(a) Create arrays `A` and `B` with the data above.

(b) Find the indices (single-index) of elements of B that are less than 3. Assign this (column-vector) to variable named `validIndex`.

(c) Find the maximum value of A, considering only the locations where the B constraint is satisfied. Assign the maxinum value to `AmaxbyB` and the index where the max occurs to `Idx`. **Note:** In this example, the proper value for the maximizing index is 5. However, this will not be the index directly returned by `max`, since `max` is only "receiving" an array with 4 elements. Be careful here. There is one additional step.

(d) Use `ind2sub` to convert the single-index value `Idx` into the correct row and column indices. Assign these to `rowBest` and `colBest`.

5. All forms of quantitative design involves choices (ie., the "design"), and almost all choices involve tradeoffs. Complex dependencies make it critical to merge extensive domain-specific expertise with modern theoretical and computational optimization tools to determine the optimal choices.

   For problems with just a few (say 2) design choices, the incredible speed of today's laptop computers can be exploited, and quick solutions can be obtained to get a basic understanding of the dominant issues in a given design problem. In this problem, you will use computation to solve a simple design problem of this form. The ideas learned here can easily be applied to other problems that share the same basic structure.

   The design involves the choice of two parameters, b and h, which must be positive, and satisfy 4 constraints, listed below

   $$\frac{6P_v}{bh^2} \le 10^8, \quad \frac{6P_h}{b^2h} \le 10^8, \quad \frac{4P_v}{2\cdot 10^{11}bh^3} \le 0.001, \quad \frac{4P_h}{2\cdot 10^{11}b^3h} \le 0.001$$

   where $P_v = 1000$ and $P_h = 100$ are known, given values.

   Looking at the formula, it is clear that since b and h are both in the denominator of all of the fractions, simply picking them both "large enough" will satisfy the 4 constraints. However, the tradeoff is that it is also important to keep the quantity bh as small as possible.

   The general theory of optimization, which relies heavily on calculus can be used to understand (and determine through computation) the best choices. Our approach here will be simpler - use brute-force calculations, computing all relevant quantities for a large number of combinations of b and h values, check which choices satisfy the constraints, and then determine which value for b and h is the best at also making bh small. In the end, you will do this with just a handful of Matlab commands. The problem is broken into many parts, each corresponding to a code-cell in the template file.

   (a) Define parameters

   $$b_{\min} = 0.001, \quad b_{\max} = 0.2, \quad h_{\min} = 0.001, \quad h_{\max} = 0.2$$

   using variable names `bMin`, `bMax`, `hMin`, `hMax`. Also define variables `Pv` and `Ph` using the values given in the problem introduction.

(b) As mentioned, we will simply compute the relevant quantites for a large number of cominations of value of b and h. Let's use 500 different values for b and 500 different values for h, so 250000 different combinations will be considered. Define a variable $N = 500$, using the variable name `N`. **Remark:** Because of this choice, some of the arrays you define in subsequent parts are going to be very large, and unmanagable to look at. You should start by solving the problem with $N = 5$ instead, which will allow you to examine the arrays that are being created, and more carefully analyze the effect of the commands you write. You won't have the autograder to rely on, but you'll learn more, and that's what we're striving for. Once everything is behaving well, increase N to 10, and then to 500, and you'll probably have a complete, correct solution to the problem.

(c) Create a row-vector `bValues`, with N elements, uniformly-spaced from $b_{\min}$ to $b_{\max}$. For the purposes of describing calculations, denote these vectors

$$\texttt{bValues} = \begin{bmatrix} b_1 & b_2 & \cdots & b_N \end{bmatrix}$$

where $b_1 = b_{\min}$ and $b_N = b_{\max}$. Do the same for the values for h, giving

$$\texttt{hValues} = \begin{bmatrix} h_1 & h_2 & \cdots & h_N \end{bmatrix}$$

(d) Create an array B of dimension $N \times N$ with the values

$$\texttt{B} = \begin{bmatrix} b_1 & b_1 & \cdots & b_1 \\ b_2 & b_2 & \cdots & b_2 \\ \vdots & \vdots & \ddots & \vdots \\ b_N & b_N & \cdots & b_N \end{bmatrix}$$

Use the variable name `B`. Note that the (i, j) entry of B is b_i. Also create an array H of dimension $N \times N$ with the values

$$\texttt{H} = \begin{bmatrix} h_1 & h_2 & \cdots & h_N \\ h_1 & h_2 & \cdots & h_N \\ \vdots & \vdots & \ddots & \vdots \\ h_1 & h_2 & \cdots & h_N \end{bmatrix}$$

Use the variable name `H`. Note that the (i, j) entry of H is h_j.

(e) Create a $N \times N$ array $S1$ whose (i,j) element is equal to

$$S1(i, j) = \frac{6P_v}{b_i h_j^2}$$

Use the variable name `S1`.

(f) Create a $N \times N$ array $S2$ whose (i,j) element is equal to

$$S2(i, j) = \frac{6P_h}{b_i^2 h_j}$$

Use the variable name `S2`.

(g) Create a $N \times N$ array $D1$ whose (i,j) element is equal to

$$D1(i,j) = \frac{4P_v}{2 \cdot 10^{11} b_i h_j^3}$$

Use the variable name `D1`.

(h) Create a $N \times N$ array $D2$ whose (i,j) element is equal to

$$D2(i,j) = \frac{4P_h}{2 \cdot 10^{11} b_i^3 h_j}$$

Use the variable name `D2`.

(i) Create a $N \times N$ array A whose (i,j) element is equal to

$$A(i,j) = b_i h_j$$

Use the variable name `A`.

(j) Write a one-line command to determine how many pairs of (b,h) (from the N^2 pairs considered) satisfy the 4 design constraints

$$S1 < 10^8, \quad S2 < 10^8, \quad D1 < 0.001, \quad D2 < 0.001$$

We'll refer to the combinations which satisfy all 4 constraints as the "valid designs." Assign this integer number to a variable `nValidDesigns`.

(k) We don't just want to know how many valid designs there are - we actually want a valid design. So, we'll redo the main computation that you just used in getting `nValidDesigns`, but to obtain additional useful information in the process. Write a 1-line command to determine the indices (in single-index form) of the valid designs. Assign this column vector (whose length is equal to `nValidDesigns`) to a variable named `indexValidDesigns`.

(l) It is of course true that the minimum value of **all** the entries of A is simply $b_{\min} h_{\min}$. Determine what is the minimum value of A **corresponding to the valid designs**. Assign this to a variable named `Amin`. **Hint:** In this part (and the next part too) refer to the ideas you learned in Problem 4.

(m) Calculate the index (single-index) where (within the N^2 choices) the best design (satisfies all constraints and minimizes A) is located, assigning it to a variable named `indexBest`.

(n) Convert `indexBest`, which is relative to a $N \times N$ array, to the corresponding row and column. Use `rowBest` and `colBest` for the variable names.

(o) Using the row and column index associated with the best design, extract the optimal design parameters from `bValues` and `hValues`, respectively, storing them in variables named `bChoice` and `hChoice`. Use the command `disp` and `num2str` to print (to the Command Window) lines like

```
The best value for b is:  0.112

The best value for h is:  0.051

These choices yield the minimum value of A is:  0.005712
```

Note that these are not the correct answers, and only used to illustrate what character string we would like you to print.

Test Grades

In the assignment download there is a file called `TestGrades.mat` that contains a two-dimensional array named `Grades`. This array will be used for Problems **6 to 14**. Each row of this array corresponds to the grades for an individual student and each column corresponds to an individual test. Hence, the value of the (2,3)-element corresponds to the second student's grade on the third test.

The scalar variable `sNum` is the student number that you want to determine information about and the scalar variable `testNum` is the test number that you want to determine information about. Using relational operators and other `Matlab` functions write one-line `Matlab` expressions that perform the following tasks. These expressions should work for arrays of any size.

6. Assign `true` to the variable `everLowest` if the student indicated by `sNum` *has the lowest score on any of the tests*, or `false` if otherwise. Here, "lowest" is to mean that the student's score is less than or equal to all the other students.

7. Assign `true` to the variable `neverLowest` if the student indicated by `sNum` *does not have the lowest score on any of the tests*, or `false` if otherwise. Here, again, "lowest" is to mean that the student's score is less than or equal to all the other students.

8. Assign `true` to the variable `alwaysLowest` if the student indicated by `sNum` *has the lowest score on all of the tests*, or `false` if otherwise. As before, "lowest" is to mean that the student's score is less than or equal to all the other students.

9. Assign `true` to the variable `everHighest` if the student indicated by `sNum` *has the highest score on any of the tests*, or `false` if otherwise. Here, "highest" is to mean that the student's score is greater than or equal to all the other students.

10. Assign `true` to the variable `neverHighest` if the student indicated by `sNum` *does not have the highest score on any of the tests*, or `false` if otherwise. Here, again, "highest" is to mean that the student's score is greater than or equal to all the other students.

11. Assign `true` to the variable `alwaysHighest` if the student indicated by `sNum` *has the highest score on all of the tests*, or `false` if otherwise. As before, "highest" is to mean that the student's score is greater than or equal to all the other students.

12. Assign to the variable `highScoreOn` a 1-by-B array (B may be 0) of the test numbers where the student indicated by `sNum` scored the highest. Again, "highest" is to mean greater than or equal to all other students.