Homework 3

This homework is designed to give you practice writing functions to solve problems. The problems in this homework are very common and you will surely encounter similar ones in your research or future classes. As before, the names of helpful functions are provided in **bold** where needed. **Homework must be submitted before the start of the next class.**

What to turn in: Copy the text from your scripts and paste it into a document. 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(x) or *.pdf file.

Keep all your code in scripts/functions. If a specific name is not mentioned in the problem statement, you can choose your own script names.

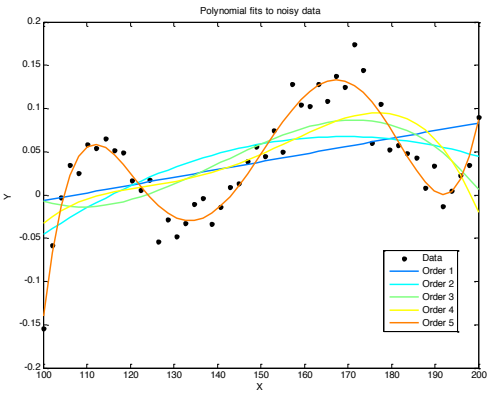
1. **Linear system of equations.** Solve the following system of equations using ****. Compute and display the error vector

$$\begin{aligned} 3a+6b+4c &= 1 \\ a+5b &= 2 \\ 7b+7c &= 3 \end{aligned}$$

2. **Numerical integration.** What is the value of: $\int_0^5 xe^{-x/3}dx$? Use **trapz** or **quad**. Compute and display the difference between your numerical answer and the analytical answer: $-24e^{-5/3}+9$.

3. **Computing the inverse.** Calculate the inverse of $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and verify that when you multiply the original matrix by the inverse, you get the identity matrix (**inv**). Display the inverse matrix as well as the result of the multiplication of the original matrix by its inverse.

4. **Fitting polynomials**. Write a script to load the data file `randomData.mat` (which contains variables `x` and `y`) and fit first, second, third, fourth, and fifth degree polynomials to it. Plot the data as blue dots on a figure, and plot all five polynomial fits using lines of different colors on the same axes. Label the figure appropriately. To get good fits, you'll have to use the centering and scaling version of **polyfit** (the one that returns three arguments, see **help**) and its counterpart in **polyval** (the one that accepts the centering and scaling parameters). It should look like this:



5. **Hodgkin-Huxley model of the neuron.** You will write an ODE file to describe the spiking of a neuron, based on the equations developed by Hodgkin and Huxley in 1952 (they received a Nobel Prize for this work). The main idea behind the model is that ion channels in the neuron's membrane have voltage-sensitive gates that open or close as the transmembrane voltage changes. Once the gates are open, charged ions can flow through them, affecting the transmembrane voltage. The equations are nonlinear and coupled, so they must be solved numerically.
   a. Download the HH.zip file from the class website and unzip its contents into your homework folder. This zip folder contains 6 m-files: `alphah.m`, `alpham.m`, `alphan.m`, `betah.m`, `betam.m`, `betan.m`. These functions return the voltage-dependent opening ($\alpha(V)$) and closing ($\beta(V)$) rate constants for the h, m, and n gates.
   b. Write an ODE file to return the following derivatives (you don't need to understand what they mean):

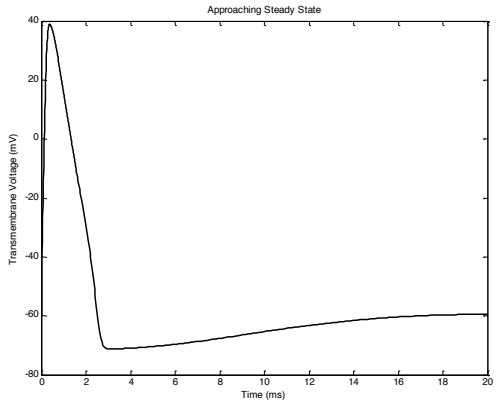
$$\frac{dn}{dt} = (1-n)\alpha_n(V) - n\beta_n(V)$$
$$\frac{dm}{dt} = (1-m)\alpha_m(V) - m\beta_m(V)$$
$$\frac{dh}{dt} = (1-h)\alpha_h(V) - h\beta_h(V)$$
$$\frac{dV}{dt} = -\frac{1}{C}\left(G_K n^4 (V - E_K) + G_{Na} m^3 h (V - E_{Na}) + G_L (V - E_L)\right)$$

   and the following constants (C is membrane capacitance, G are the conductances and E are the reversal potentials of the potassium (K), sodium (Na), and leak (L) channels):

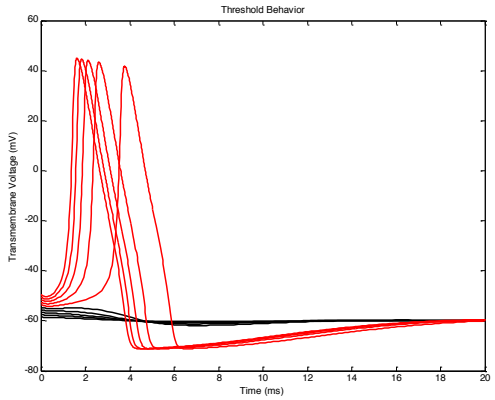
$$C = 1$$
$$G_K = 36$$
$$G_{Na} = 120$$
$$G_L = 0.3$$
$$E_K = -72$$
$$E_{Na} = 55$$
$$E_L = -49.4$$

   c. Write a script called `HH.m` which will solve this system of ODEs and plot the transmembrane voltage. First, we'll run the system to steady state. Run the simulation for 20ms (the timescale of the equations is ms) with initial values: $n = 0.5; m = 0.5; h = 0.5; V = -60$ (**ode45**). Store the steady-state value of all 4 parameters in a vector called `ySS`. Make a new figure and plot the timecourse of $V(t)$

to verify that it reaches steady state by the end of the simulation. It should look something like this:



d. Next, we'll explore the trademark feature of the system: the all-or-none action potential. Neurons are known to 'fire' only when their membrane surpasses a certain voltage threshold. To find the threshold of the system, solve the system 10 times, each time using `ySS` as the initial condition (**ode45** will take this as an input argument) but increasing the initial value of V by 1, 2, … 10 mV from its steady state value. After each simulation, check whether the peak voltage surpassed 0mV, and if it did, plot the voltage with a red line, but if it didn't, plot it with a black line. Plot all the membrane voltage trajectories on the same figure, like below. We see that if the voltage threshold is surpassed, then the neuron 'fires' an action potential; otherwise it just returns to the steady state value. You can zoom in on your figure to see the threshold value (the voltage that separates the red lines from the black lines).



Optional Problem

1. **Linear regression, in multiple ways.** Linear regression is a widely-used class of statistical models that attempts to fit a relationship between a scalar dependent variable y and one or more independent variables x. Suppose you run an experiment with p independent variables, with values $\boldsymbol{x} = [x_1, \dots, x_p]$, and as a result you measure a real number y. Repeating this n times, you can assemble the following matrices:

$$\boldsymbol{X} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,p} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,p} \end{bmatrix} \; ; \; \boldsymbol{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

The goal of linear regression is the fit each y_m to its corresponding $\boldsymbol{x_m} = [x_{m,1}, \dots, x_{m,p}]$, by finding the appropriate $\boldsymbol{\theta} = [\theta_1, \dots, \theta_p]^{\mathrm{T}}$ and b that fits the following set of n equations:

$$y_1 = \theta_1 x_{1,1} + \theta_2 x_{1,2} + \cdots + \theta_p x_{1,p} + b = \boldsymbol{x_1} * \boldsymbol{\theta} + b$$
$$\vdots$$
$$y_n = \theta_1 x_{n,1} + \theta_2 x_{n,2} + \cdots + \theta_p x_{n,p} + b = \boldsymbol{x_n} * \boldsymbol{\theta} + b$$

In other words, we would like to model the relationship as follows:

$$y = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_p x_p + b = \boldsymbol{x} * \boldsymbol{\theta} + b$$

You might be familiar with the case where $p = 1$: finding the line of best fit. In particular, least-squares is the most well-known approach. In this exercise we will consider multiple ways to perform least squares, each with its own merits, along with some natural generalizations.

   a. Download the file `regression.mat`
   b. from the class website and load it into your workspace. This will provide you with data matrices `X` (data) and `Y` (labels,) where in this case they are actually column vectors ($n = 100, p = 1$).
   c. The first thing you should do when you have data is to visualize it! Fortunately, the given matrices `X` and `Y` are 1-D, so a 2-D **scatter** plot suffices. Plot the data as black dots.
   d. Examining the plot, the data mostly indicates a linear relationship, with some regional outliers. Let's try fitting a line using least-squares regression. We want to find $\boldsymbol{\theta}$ and b that fits the data well by minimizing the sum-of-squared-errors (hence "least squares"):

$$J(\boldsymbol{\theta}, b) = \sum_{m=1}^{n} (y_m - (\boldsymbol{x_m} * \boldsymbol{\theta} + b))^2$$

      Intuitively, if each $(\boldsymbol{x_m} * \boldsymbol{\theta} + b)$ matches closely to y_m, then we have found a good fit. We will minimize this cost function $J(\boldsymbol{\theta}, b)$ by numerical optimization, using **fminsearch**. Before we so, it will be more convenient to concatenate $\boldsymbol{\theta}$ and b into a single parameter vector $\boldsymbol{\beta} = [\boldsymbol{\theta}; b]$. Notice that $(\boldsymbol{x_m} * \boldsymbol{\theta} + b) = [\boldsymbol{x_m}, 1] * \boldsymbol{\beta}$.
      Create an augmented data matrix `XAug` by concatenating a column of ones to `X`.
   e. Recall how **fminsearch** is used: We provide it with a function that evaluates, for any $\boldsymbol{\beta}$, the objective function being optimized ($J(\boldsymbol{\beta})$ in this case). Write (in a separate file) a function `cost = squaredCost(beta, dataAug, labels)` that computes $J(\boldsymbol{\beta})$ using the given augmented data points in `dataAug` and labels in `labels`.

Note: Do not use `XAug` and `Y`! The function cannot access variables in the workspace; they will be passed in through the arguments of `squaredCost`.

Optional: Can you do this without using a for-loop? Optimization will be much faster.

f. Note that our function has three input arguments, however we only wish to minimize the cost function with respect to `beta`, while keeping `dataAug` and `labels` as parameters which are constant in our minimization. In order to use **fminsearch** we will need to define a new, anonymous function which only has one parameter (`beta`). This function will call our previously declared `squaredCost` function and pass on our workspace variables `XAug` and `Y` as parameters. Complete the following command to declare the new anonymous function `squaredCostReduced = @(beta)…`

g. Now call **fminsearch** by using `squaredCostReduced` and an initial value. See **help** to find out what the first two arguments should be (they're the same as introduced in lecture). Use zeros for the initial value, noting that it needs have the same size as $\boldsymbol{\beta}$. Display `beta`, and compute the cost $J(\boldsymbol{\beta})$ using `squaredCost` (or `squaredCostReduced`) and `beta`. Verify by plotting the fitted line (with coefficients in `beta`) with the data points.

h. Since there is only a single independent variable, we can try **polyfit** on the data as well. Call **polyfit** on `X` and `Y` to fit a straight line, and verify that the coefficients returned match those from **fminsearch** (since **polyfit** does least-squares regression too).

i. It turns out that there is an analytical solution for least-squares regression, known as the normal equations (which you can find by differentiating $J(\boldsymbol{\beta})$ and setting equal to zero):

$$\boldsymbol{\beta} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

Verify that this gives the same set of coefficients (use `XAug`).

j. In fact, because we have an over-determined system of linear equations, we can solve for $\boldsymbol{\beta}$ in the same way you did in Q1. Verify your answer.

k. At this point, you may think that we've wasted our time trying to re-invent the wheel. Well, not quite! First, notice that **polyfit** cannot handle more than one independent variable. Second, the analytical solutions only work for the beautiful quadratic cost function we've seen above, the sum-of-squares error. This cost function is motivated by a simple probabilistic modeling assumption, that the sampled outputs `Y` follow the linear relationship with added normally-distributed (Gaussian) noise.[1] However, this noise distribution is unrealistic in many cases, and modifications are needed.

Looking at the plots you've made, you will notice that the fitted line does not fit the majority of data well because of a few outliers. It turns out that, instead of using the sum-of-squares error, using absolute-deviation error significantly improves robustness against outliers (Optional: Why is that? What does the cost function mean?):

[1] Incidentally, Gauss, the "Prince of Mathematics", is widely credited with developing and advancing the theory behind the least-squares method in the 18th century. Along with many other achievements: http://www.gaussfacts.com

$$J_{abs}(\boldsymbol{\theta}, b) = \sum_{m=1}^{n} abs(\, y_m - (\boldsymbol{x_m} * \boldsymbol{\theta} + b)\,)$$

Repeats parts (d)-(f), this time writing a different function `cost = absoluteCost(beta, dataAug, labels)`. Note that it's not valid to compare the values from the two different cost functions. Plot the data, the least-squares line (in blue), and the least-absolute-deviations line (in red) together.

There's no analytical solution for this cost function![2]

l. Optional: Investigate what happens when you use other p-norms in the cost function:

$$J_{abs}(\boldsymbol{\theta}, b) = \sum_{m=1}^{n} [\, abs(\, y_m - (\boldsymbol{x_m} * \boldsymbol{\theta} + b)\,)\,]^p$$

The case of $p = 1$ corresponds to least-absolute-deviations, $p = 2$ to least-squares.

Try modifying your loss function to take in p as well so that you only need to write one!

2. **Optional, but highly recommended: Julia Sets.** In this problem you will generate quadratic Julia Sets. The following description is adapted from Wikipedia at http://en.wikipedia.org/wiki/Julia_Sets. For more information about Julia Sets please read the entire article there.

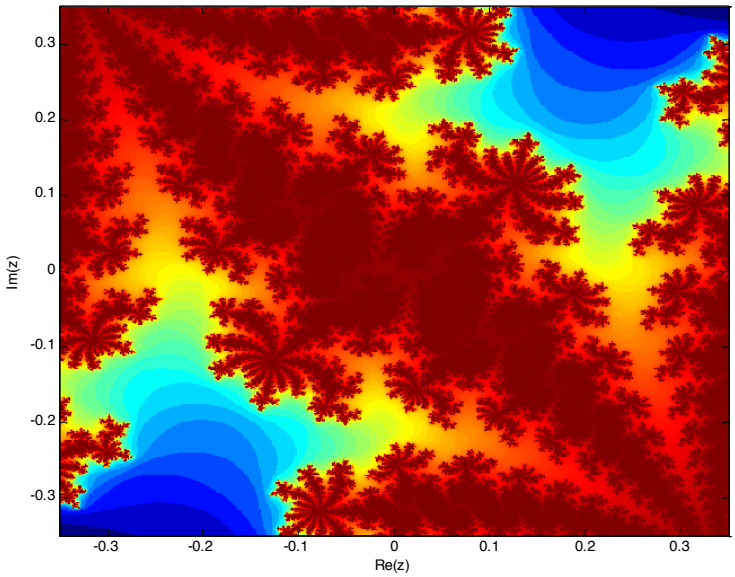
Given two complex numbers, c and z_0, we define the following recursion:

$$z_n = z_{n-1}^2 + c$$

This is a dynamical system known as a quadratic map. Given a specific choice of c and z_0, the above recursion leads to a sequence of complex numbers $z_1, z_2, z_3 \ldots$ called the orbit of z_0. Depending on the exact choice of c and z_0, a large range of orbit patterns are possible. For a given fixed c, most choices of z_0 yield orbits that tend towards infinity. (That is, the modulus $|z_n|$ grows without limit as n increases.) For some values of c certain choices of z_0 yield orbits that eventually go into a periodic loop. Finally, some starting values yield orbits that appear to dance around the complex plane, apparently at random. (This is an example of chaos.) These starting values, z_0, make up the Julia set of the map, denoted J_c. In this problem, you will write a MATLAB script that visualizes a slightly different set, called the filled-in Julia set (or Prisoner Set), denoted K_c, which is the set of all z_0 with orbits which do not tend towards infinity. The "normal" Julia set J_c is the edge of the filled-in Julia set. The figure below illustrates a Julia Set for one particular value of c. You will write MATLAB code that can

[2] Another way to deal with outliers is to use weighted least-squares regression, where each data point has a different weight in the cost function. Outliers are first detected and given lower weights, i.e., less importance in the optimization.

generate such fractals in this problem.



a. It has been shown that if the modulus of z_n becomes larger than 2 for some n then it is guaranteed that the orbit will tend to infinity. The value of n for which this becomes true is called the 'escape velocity' of a particular z_0. Write a function that returns the escape velocity of a given z_0 and c. The function declaration should be: `n=escapeVelocity(z0,c,N)` where `N` is the maximum allowed escape velocity (basically, if the modulus of z_n does not exceed 2 for `n<N`, return `N` as the escape velocity. This will prevent infinite loops). Use **abs** to calculate the modulus of a complex number

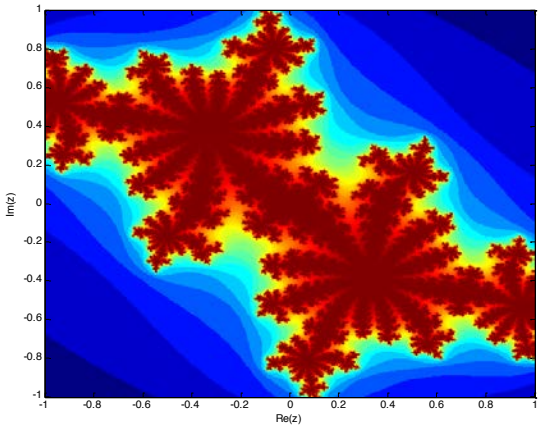
b. To generate the filled in Julia Set, write the following function `M=julia(zMax,c,N)`. `zMax` will be the maximum of the imaginary and complex parts of the various z_0's for which we will compute escape velocities. `c` and `N` are the same as defined above, and M is the matrix that contains the escape velocity of various z_0 's.

   i. In this function, you first want to make a 500x500 matrix that contains complex numbers with real part between `-zMax` and `zMax`, and imaginary part between `-zMax` and `zMax`. Call this matrix `Z`. Make the imaginary part vary along the y axis of this matrix. You can most easily do this by using **linspace** and **meshgrid**, but you can also do it with a loop.

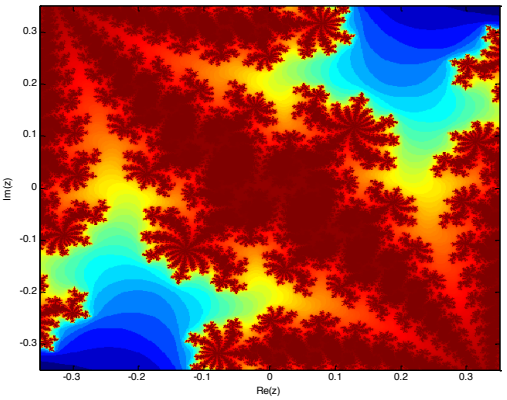
   ii. For each element of `Z`, compute the escape velocity (by calling your `escapeVelocity`) and store it in the same location in a matrix `M`. When done, the matrix `M` should be the same size as `Z` and contain escape velocities with values between 1 and `N`.

iii. Run your `julia` function with various `zMax`, `c`, and `N` values to generate various fractals. To display the fractal nicely, use **imagesc** to visualize `atan(0.1*M)`, (taking the arctangent of `M` makes the image look nicer; you may also want to use **axis xy** so the y values aren't flipped). WARNING: this function may take a while to run.

The figure below was created by running:
`M=julia(1,-.297491+i*0.641051,100);` and visualizing it as described above.



The figure below was generated by running the same `c` parameter as above, but on a smaller range of `z` values and with a larger `N`:
`M=julia(.35,-.297491+i*0.641051,250);`



3. **Optional: Solve a Sudoku game.** Sudoku is a popular number placement puzzle game. The objective of the game is to fill a 9x9 grid with 9 sets of the numbers from 1 to 9, such that each number (from 1 to 9) occurs only once every row, column and 3x3 sub-block. Below is an example of a Sudoku board before and after solving.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 1 Unsolved (left) and solved (right) Sudoku boards

In this exercise we will write a simple Matlab function that receives a 9x9 matrix input representing an incomplete Sudoku game and returns the correctly solved board. There are several approaches for solving such a problem; we will implement a simple backtracking algorithm.

The concept of backtracking involves guessing a certain solution which does not violate the game rules and proceeding until we reach a point where we cannot satisfy the rules. At this point, we reverse our steps (backtrack) to the last known working solution and make a different choice among the available possibilities until the entire game is solved.

a. Download the file `sudoku.mat` from the class website and load it into your workspace. You will now have an example for an unsolved Sudoku board in your workspace named `unsolvedBoard`.

b. Write a function with the declaration `safe = checkSudoku(board, row, col, num)`

   `board` is a 9x9 matrix which contains the Sudoku board to check. Unfilled board cells are indicated by a NaN value.

   `row` and `col` are indexes to the check if the number `num` (between 1 and 9) can be inserted without causing a violation. i.e. does `num` already exist in the given row, column or 3x3 sub-block. Note that we do not need to check the entire board's validity (although you could try to do so as practice) since we will be filling our board one element at a time.

   The function returns a logical `true` or `false` value in the output `safe` depending on the outcome of the check.

   There are several ways to accomplish this functionality. Possible functions you may want to use are **sort**, **unique**, **ismember, any, all**. Note that NaN values are unequal to one another!

   Test your function with various inputs to see it is operating as expected!

   For example:

```
>> checkSudoku(unsolvedBoard, 3, 1, 1)
ans =
     1
```

```
>> checkSudoku(unsolvedBoard, 3, 1, 5)
ans =
     0
```

c. Write a function with the declaration `solvedBoard = solveSudoku(board)`. The input to this function will be an incomplete Sudoku board and the output will be the solved board.

   Below are hints and suggestions for steps to help you implement this function:

   i. Find the indices of all the missing values in the table and store them in the variable `emptyInd` (**find**, **isnan**)

   ii. Create a variable called `ind` which will indicate our current location within the `emptyInd` array

   iii. Write a loop that will traverse the `emptyInd` array. At each location check what number can be placed in that location using your function from part (b) (Which numbers do you need to check?). If it's not valid, try a different number, if none work, replace it with a NaN, and go back to the previous empty index and try a different number than before. (Which loop type would be most suitable for this scenario?) (**ind2sub**, **checkSudoku**)

d. Test your function with the given example `unsolvedBoard` and compare the result to the image shown above.

e. **Optional**: Write a function `displaySudoku(board)` for displaying the Sudoku game board in a more "human friendly" form. See an example for such a display below.

```
>> load sudoku
>> displaySudoku(unsolvedBoard)
+=============================+
| 5  3     |     7     |          |
| 6        | 1  9  5   |          |
|     9  8 |           |     6    |
|=============================|
| 8        |     6     |        3 |
| 4        | 8     3   |        1 |
| 7        |     2     |        6 |
|=============================|
|     6    |           | 2  8     |
|          | 4  1  9   |        5 |
|          |     8     |     7  9 |
+=============================+

>> solvedBoard = solveSudoku(unsolvedBoard);
>> displaySudoku(solvedBoard)
+=============================+
| 5  3  4  | 6  7  8  | 9  1  2  |
```

```
| 6  7  2 | 1  9  5 | 3  4  8 |
| 1  9  8 | 3  4  2 | 5  6  7 |
|=========================|
| 8  5  9 | 7  6  1 | 4  2  3 |
| 4  2  6 | 8  5  3 | 7  9  1 |
| 7  1  3 | 9  2  4 | 8  5  6 |
|=========================|
| 9  6  1 | 5  3  7 | 2  8  4 |
| 2  8  7 | 4  1  9 | 6  3  5 |
| 3  4  5 | 2  8  6 | 1  7  9 |
+=========================+
```

6.057 Introduction to MATLAB

IAP 2019