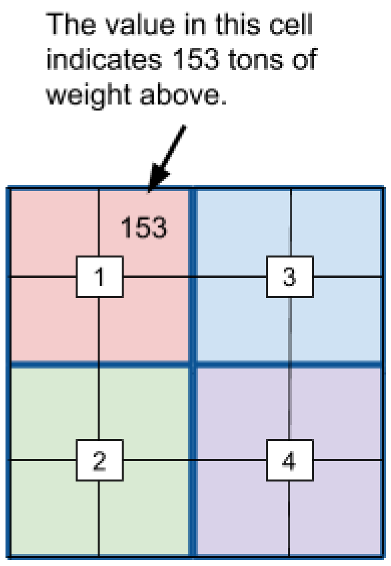
The next task is to compute the actual load the support poles will need to bear. This is based on the weight of buildings above the structure, and we are primarily concerned with finding the single pole that bears the most weight (it would be the first to buckle).



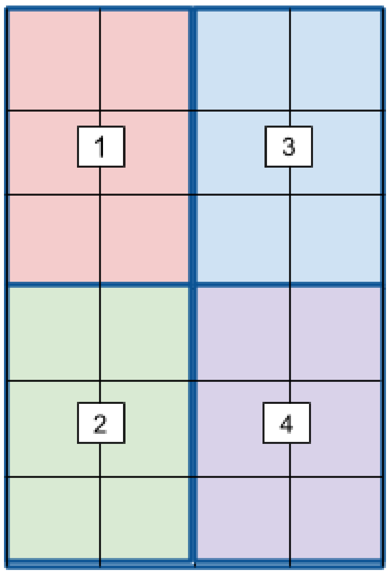
For this task, we assume the structure is rectangular in shape and that four support poles are used to hold up the "ceiling" (and the buildings on top of it). The structure is divided into four quadrants, with a support pole placed in the center of each. We assume each pole carries all of the load in its quadrant.

We will model the layout of the structure with a 2D matrix, where each element in the matrix represents a 10x10ft area on top of the structure. The value of each element in the matrix is the load applied on that portion of the underground structure, due to the weight of the buildings and/or other objects on top of the structure in that 10x10ft area.

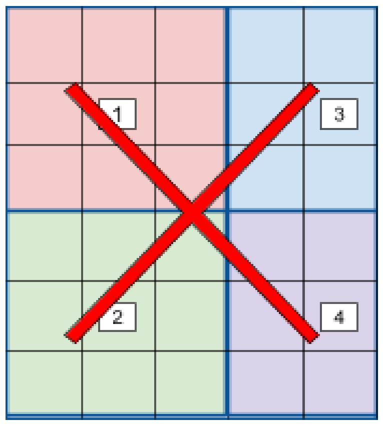
You may assume that the matrix representing the loads on the structure can be exactly divided into four equally sized submatrices representing the four quadrants without any rows or columns overlapping. The figure below illustrates the quadrants and locations of the support poles.



A 40x40ft structure
4x4 matrix
OK



A 60x40ft structure
6x4 matrix
OK

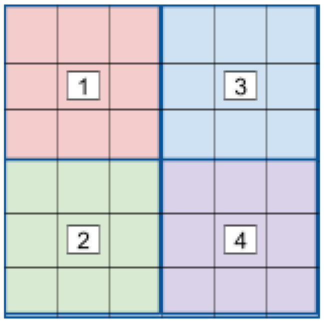


A 60x50ft structure
6x5 matrix
NOT ALLOWED

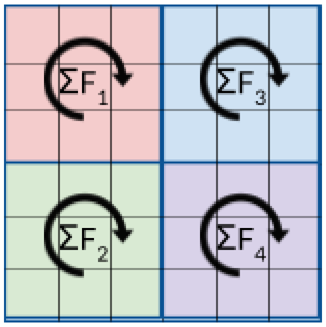
For this task, write the `actualLoad` function, which computes the largest load placed on any individual support pole.

Use the following procedure to compute the function result:

Step 1:
Use indexing to select four equally sized submatrices for each quadrant.



Step 2:
Compute the sum of the loads in each quadrant (i.e. the sum of elements in each submatrix).



Step 3:
Determine the **maximum** load among the four quadrants and return that value.

Function definition and description

The function header and comments are provided for you in a file called `actualLoad.m`.

```
function [maxLoad] = actualLoad(W)
%actualLoad computes the largest load any of the support poles would
%  need to bear based on the distribution of forces in the W matrix. We
%  assume each pole is solely responsible for the forces in its
%  quadrant.
%    W: Weights matrix (Distribution of weight on the roof)
%
%    maxLoad: the largest load among any of the four support poles
```

Write your implementation of the function in this file.

Testing your `actualLoad` function (REQUIRED)

You will write a script containing several unit tests for the `actualLoad` function. Each unit test provides a different set of inputs to the function and uses `assert` and `almostEqual` to verify the output of the function matches the expected result. (See the *Unit Testing* section above.)

We have provided starter code in the `TestActualLoad.m` file with a sample unit test.

`TestActualLoad.m`:

```
%% Unit test for a case with the largest load in quadrant 1
W = [5, 4, 4, 3 ;
     3, 9, 8, 3 ;
     4, 2, 1, 8 ;
     3, 4, 1, 2 ];

result = actualLoad(W);
expectedResult = 21;
assert(almostEqual(result, expectedResult));

% REQUIRED - Add more tests here

%% We only get to this point if all asserts passed successfully
disp('actualLoad: ALL TESTS PASS');
```

If your `actualLoad` function passes the tests, you should see the "ALL TESTS PASS" message.

Command Window

```
>> TestActualLoad
actualLoad: ALL TESTS PASS
```

If it produces the wrong output for a test, you will see an assertion failure error and line number.

Command Window

```
>> TestActualLoad
Error using TestActualLoad (line 9)
Assertion failed.
```

However, this single test doesn't check for correct behavior in all cases! **Write additional unit tests in the `TestActualLoad.m` and submit to the autograder.** In order to receive full credit, your unit tests must catch all 5 of the following 5 potential bugs:

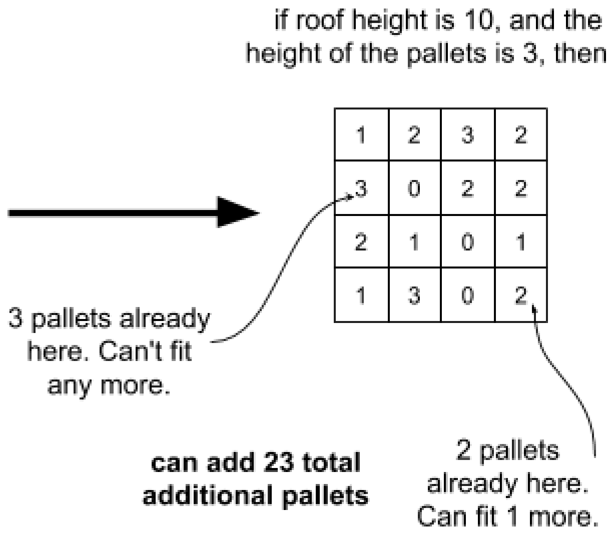
- `actualLoad` fails when the maximum load is in the 1st quadrant.
- `actualLoad` fails when the maximum load is in the 2nd quadrant.
- `actualLoad` fails when the maximum load is in the 3rd quadrant.
- `actualLoad` fails when the maximum load is in the 4th quadrant.
- `actualLoad` fails for non-square matrices (e.g. a 2x6 matrix).

See "How we grade your tests" in the Unit Testing section for more information about grading.

Task 3: Determining Storage Capacity

A portion of the structure will be used for long-term storage of supplies (e.g. non-perishable food, medicine, old ENGR 101 exams, etc.). The supplies are packed into pallets, which are arranged into a grid; each grid space is the length and width of one pallet. The pallets can be stacked on top of each other, assuming their height does not exceed the height of the ceiling.

Assume that all pallets are exactly the same size. They have square bases and a fixed height. Pallets are stored in a rectangular grid, which we will represent with a matrix. Each cell in the matrix contains a number to indicate how many pallets are stacked there. Your task is to write a function that computes the *additional* number of pallets that can be stored, based on the existing pallets, the height of the storage area, and the height of each pallet.


if roof height is 10, and the height of the pallets is 3, then...
1 2 3 2
3 0 2 2
2 1 0 1
1 3 0 2
3 pallets already here. Can't fit any more.
2 pallets already here. Can fit 1 more.
can add 23 total additional pallets

(Left) Supplies on pallets; pallets stacked in designated spaces in the storage facility.
(Right) A matrix showing how many pallets are currently stored in each of the spaces.

Function definition and description

The function header and comments are provided for you in a file called `additionalStorage.m`.

```
function [numPallets] = additionalPallets(roofHeight, pallets,
                                          palletHeight)
%additionalPallets computes the number of additional storage pallets
%     that can fit in the storage area of the structure
%     roofHeight: scalar representing the height of the roof
%     pallets: a matrix representing the number of pallets in
%              each storage cell of the storage area
%     palletHeight: scalar representing the height of a single pallet
%
%     numPallets: number of additional pallets that can fit in
%                 the storage area
```

Write your implementation of the function in this file.

Hint: You may find the `floor()` function helpful.

Note: You can assume that the values in the pallets matrix will be less than or equal to the maximum number that could fit given the the roof height. In other words, no storage cell will already be exceeding the capacity.

Testing your `additionalPallets` function

To test your `additionalPallets` function, use the testing script provided with the starter files.

`TestAdditionalPallets.m`:

```
roofHeight = 10;
pallets = [3, 2, 4;
           4, 1, 3;
           1, 4, 5];
palletHeight = 2;

ap = additionalPallets(roofHeight, pallets, palletHeight);
display(ap);
```

If your `additionalPallets` function is working correctly, you should get the following output when running the test script:

Command Window:

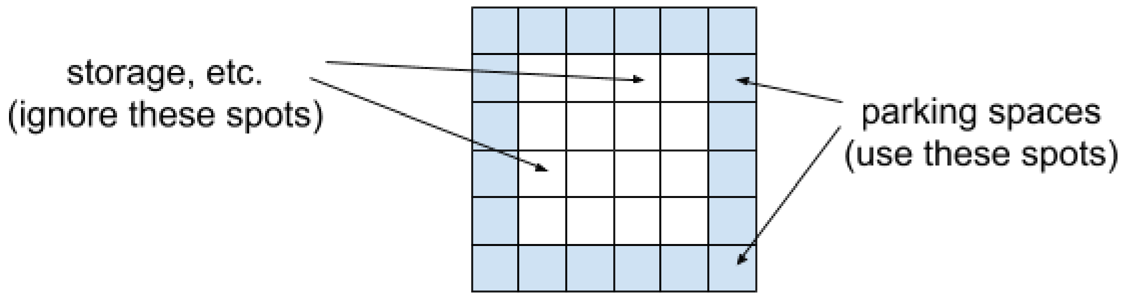
```
>> TestAdditionalPallets

ap =

    18
```

We recommend you test this function, especially with a variety of initial configurations and pallet heights, but you do not have to turn in any of your tests for this task.

Task 4: Parking

The underground structure will also provide parking for our fleet of rovers. To help fund ongoing maintenance of the structure, we plan to charge parking fees for each space. Your task is to compute the total amount earned in a day based on usage for each space and the fee charged (some spaces charge higher rates per hour).

The underground structure is divided into equal sized spaces, each of which has different designations: parking and storage. Parking spaces are located around the edge of the structure. We will use matrices to represent the data associated with each space (because we also rent out the storage spaces), but only the values on the edges of the matrix are relevant for the parking revenue calculation. Your code should <u>not</u> use values from the middle of the matrices.



A matrix representing all of the spaces that we will rent out to settlers.
The storage spaces are in the middle and the parking spaces are along the edges.

You will write a function called `parkingRevenue` that computes the total amount earned for a particular day. Its inputs are two matrices, one that represents the rental price per hour for each space, and another that represents the number of hours that space was in use for the day. You may assume these matrices are the same size. Again, only the edge values in the matrix should be considered, because that is where the parking spots are located.

0.7	4.2	8.0	6.5	2.5	1.8
0.9	?	?	?	?	5.5
3.3	?	?	?	?	1.1
4.9	?	?	?	?	2.3
6.5	?	?	?	?	7.8
2.2	2.2	3.5	5.2	2.4	0.0

matrix representing the number of hours each space was rented

1.2	1.2	1.2	1.5	1.5	1.5
1.2	?	?	?	?	1.5
0.7	?	?	?	?	0.8
0.7	?	?	?	?	0.8
1.8	?	?	?	?	1.9
1.8	1.8	1.7	1.6	1.9	1.9

matrix representing the price per hour of each space

Function definition and description

The function header and comments are provided for you in a file called `parkingRevenue.m`.

```
function [revenue] = parkingRevenue(timeUsed, price)
%parkingRevenue computes the revenue from parking fees
%     timeUsed: a matrix with the number of hours each spot was used
%     price: a matrix with the price per hour for each spot
%     ONLY THE EDGE VALUES FROM EACH MATRIX SHOULD BE USED
%
%     revenue: the total amount earned from all parking spots
```

Write your implementation of the function in this file.

Testing your `parkingRevenue` function

To test your `parkingRevenue` function, use the testing script provided with the starter files.

`TestParkingRevenue.m`:

```
timeUsed = [0.5 4.0 8.0 1.2;
            3.0 0.0 0.0 5.0;
            7.4 0.0 0.0 3.5;
            2.0 1.0 1.5 4.0];

price = [0.8 0.8 0.9 0.9;
         0.8 0.0 0.0 0.9;
         1.5 0.0 0.0 1.5;
         1.5 1.5 1.5 1.9];

revenue = parkingRevenue(timeUsed, price);
display(revenue);
```

If your `parkingRevenue` function is working correctly, you should get the following output when running the test script:

Command Window:

```
>> TestParkingRevenue

revenue =

   49.4800
```

We recommend you test this function, especially with a variety of matrices (values and sizes) for the time used and the price per space, but you do not have to turn in any of your tests for this task.

Submission and Grading

The deliverables for this project are MATLAB function files and unit tests:

- Task 1 - `criticalLoad.m`
- Task 2 - `actualLoad.m`
- Task 2 - `TestActualLoad.m`
- Task 3 - `additionalPallets.m`
- Task 4 - `parkingRevenue.m`

Submit these files to the autograder (autograder.io) for grading.

The autograder will run a set of public tests - these are the same as the test scripts provided in this project specification. It will give you your score on these tests, as well as feedback on their output.

The autograder also runs a set of hidden tests. The are additional tests of your code (e.g. special cases). You still see your score on these tests, but you do not receive feedback on their output. The reason we do not give full feedback on all tests is because part of the project goal is for you to learn to develop effective testing practices of your own.

Your score on the public and hidden tests combined makes up your final autograder score. If you make multiple submissions, your autograder score is the best score from any submission. The submission with the best score is also the one used for style grading. If multiple submissions have the best score, we use the most recent one for style grading.

For example, let's assume that I turned in the following:

Submission #	1	2	3	4
Score	50	100	100	50

Then my grade for the Autograded portion would be 100 points (the best score of all submissions) and Submission #3 would be style graded since it is the latest submission with the best score.

You are limited to 5 submissions with feedback on the autograder per day. After the 5th submission, you are still able to submit, but all feedback will be hidden other than confirmation that your code was submitted.

Please refer to the syllabus on more information regarding partner groups and general information about the autograder.

After the due date, your autograder score is scaled to be out of 100 points. Next, your code is graded by hand to evaluate style and commenting, which are worth 10 additional points. The overall project grade is out of 110 points.

Autograder Details

There are some specific details to be aware of when working with the autograder. Make sure you understand these so you will have successful submissions.

Suppressing Output

The test scripts discussed in these project specifications display output in the command window, but your function implementations for each of tasks 1-4 should ***not*** display anything in the command window when they run. Suppress all output within functions by using semicolons, as is best practice.

The `TestActualLoad.m` script should only display messages relating to whether the unit tests have been passed. All other output should be suppressed with semicolons.

Acceptable Functions

Due to the structure of the autograder, there are certain limitations on which functions you can use and how you need to use them:

- If you use the `sum()` function, do not use the `'all'` option. MATLAB supports the `'all'` option, but the autograder does not.

If you fail to follow these guidelines, your functions will not run in the autograder.

Precision

Floating point numbers (i.e. numbers with a decimal point) can only be represented with a finite amount of precision on a computer. This means we can run into issues with roundoff error, where two different (correct) methods of computing the same result can yield numbers that are not precisely equal. Because of this, the autograder allows a certain amount of tolerance in the numeric results of your code - anything within 0.01 of the correct result will be treated as correct.

Style and Commenting

Maximum Score: 10 points

2 pts - Each submitted file has Name, Partner Uniqname (or "none"), Section Number, and Date Submitted included in a comment at the top
2 pts - Comments are used appropriately to describe your code (e.g. major steps are explained)
2 pts - Indenting and whitespace are appropriate (including functions are properly formatted)
2 pts - Variables are named descriptively (must rename `M`, `x`, `M1`, etc. in the functions)
2 pts - Other factors (variable names aren't all caps, etc...)