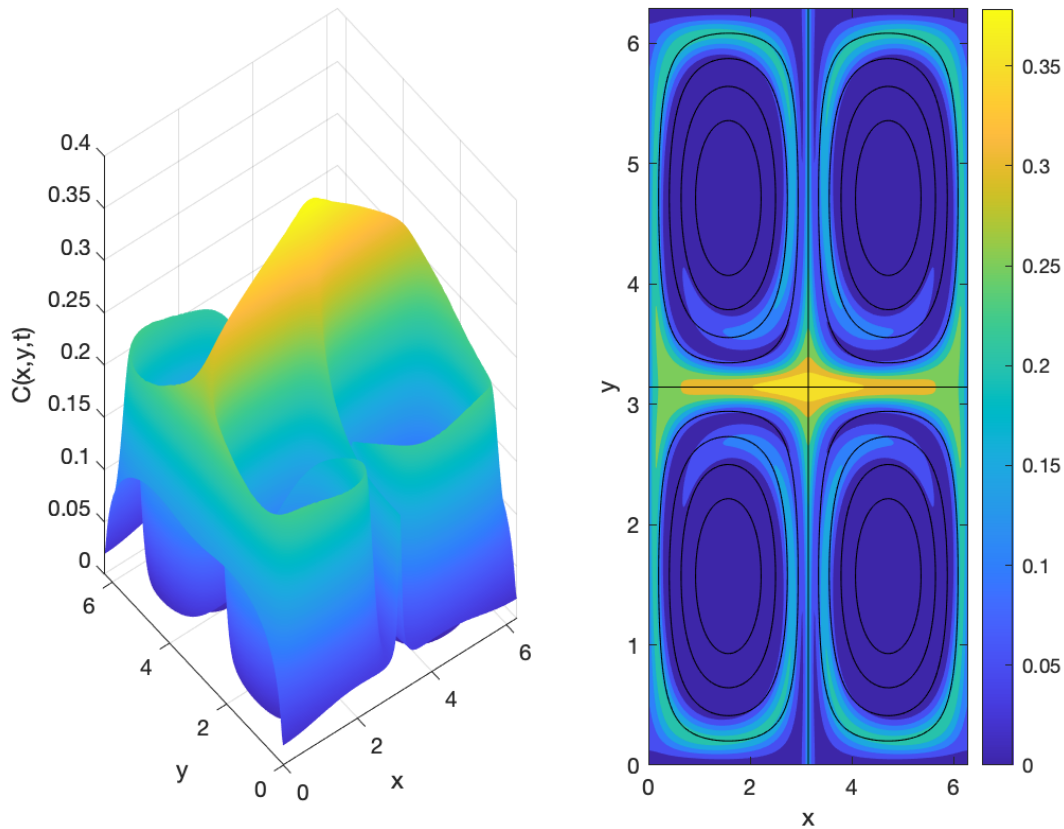


COLORADO SCHOOL OF MINES

# MATLAB Training Module

An intro to MATLAB with tips from a grad student.

$C(x, y, 20)$  , with  $\beta = 1$ ,  $D = 0.001$



DAVE MONTGOMERY

FALL 2020

## Welcome!

The purpose of this training module is to introduce students to MATLAB and provide resources for success in their studies at Mines. This module was written by a Mines grad student and contains tips for how to turn in homework assignments as well as tools for speeding up linear algebra computations in MATLAB. The contents of this tutorial can be downloaded from the following Dropbox link: <https://www.dropbox.com/sh/bep5gdxk2f1oeny/AAA-QkH-uT1lf4C0msm08octa?dl=0>

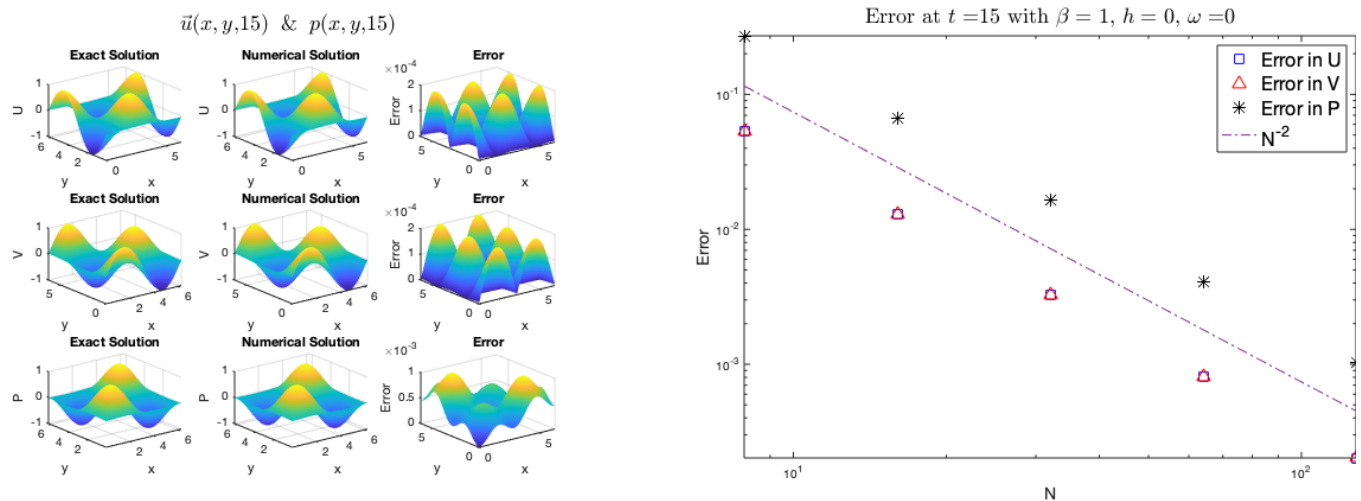


Figure 0.1: Error analysis of a finite volume method written in MATLAB that solves the 2D Navier-Stokes equations.

## Learning Objectives

1. Students will be able to implement codes for scientific computing in MATLAB.
2. Students will be able to analyze errors and debug codes in MATLAB.
3. Students will be able to turn in MATLAB codes for assignments in a professional manner.

## Additional Materials for MATLAB and Simulink Training

Throughout this module we will utilize materials from MATLAB's Onramp tutorial.

**Onramp:** Learn the essentials of MATLAB through this free, two-hour introductory tutorial on commonly used features and workflows. <https://www.mathworks.com/learn/tutorials/matlab-onramp.html>

At this time, please visit the Onramp MATLAB page and login to your Mathworks account. We will be linking you to specific sections of this tutorial throughout the following sections.

# Contents

<b>1</b>	<b>Getting Started: Basics of MATLAB</b>	<b>3</b>
1.1	Installing MATLAB . . . . .	3
1.2	Getting to know the MATLAB Editor . . . . .	3
1.2.1	Working in the Command Window . . . . .	4
1.2.2	Command Window Shortcuts: Clearing the Command Window and Stored Variables . . . . .	4
1.2.3	Running Scripts . . . . .	5
1.3	Arrays: Matrices and Vectors . . . . .	7
1.3.1	Manually Entering Arrays . . . . .	7
1.3.2	Creating Evenly-Spaced Vectors . . . . .	7
1.3.3	Array Creation Functions . . . . .	8
1.3.4	Indexing Arrays . . . . .	8
1.3.5	Extracting Multiple Elements . . . . .	8
1.3.6	Changing Values in Arrays . . . . .	9
1.3.7	Matrix and Vector Operations . . . . .	10
1.4	Built-in Functions . . . . .	11
1.4.1	Mathematical Functions . . . . .	12
1.4.2	Linear Algebra . . . . .	12
1.4.3	Timing Computations . . . . .	13
1.5	Loops . . . . .	13
1.5.1	For Loops . . . . .	13
1.5.2	While Loops . . . . .	14
1.5.3	Nested Loops . . . . .	14
1.6	If, Elseif, Else Statements . . . . .	16
<b>2</b>	<b>Work Flow</b>	<b>17</b>
2.1	Functions . . . . .	18
2.1.1	Function Files . . . . .	18
2.1.2	Function Handles . . . . .	19
2.2	Saving Data . . . . .	19
<b>3</b>	<b>Plotting</b>	<b>20</b>
3.1	2D Plots . . . . .	20
3.2	3D Figures . . . . .	22
3.3	Subplot . . . . .	23
3.4	Figure Editor . . . . .	23
<b>4</b>	<b>Debugging</b>	<b>26</b>
4.1	Errors and Breakpoints . . . . .	26
<b>5</b>	<b>How to Turn in MATLAB Code for Assignments</b>	<b>28</b>
5.1	Publish . . . . .	28
5.2	Listings in L <sup>A</sup> T <sub>E</sub> X . . . . .	29
<b>6</b>	<b>Extra's from a Grad Student</b>	<b>30</b>
6.1	Color Schemes . . . . .	30
6.2	Sparse Matrices . . . . .	30
6.3	Cell Arrays . . . . .	31
6.4	Symbolic Computing . . . . .	32
<b>7</b>	<b>Coding Projects</b>	<b>33</b>
7.1	Real Roots of a Quadratic . . . . .	33
7.2	Self Portrait . . . . .	33
7.3	Stellar Motion - MATLAB Onramp . . . . .	33

# 1 Getting Started: Basics of MATLAB

In order to solve hardcore math problems, we will start with some MATLAB basics. Topics in this section include installing MATLAB, getting to know the MATLAB editor, datatypes, built-in functions, loops, and logical statements.

## 1.1 Installing MATLAB

MATLAB is available for *free* to the Mines community, courtesy of Tech Fee. To install MATLAB on your machine visit:

<https://www.mathworks.com/academia/tah-portal/colorado-school-of-mines-40579029.html>

You will be prompted to create a MATLAB account; make sure to use your university email address. This download may take a while depending on your internet speed.

The Colorado School of Mines has access to many other software titles. That software is available to the Mines community in accordance with the license agreement for each package. To see the ITS Software Index visit <https://its.mines.edu/software/>.

## 1.2 Getting to know the MATLAB Editor

If you haven't already, please open MATLAB on your desktop and visit the Onramp MATLAB page. <https://www.mathworks.com/learn/tutorials/matlab-onramp.html>. We will be linking you to specific sections of this tutorial throughout the following sections.

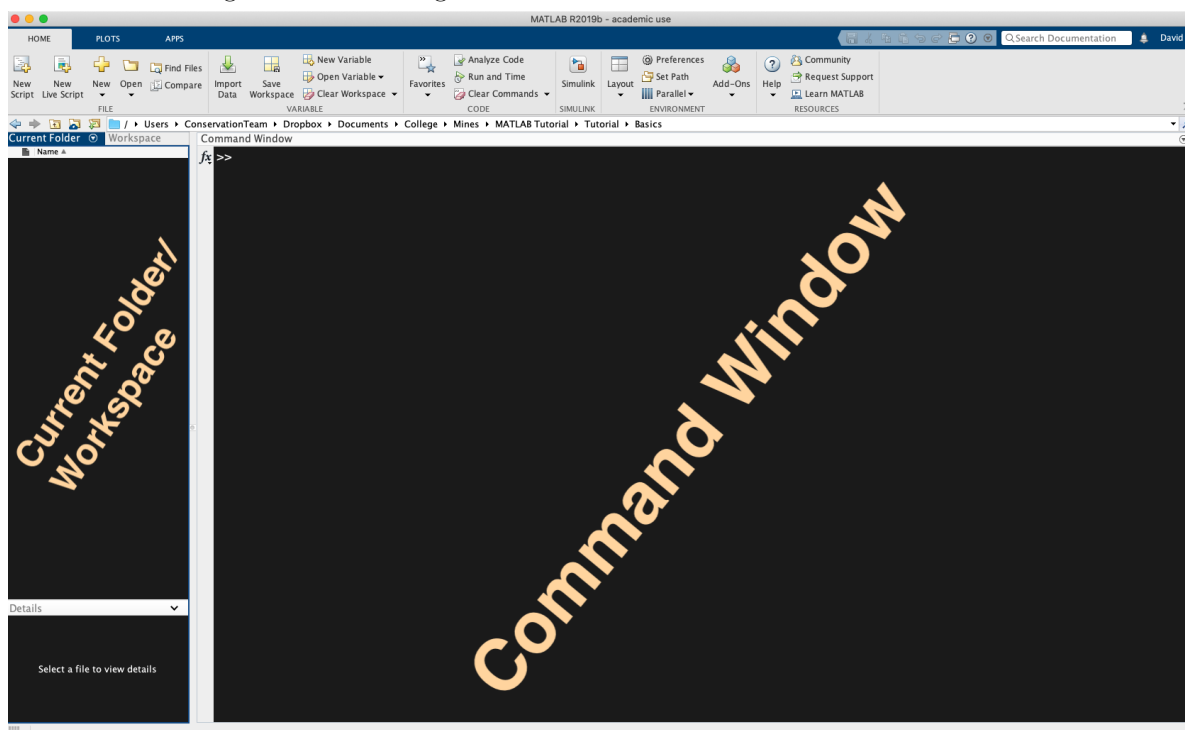


Figure 1.1: The default MATLAB editor. Note that I have changed the background and text colors to be easier on my aging eyes. For details on how to do this, see the “Color Schemes” subsection in the Extra’s section.

**Command Window:** The command window is where you will see outputs from your codes, but it is also a great tool for checking syntax, making quick calculations, and debugging.

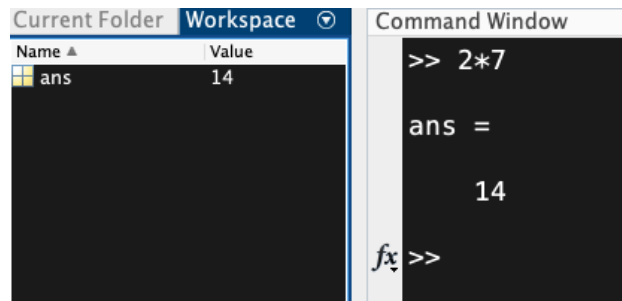
**Current Folder:** This tab will display the contents of the current folder that you are working in.

**Workspace:** This tab will display all of the current variables being used.

### 1.2.1 Working in the Command Window

Let's try out some basic commands/calculations in the Command Window.

1. Click anywhere in the Command Window to get your cursor next to the ">>".
2. Let's see if MATLAB can handle multiplying 2 and 7. Type `2*7` in the Command Window and press enter. You just created your first MATLAB variable and you didn't even get to name it...



By default, MATLAB will store calculations in the Command Window as the variable "ans". Note that in the Workspace tab, there is now a variable named "ans" whose value is 14.

Now that we've seen how this works on our desktops, let's go to the MATLAB Onramp tutorial and try out some other commands. Task 1 should be similar to what we just did.

**Onramp Section 2.1 Entering Commands:** [https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s\\_tid=course\\_mlor\\_start1%23chapter=2&lesson=1&section=1#chapter=2&lesson=1&section=1](https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s_tid=course_mlor_start1%23chapter=2&lesson=1&section=1#chapter=2&lesson=1&section=1)

#### Key Takeaways: Entering Commands

- By default MATLAB stores calculations in the Command Window in the variable "ans".
- Using the up-arrow on your keyboard allows you to recall previous commands. This will save you a lot of time.
- Tasks in the Command Window are performed in serial. If the variable `y` was defined as `y = m/2`. Then reassigning `m = 18` does not change the variable `y` as it was defined prior to changing `m`.
- You can name variables anything you want as long as they start with a letter, and don't contain a "space". For example, `dave_is_wonderful2020` is a perfectly fine variable.

### 1.2.2 Command Window Shortcuts: Clearing the Command Window and Stored Variables

Let's return to our desktop editor and try out some shortcuts for clearing variables and the contents of the command window.

1. Define `M = pascal(11)` and press enter. This should fill your command window with an  $11 \times 11$  Pascal matrix.
2. For more info on the pascal function, type "`help pascal`" into the Command Window and press enter.
3. Next, type the word, "`why`", into the the Command Window and press enter.
4. Repeat the previous step.
5. Repeat the previous step.
6. Repeat until it is no longer funny...

7. Okay, okay I'll get back to the tutorial now. We've successfully filled the Command Window, but now I want to get rid of everything. Type "clc" in the Command Window and press enter.

*This clears the Command Window, but does not get rid of the stored variables. Note that M and "ans" still appear in the Workspace.*

8. Let's make sure M is still an  $11 \times 11$  Pascal matrix. Type M in the Command Window and press enter.
9. Maybe you don't like how M appears in the Command Window. Double-click on M in the Workspace tab. This enables you to see the matrix in a tabular setting.
10. Close the variables-M tab by clicking the "x" on the far right.
11. Clear the contents of the Command Window by entering "clc".
12. Clear the stored variables by entering "clear".
13. Just to be sure this worked, type M in the Command Window and press enter.

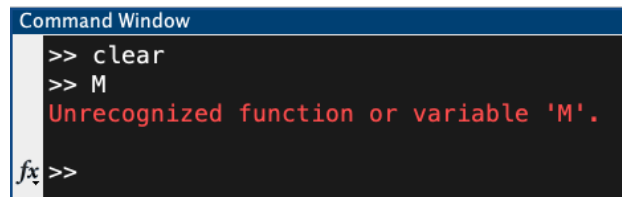


Figure 1.1: Your first error message :(

### Key Takeaways: Command Window Shortcuts

- "help" is a great tool for looking up MATLAB functions.
- "why" is a good way to get a laugh when your codes are making you sad.
- "clc" clears the Command Window but does not clear variables.
- "clear" clears variables but does not clear the Command Window.

### 1.2.3 Running Scripts

Working in the Command Window is great and all, but for more complicated tasks we will want to write scripts. In this section we will create our first .m file and run the script.

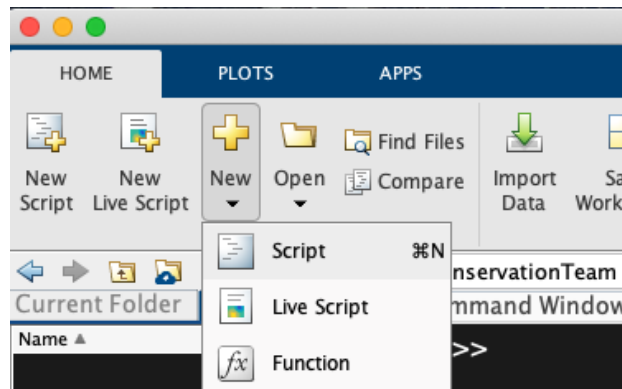


Figure 1.2: Creating a new script.

1. In the upper left corner of the Home tab, click the large + with the word New underneath it. Select the first option in the drop-down menu, “Script”. This will open up an empty .m file in a window above the Command Window. This will also toggle you away from the Home tab to a new tab called Editor.
2. Save this file as “**firstScript.m**” in an empty folder on your computer. This folder is now the Current Folder that MATLAB will access.
3. In the first line of **firstScript.m**, type **clear**. This ensures we start with a clean slate of variables.
4. In the second line of **firstScript.m**, define  $\mathbf{v} = (0 : 1 : 5)'$ ;  
*This creates a column vector  $\vec{v} = [0 \ 1 \ 2 \ 3 \ 4 \ 5]^T$ . Note that the ' is MATLAB syntax for transpose. We'll talk more about arrays later.*
5. In the third line, define  $\mathbf{w} = \text{flipud}(\mathbf{v})$ . Since we did not include a semicolon, MATLAB highlights the equals sign in red to remind us that the script will print  $\vec{w}$  each time we run it. Ignore this for now.  
*The function `flipud` flips a given vector upside-down and sets  $\vec{w} = [5 \ 4 \ 3 \ 2 \ 1 \ 0]^T$ . There is also a `fliplr` function for row vectors.*
6. Save the script, then run the script using one of the following options.
  - (a) Press the Run button in the editor tab.
  - (b) Type the file name “**firstScript**” into the Command Window and press enter.

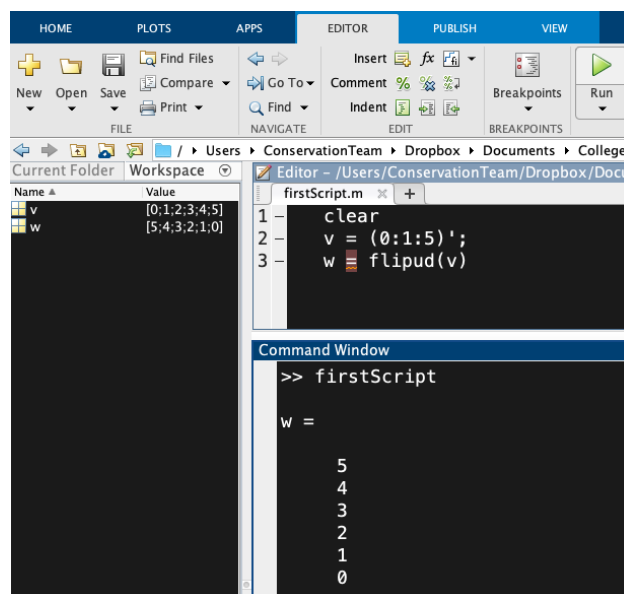


Figure 1.2: Running your first script

Regardless of which method you used, you will see an output for  $\vec{w}$  in the Command Window.

### Key Takeaways: Running Scripts

- MATLAB will highlight equals signs red if a semicolon is not used (there are other reasons too).
- To run a script, save the script then either press Run or type the file name into the Command Window.

### 1.3 Arrays: Matrices and Vectors

In MATLAB, a numeric data type is an array of integers or floating-point data. Matrices and arrays are the fundamental representation of information in MATLAB. Even a scalar is defined as a  $1 \times 1$  array. In this section we will introduce several techniques for creating arrays and accessing information within an array.

#### 1.3.1 Manually Entering Arrays

Let's get started with entering vectors and matrices manually by walking through the associated MATLAB Onramp tutorial.

**Onramp Section 4.1 Manually Entering Arrays:** [https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s\\_tid=course\\_mlor\\_start1#chapter=4&lesson=1&section=1](https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s_tid=course_mlor_start1#chapter=4&lesson=1&section=1)

#### Key Takeaways: Manually Entering Arrays

- Use the square brackets `[]` to define vectors and matrices.
- Using a semicolon within square brackets indicates a new row. For example

$$[1\ 2\ 3\ 4; 5\ 6\ 7\ 8] = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, \quad \text{and} \quad [1; 2; 3] = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

#### 1.3.2 Creating Evenly-Spaced Vectors

Often times we need to create long, evenly spaced vectors. For example time discretizations. This is easily done in MATLAB using the colon operator or linspace function. The next Onramp section provides a great introduction to creating evenly spaced vectors in MATLAB.

**Onramp Section 4.2 Creating Evenly-Spaced Vectors:** [https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s\\_tid=course\\_mlor\\_start1#chapter=4&lesson=2&section=1](https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s_tid=course_mlor_start1#chapter=4&lesson=2&section=1)

#### Key Takeaways: Creating Evenly Spaced Vectors

- `x = a : Δx : b` creates a row vector from  $a$  to  $b$  with step size  $\Delta x$ .  
*Note: the colon operator does not guarantee the vector ends at  $b$ . For example `x = 1 : 3 : 9` produces  $\vec{x} = [1\ 4\ 7]$ . The choice of  $\Delta x$  determines the last entry.*
- The `linspace` function produces an evenly spaced row vector with  $N$  elements ranging from  $a$  to  $b$ . Syntax: `x = linspace(a, b, N)`.
- To create evenly spaced column vectors, use the transpose operator `'`. For example `x = (a : Δx : b)'` or `x = (linspace(a, b, N))'`.



### 1.3.3 Array Creation Functions

We have already seen the pascal function, which creates a Pascal matrix of a given size. There are numerous other functions for creating matrices of a given size in MATLAB.

#### Key Takeaways: Array Creation Functions

- **Zeros:** Great for initializing matrices. The zeros function creates an  $m \times n$  matrix of zeros. Syntax: `A = zeros(m,n)`.
- **Ones:** The ones function creates an  $m \times n$  matrix of ones. Syntax: `A = ones(m,n)`.
- **Eye:** The eye function creates an  $m \times n$  identity matrix. Syntax: `A = eye(m,n)`.
- **Rand:** The rand function creates an  $m \times n$  matrix with random entries ranging between 0 and 1. For more info type “help rand” in the Command Window.

### 1.3.4 Indexing Arrays

Indexing for arrays starts at 1. Unlike other languages, there is no  $0^{th}$  element in MATLAB arrays. So given an  $m \times n$  matrix **A**, such that

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix},$$

`A(i,j)` is the syntax for retrieving the element in the  $i^{th}$  row and  $j^{th}$  column,  $a_{i,j}$ .

When working with row and column vectors, MATLAB does not *require* two indices, users only need to supply the element number. For example, say we have

$$\vec{\mathbf{u}} = [1 \quad 2 \quad 3], \quad \text{and} \quad \vec{\mathbf{v}} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

Then,  $\vec{\mathbf{u}}(2) = 2$  and  $\vec{\mathbf{v}}(2) = 5$ . However, a user may also use row and column indexing to retrieve elements in a vector. For instance,  $\vec{\mathbf{u}}(1,2) = \vec{\mathbf{u}}(2) = 2$  and  $\vec{\mathbf{v}}(2,1) = \vec{\mathbf{v}}(2) = 5$ .

#### Key Takeaways: Indexing Arrays

- `A(i,j)` calls the element in the  $i^{th}$  row and  $j^{th}$  column of an array.
- Elements in row/column vectors can be retrieved using the element number, or  $i,j$  indexing.
- “end” can be used to extract the last entry. For example `A(end,end)` returns the last element in the array.

### 1.3.5 Extracting Multiple Elements

Extracting multiple elements from an array is easy with the colon operator (`:`). Let’s return to MATLAB Onramp for some practice.

**Onramp Section 5.2 Extracting Multiple Elements:** [https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s\\_tid=course\\_mlor\\_start1#chapter=5&lesson=2&section=1](https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s_tid=course_mlor_start1#chapter=5&lesson=2&section=1)

For the following *Key Takeaways*, assume  $\mathbf{A} \in \mathbb{R}^{m \times n}$ , such that

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix}.$$

### Key Takeaways: Extracting Multiple Elements

- The colon operator ( $:$ ) is used to specify a range of values for the  $i$  and/or  $j$  index, or to indicate all values.
- Example: Extracting the  $j^{th}$  column of  $\mathbf{A}$ . We use ( $:$ ) in the  $i^{th}$  position of  $\mathbf{A}(\mathbf{i}, \mathbf{j})$  to specify that we want all the elements in the  $j^{th}$  column.

$$\vec{\mathbf{a}}_j = \mathbf{A}(:, j) = \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ \vdots \\ a_{m,j} \end{bmatrix}$$

- Example: Extracting the  $i^{th}$  row of  $\mathbf{A}$ . We use ( $:$ ) in the  $j^{th}$  position of  $\mathbf{A}(\mathbf{i}, \mathbf{j})$  to specify that we want all the elements in the  $i^{th}$  row.

$$\mathbf{A}(\mathbf{i}, :) = [a_{i,1} \quad a_{i,2} \quad \dots \quad a_{i,n}]$$

- Example: Removing the left and right columns, as well as the bottom and top rows of  $\mathbf{A}$ .

$$\mathbf{A}(2 : \text{end} - 1, 2 : \text{end} - 1) = \mathbf{A}(2 : m - 1, 2 : n - 1) = \begin{bmatrix} a_{2,2} & \dots & a_{2,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,2} & \dots & a_{m-1,n-1} \end{bmatrix},$$

### 1.3.6 Changing Values in Arrays

Elements in an array can be altered by combining indexing with assignment.

For the following *Key Takeaways*, assume  $\mathbf{A} \in \mathbb{R}^{m \times n}$ , such that

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix}.$$

**Key Takeaways: Changing Values in Arrays**

- Example: Replace the first super diagonal entry,  $a_{1,2}$  with  $\pi$ .

$$A(1,2) = \pi, \quad \Rightarrow \quad \mathbf{A} = \begin{bmatrix} a_{1,1} & \pi & \dots & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix}$$

- Example: Replace the the last  $m - 1$  entries of the first column of  $\mathbf{A}$  with  $[2 \ 3 \ \dots \ m]^T$

$$A(2:m, :) = (2:1:m)', \quad \Rightarrow \quad \mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & \dots & a_{1,n} \\ 2 & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ m & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix}$$

**1.3.7 Matrix and Vector Operations**

Operations with vectors and matrices is easily done in MATLAB with minimal syntax. To get used to the syntax let's go back to Onramp.

**Onramp Section 6.1 Performing Array Operations on Vectors:** [https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s\\_tid=course\\_mlor\\_start1#chapter=6&lesson=1&section=1](https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted&s_tid=course_mlor_start1#chapter=6&lesson=1&section=1)

For the following *Key Takeaways*, assume  $m \neq n$ , and that  $\mathbf{A} \in \mathbb{R}^{m \times n}$ ,  $\vec{\mathbf{x}} \in \mathbb{R}^m$ ,  $\vec{\mathbf{y}} \in \mathbb{R}^m$ ,  $\vec{\mathbf{u}} \in \mathbb{R}^n$  such that

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & \dots & a_{2,n} \\ \vdots & & \ddots & & \vdots \\ a_{m,1} & a_{m,2} & \dots & \dots & a_{m,n} \end{bmatrix}, \quad \vec{\mathbf{x}} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad \vec{\mathbf{y}} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}, \quad \vec{\mathbf{u}} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}.$$

**Key Takeaways: Matrix and Vector Operations**

- **Adding a scalar to each entry:** Given  $c \in \mathbb{R}$ ,

$$\mathbf{A} + c = \begin{bmatrix} a_{1,1} + c & a_{1,2} + c & \dots & \dots & a_{1,n} + c \\ a_{2,1} + c & a_{2,2} + c & \dots & \dots & a_{2,n} + c \\ \vdots & & \ddots & & \vdots \\ a_{m,1} + c & a_{m,2} + c & \dots & \dots & a_{m,n} + c \end{bmatrix}, \quad \mathbf{x} + c = \begin{bmatrix} x_1 + c \\ x_2 + c \\ \vdots \\ x_m + c \end{bmatrix}.$$

- **Multiplying by a scalar:** Given  $c \in \mathbb{R}$ ,

$$c * \mathbf{A} = \mathbf{A} * c = \begin{bmatrix} c a_{1,1} & c a_{1,2} & \dots & \dots & c a_{1,n} \\ c a_{2,1} & c a_{2,2} & \dots & \dots & c a_{2,n} \\ \vdots & & \ddots & & \vdots \\ c a_{m,1} & c a_{m,2} & \dots & \dots & c a_{m,n} \end{bmatrix}, \quad c * \mathbf{x} = \mathbf{x} * c = \begin{bmatrix} c x_1 \\ c x_2 \\ \vdots \\ c x_m \end{bmatrix}.$$

- **Function evaluations of each entry:** Example: Find  $\sin(a_{i,j})$  for  $i = 1, \dots, m$ ,  $j = 1, \dots, n$ .

$$\sin(\mathbf{A}) = \begin{bmatrix} \sin a_{1,1} & \sin a_{1,2} & \dots & \dots & \sin a_{1,n} \\ \sin a_{2,1} & \sin a_{2,2} & \dots & \dots & \sin a_{2,n} \\ \vdots & & \ddots & & \vdots \\ \sin a_{m,1} & \sin a_{m,2} & \dots & \dots & \sin a_{m,n} \end{bmatrix}$$

- **Dot operator:** The dot operator is a shortcut for element-wise operations. Generally users use the dot operator with multiplication, division and exponentiation.

Example: Square each entry of  $\vec{\mathbf{x}}$ .

$$\mathbf{x} . ^ 2 = \begin{bmatrix} x_1^2 \\ x_2^2 \\ \vdots \\ x_m^2 \end{bmatrix}.$$

Example: Let  $\vec{\mathbf{v}} \in \mathbb{R}^m$  such that  $v_i = x_i y_i$  for  $i = 1, \dots, m$ .

$$\vec{\mathbf{v}} = \mathbf{x} . * \mathbf{y} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_m y_m \end{bmatrix}.$$

Example: Assume  $x_i \neq 0$  for  $i = 1, \dots, m$ . Let  $\mathbf{B} \in \mathbb{R}^{m \times n}$  such that  $b_{i,j} = a_{i,j}/x_i$  for  $i = 1, \dots, m$ . That is, divide each entry of  $\mathbf{A}$  by the corresponding row entry of  $\vec{\mathbf{x}}$ .

$$\mathbf{B} = \mathbf{A} ./ \mathbf{x} = \begin{bmatrix} a_{1,1}/x_1 & a_{1,2}/x_1 & \dots & \dots & a_{1,n}/x_1 \\ a_{2,1}/x_2 & a_{2,2}/x_2 & \dots & \dots & a_{2,n}/x_2 \\ \vdots & & \ddots & & \vdots \\ a_{m,1}/x_m & a_{m,2}/x_m & \dots & \dots & a_{m,n}/x_m \end{bmatrix}.$$

- **Matrix/Matrix-vector multiplication:** Provided matrices and vectors are of proper dimensions, then matrix multiplication and matrix-vector multiplication is done using the multiplication operator  $*$ .

Example: Multiply  $\mathbf{A}^T$  by  $\mathbf{A}$ .

$$\mathbf{A}^T \mathbf{A} = \mathbf{A}' * \mathbf{A}$$

Example: Multiply  $\mathbf{A}$  by  $\vec{\mathbf{u}}$ .

$$\mathbf{A} \vec{\mathbf{u}} = \mathbf{A} * \mathbf{u}$$

Example: Find the standard inner product of  $\vec{\mathbf{x}}$  and  $\vec{\mathbf{y}}$ .

$$\langle \vec{\mathbf{x}}, \vec{\mathbf{y}} \rangle_2 = \vec{\mathbf{x}}^T \vec{\mathbf{y}} = \mathbf{x}' * \mathbf{y}$$

## 1.4 Built-in Functions

MATLAB has many built-in functions such as trig functions, root finding functions, PDE solvers, and matrix factorizations. In this subsection we introduce some of the most common functions used in your coursework in AMS at Mines.

### 1.4.1 Mathematical Functions

A complete list of mathematical functions in MATLAB can be found at

<https://www.mathworks.com/help/symbolic/mathematical-functions.html>

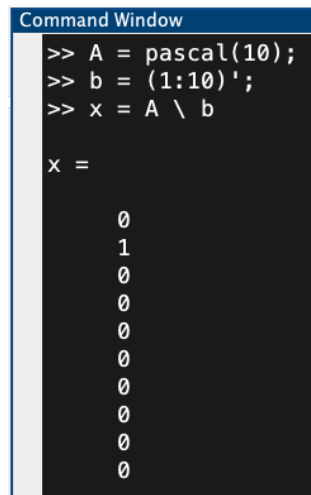
### 1.4.2 Linear Algebra

In the computational linear algebra course you will learn and code several algorithms for solving linear systems, finding eigenvalues/vectors, and computing matrix decomposition's. However, there are a handful of built in functions in MATLAB that are superior in accuracy and efficiency.

**Solve  $\mathbf{A}\vec{x} = \vec{b}$ :** The most efficient way to solve a linear system in MATLAB is to use the backslash operator (`\`). Backslash is a very robust solver that will determine whether a system should be solved directly, or with an iterative solver. Additionally, the backslash command works with sparse matrices. We will talk more about solving sparse systems in the *Extra's from a Grad Student* section on *Factorize*. For more info on the backslash command see

<https://www.mathworks.com/help/matlab/ref/mldivide.html>.

Example: Solve  $\mathbf{A}\vec{x} = \vec{b}$ , where  $\mathbf{A} \in \mathbb{R}^{10 \times 10}$  is a Pascal matrix, and  $\vec{b} \in \mathbb{R}^{10}$  such that  $b_i = i$ , for  $i = 1, 2, \dots, 10$ .



```
Command Window
>> A = pascal(10);
>> b = (1:10)';
>> x = A \ b

x =

     0
     1
     0
     0
     0
     0
     0
     0
     0
     0
```

Figure 1.3: Solving  $\mathbf{A}\vec{x} = \vec{b}$  using the backslash operator.

**Matrix Factorizations:** MATLAB has some great built in functions for generating different factorizations of a given matrix  $\mathbf{A}$ . The most commonly used factorizations are *LU*, *QR*, Cholesky, and Singular Value Decomposition (SVD). For more information on built in factorizations see

<https://www.mathworks.com/help/matlab/math/factorizations.html>.

**Eigenvalues and Eigenvectors:** The `eig` function will return the eigenvalues and eigenvectors of a matrix. For more information and syntax see

<https://www.mathworks.com/help/matlab/ref/eig.html>.

**Test Matrices:** The `gallery` function is a useful tool for generating test matrices. For more information see

<https://www.mathworks.com/help/matlab/ref/gallery.html>.

Additionally, there is an example in the *Timing Computations* sections that uses the `gallery` function.

### 1.4.3 Timing Computations

`Tic` and `toc` are used to time sections of code. This is typically done by typing `tic` above a computation to start the timer, and `toc` after the computation to stop the timer. You can store the time by assigning a variable equal to `toc`.

Example: Time how long it takes to solve  $\mathbf{A}\vec{x} = \vec{b}$ , where  $\mathbf{A} \in \mathbb{R}^{1000 \times 1000}$  tridiagonal matrices obtained using the `gallery` function. Observe that times will vary on each machine.

```

1 - A = gallery('tridiag',1000);
2 - b = A * ones(1000,1);
3
4 - tic
5 - x = A \ b;
6 - time_sparseA = toc
7
8 - tic
9 - x = full(A) \ b;
10 - time_fullA = toc

```

(a) Script for timing.

```

Command Window
>> ticToc

time_sparseA =

    7.7722e-05

time_fullA =

    0.0298

```

(b) Results from a 2012 macbook pro.

*Note:* `gallery('tridiag',1000)` returns a sparse tridiagonal matrix of size  $1000 \times 1000$  with subdiagonal entries -1, diagonal entries 2, and superdiagonal entries -1. You will see this matrix again in your numerical solutions to partial differential equations course. `full(A)` turns a sparse matrix  $\mathbf{A}$  into a full sized matrix. We'll talk more about sparse matrices in the "Extra's from a Grad Student" section

## 1.5 Loops

Loop control statements are used to repeatedly execute a block of code. There are two types of loops in MATLAB: `for` loops and `while` loops.

[https://www.mathworks.com/help/matlab/matlab\\_prog/loop-control-statements.html](https://www.mathworks.com/help/matlab/matlab_prog/loop-control-statements.html)

### Key Takeaways: Loops

- For loops stop after a specified number of iterations.
- While loops continue looping as long as a condition is true.
- If you accidentally create an infinite loop, type `Ctrl+C` to stop execution.
- It is customary to indent the contents of a loop for readability.

#### 1.5.1 For Loops

For statements are used to loop through a block of code for a specified number of times. They are often used in computational linear algebra and when solving PDE's as you typically know when you need to stop the loop.

Example: Use a `for` loop to fill a vector with the first 7 Fibonacci numbers. Recall that the Fibonacci sequence is

$$F_n = F_{n-1} + F_{n-2}, \text{ for } n > 1,$$

$$\text{with } F_0 = 0, F_1 = 1.$$

*Note:* MATLAB does not have a  $0^{th}$  array element, so you will need to shift the indexing by one.

```

1 F = zeros(1,7); % Initialize vector to store F_n
2 F(2) = 1; % Set second entry of F
3
4 for n = 3:7
5     F(n) = F(n-1) + F(n-2);
6 end
7
8 F % displays the vector F

```

Command Window

```

>> fibonacciSeq
F =
    0    1    1    2    3    5    8

```

Figure 1.5: Fibonacci numbers using a for loop.

### 1.5.2 While Loops

While statements are used to loop through a block of code as long as a condition is true. They are often used in root finding problems as you don't usually know exactly how many iterations are necessary for a method to converge.

Example: Use a while loop to determine how many iterations it takes Newton's method to converge to the positive root of  $f(x) = x^2 - \pi$ , within a tolerance of  $10^{-10}$ . That is, find  $n$  such that  $|x_n - \sqrt{\pi}| < 10^{-10}$ . Recall Newton's method is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \dots$$

We will start using an initial guess of  $x_0 = 2$ . Plugging the function into Newton's method we have

$$x_{n+1} = x_n - \frac{x_n^2 - \pi}{2x_n}, \quad x_0 = 2.$$

```

1 x0 = 2; % Initial Guess
2 n = 0; % Iteration number
3
4 while abs(x0 - sqrt(pi)) > 1e-10
5     n = n + 1; % update iteration number
6     x = x0 - (x0^2 - pi) / (2 * x0);
7     x0 = x; % update previous approximation
8 end
9
10 n

```

Command Window

```

>> newtonsWhile
n =
    4

```

Figure 1.6: Newton's method converges in 4 iterations.

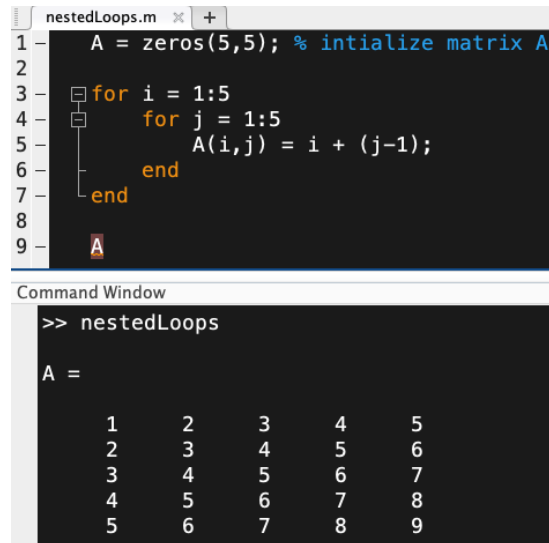
### 1.5.3 Nested Loops

A Nested loop is any block of code that has a loop inside of another loop. Nested loops are commonly used to fill the entries of a matrix. However, nested loops are slow and can often be avoided.

Example: Build the following matrix in MATLAB without manually entering each element.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}.$$

One solution is to build this matrix one row and one column at a time using 2 for loops.



The figure shows a MATLAB script named `nestedLoops.m` with the following code:

```
1 A = zeros(5,5); % initialize matrix A
2
3 for i = 1:5
4     for j = 1:5
5         A(i,j) = i + (j-1);
6     end
7 end
8
9 A
```

The Command Window shows the output of running the script:

```
>> nestedLoops

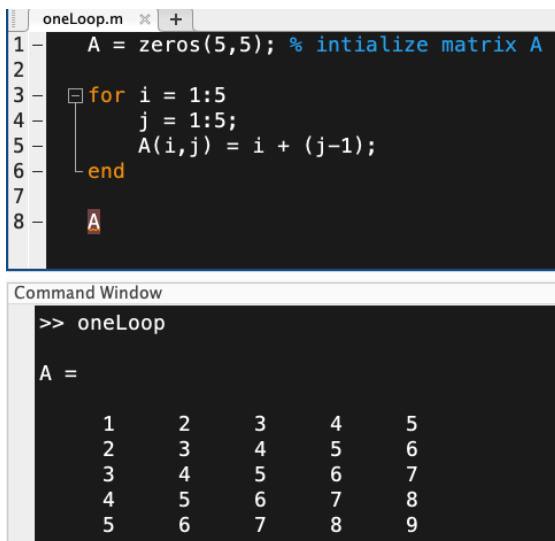
A =

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

Figure 1.7: Building  $\mathbf{A}$  using double for loops.

Another solution is to build this matrix one row at a time by adding a scalar to each entry of the vector  $\vec{j} = [1 \ 2 \ 3 \ 4 \ 5]$ . See (a) below.

The tricky solution is to “break math”. In theory you cannot add a row vector and a column vector. However, in MATLAB adding a row vector with a column vector generates a matrix. This is a great trick for generating matrices in just a few lines of code. See (b) below.



The figure shows a MATLAB script named `oneLoop.m` with the following code:

```
1 A = zeros(5,5); % initialize matrix A
2
3 for i = 1:5
4     j = 1:5;
5     A(i,j) = i + (j-1);
6 end
7
8 A
```

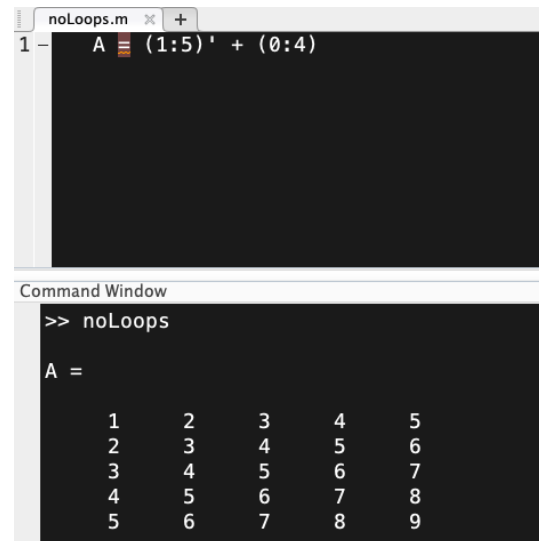
The Command Window shows the output of running the script:

```
>> oneLoop

A =

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

(a) Building  $\mathbf{A}$  using 1 for loop.



The figure shows a MATLAB script named `noLoops.m` with the following code:

```
1 A = (1:5)' + (0:4)
```

The Command Window shows the output of running the script:

```
>> noLoops

A =

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

(b) Building  $\mathbf{A}$  without loops.

Figure 1.8: Avoiding an unnecessary double for loop.



## 1.6 If, Elseif, Else Statements

`if`, `elseif`, `else` commands are used as decision branches. They can be used to make plots at specified times during a for loop, or to fill matrices. For more information see <https://www.mathworks.com/help/matlab/ref/if.html>

**Syntax:**

```

if (conditional statement)
    executable code
    :
elseif (conditional statement)
    executable code
    :
else
    executable code
    :
end

```

**Conditional Statements:** Conditional statements generally have the form:

1. If  $p$  then  $q$ .

That is, if  $p$  is true, then  $q$  is computed.

2. If not  $p$  then  $q$ .

This time if  $p$  is not true, then  $q$  is computed.

3. If  $p$  and  $q$ , then  $r$ .

Here both  $p$  and  $q$  need to be true in order for  $r$  to be computed.

4. If  $p$  or  $q$ , then  $r$ .

This time either  $p$  or  $q$  need to be true, but not necessarily both, in order for  $r$  to be computed.

MATLAB uses the following syntax to handle these types of statements.

Conditional Statement	MATLAB Syntax
if $p = 1$	if <code>p == 1</code>
if $p \neq 1$	if <code>p ~= 1</code>
if $p > 1$ and $q < 10$	if <code>(p &gt; 1) &amp;&amp; (q &lt; 10)</code>
if $p > 1$ or $q < 10$	if <code>(p &gt; 1)    (q &lt; 10)</code>

*Note: In MATLAB  $0 \equiv \text{false}$ , and  $1 \equiv \text{true}$ . Therefore the following two statements are equivalent:*

if <code>p == 1</code>	if <code>p</code>
if <code>p ~= 1</code>	if <code>~p</code>

## 2 Work Flow

As the problems you work on become more complicated, your codes will too. Much like your kitchen at home, codes work best, and are easier to use when they are well organized. One approach for organizing codes is to use a single *run file* with editable parameters, then call on subroutines for computations.

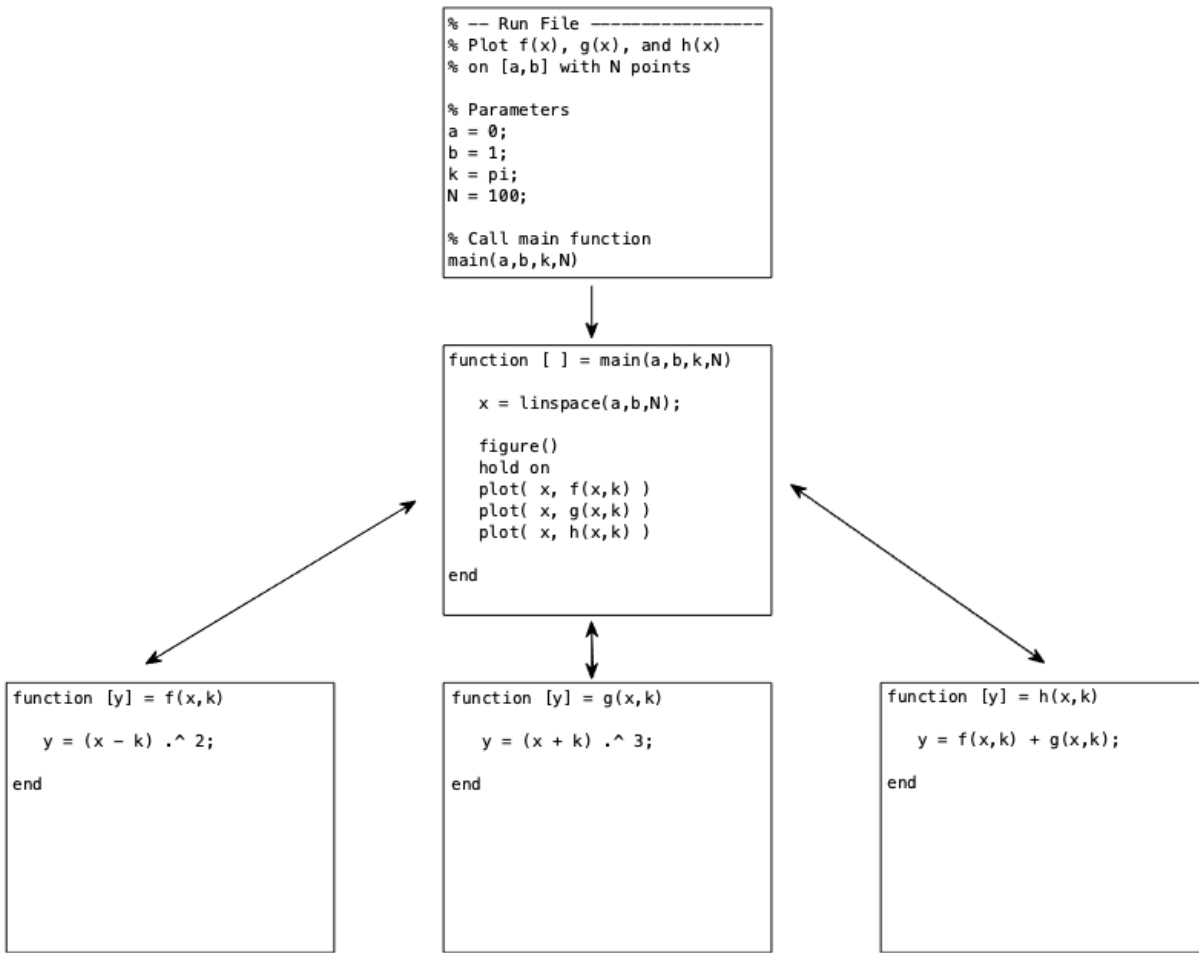


Figure 2.1: Flow chart of how a run file can be used to edit parameters that get sent to several subroutines. (*Note: Flow chart was made in MATLAB.*)

The above code illustrates how all inputs are edited in the run file then handled by several routines to accomplish a task. Observe that the `main.m` file is used to organize the computations. There are several benefits to using subroutines to piecemeal your code: they increase readability; they can easily be employed by other codes; they make debugging easier. Lastly, since variables are defined locally, you don't have to be as creative with naming your variables. Note that we use the variable  $y$  in the three lowest subroutines without conflict.

## 2.1 Functions

There are two types of user created functions in MATLAB: Function files, and function handles.

### 2.1.1 Function Files

Function files are .m files that take inputs and return outputs. For more information visit <https://www.mathworks.com/help/matlab/ref/function.html>.

To create a new function file, simply open a new script and use the following syntax in the first line:

```
function [y1,...,yM] = functionName(x1,...,xN)
```

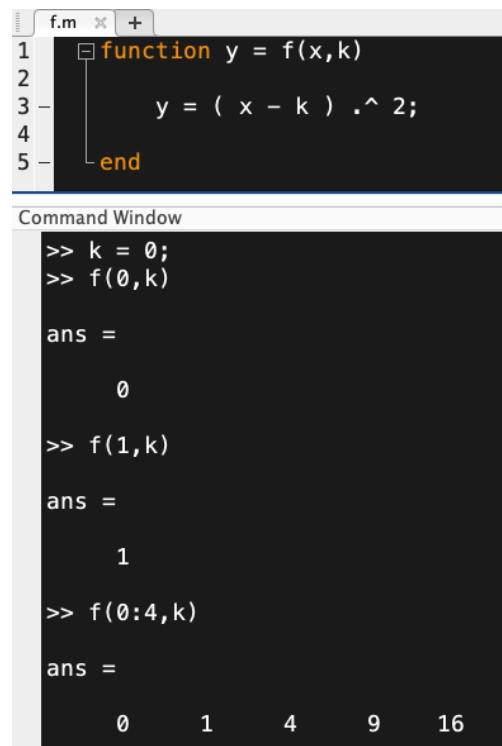
- [y1,...,yM] are the outputs.
- functionName is the name of your function.
  - Avoid using names like **zeros**, **poly**, **roots**, which are built-in MATLAB functions. When in doubt, type **help functionName** into the command line to see if the name is already used by MATLAB.
  - When you save the file, the function name must match the file name, e.g. **functionName.m**
- (x1,...,xN) are the inputs for the function.

When a function is saved in the same folder as a run file, use the following syntax to call the function:

```
[y1,...,yM] = functionName(x1,...,xN);
```

#### Example:

- Create a function file called **f.m** that computes  $f(x) = (x - k)^2$  where  $k \in \mathbb{R}$  is a given parameter  $k$ .
- Call your function using the command line and test it for different  $x$  and  $k$ .



```
f.m
1 function y = f(x,k)
2
3     y = ( x - k ) .^ 2;
4
5 end
```

```
Command Window
>> k = 0;
>> f(0,k)

ans =

     0

>> f(1,k)

ans =

     1

>> f(0:4,k)

ans =

     0     1     4     9    16
```

Figure 2.2:  $f(x)$  being evaluated with  $k = 0$ , and various  $x$  values.

### 2.1.2 Function Handles

A function handle is a MATLAB data type that represents a function. A typical use of function handles is to pass a function to another function. For example, if you are creating a code that solves

$$\frac{dy}{dt} = f(t)$$

for  $y(t)$  given the function  $f(t)$ . You may want to pass in different  $f$  into your general solver. This can be done by creating a function handle for the function  $f$  in a run file that gets passed to the solver.

To create a function handle, we use the `@` operator. For example

```
f = @(x,k) (x - k) .^ 2;
```

Here we use the “.” operator so that the function handle can operate on scalars and arrays. To illustrate how to pass function handles, we can recreate the functions as in the flow chart above and pass them to the main function.

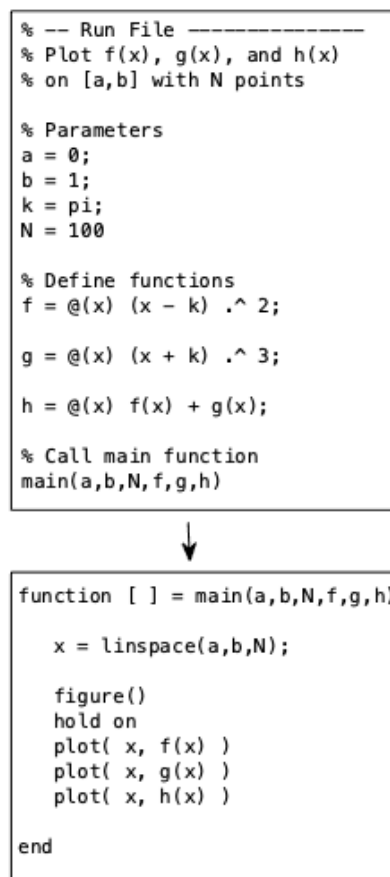


Figure 2.3: Creating and passing function handles. Note that  $k$  is defined in the run file where the function handles are defined.

## 2.2 Saving Data

If your code takes a long time to run, it is in your best interest to save the data so you can work with it later. When you save data in MATLAB, you will save it as a `.mat` file. You can save an entire work space, or specific variables.

Saving Data: <https://www.mathworks.com/help/matlab/ref/save.html>

Loading Data: <https://www.mathworks.com/help/matlab/ref/load.html>

### 3 Plotting

There are many ways to visualize data in MATLAB. In our program the most commonly used plotting functions are `plot`, `surf`, and `contour`.

**Saving Figures:** Once you have created a figure in MATLAB, I highly recommend saving the figure as a `.fig` file first. This will allow you to open the figure later in the figure editor and edit line thicknesses, axis info, titles, etc. I'm embarrassed to say exactly how long it took me to "figure" this out, but `.fig` files have saved me a lot of time and suffering. Anyway, once you have saved a figure as a `.fig`, there are several file types available that play well with other programs including `.png`, `.jpg`, `.pdf`, etc.

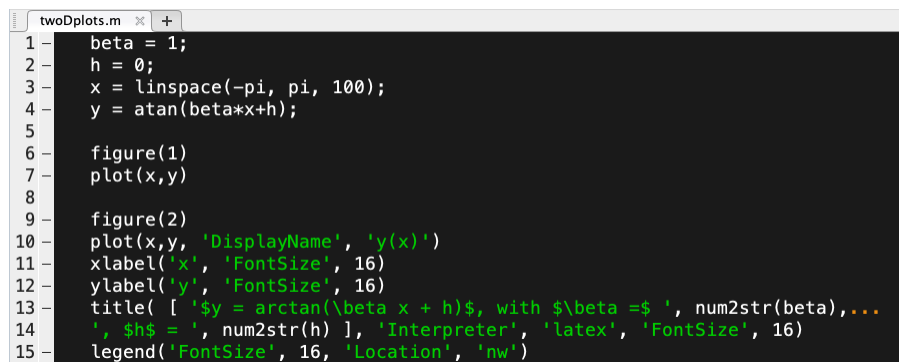
#### 3.1 2D Plots

With the exception of the symbolic plotting function `fplot`, plotting is done discretely. That is, a function is sampled at multiple points across a domain versus continuously throughout the entire domain. For more examples see

[https://www.mathworks.com/help/matlab/creating\\_plots/using-high-level-plotting-functions.html](https://www.mathworks.com/help/matlab/creating_plots/using-high-level-plotting-functions.html).

Example: Plot  $y(x) = \arctan(\beta x + h)$  on  $[-\pi, \pi]$  using  $N = 100$  points, with  $\beta = 1$  and  $h = 0$ .

*Note: In the code below you will notice in lines 6 and 9 the command `figure(#)`. This tells MATLAB to open a new window for a figure. All plots will be added to this window until another window is opened.*

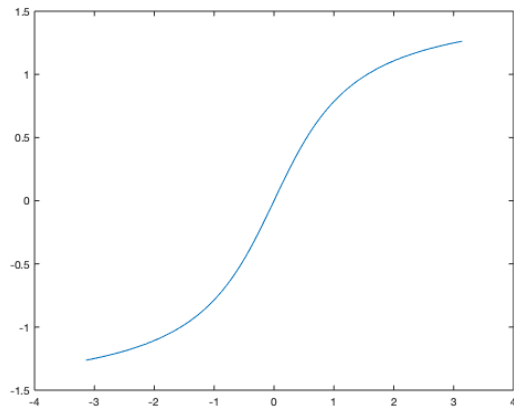
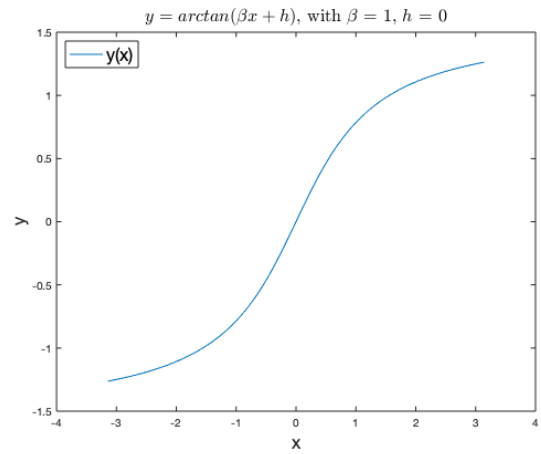


```

1  beta = 1;
2  h = 0;
3  x = linspace(-pi, pi, 100);
4  y = atan(beta*x+h);
5
6  figure(1)
7  plot(x,y)
8
9  figure(2)
10 plot(x,y, 'DisplayName', 'y(x)')
11 xlabel('x', 'FontSize', 16)
12 ylabel('y', 'FontSize', 16)
13 title(['$y = \arctan(\beta x + h)$, with $\beta = $', num2str(beta), ...
14        '$h = $', num2str(h) ], 'Interpreter', 'latex', 'FontSize', 16)
15 legend('FontSize', 16, 'Location', 'nw')

```

Figure 3.1: Plotting  $y(x)$ . `figure(1)` is a basic plot without labels. `figure(2)` uses the L<sup>A</sup>T<sub>E</sub>X interpreter to create a nice title. Note that integers/floats must be converted to strings if you wish to include them in a title.

(a) `figure(1)` from code.(b) `figure(2)` with fancy title, labels, and legend.Figure 3.2: Plots produced from `twoDplots.m`

### 3.2 3D Figures

The two most common ways to visualize 2D data is with a surf or contour plot. In older versions of MATLAB, 3D plots required a `meshgrid` of the spatial domain  $\Omega = [a, b] \times [c, d]$ . However, in newer versions, you can simply provide the surf and contour functions with vectors  $(\vec{x}, \vec{y})$ . For more info on meshgrid visit <https://www.mathworks.com/help/matlab/ref/meshgrid.html>

Example: Use meshgrid to generate 2D data from a function  $g(x, y)$ .

1. Let  $\Omega = [0, 2\pi] \times [0, 2\pi]$ . That is create two vectors that discretize the domain with  $N = 8$  grid points.
2. Create a function handle for  $g(x, y) = \sin(x) \cos(y)$ , such that `g = @(x,y) sin(x) .* cos(y);`  
*Note the use of the dot operator.*
3. Create a meshgrid of the domain: `[X,Y] = meshgrid(x,y);`
4. Display `X` and `Y`. Can you see what meshgrid did?
5. Let `Z = g(X,Y)`. This creates a matrix `Z` where  $z_{1,1} = g(x_1, y_1)$ ,  $z_{1,2} = g(x_2, y_1), \dots, z_{1,N} = g(x_N, y_1)$ , etc.

*Note: The x values change with the columns, and the y values change with the rows. Just like a Cartesian grid.*

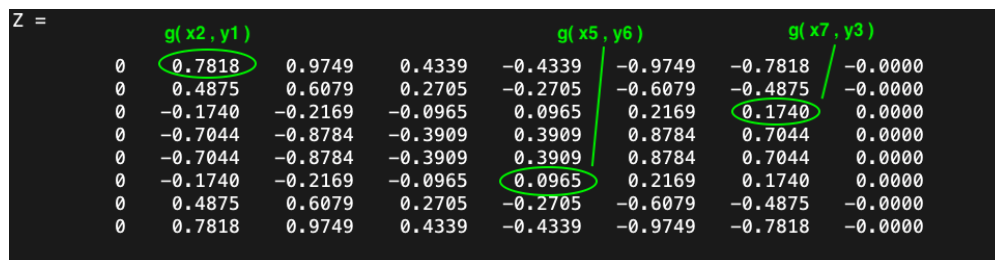


Figure 3.3: Generating data from a 2D function using meshgrid.

**Surf:** Surf creates a 3D surface plot of the values in a matrix `Z` as heights above a grid in the  $x - y$  plane. For more information see <https://www.mathworks.com/help/matlab/ref/surf.html>

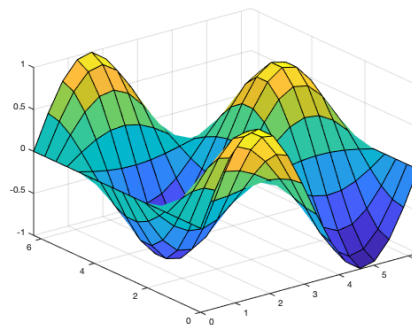
Example: Create a surf of the function  $g(x, y)$  above on  $\Omega = [0, 2\pi] \times [0, 2\pi]$

```

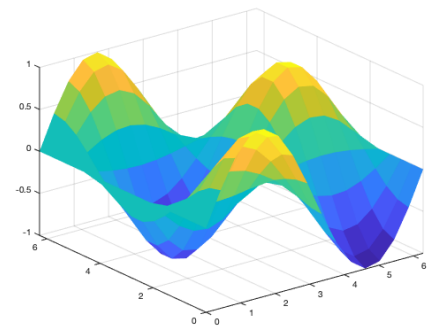
1 N = 16;
2 x = linspace(0,2*pi,N);
3 y = x;
4
5 g = @(x,y) sin(x) .* cos(y);
6
7 [X,Y] = meshgrid(x,y);
8
9 Z = g(X,Y);
10
11 figure(1)
12 surf(X,Y,Z)
13
14 figure(2)
15 surf(x,y,Z, 'EdgeColor', 'none')

```

(a) surfEx.m



(b) figure(1) from code.



(c) figure(2) has no grid lines.

Figure 3.4: Plots produced from surfEx.m

**Contour:** The contour function creates a contour plot containing the isolines of a matrix **Z**, where **Z** contains the height values on the  $x$ - $y$  plane. For more information see <https://www.mathworks.com/help/matlab/ref/contour.html>.

Example: Create a contour plot of the function  $g(x, y)$  above on  $\Omega = [0, 2\pi] \times [0, 2\pi]$ . We use `contour(X,Y,Z)` and `colorbar()` to create the contour plot and associated color bar legend.

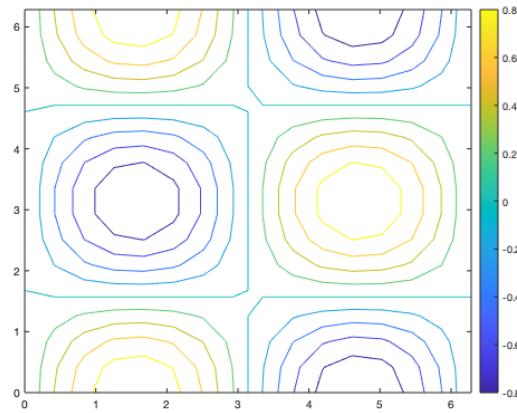


Figure 3.5: Contour plot of  $g(x, y)$  with associated color bar.

### 3.3 Subplot

Subplot provides users with the ability to tile several plots into one figure. For more information see <https://www.mathworks.com/help/matlab/ref/subplot.html>.

One thing worth noting is that in newer versions of MATLAB you can easily create a title for the entire subplot using `sgtitle('...')`.

### 3.4 Figure Editor

The figure editor in MATLAB is an incredibly versatile tool that can be used for much more than plotting your data. For example I created the figure below to illustrate the variables for a distance function I used in an immersed boundary method. Figures like this can be made in L<sup>A</sup>T<sub>E</sub>X using the Tikz package, but they can be time consuming to create. This figure took roughly 5 minutes to make in MATLAB.

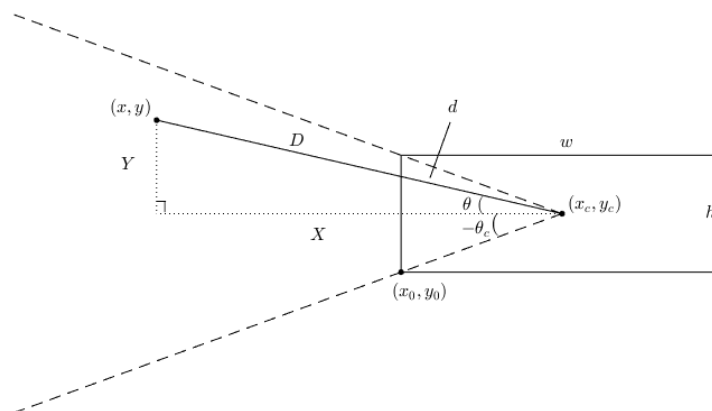


Figure 3.6: A figure created in MATLAB's figure editor.



Example: Let's create a basic figure that labels parts of a triangle.

1. In the Command Window, type `close` to close any open figures, then open up a new figure using `figure(1)`.
2. Next, type `axis([0 3 0 3])` to give a plotting domain of  $\Omega = [0, 3] \times [0, 3]$ .
3. In the next line type `hold on`. This allows multiple lines to be added to the same figure.
4. Now create a triangle in this domain. The code below gives an example.

*Note: 'k-' indicates the line will be a solid black line.*

```
Command Window
>> figure(1)
>> axis([0 3 0 3])
>> hold on
>> plot([0.5 2.5], [0.5 0.5], 'k-') % Bottom of Triangle
>> plot([0.5 1], [0.5 2.5], 'k-') % Left side
>> plot([1 2.5], [2.5 0.5], 'k-') % Right side
>> plot([1 1], [0.5 2.5], 'k-') % Height
```

Figure 3.7: Code to plot a triangle.

5. In the upper menu of the figure window you will see *view*. Click *view* → *property editor*. The property editor will allow you to edit just about anything in the figure.
6. We want the height line to be dashed. Click on the vertical height line, then select '--' from the *Line* drop down menu.

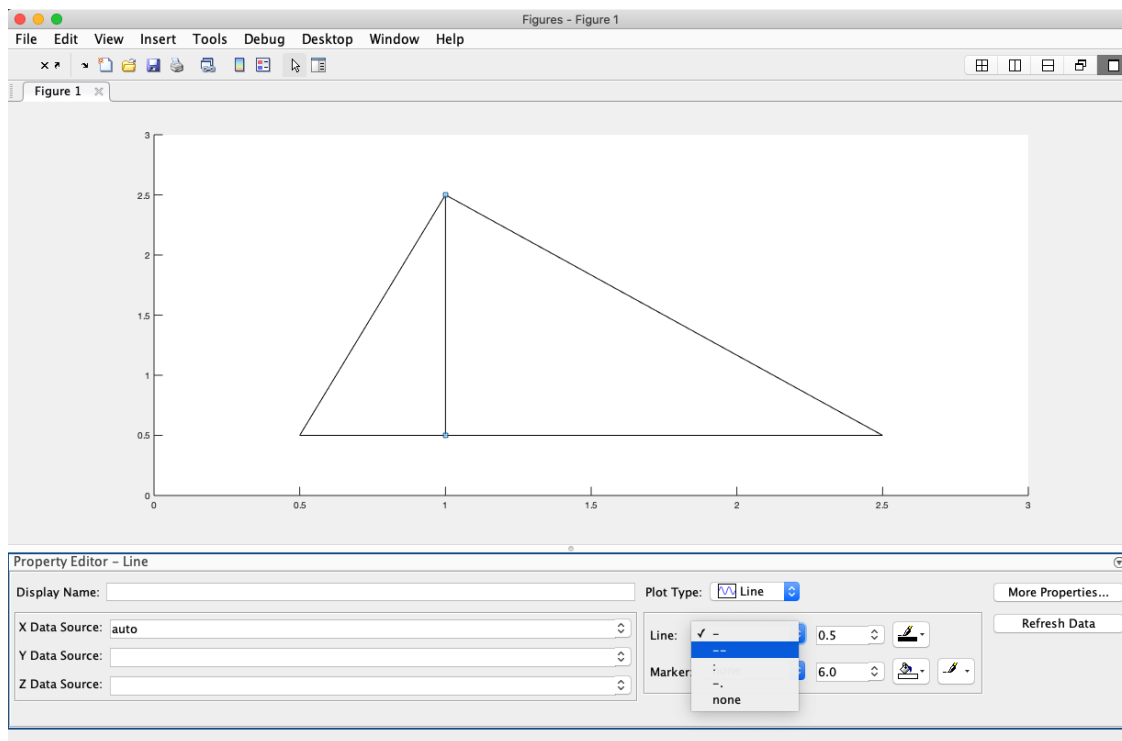


Figure 3.8: Making the height a dashed line.

7. Next, let's add some labels. Label each side  $a, b, c$ , respectively, and the height line  $h$ , by inserting a text box from the *Insert* tab in the upper menu bar.

*Note: It may be quicker to copy and paste a letter versus creating an entirely new text box each time as new text boxes always have a black border.*

8. Every triangle needs an angle labeled right? Label one of the angles  $\theta$ .

*The text editor defaults to a "tex" interpreter. That means greek letters and basic tex show up without using  $\\$\\$$ . If the "tex" you entered isn't registering. Try using the "latex" interpreter and  $\\$\\$$  as usual.*

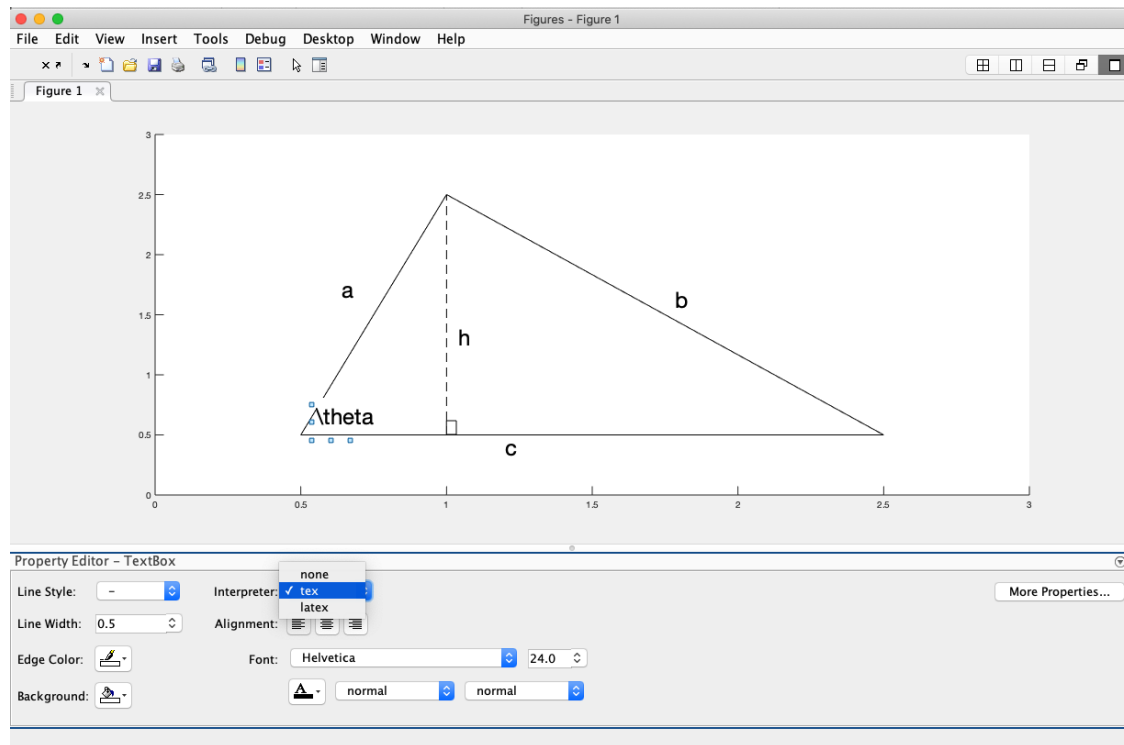


Figure 3.9: Changing the interpreter.

9. Next add a small rectangle to the bottom of the height line to indicate a right angle. To do this click Insert → Rectangle.
10. Remove the axis ticks by clicking on the either axis. On the right side of the property editor you will see the a button called *ticks*. Click on this to open the Edit Axes Ticks window. Click on one of the locations then click delete.
11. Lastly, remove the black border by selecting the border, then changing the color to white in the property editor.

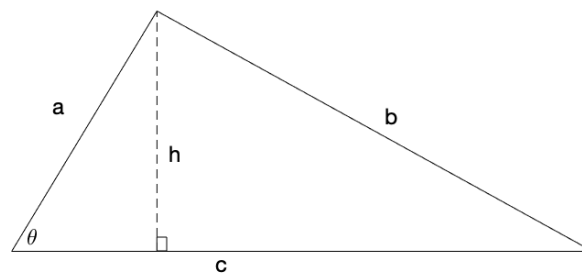


Figure 3.10: Your first labeled figure.

## 4 Debugging

Debugging or finding errors in codes can be a maddening process. Luckily MATLAB has some intuitive built in features that aid users in their hunt for the tinniest of bugs.

### 4.1 Errors and Breakpoints

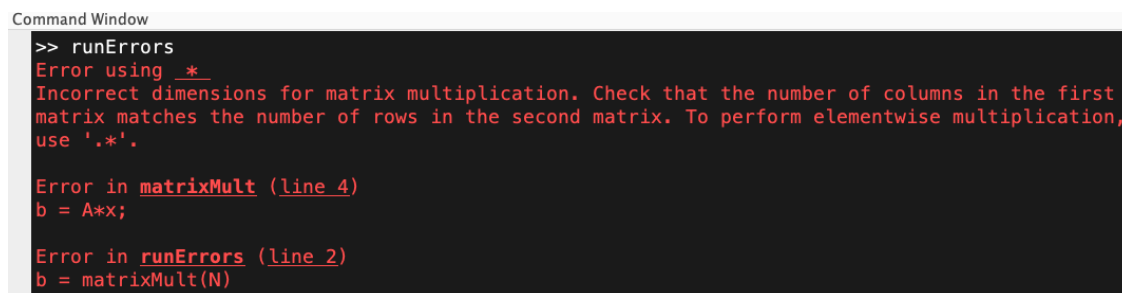
When an error occurs, MATLAB will generate a detailed report of the error in the command window. Included in the report is the location and line number of where the error occurred.

To learn about errors and how to debug them, let's create two files: a function called `matrixMult.m` that takes a single input  $N$  and returns a vector  $\vec{b}$ ; a run file that called `runErrors.m` that calls the function with  $N = 5$ .

<pre>function b = matrixMult(N) A = pascal(N); x = 1:N; b = A*x; end</pre>	<pre>N = 5; b = matrixMult(N)</pre>
--	-------------------------------------

Figure 4.1: `matrixMult.m` (left) and `runErrors.m` (right).

Copy the text above to create both files. Save the files, then run `runErrors.m`. You should get the following error.



```
Command Window
>> runErrors
Error using .*
Incorrect dimensions for matrix multiplication. Check that the number of columns in the first
matrix matches the number of rows in the second matrix. To perform elementwise multiplication,
use '.*'.

Error in matrixMult (line 4)
b = A*x;

Error in runErrors (line 2)
b = matrixMult(N)
```

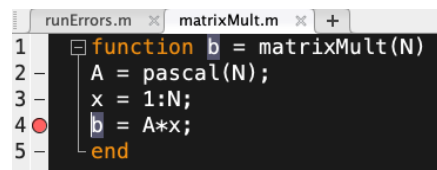
Figure 4.2: Error!!!!

This tells us that there is a problem with the matrix multiplication. Namely, that the dimensions are incorrect in **line 4** of the `matrixMult` file. While you might already see the error in the code above, humor me for a few minutes while I introduce breakpoints for trouble shooting MATLAB code.

**Breakpoints:** Breakpoints pause the execution of a program so that you can examine values where you think a problem might be. You can set breakpoints using the Editor or by using functions in the Command Window. For the full reference guide, see [https://www.mathworks.com/help/matlab/matlab\\_prog/set-breakpoints.html](https://www.mathworks.com/help/matlab/matlab_prog/set-breakpoints.html).

**Using Breakpoints to Debug:** Take the following steps to debug our code.

1. Click on **line 4** in the error report next to `matrixMult` to open `matrixMult.m` and jump to the line that threw the error.
2. Next to the line number 4, click on the “-” to place a breakpoint where the error occurred.



```
runErrors.m x matrixMult.m x +
1 function b = matrixMult(N)
2 A = pascal(N);
3 x = 1:N;
4 b = A*x;
5 end
```

Figure 4.3: Adding a breakpoint where the error occurred.

3. Rerun `runErrors.m`. In the command window you should see the following:

```
>> runErrors
4   b = A*x;
K>>
```

Figure 4.4: Debugging mode in MATLAB.

`K>>` indicates that you are now in the debugging mode. That means our workspace will contain variables from the function where the breakpoint is.

4. The error said incorrect dimensions for matrix multiplication. Therefore, let's check the dimensions of the arrays `A` and `x` using the `size` function. In the Command Window enter `size(A)`, then `size(x)`.

```
K>> size(A)
ans =
     5     5
K>> size(x)
ans =
     1     5
```

Figure 4.5: Welp, there's the problem right there...

Looks like we tried to multiply a  $5 \times 5$  matrix by a  $1 \times 5$  row vector... whoops!

5. Redefine `x = (1:N)'` to make `x` a column vector, then remove the breakpoint by clicking on it. Save `matrixMult.m`.
6. In the upper right part of the Editor menu, click the red rectangle to Quit Debugging.
7. Run `runErrors.m`. You should see that  $\vec{b} = [15 \ 55 \ 140 \ 294 \ 546]^T$

Congratulations! You have successfully debugged a code the MATLAB way!

## 5 How to Turn in MATLAB Code for Assignments

So you've got this assignment that involves some MATLAB code that you need to turn in. While we've been using a lot of screen shots in this tutorial, that is not acceptable or practical for assignments. We will present two methods for exporting code, both of which can be used for electronic or hard copy submission. If you are creating a  $\text{\LaTeX}$  document, we highly recommend using listings to import `.m` files directly into your tex.

### 5.1 Publish

The first option utilizes an internal MATLAB feature to export your code to a `.pdf` file.

1. In MATLAB, open the `.m` file that you wish to export. We are going to export a file called `streamFxn.m` which calculates, you guessed it, a stream function  $\psi$  for a given 2D velocity field.
2. In the upper menu of the MATLAB editor, toggle to the Publish menu.
3. On the right side of the Publish menu, you will see a button with a green arrow/document with the word Publish beneath it. Click the small black downward facing arrow at the bottom of this button to access a drop down menu. Click "Edit Publishing Options..."

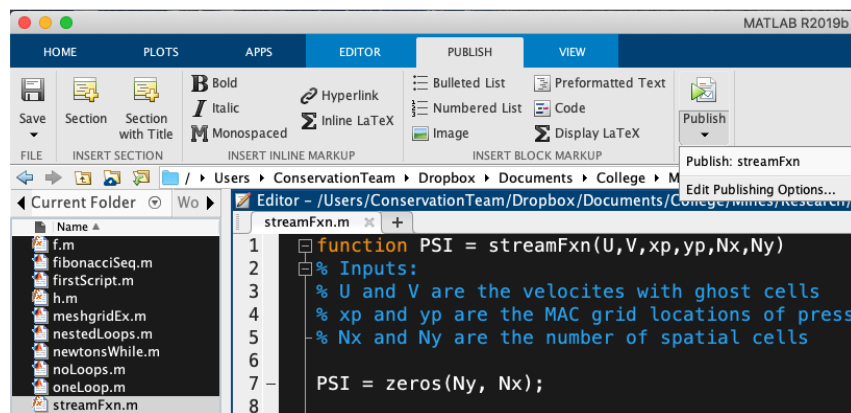


Figure 5.1: Opening up the Edit Publishing Options window.

4. In the Output settings drop down, click to the right of "Output file format" (should say html by default). Select pdf. *We'll talk about  $\text{\LaTeX}$  exports in the next section.*

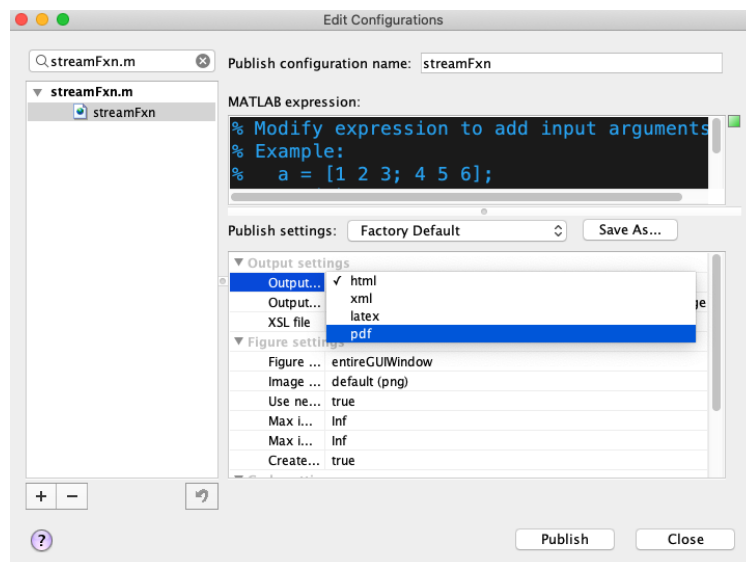


Figure 5.2: Selecting `.pdf` for the export file format.

- Click Publish at the bottom. By default MATLAB saves the .pdf in a new folder within the current folder called “html”. You can change the Output folder if you wish.

## 5.2 Listings in L<sup>A</sup>T<sub>E</sub>X

In our opinion, the best way to turn in code for assignments is by importing the .m file directly into the L<sup>A</sup>T<sub>E</sub>X document you have already created for your assignment. There are several benefits to this approach:

- Any edits you make to the .m file will be immediately updated in your .tex file.
- You can clearly show and explain the math/code simultaneously.

The most common way to import an .m file into your L<sup>A</sup>T<sub>E</sub>X document is to use the listings package. For documentation see: <http://texdoc.net/texmf-dist/doc/latex/listings/listings.pdf>.

Alternatively, you can use the Matlab Prettifier package which is built on top of the listings package. For the sake of time, we will provide all the necessary code for importing .m files using the Matlab Prettifier package. More info about this package can be found at <http://mirror.las.iastate.edu/tex-archive/macros/latex/contrib/matlab-prettifier/matlab-prettifier.pdf>.

- Add the following package in the preamble of your L<sup>A</sup>T<sub>E</sub>X document.

```
\usepackage{matlab-prettifier}
```

- In the document body, use the following code to import your .m file.

```
\lstinputlisting[style=Matlab-editor, basicstyle=\color{black}\ttfamily
\scriptsize,frame=single]{../Basics/streamFxn.m}
```

*Note: All you need to edit is the file name and directory location from you the folder where your L<sup>A</sup>T<sub>E</sub>X document is located: ../Basics/streamFxn.m*

```
function PSI = streamFxn(U,V,xp,yp,Nx,Ny)
% Inputs:
% U and V are the velocities with ghost cells
% xp and yp are the MAC grid locations of pressure
% Nx and Ny are the number of spatial cells

PSI = zeros(Ny, Nx);

% Compute first column of PSI
for I = 2:Ny
    % Compensate for Ghost Points
    i = I + 1;

    dy = yp(i) - yp(i-1);

    uw = U(i-1,1);
    ue = U(i-1,2);
    unw = U(i,1);
    une = U(i,2);

    un = 0.5 * ( unw + une);
    up = 0.5 * ( uw + ue );

    PSI(I,1) = PSI(I-1,1) + 0.5 * dy * ( un + up );
end

% Compute all other columns
I = 1:Ny;
for J = 2:Nx
    j = J+1;
    dx = xp(j) - xp(j-1);

    vn = V(I+1, j-1);
    vs = V(I, j-1);
    vne = V(I+1, j);
    vse = V(I, j);

    ve = 0.5 * ( vne + vse);
    vp = 0.5 * ( vn + vs );

    PSI(I,J) = PSI(I,J-1) - 0.5 * dx .* ( ve + vp );
end
```

## 6 Extra's from a Grad Student

### 6.1 Color Schemes

If you prefer a different color scheme than the default white background, check out MATLAB Schemer in the MathWorks File Exchange: <https://www.mathworks.com/matlabcentral/fileexchange/53862-matlab-schemer>

### 6.2 Sparse Matrices

A sparse matrix is a matrix with a large percentage of zeros. The sparse function in MATLAB provides an efficient way store sparse matrices. In MATLAB, a sparse matrix is easily created from three,  $k$ -dimensional vectors  $\mathbf{i}, \mathbf{j}, \mathbf{a}$ . The vectors  $\mathbf{i}$  and  $\mathbf{j}$  store the row and column indices for each non-zero entry  $a_{i,j}$ . The vector  $\mathbf{a}$  contains the value of  $a_{i,j}$ .

Example: Create an  $8 \times 8$  sparse matrix  $\mathbf{A}$ , such that

$$A = \begin{bmatrix} 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 & 0 & 4 & 0 \\ 0 & 4 & 0 & 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 1 \end{bmatrix}.$$

There are a few ways to build this matrix. We present a long and short option.

1. One option is to build this matrix using a for loop and the `sparse` function. For more info/examples of sparse matrices in MATLAB, see <https://www.mathworks.com/help/matlab/ref/sparse.html>.

```
% Option 1
N = 8;

% Initialize the 18 entries
row = zeros(18,1); % stores the index for ea. row
col = zeros(18,1); % stores the index for ea. col
a = zeros(18,1); % stores the associated A(i,j) entry

ctr = 0; % start a counter for row/col indexing

for i = 1:N

    % Main Diagonal Entries
    ctr = ctr + 1;
    row(ctr) = i;
    col(ctr) = i;
    a(ctr) = 1;

    % Sup Diagonal Entries
    if (i < 6)
        ctr = ctr + 1;
        row(ctr) = i;
        col(ctr) = i + 3;
        a(ctr) = 4;
    end

    % Sup Diagonal Entries
    if (i > 3)
        ctr = ctr + 1;
        row(ctr) = i;
        col(ctr) = i - 3;
        a(ctr) = 4;
    end

end

A = sparse(row,col,a,N,N); % Create N x N sparse matrix from row, col, a
```

2. The more efficient way to build this matrix is to use the `spdiags` function. For more info see <https://www.mathworks.com/help/matlab/ref/spdiags.html>. The following 3 lines of code build matrix  $\mathbf{A}$ .

```
% Option 2
N = 8;
e = ones(N,1);
A = spdiags([4*e e 4*e], [-3 0 3], N,N);
```

In the last line, we provide an  $8 \times 3$  matrix `[4*e e 4*e]` for the values on the diagonals. The second entry, `[-3 0 3]`, is a vector that tells `spdiags` which diagonals to put the columns of `[4*e e 4*e]` on. Note that the main diagonal is the 0 diagonal. Thus the 4's go on the  $-3$  and  $3$  diagonals.

Regardless of which method you use, you will see the following output for the sparse matrix **A**.

```
A =
(1,1)      1
(4,1)      4
(2,2)      1
(5,2)      4
(3,3)      1
(6,3)      4
(1,4)      4
(4,4)      1
(7,4)      4
(2,5)      4
(5,5)      1
(8,5)      4
(3,6)      4
(6,6)      1
(4,7)      4
(7,7)      1
(5,8)      4
(8,8)      1
```

(a) Output for sparse matrix **A**.

```
>> full(A)
ans =
     1     0     0     4     0     0     0     0
     0     1     0     0     4     0     0     0
     0     0     1     0     0     4     0     0
     4     0     0     1     0     0     4     0
     0     4     0     0     1     0     0     4
     0     0     4     0     0     1     0     0
     0     0     0     4     0     0     1     0
     0     0     0     0     4     0     0     1
```

(b) Using the `full` function to visualize **A**.

Figure 6.1: Sparse matrix **A** and non-sparse `full(A)`. Note that MATLAB only needs to store 18 entries when a matrix is created as a sparse matrix. This is a huge improvement on the 64 entries of a full matrix.

### 6.3 Cell Arrays

A cell array is a data type with indexed data containers called cells, where each cell can contain *any* type of data. Cell arrays commonly contain either lists of text, combinations of text and numbers, numeric arrays of different sizes, or function handles. Refer to sets of cells by enclosing indices in smooth parentheses, `()`. Access the contents of cells by indexing with curly braces, `{}`. For more info see <https://www.mathworks.com/help/matlab/ref/cell.html>.

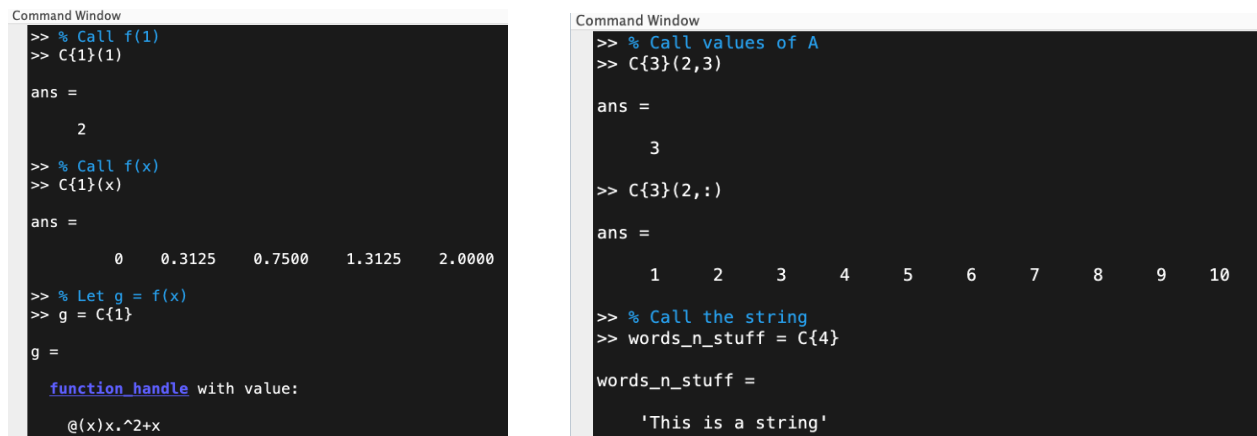
Example: The following code demonstrates how to create and call the contents of a cell array.

```
cellArrayEx.m
1 - f = @(x) x.^2 + x;
2 - x = 0:.25:1;
3 - A = pascal(10);
4 - str = 'This is a string';
5
6 - C = {f x A str}

Command Window
>> cellArrayEx
C =
1x4 cell array
    {@(x)x.^2+x}    {1x5 double}    {10x10 double}    {'This is a string'}
```

Figure 6.2: Creating a cell array that contains a function handle, vector, matrix and a string.





The figure consists of two side-by-side screenshots of the MATLAB Command Window. The left window shows a sequence of commands: calling a function `f(1)` to get the value 2, calling `C{1}(1)` to get the same value, calling `f(x)` to get a vector of values, and assigning a function handle to `g`. The right window shows: calling `C{3}(2,3)` to get the value 3, calling `C{3}(2,:)` to get a row vector of values 1 through 10, and assigning a string to `words_n_stuff` from `C{4}`.

```
Command Window
>> % Call f(1)
>> C{1}(1)

ans =

     2

>> % Call f(x)
>> C{1}(x)

ans =

     0     0.3125     0.7500     1.3125     2.0000

>> % Let g = f(x)
>> g = C{1}

g =

function_handle with value:

 @(x)x.^2+x

Command Window
>> % Call values of A
>> C{3}(2,3)

ans =

     3

>> C{3}(2,:)

ans =

     1     2     3     4     5     6     7     8     9    10

>> % Call the string
>> words_n_stuff = C{4}

words_n_stuff =

'This is a string'
```

Figure 6.3: Calling the contents of a cell array.

## 6.4 Symbolic Computing

MATLAB has a powerful symbolic toolbox that you may find useful in your studies here at Mines. In particular, it can be helpful for checking derivatives, integrals, or factoring/simplifying exciting expressions. We won't get into the details here, but if you'd like to learn more about symbolic computations in MATLAB visit

<https://www.mathworks.com/help/symbolic/symbolic-computations-in-matlab.html>.

## 7 Coding Projects

In this section we challenge you to write your own codes that solve the given problem.

### 7.1 Real Roots of a Quadratic

We all know that the roots of a quadratic,  $ax^2 + bx + c = 0$ , are given by

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Write a function with inputs  $(a, b, c)$  that returns the *real* roots of a quadratic. Consider the following problems and how to handle them in your code:

1. How do you return a double root versus two unique roots.
2. What will your code return if the root is imaginary? Will you notify users that the root is imaginary?

Test your code with various  $a, b, c$ .

### 7.2 Self Portrait

Use the plotting features in MATLAB to create a self portrait. The `rectangle` function may be useful: <https://www.mathworks.com/help/matlab/ref/rectangle.html>. How much detail can you achieve?

### 7.3 Stellar Motion - MATLAB Onramp

<https://matlabacademy.mathworks.com/R2020a/portal.html?course=gettingstarted#chapter=14&lesson=1&section=1>