Hands-on Exercise 1

This exercise is designed to teach you to think in terms of matrices and vectors because this is how MATLAB organizes data. You will find that complicated operations can often be done with one or two lines of code if you use appropriate functions and have the data stored in an appropriate structure.
The other purpose of this homework is to make you comfortable with using **help** to learn new functions. The names of the functions you'll need to look up are provided in **bold** where needed.
Also, in case, recall we cannot use space in the script's name.
Exercise reports must be submitted on the website(超星学习通) before the end of the course.

What to turn in: Copy the text from your scripts and paste it into a document(use Matlab-报告模板.docx). If a question asks you to plot or display something to the screen, also include the plot and screen output your code generates. Submit either a *.doc or *.pdf file. **Name it as "学号_姓名_第X次实验报告".**

For problems 1-6, write a script called `shortProblems.m` and put all the commands in it. Separate and label different problems using comments.

1. **Scalar variables.** Make the following variables
   a. $a = 10$
   b. $b = 2.5 \times 10^{23}$
   c. $c = 2 + 3i$, where i is the square root of -1
   d. $d = e^{j2\pi/3}$, where j is the square root of -1 and e is Euler's number[1] (use **exp**, **pi**)

2. **Vector variables.** Make the following variables
   a. $aVec = \begin{bmatrix} 3.14 & 15 & 9 & 26 \end{bmatrix}$
   b. $bVec = \begin{bmatrix} 2.71 \\ 8 \\ 28 \\ 182 \end{bmatrix}$
   c. $cVec = \begin{bmatrix} 5 & 4.8 & \cdots & -4.8 & -5 \end{bmatrix}$ (all the numbers from 5 to -5 in increments of -0.2)
   d. $dVec = \begin{bmatrix} 10^{0} & 10^{0.01} & \cdots & 10^{0.99} & 10^{1} \end{bmatrix}$ (Logarithmically spaced numbers between 1 and 10, use **logspace**, make sure you get the length right!)
   e. $eVec = Hello$ ($eVec$ is a string, which is a vector of characters)

3. **Matrix variables.** Make the following variables

   a. $aMat = \begin{bmatrix} 2 & \cdots & 2 \\ \vdots & \ddots & \vdots \\ 2 & \cdots & 2 \end{bmatrix}$ a 9x9 matrix full of 2's (use **ones** or **zeros**)

   b. $bMat = \begin{bmatrix} 1 & 0 & \cdots & & 0 \\ 0 & \ddots & 0 & \ddots & \\ \vdots & 0 & 5 & 0 & \vdots \\ & \ddots & 0 & \ddots & 0 \\ 0 & & \cdots & 0 & 1 \end{bmatrix}$ a 9x9 matrix of all zeros, but with the values $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$ on the main diagonal (use **zeros**, **diag**).

   c. $cMat = \begin{bmatrix} 1 & 11 & \cdots & 91 \\ 2 & 12 & \ddots & 92 \\ \vdots & \vdots & \ddots & \vdots \\ 10 & 20 & \cdots & 100 \end{bmatrix}$ a 10x10 matrix where the vector 1:100 runs down the columns (use **reshape**).

   d. $dMat = \begin{bmatrix} NaN & NaN & NaN & NaN \\ NaN & NaN & NaN & NaN \\ NaN & NaN & NaN & NaN \end{bmatrix}$ a 3x4 NaN matrix (use **nan**)

   e. $eMat = \begin{bmatrix} 13 & -1 & 5 \\ -22 & 10 & -87 \end{bmatrix}$

   f. Make $fMat$ be a 5x3 matrix of random integers with values on the range -3 to 3 (First use **rand** and **floor** or **ceil.** Now only use **randi**)

4. **Scalar equations.** Using the variables created in 1, calculate x, y, and z.

   a. $x = \dfrac{1}{1+e^{\left(-(a-15)/6\right)}}$

   b. $y = \left(\sqrt{a} + \sqrt[21]{b}\right)^{\pi}$, recall that $\sqrt[g]{h} = h^{1/g}$, and use **sqrt.** You can also use **nthroot** (refer to the MATLAB help to understand the difference between **nthroot** and a fractional power)

   c. $z = \dfrac{\log\left(\Re\left[\left(c+d\right)\left(c-d\right)\right]\sin\left(a\pi/3\right)\right)}{c\bar{c}}$ where $\Re$ indicates the real part of the complex number in brackets, $\bar{c}$ is the complex conjugate of c, and $\log$ is the *natural* log (use **real**, **conj**, **log**).

5. **Matrix equations.** Using the variables created in 2 and 3, find the values of xMat, yMat and zMat below. Use matrix operators.
    a. $\mathrm{xMat} = (\mathrm{aVec} \cdot \mathrm{bVec}) \cdot \mathrm{aMat}^2$
    b. $\mathrm{yMat} = (\mathrm{bVec} \cdot \mathrm{aVec})$, note that this is *not* the same as $(\mathrm{aVec} \cdot \mathrm{bVec})$
    c. $\mathrm{zMat} = |\mathrm{cMat}|(\mathrm{aMat} \cdot \mathrm{bMat})^{\mathrm{T}}$, where $|cMat|$ is the determinant of $cMat$, and T again indicates the transpose (use **det**).

6. **Common functions and indexing.**
    a. Make $cSum$ the column-wise sum of $cMat$. The answer should be a row vector (use **sum**).
    b. Make $eMean$ the mean across the rows of $eMat$. The answer should be a column (use **mean**).
    c. Replace the top row of $eMat$ with $\begin{bmatrix}1 & 1 & 1\end{bmatrix}$.
    d. Make $cSub$ the submatrix of $cMat$ that only contains rows 2 through 9 and columns 2 through 9.
    e. Make the vector $lin = \begin{bmatrix}1 & 2 & \cdots & 20\end{bmatrix}$ (the integers from 1 to 20), and then make every even value in it negative to get $lin = \begin{bmatrix}1 & -2 & 3 & -4 & \cdots & -20\end{bmatrix}$.
    f. Make r a 1x5 vector using **rand**. Find the elements that have values <0.5 and set those values to 0 (use **find**).

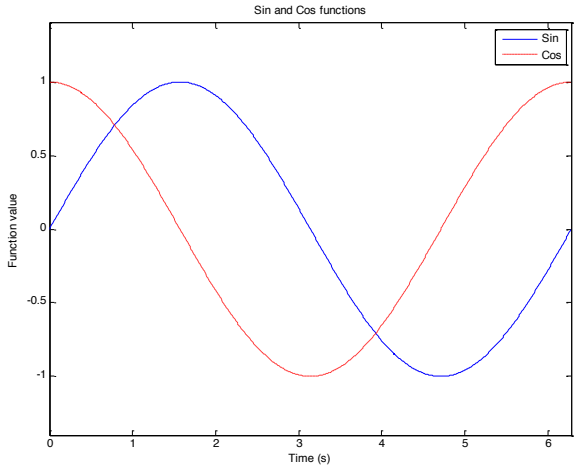
7. **Plotting multiple lines and colors.** In class we saw how to plot a single line in the default blue color on a plot. You may have noticed that subsequent plot commands simply replace the existing line. Here, we'll write a script to plot two lines on the same axes.
    a. Open a script and name it `twoLinePlot.m`. Write the following commands in this script.
    b. Make a new figure using **figure**
    c. We'll plot a sine wave and a cosine wave over one period
        i. Make a time vector t from 0 to 2π with enough samples to get smooth lines
        ii. Plot $\sin(t)$
        iii. Type **hold on** to turn on the 'hold' property of the figure. This tells the figure not to discard lines that are already plotted when plotting new ones. Similarly, you can use **hold off** to turn off the hold property.
        iv. Plot $\cos(t)$ using a red dashed line. To specify line color and style, simply add a third argument to your plot command (see the third paragraph of the **plot** help).

This argument is a string specifying the line properties as described in the help file. For example, the string 'k:' specifies a black dotted line.

d. Now, we'll add labels to the plot
   i. Label the x axis using **xlabel**
   ii. Label the y axis using **ylabel**
   iii. Give the figure a title using **title**
   iv. Create a legend to describe the two lines you have plotted by using **legend** and passing to it the two strings 'Sin' and 'Cos'.

e. If you run the script now, you'll see that the x axis goes from 0 to 7 and y goes from -1 to 1. To make this look nicer, we'll manually specify the x and y limits. Use **xlim** to set the x axis to be from 0 to 2π and use **ylim** to set the y axis to be from -1.4 to 1.4.

f. Run the script to verify that everything runs right. You should see something like this:



8. **Manipulating variables.** Write a script to read in some grades, curve them, and display the overall grade. To do this, you'll need to use the file `classGrades.mat` and put it in the same folder as your script.
    a. Open a script and name it `calculateGrades.m`. Write all the following commands in this script.
    b. Load the `classGrades` file using **load**. This file contains a single variable called $namesAndGrades$
    c. To see how $namesAndGrades$ is structured, display the first 5 rows on your screen. The first column contains the students 'names', they're just the integers from 1 to 38. The remaining 7 columns contain each student's score on each of 7 assignments. There are also some NaNs which indicate that a particular student was absent on that day and didn't do the assignment.
    d. We only care about the grades, so extract the submatrix containing all the rows but only columns 2 through 8 and name this matrix $grades$ (to make this work on any size matrix, don't hard-code the 8, but rather use **end** or **size(namesAndGrades,2)**).
    e. Calculate the mean score on each assignment. The result should be a 1x7 vector containing the mean grade on each assignment.
        i. First, do this using **mean**. Display the mean grades calculated this way. Notice that the NaNs that were in the grades matrix cause some of the mean grades to be NaN as well.
        ii. To fix this problem, do this again using **nanmean**. This function does exactly what you want it to do, it computes the mean using only the numbers that are not NaN. This means that the absent students are not considered in the calculation, which is what we want. Name this mean vector $meanGrades$ and display it on the screen to verify that it has no NaNs
    f. Normalize each assignment so that the mean grade is 60 . You'll want to divide each column of $grades$ by the correct element of $meanGrades$.

        i. Make a matrix called $meanMatrix$ such that it is the same size as $grades$, and each row has the values $meanGrades$. Do this by taking the outer product of a 38x1 vector of ones and the vector $meanGrades$, which is a row (use **ones**, *****). Display $meanMatrix$ to verify that it looks the way you want.

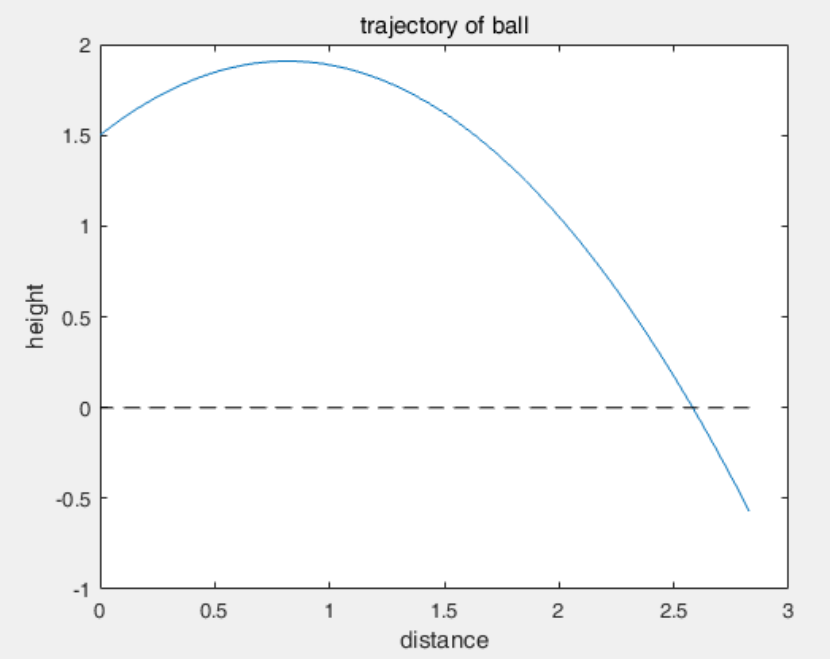
        ii. To calculate the curved grades, do the following: $curvedGrades = 60\left(grades / meanMatrix \right)$. Keep in mind that you want to do the division elementwise.
        iii. Compute and display the mean of $curvedGrades$ to verify that they're all 60 (**nanmean**).
        iv. Because we divided by the mean and multiplied by 60, it's possible that some grades that were initially close to 100 are now larger than 100. To fix this, find all the

elements in *curvedGrades* that are greater than 100 and set them to 100.

g. Calculate the total grade for each student and assign letter grades
    i. To calculate the *totalGrade* vector, which will contain the numerical grade for each student, you want to take the mean of *curvedGrades* across the columns (use **nanmean**, see help for how to specify the dimension). Also, we only want to end up with numbers from 0 to 100, so calculate the ceiling of the *totalGrade* vector (use **ceil**).
    ii. Make a string called *letters* that contains the letter grades in increasing order: BA
    iii. Make the final letter grades vector *letterGrades* by using *totalGrade* (which after the ceil operation should only contain values between 1 and 100) to index into *letters* .
    iv. Finally, display the following using **disp**: Grades: *letterGrades*
h. Run the script to verify that it works.

9. **Throwing a ball.** Below are all the steps you need to follow, but you should also add your own meaningful comments to the code as you write it.
    a. Start a new file in the MATLAB Editor and save it as `throwBall.m`
    b. At the top of the file, define some constants (you can pick your own variable names)
        i. Initial height of ball at release = 1.5 m
        ii. Gravitational acceleration = 9.8 m/s^2
        iii. Velocity of ball at release = 4 m/s
        iv. Angle of the velocity vector at time of release = 45 degrees
    c. Next, make a time vector that has 1000 linearly spaced values between 0 and 1, inclusive.
    d. If x is distance and y is height, the equations below describe their dependence on time and all the other parameters (initial height h, gravitational acceleration g, initial ball velocity v, angle of velocity vector in degrees θ). Solve for x and y
        i. $x(t) = v\cos\left(\theta\frac{\pi}{180}\right)t$. We multiply θ by $\frac{\pi}{180}$ to convert degrees to radians.
        ii. $y(t) = h + v\sin\left(\theta\frac{\pi}{180}\right)t - \frac{1}{2}gt^2$
    e. Approximate when the ball hits the ground.
        i. Find the index when the height first becomes negative (use **find**).
        ii. The distance at which the ball hits the ground is value of x at that index
        iii. Display the words: *The ball hits the ground at a distance of X m*eters. (where X is the distance you found in part ii above)
    f. Plot the ball's trajectory
        i. Open a new figure (use **figure**)
        ii. Plot the ball's height on the y axis and the distance on the x axis (**plot**)
        iii. Label the axes meaningfully and give the figure a title (use **xlabel**, **ylabel**, and **title**)
        iv. Hold on to the figure (use **hold on**)
        v. Plot the ground as a dashed black line. This should be a horizontal line going from 0 to the maximum value of x (use **max**). The height of this line should be 0. (see **help plot** for line colors and styles)
    g. Run the script from the command window and verify that the ball indeed hits the ground around the distance you estimated in e-ii. You should get something like this:

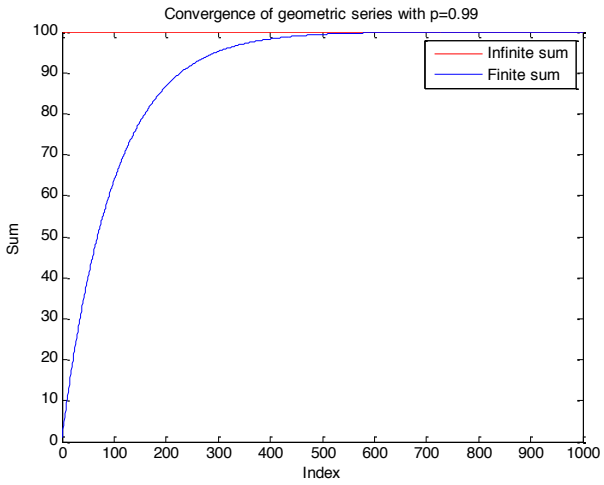
```
>> throwBall
The ball hits the ground at a distance of 2.5821 meters
```

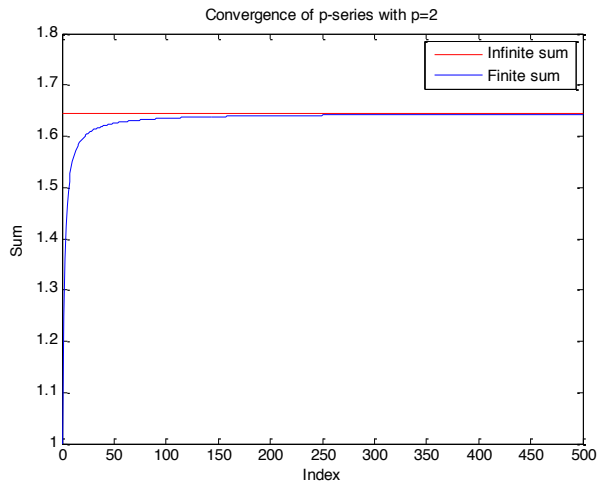


Optional Problems

2. **Convergence of infinite series.** We'll look at two series that converge to a finite value when they are summed.
    a. Open a new script in the MATLAB Editor and save it as `seriesConvergence.m`
    b. First, we'll deal with a geometric series $G = \sum_{k=0}^{\infty} p^k$. We need to define the value of p and the values of k
        i. p =0.99.
        ii. k is a vector containing the integers from 0 to 1000, inclusive.
    c. Calculate each term in the series (before summation)
        i. $geomSeries = p^k$ (this should be done elementwise)
    d. Calculate the value of the infinite series
        i. We know that $G = \sum_{k=0}^{\infty} p^k = \frac{1}{1-p}$
    e. Plot the value of the infinite series
        i. Plot a horizontal red line that has x values 0 and the maximum value in k (use **max**), and the y value is constant at G.
    f. On the same plot, plot the value of the finite series for all values of k
        i. Plot the cumulative sum of $geomSeries$ versus k. The cumulative sum of a vector is a vector of the same size, where the value of each element is equal to the sum of all the elements to the left of it in the original vector. (use **cumsum**, and try cumsum([1 1 1 1 1]) to understand what it's doing.) Use a blue line when plotting.
    g. Label the x and y axes, and give the figure a title (**xlabel**, **ylabel**, **title**). Also create a legend and label the first line 'Infinite sum', and the second line 'Finite Sum' (**legend**).
    h. Run the script and note that the finite sum of 1000 elements comes very close to the value of the infinite sum.
    i. Next, we will do a similar thing for another series, the p-series: $P = \sum_{n=1}^{\infty} \frac{1}{n^p}$
    j. At the bottom of the same script, initialize new variables
        i. $p = 2$
        ii. n is a vector containing all the integers from 1 to 500, inclusive.
    k. Calculate the value of each term in the series
        i. $pSeries = \frac{1}{n^p}$
    l. Calculate the value of the infinite p-series. The infinite p-series with $p = 2$ has been proven to converge to $P = \sum_{n=1}^{\infty} \frac{1}{n^p} = \frac{\pi^2}{6}$.

m. Make a new figure and plot the infinite sum as well as the finite sum, as we did for the geometric series
    i. Make a new figure
    ii. Plot the infinite series value as a horizontal red line with x values 0 and the maximum value in n, and the y value is constant at P.
    iii. Hold on to the figure, and plot the cumulative sum of $pSeries$ versus n (use **hold on**, **cumsum**).
    iv. Label the x and y axes, give the figure a title, and make a legend to label the lines as 'Infinite sum', and 'Finite sum' (use **xlabel**, **ylabel**, **title**, **legend**)
n. Run the script to verify that it produces the expected output. It should look something like this:





3. **Encryption Algorithm（加密算法）.** Write a simple shuffling 'encryption' algorithm.
    a. Open a new script and save it as `encrypt.m`
    b. At the top of the script, define the *original* string to be: This is my top secret message!
    c. Next, let's shuffle the indices of the letters. To do this, we need to make a string of encoding indices
        i. Make a vector that has the indices from 1 to the length of the original string in a randomly permuted（随机排列） order. Use **randperm** and **length**
        ii. Encode the original string by using your encoding vector as indices into *original*. Name the encoded message *encoded*.
    d. Now, we need to figure out the decoding key to match the encoding key we just made.
        i. Assemble a temporary matrix where the first column is the encoding vector you made in the previous part and the second column are the integers from 1 to the length of the original string in order. Use **length**, and you may need to transpose some vectors to make them columns using **'**.
        ii. Next, we want to sort the rows of this temporary matrix according to the values in the first column. Use **sortrows**.
        iii. After it's been sorted, extract the second column of the temporary matrix. This is your decoding vector.
        iv. To make the *decoded* message, use the decoding vector as indices into *encoded*.
    e. Display the original, encoded, and decoded messages
        i. Display the following three strings, where : *original*, *encoded*, and *decoded* are the strings you made above. Use **disp**
        Original: *original*
        Encoded: *encoded*
        Decoded: *decoded*
    f. Compare the original and decoded strings to make sure they're identical and display the result
        i. Use **strcmp** to compare the *original* and *decoded* strings. Name the output of this operation *correct*. *correct* will have the value 1 if the strings match and the value 0 if they don't
        ii. Display the following string: Decoded correctly (1 true, 0 false): *correct*
        use **disp** and **num2str**
    g. Run the script a few times to verify that it works well. You should see an output like this:

```
>> encrypt
Original: This is my top secret message!
Encoded : sisrpigct tessaoheem m  yTse!
Decoded : This is my top secret message!
Decoded correctly (1 true, 0 false): 1
```