

# A MATLAB<sup>®</sup> Reference Guide for Undergraduate STEM Majors

*Department of Mathematical Sciences,  
The College of Arts and Sciences, University of Delaware, Newark, DE*

Last updated: 17 December 2021

MATLAB<sup>®</sup> is a software package with several built-in functions to help with a variety of tasks. Most science and engineering students learn MATLAB<sup>®</sup> by their sophomore and junior years, oftentimes paired as a lab component to another course (e.g., linear algebra or ordinary differential equations). From there, the typical undergraduate STEM student goes on to use it for their upper-level courses and research endeavors. And yet, rarely is an introductory course on MATLAB<sup>®</sup> offered in an undergraduate curriculum. Students are thrown into labs with only a handful of examples from a worksheet to work with. Moreover, we have found that several students have no prior coding experience with the exception of perhaps a calculus sequence that used Mathematica.

That is where this reference guide comes into the picture. The purpose of this guide is to provide an informal reference to common functions and applications of MATLAB<sup>®</sup> that a typical STEM student might encounter in their undergraduate education. Please note that we have chosen to emphasize the *mathematical topics* that we feel most STEM majors will encounter at some point. Our guide is broken up into three parts: (1) “teaching” several common functions through examples and exercises; (2) an appendix of some advanced functions seen in junior/senior year courses; and (3) a series of optional labs on additional advanced topics. Our solutions are provided, but it is in the best interest of the reader to earnestly attempt the labs on their own. If the reader just wants a *gentle introduction* to MATLAB<sup>®</sup>, then the first part of this guide should suffice. The reader should read the first part in the presented order since the exercises assume knowledge from previous sections.

This reference guide is by no means exhaustive, neither in MATLAB<sup>®</sup> capabilities nor in mathematical scope. Should the reader desire an actual book that is easily accessible and wide in scope, we highly recommend *MATLAB<sup>®</sup>, A Practical Introduction to Programming and Problem Solving*, by Stormy Attaway. It is an excellent book! Please note that we use MATLAB<sup>®</sup> R2019b, and as such the functions covered might change in updated versions. If the reader finds any typos or feels additional topics would benefit this guide, then please send an email to [nakaoj@udel.edu](mailto:nakaoj@udel.edu) or [dphayes@udel.edu](mailto:dphayes@udel.edu). With all this said, we hope this reference guide serves you well!

Joseph H. Nakao  
Daniel P. Hayes

# Contents

1	The MATLAB <sup>®</sup> Desktop Environment: What Does Everything Mean? . . .	4
2	Help! I've Never Coded Before. What Are Variables? . . . . .	6
3	Starting to Code, Logical Operators, If Statements . . . . .	10
4	Loops: for, while . . . . .	14
5	Vectors, Matrices, Arrays, and "Two" Types of Multiplication/Division . . .	18
6	Cell Arrays . . . . .	24
7	Debugging and Break Points . . . . .	26
8	Function Handles . . . . .	30
9	Plotting Functions of One and Two Variables . . . . .	34
10	Creating Your Own Functions . . . . .	42
11	Saving and Importing Data . . . . .	47
12	Appendix (Other Useful Functions) . . . . .	52
13	Lab 1 – Newton's Method . . . . .	53
14	Lab 2a – Trapezoid Rule . . . . .	53
15	Lab 2b – Simpson's Rule . . . . .	54
16	Lab 2c – Gauss-Legendre Quadrature . . . . .	54
17	Lab 3a – Forward Euler Method . . . . .	55
18	Lab 3b – Heun's Method . . . . .	56
19	Lab 3c – 4th Order Runge-Kutta (RK4) . . . . .	56
20	Lab 4 – Linear Regression . . . . .	57
21	Lab 5 – Least Squares Problems: Solving a Basic Linear System . . . . .	58
22	Lab 6a – Lagrange Polynomials . . . . .	60
23	Lab 6b – Hermite Polynomials . . . . .	61
24	Lab 6c – Cubic Splines . . . . .	62
25	Lab 7 – The 1D Heat Equation (Separation of Variables) . . . . .	63

26 Lab 8 – The 2D Laplace Equation (Separation of Variables) . . . . .	63
27 Lab 9 – The 1D Linear Advection Equation (Finite Differences) . . . . .	64
28 Lab Solutions . . . . .	65

# The MATLAB<sup>®</sup> Desktop Environment: What Does Everything Mean?

Upon opening MATLAB<sup>®</sup> you should see a user interface similar to below. Note that the configuration shown below might differ slightly from yours. The configuration can be easily changed by clicking and dragging the windows.

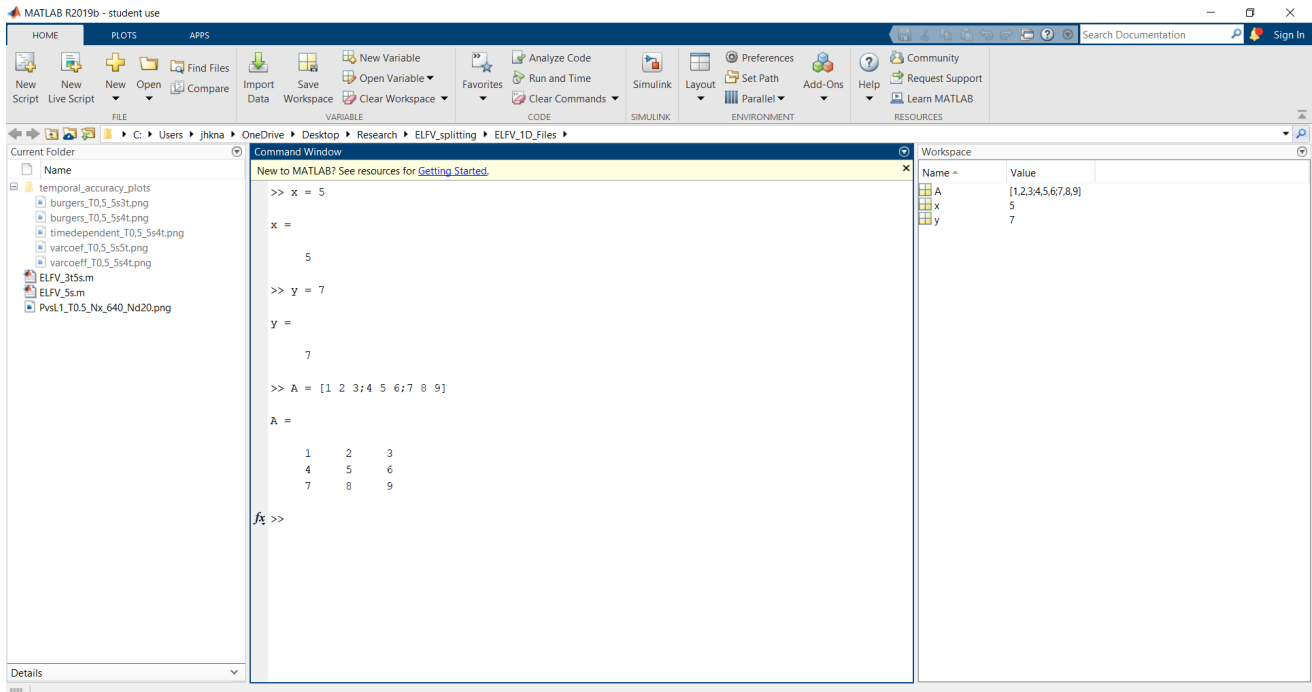


Figure 1: MATLAB<sup>®</sup> user interface.

Observe that in figure 1 there are three windows shown (for now): current folder, command window, and workspace.

- **Current Folder** tells you the folder that you are currently working in. This means that the files and (sub)folders shown are accessible to you. In other words, if you have a photo saved in a completely separate folder, then you will not be able to access that photo. Looking at the path (i.e., the line “above” the command window and “below” the toolbar), we are currently in the folder ELFV\_1D\_Files. And within this folder are the files and (sub)folders shown in the current folder. In this case, ELFV\_1D\_Files contains: the folder temporal\_accuracy\_plots with several photos, ELFV\_3t5s.m, ELFV\_5s.m, and PvsL1\_T0.5\_Nx\_640\_Nd20.png.

To change the current folder (*path*), you can click the path line below the toolbar accordingly. For instance, we could click Desktop to change our current folder (*path*) to be the entire Desktop.

To run a .m MATLAB<sup>®</sup> file that is in your current folder (*path*), simply type the file name in the **Command Window**. For instance, if we wanted to run the file ELFV\_3t5s.m, we simply need to type ELFV\_3t5s in the command window and click the “Enter (or Return)” key on your keyboard.

- **Command Window** is the window that you use to interact with MATLAB<sup>®</sup>. In other words, you will use the command window to execute commands or expressions, including running .m MATLAB<sup>®</sup> files. For instance, observe that in figure 1 we defined  $x$  to be 5,  $y$  to be 7, and  $A$  to be the matrix  $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ . Don’t worry too much about how to define these variables or how to *code*. We will cover such things in the next section. The point here is that the command window is used to interact with MATLAB<sup>®</sup>.

If you ever want to clear the command window (perhaps because it’s getting a bit too cluttered), you can type **clc** in the command window.

- **Workspace** is the window that gives you a list of all the variables you have defined, as well as their values and data structure. For instance, since we defined  $x$ ,  $y$  and  $A$  in the command window, they are now **variables**. The workspace shows us that  $x = 5$ ,  $y = 7$ , and  $A = [123; 456; 789]$  (this is MATLAB<sup>®</sup> notation for a matrix, but don’t worry about that yet).

If you ever want to delete a variable (e.g., you no longer want  $x$  to be a thing), then you can type **clear x** in the command window. Similarly, if you want to delete the variable  $A$ , then type **clear A**. If you want to delete *all* the variables in your workspace, then you can type **clear all** or just **clear** in the command window.

We must make a couple remarks about the **toolstrip**. For the sake of this reference guide (and to a lesser extent the sake of an undergraduate education), you will probably never touch 95% of the applications in the toolstrip. In fact, the authors have gone eight years since learning MATLAB<sup>®</sup> only ever having to use maybe five applications. The two applications we will predominantly use are **New Script** and **Open (Folders/Files)**, shown in the top left of figure 1. Perhaps you’ll use **Import Data** or **Save Workspace**, but we shan’t concern ourselves with those at the moment.

---

○

## EXERCISES

**Exercise 1.** Referring to figure 1, what would we type in the command window if we wanted to run the MATLAB<sup>®</sup> file ELFV\_5s.m?

**Exercise 2.** What are the commands for (a) clearing the entire command window; (b) clearing

all the variables in the workspace?

**Exercise 3.** Open MATLAB<sup>®</sup>. In the command window, define the variables  $x = 4$ ,  $y = 5$ , and  $z = -9$  (you can refer to figure 1 for an example). Then delete everything in the command window. Does this inadvertently clear all of the variables in the workshop?

## Help! I've Never Coded Before. What Are Variables?

This section assumes that the reader has no prior coding experience. The authors want to assure the readers that fall into this camp that YOU WILL BE OKAY! Yes, learning to code will first feel like a steep hill to climb. However, it quickly becomes easier (not without effort, might we add) and MATLAB<sup>®</sup> is a very user-friendly first computer language to learn from. Some people might tell you that MATLAB<sup>®</sup> is not a “real” computer language; ignore those people. If you pay attention to the examples in the first part of this reference guide and do the exercises, then you should have a sufficient foundation to learn more about MATLAB<sup>®</sup> on your own.

This section is pretty standard for an introduction to coding, albeit with a MATLAB<sup>®</sup> flavor. We shall start with a very simple question – what is a variable? A **variable** is an object that stores a value (or values) in MATLAB<sup>®</sup>. In simple terms, you will choose a letter or name that represents some object in MATLAB<sup>®</sup>.

Each variable has a **type/class**. For the sake of this reference guide, we shall only concern ourselves with three different types.

- **double** means that the real value is double precision. Similarly, **single** means that the real value is single precision. Do not worry about the difference between the two. For the sake of this reference guide, just know that by default MATLAB<sup>®</sup> stores real numbers with double precision.

As seen in figures 2 and 3, the variables  $x$  and  $A$  are both of type **double** since both are comprised of real numbers.  $A$  being a matrix/array doesn't matter when referring to the type.

- **char** is the type that signifies a single character (e.g., a letter in the alphabet) or a **string** (e.g., a word comprised of letters).

As seen in figures 2 and 3, the variable `myname` is of type **char** since it is a string. When defining variables of type `char`, you put the string in single quotation marks.

- **logical** is the type that signifies whether a statement is true (logical 1) or false (logical 0).

As seen in figures 2 and 3, then variable `trueorfalse` is of the type **logical** since it is checking is the statement “if 4 equal to 4?” is true or false. The logical is 1 because the statement is true. When defining variables of type `logical`, you must check a statement in the function `logical( )`, as seen in figure 2.

- **int8**, **int16**, **int32**, **int64**, **uint8**, **uint16**, **uint32**, **uint64** are other types that are used for integer variables. We are mentioning them here so that certain people don't get

angered by their omission. But as mentioned, we shall not concern ourselves with these in this reference guide.

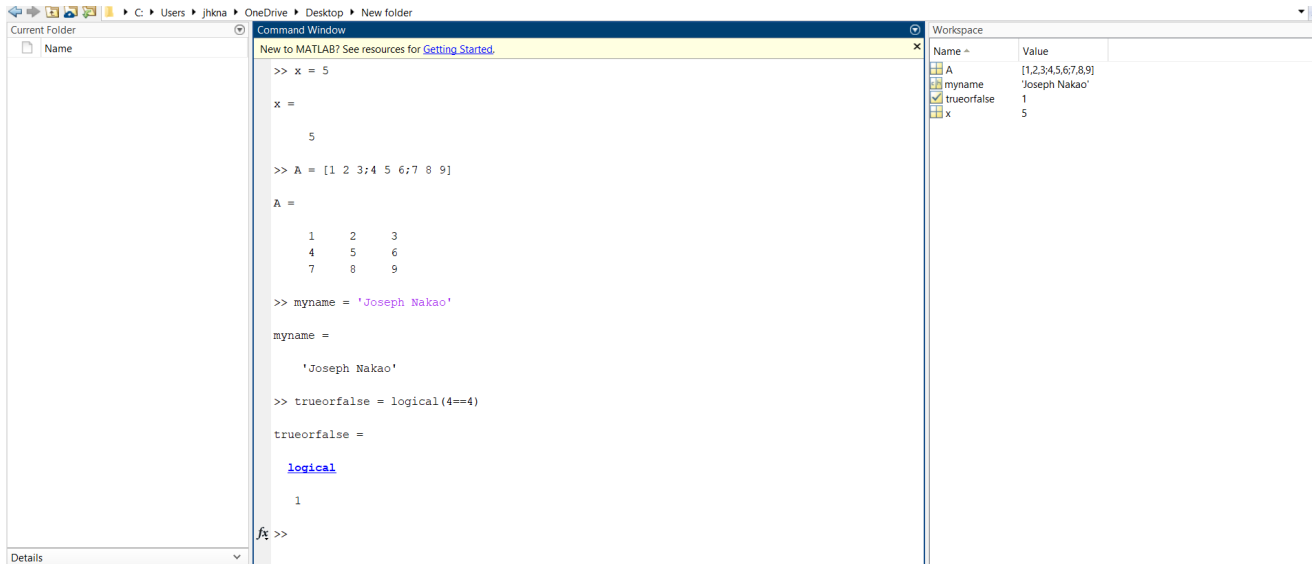


Figure 2: Defining variables of various types.

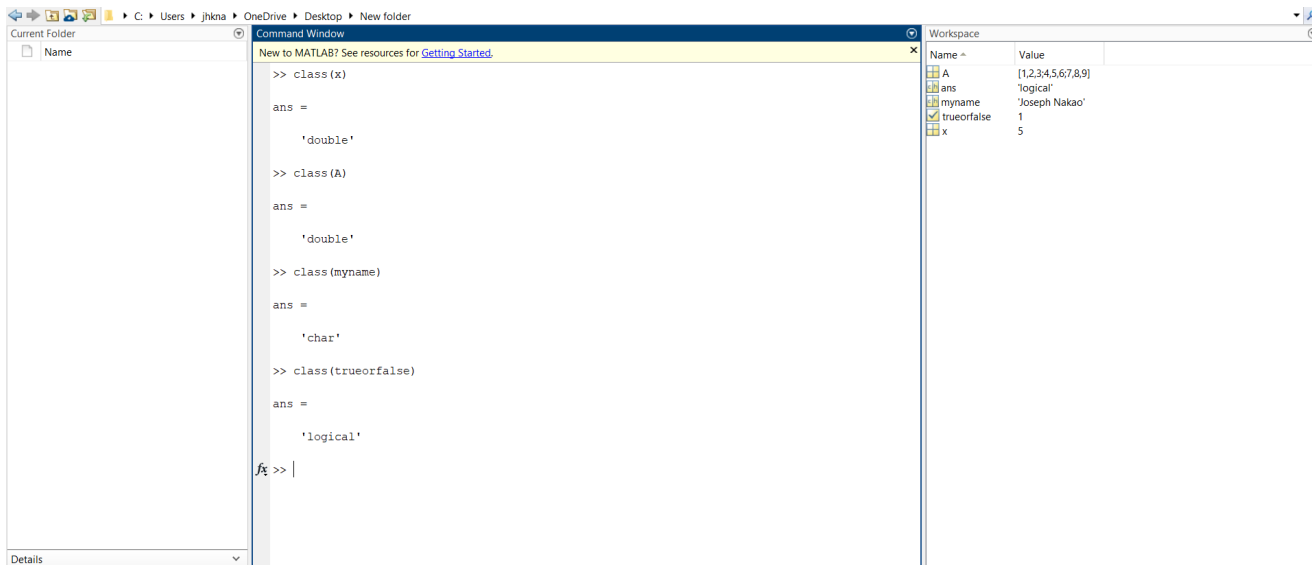


Figure 3: Checking the classes/types of the variables in figure 2.

**Concatenating strings.** What if we want to combine multiple strings (words) into a single string (word)? For instance, what if we have two variables *firstname* = 'Joseph' and *lastname* = 'Nakao' and want to combine them into a single string that is the entire name? There are two methods of doing so.

**Method 1.** We can simply put multiple strings into brackets [ ], being sure to separate the strings with either a comma or a space. As seen in figure 4, this method spits out the string 'JosephNakao'. There is not a space in between the first and last names because we quite literally combined the two strings into one. Whereas, as seen in figure 4, when I put a space at the end of the first name, we get 'Joseph Nakao'.

**Method 2.** We can also use the function **strcat** (i.e., string concatenate). If you want to combine the strings *firstname* and *lastname*, then you would enter *strcat(firstname,lastname)*, as seen in figure 4. The difference between **strcat** and using brackets [ ] is that **strcat** removes all spaces, even if they are included in the strings we are combining. This is also demonstrated in figure 4.

```
>> firstname = 'Joseph'

firstname =

    'Joseph'

>> lastname = 'Nakao'

lastname =

    'Nakao'

>> [firstname,lastname]

ans =

    'JosephNakao'

>> ['Joseph ', 'Nakao']

ans =

    'Joseph Nakao'

>> strcat('Joseph ', 'Nakao')

ans =

    'JosephNakao'
```

Figure 4: Demonstrating the concatenation of multiple strings.



**Turning numbers into strings.** You can turn a number into a string using the function `num2str` (i.e., number to string). As seen in figure 5 we turn the double variable `age` into a string and make a sentence (string) using the concatenation method from the previous example.

```
>> age = 25

age =

    25

>> name = 'Samantha'

name =

    'Samantha'

>> [name, ' is ', num2str(age), ' years old. ']

ans =

    'Samantha is 25 years old.'

>> |
```

Figure 5: Turning a number (double) into a string (char).

---

## EXERCISES

**Exercise 1.** Using the command line, use the type logical to verify that 4 is less than 5.

**Exercise 2.** Using the command line, define your first and last names using the variables `firstname` and `lastname`. Put them together (without a space in between) and output your entire name using: (a) brackets; and (b) `strcat`.

**Exercise 3.** Using the command line, let `x = 5` and `myname = 'Joseph Nakao'` as in figure 2. *\*be sure to clear the command window and workspace between exercises.*

(a) Check the types of `x` and `myname`.

(b) Why should we expect `x + myname` to spit out a weird result?

(c) Enter `x + myname` in the command line. You should get a row of numbers. What happened?! To investigate, enter `double(myname)` in the command line and try to figure out what happened.

**Exercise 4.** Using the command line, define the variables `hours = 4`, `minutes = 56`, `seconds = 13`. What is the type of these three variables? Using these variables, produce the string/sentence

‘The current time is 4 hours 56 minutes 13 seconds after noon.’

# Starting to Code, Logical Operators, If Statements

It is now time to start creating more complicated codes! Although the command window is nice for short little assignments like in the previous section, it would be a nightmare to code anything longer in it! That is why we typically want to write longer codes in their own files/scripts (i.e., .m files). Unless otherwise stated, you should make it a habit to write code in their own designated files/scripts. You typically use the command window to investigate the variables defined in your code. This all being said, we need stronger tools to start coding more intricate problems.

To create a file, you can click **New Script** in the toolbar. Afterwards, you should notice a new tab called **Editor** in the toolbar. You are able to run and debug your code using the Editor tab. You can save the file by clicking **Save As** and saving it similar to how you would save any other document (e.g., Word or Power Point). As seen in figure 6, I named the MATLAB<sup>®</sup> file firstcode.

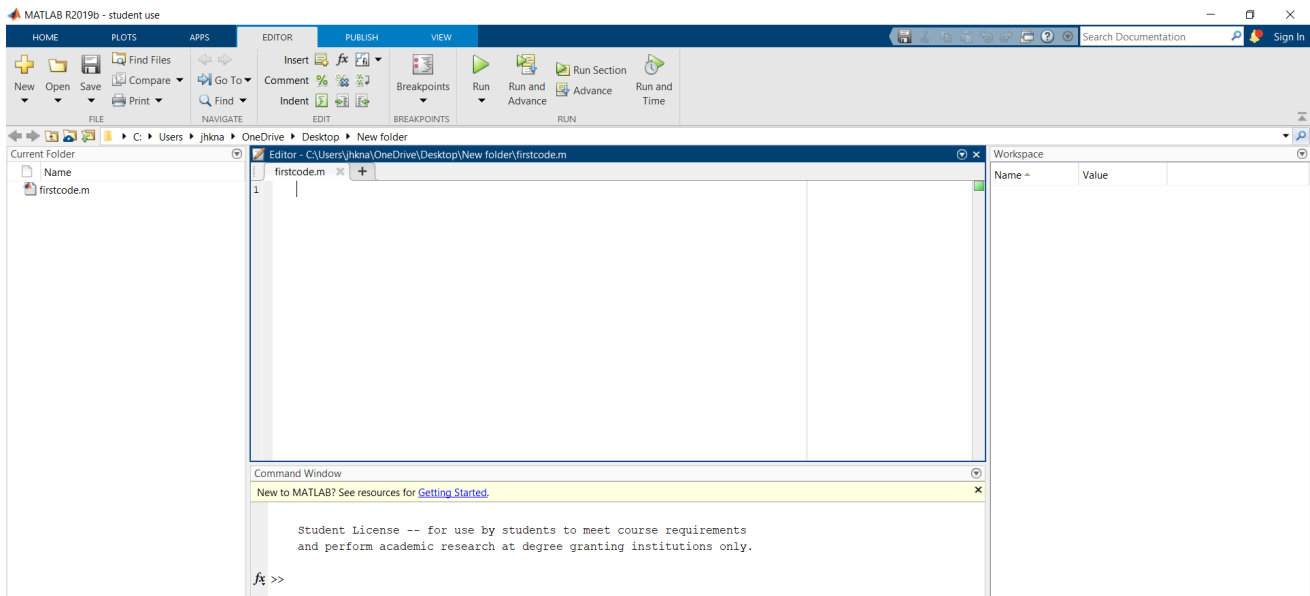


Figure 6: Creating and saving a MATLAB<sup>®</sup> file.

**Coding Tips and How to Run Your Code.** It is imperative that we discuss coding habits that will help you avoid headaches later. We shall simply list a few tips to get you started.

- Begin your code with a header of some sort using comments. After any header/initial comments, use **clear all** to clear/delete the workshop. This way, you will get rid of any old variables that might interfere with the code you're about to run. If desired, you could also use **clc** to clear/delete the command window. However, this might not always be desired since viewing old outputs could be useful.
- If you DO NOT want a line to show an output (e.g., if you want  $x = 5$  in your code, but you don't want to explicitly print  $x = 5$  in the command window), then end the line with a **semicolon**.
- COMMENT YOUR CODE AS YOU GO! There is nothing worse than a code that someone did not comment. It's incredibly important that you write little comments here and there to tell the reader what a certain part of your code is doing. Of course, don't go overboard with comments. You don't need an entire paragraph, just a few words typically suffices.
- Be sure to save your code periodically. It saves each time you run it, but it doesn't hurt to save in between runs.
- If your code keeps running forever without end (i.e., your computer is about to crash) or you want to terminate a code prematurely, click the command window and use the keyboard with **CTRL+C**.

**Commenting Code.** One of the most important components of learning to code is learning how to comment your code. Roughly speaking, a "comment" is just stuff you want to write (i.e., literally just text). However, you clearly don't want MATLAB<sup>®</sup> to think that your comments is code to run. As seen in figure 7, you make a comment in MATLAB<sup>®</sup> by starting what you want to write with an exclamation mark **!**. We typically use comments for three things:

- Writing your name, date, etc...
- Writing instructions on how to use your code.
- Commenting on what a certain part of your code is doing so that other people (and your future self) know what the heck your code is doing.

It's important to note that comments can be written in the same line as part of your code. However, anything typed *after* the exclamation point will be considered a comment by MATLAB<sup>®</sup>.

```
% Joseph Nakao
% Date: 23 April 2021
% Purpose: First code example for MATLAB reference guide.
```

Figure 7: Commenting a MATLAB<sup>®</sup> file.

**Logical Operators** are used when creating logical statements such as the following: “if  $A$  is less than or equal to  $B$ , then add  $A$  and  $B$  together”; “if  $A$  is less than zero and  $B$  is greater than zero, then multiply  $A$  and  $B$  together”. In order to code these conditional statements we need a way to code statements such as *and*, *or*, *less than*, *greater than or equal to*, *equal to*, etc...

- **and** is typed using `&` or `&&`.

The difference between `&` and `&&` is subtle. Both are logical operators. However, `&&` tells MATLAB<sup>®</sup> to evaluate the second argument only if the first argument is satisfied. This is useful for increasing performance when dealing with scalar values. We typically use `&&` by default.

- **or** is typed using `|` or `||`.

The difference between `|` and `||` is similar to that of `&` and `&&`. We typically use `||` by default.

- **equal to** is typed using two equal signs `==`.
- **less than**, **greater than** are typed using `<` and `>`, respectively.
- **less than or equal to**, **greater than or equal to** are typed using `<=` and `>=`, respectively.
- **not** is typed using `~`.

For instance, if you want to say “ $x$  not equal to zero,” then you would type  `$x \sim= 0$` .

There are more subtleties to logical operators that we are grossly avoiding. For the purposes of what a typical undergraduate STEM major will be using MATLAB<sup>®</sup> for –and keeping in mind this is supposed to be introductory– we have decided to omit such subtleties. The reader can find further information online or in Stormy Attaway’s book.

**If (and If-Else) Statements.** We just covered logical operators. However, we still need to talk about how to implement the conditional statement itself. In other words, how do we check the “if, then” statement?

Figure 8 shows how to structure an if statement. The notation for an **if statement** in MATLAB<sup>®</sup> is to start with “if STATEMENT” and end with “end”.

The two other conditions are **elseif** and **else**. The **elseif** line reads “if this other statement is true, then...” and the **else** line reads “if anything else other than the conditions already stated is true, then...”

As an example of an if-else statement is shown in figure 8. We define two variables,  $x = 5$  and  $y = -1$ . The first if statement reads “if  $x + y = 4$ , then print/display their product.” Since  $x + y$  does in fact equal four, the code spits out  $xy = (5)(-1) = -5$ .

The second if-else statement in figure 8 reads “if  $x > 0$  and  $y > 0$ , then spit out their product;

if instead  $x < 0$  and  $y > 0$ , then spit out their sum; if anything else is true, then spit out zero.” The choice of  $x = 5 > 0$  and  $y = -1 < 0$  falls into the *else* statement, so the code spits out zero.

```
x = 5;
y = -1;
% IF STATEMENT #1
if x+y==4
    disp(x+y)
end

% IF STATEMENT #2
if x>0 && y>0
    disp(x*y)
elseif x<0 && y>0
    disp(x+y)
else
    disp(0)
end
```

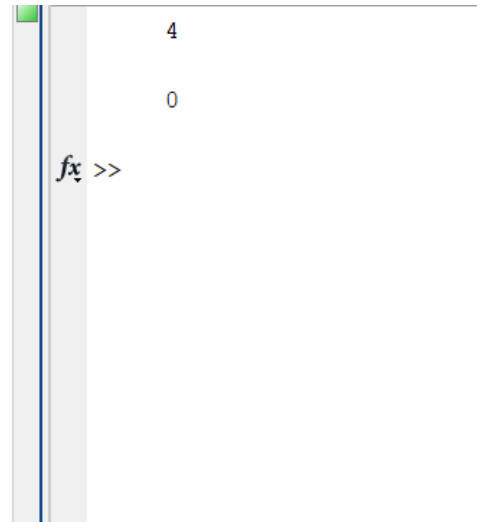


Figure 8: Demonstrating an if statement and if-else statement.

## EXERCISES

**Exercise 1.** Define two variables *firstname* and *lastname* using a made-up person’s name (don’t use your name). Write a MATLAB<sup>®</sup> code that checks whether or not: this person shares the same first initial as you but not the same last initial; this person shares the same last initial as you but not the first; this person shares both the same first and last initials as you; this person shares neither your first initial nor last initial.

*\*note – given a variable of the type char (e.g., name = ‘Joseph’) then name(1) will spit out the first letter of the variable.*

**Exercise 2.** Read the code shown below. What will be the value of *z*?

```
x = -5;
y = 1;
if x>0
    if y>0
        z = x + 2;
    else
        z = x - 2;
    end
else
    if y<0
        z = y + 2;
    else
        z = y - 2;
    end
end
```

## Loops: for, while

**For and While Loops.** There are two types of loops we use when coding: for loops and while loops.

- **For Loops** iterate through a set of values. In other words, “for certain values, I want to perform some action.”

Ex. For  $n = 1, 2, 3, 4, 5$ , I want to compute  $n - 3$ .

Ex. Consider the set of function  $\{\sin(x), \sin(2x), \sin(3x), \dots, \sin(10x)\}$ . For each function  $f(x)$  in this set, I want to compute  $f'(x = 1)$ .

- **While Loops** keep on running through the same loop until a certain condition is met. In other words, “until a certain goal is met, I will continue to update a certain action.”

Ex. We know that as you take more and more terms in a (convergent) series we will get closer and closer to the exact value. So, let’s keep adding more and more term (one at a time) until the series approximation is within 5% of the exact solution.

The notation for a **for loop** in MATLAB<sup>®</sup> is to start the loop with “for index = startval:endval” and end the loop with “end”. In this case, “index” is the name you’re giving the index variable; “startval” and “endval” are the starting and ending index values. See the examples to come to see this more explicitly.

The notation for a **while loop** in MATLAB<sup>®</sup> is to start the loop with “while (put your statement here)” and end the loop with “end”. In this case, (put your statement here) is simply whatever condition you want to be met. See the examples to come to see this more explicitly.

**For Loop Example 1.** Figure 9 shows the first for loop example listed above. As we expect, if for each  $n = 1, 2, 3, 4, 5$  we want to print/show  $n - 3$ , then we should get the values  $-2, -1, 0, 1, 2$ .

**For Loop Example 2.** Figure 10 shows another for loop example. It’s important that the reader **learns to read code!** To read code, you need to read line by line and think about the order in which MATLAB<sup>®</sup> will do things. Let’s read through this code together!

Reading things line by line (chronologically), we see that  $x$  is defined to be 5. Then, we enter the for loop which says, “for  $n = 1$ , then  $n = 2$ , ..., then  $n = 5$ , we will **redefine**  $x$  to be its current value plus whatever the value of  $n$  is.” Note that  $x = x + n$  is saying, “ $x$  will now be set to  $x$  (the current value) plus  $n$ .”

Further note that we have include a semicolon on the line  $x = x + n$ ; because we don’t want to print the value of  $x$  each loop. However, by asking for  $x$  *after* the loop, we will print out the most recent value.

Let’s work this out. We start with  $x = 5$ . Then,  $x = 6$  because the first loop set  $x = x + n$  (which in the first loop is  $x = 5 + 1$ ). Then,  $x = 8$  because the second loop set  $x = x + n$  (which in the second loop is  $n = 8 + 2$  because now the “current”  $x$  value is 8 and  $n = 2$ ). Continuing

in this fashion, the third loop will set  $x = 8 + 3 = 11$ ; the fourth loop will set  $x = 11 + 4 = 15$ ; and the fifth loop will set  $x = 15 + 5 = 20$ . Indeed, the final output is  $x = 20$ .

```

1 % Name: Joseph Nakao
2 % Date: 23 April 2021
3
4 - clear all
5 - clc
6
7 - for n = 1:5
8 -     n-3
9 - end

```

ans =  
-2  
ans =  
-1  
ans =  
0  
ans =  
1  
ans =  
2  
fx >>

Figure 9: First for loop example.

```

1 % Name: Joseph Nakao
2 % Date: 23 April 2021
3
4 - clear all
5 - clc
6
7 - x = 5; %starting value
8 - for n = 1:5
9 -     x = x + n;
10 - end
11 - x %print the final x value

```

x =  
20  
fx >>

Figure 10: Second for loop example – learning to read code.

**While Loop Example.** As the reader hopefully recalls from their calculus course,

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}, \quad |x| < 1.$$

In this example we consider this geometric series for  $x = 1/3$ . We know that the geometric series should converge to  $1/(1 - 1/3) = 1/(2/3) = 3/2$ . As seen in figure 11, we take more and more terms until we are within 0.001 of the exact value. The goal is to print two things: the approximate value; and the number of terms required.

Observe that in the while loop, we need to write  $k = k + 1$  first so that the additional term uses the *updated* index. If we'd put  $k = k + 1$  after defining the next term, then for the first iteration of the while loop, it would've used the value  $k = 0$ .

*\*note - we use the function **disp()** which outputs whatever you put in the parentheses. The advantage of disp() is that it does not also spit out the variable name. For instance,  $x = 5$  spits out that  $x = 5$ . However, **disp(x)** will just spit out 5.*

```
% Name: Joseph Nakao
% Date: 23 April 2021

clear all
clc

k = 0;      %the number of terms
tol = 0.001; %the tolerance
exact = 3/2; %the exact value
approx = 1; %the approximate value, starting with k=0. (1/3)^0 = 1.

while abs(approx-exact) >= tol %while |approx - exact| >= tolerance
    k = k + 1; %take another term.
    approx = approx + (1/3)^k; %using the "new" k value.
end

disp(['The approximate value is ',num2str(approx)])
disp(['The number of terms in the series: ',num2str(k)])
```

```
The approximate value is 1.4993
The number of terms in the series: 6
fx >>
```

Figure 11: While loop example using a geometric series.

```
number = input('Enter a number:');
word = input('Enter a word:');

class(number)
class(word)
```

```
>> firstcode
Enter a number:5.5
Enter a word:'hello'

ans =

    'double'

ans =

    'char'
```

Figure 12: How to use **input** to define variables.



**Input a Variable.** In the previous example we needed to define a tolerance. As seen in figure 11, we chose a tolerance of 0.001. But what if you wanted to change the tolerance without actually having to retype the tolerance before every run? What if we could design our code so that each run just asks us what tolerance we want?

The function to do so is `input('type your statement here')`. Whatever you put in the single quotes will show up in the command window. You will enter whatever you would've normally typed directly in the code. As seen in figure 12, we define two variables `number` and `word` using the `input` function. We enter 5.5 for `number` and 'hello' for `word`. Notice that the variable type of `number` is double, and the variable type of `word` is char.

○

## EXERCISES

**Exercise 1.** The code below is a relatively basic for loop that is supposed to update the value of  $x$  by adding one until  $x$  is larger than a tolerance of 50.2. Why won't this code run? What did we forget to do?

*\*hint – MATLAB® won't know what a variable's value is unless you've already specified it.*

```
x = 0;
while x < tol
    x = x + 1;
end
```

**Exercise 2.** Make the following modifications to the code shown in exercise 1:

(a) Fix the code; you should only have to add one more line of code. Try running it. If no error pops up in the command window, then it works.

(b) Add another line of code that will spit out the final value of  $x$ . Use the `disp()` function to do this; refer to figure 11 for an example.

**Exercise 3.** Read the code shown below. The last two lines spit out the final values of  $x$  and  $y$ . What are these values going to be?

```
y = 2;
x = y;
for k = 1:5
    y = x + k;
    x = y - 1;
end
disp(['The final value of x is ', num2str(x)])
disp(['The final value of y is ', num2str(y)])
```

**Exercise 4.** The MacLaurin series for  $e^x$  is  $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ . In particular, the  $N$ th partial sum

$e^x \approx \sum_{n=0}^N \frac{x^n}{n!}$  is a good approximation for  $-1 < x < 1$ , given that  $N$  is larger than 5. Create a

code that computes the  $N$ th partial sum of  $e^x$  for any  $x$  between -1 and 1. Design your code using `input()` to ask for an  $x$  value between -1 and 1, as well as the number of terms  $N$ . You will need to use a for loop to compute the partial sum.

*\*note – to compute  $n!$  in MATLAB<sup>®</sup> you must use the function `factorial(n)`.*

**Exercise 5.** Here, we demonstrate a nested for loop with an if statement. A nested for loop is simply two for loops with one nested inside the other. In this problem, imagine we roll two six-sided dice. Write a MATLAB<sup>®</sup> code that will compute how many possible combinations will add to 10 or greater. Below is a template of the code to get you started. Complete this code by filling in the missing lines.

```
clear all; clc;

count = 0; %number of combinations
for i = 1:6 %dice #1
    for j = 1:6 %dice #2
        if XXXcompletethisXXX
            XXXcompletethisXXX
        end
    end
end
```

## Vectors, Matrices, Arrays, and “Two” Types of Multiplication/Division

For an engineering or science major, most codes you write in MATLAB<sup>®</sup> will probably require using vectors, matrices, and arrays. Mathematically speaking, recall that a (column) vector is of dimension  $N \times 1$  (e.g., a  $4 \times 1$  column vector), and a matrix is of dimension  $M \times N$  (e.g., a  $3 \times 2$  matrix). For example, we could define a column vector  $\mathbf{v} \in \mathbb{R}^{4 \times 1}$  and a matrix  $\mathbf{A} \in \mathbb{R}^{4 \times 3}$  as:

$$\mathbf{v} = \begin{bmatrix} 2 \\ -1 \\ 5 \\ 6 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 1 & 2 & 1 \\ -4 & 7 & 1 \\ 0 & 2 & -3 \\ -4 & 2 & 3 \end{bmatrix}$$

You might be asking yourself, “How are we going to use vectors and matrices in a code?” GOOD QUESTION! Vectors and matrices can be used in several different ways. The two most common uses being:

- **Storing values.** For example, say we want to *store* the values of  $x^2$  for the  $x$ -values  $\{1, 2, 3, 4, 5\}$ . We could store these values in a row vector called *functionvals*

$$\text{functionvals} = [1 \ 4 \ 9 \ 16 \ 25]$$

- **Doing computations.** For example, say we want to compute the dot product of the vectors  $\mathbf{a} = \langle 1, 2, 3 \rangle$  and  $\mathbf{b} = \langle -1, 0, -2 \rangle$ .

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} -1 \\ 0 \\ -2 \end{bmatrix} = -1 + 0 - 6 = -7.$$

**Creating vectors and matrices.** There are many ways to build these objects in MATLAB<sup>®</sup>.

- A general vector or matrix can be built by manually entering values in between two brackets [ ]. Row entries are separated by commas; column entries are separated by semicolons. *\*Note: row entries can also just be separated by a space instead of a comma.*

Typing `[1, 2, -3, 4, -2]` will produce the row vector

$$\begin{bmatrix} 1 & 2 & -3 & 4 & -2 \end{bmatrix}$$

Typing `[1; 2; 3]` will produce the column vector

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Typing `[1, 2, 3; 4, 5, 6; 7, 8, 9]` will produce the matrix

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

- Using `a:b:c` will create a row vector of values from  $a$  to  $c$  in increments of  $b$ .

Typing `1 : 3 : 8` will produce the row vector

$$\begin{bmatrix} 1 & 4 & 7 \end{bmatrix}$$

Notice that it did not include 8 because we just said “starting from 1, increase by increments of 3 just until we pass the value 8.”

- Similar to `a:b:c`, we can use the function `linspace(a,b,n)` to create a row vector of values from  $a$  to  $b$  by discretizing the interval  $[a, b]$  with  $n$  equally spaced points. *\*Note: Unlike `a:b:c`, `linspace(a,b,n)` includes the endpoints  $a$  and  $b$ .*

Typing `linspace(1,3,6)` will produce the row vector

$$\begin{bmatrix} 1.0000 & 1.4000 & 1.8000 & 2.2000 & 2.6000 & 3.0000 \end{bmatrix}$$

- We can create a matrix by using either **`zeros(m,n)`** or **`ones(m,n)`**. The prior will create an  $m \times n$  matrix of 1s; the latter will create an  $m \times n$  matrix of 0s. We typically use **`zeros(m,n)`** to create a matrix that we will *fill in*. We usually use **`zeros(m,n)`** instead of **`ones(m,n)`** because it uses less storage. See the next example.

**Example: Filling in a matrix with values.** DISCLAIMER: There is a more efficient method to fill in a matrix with function values (using `meshgrid` and function handles), but that is explained later on. For now, we use for loops. Let  $f(x,y) = x + y$ . Compute  $f(x,y)$  for the  $x$ -values  $\{1, 2, 3, 4\}$  and the  $y$ -values  $\{-2, -1, 0\}$ . Put these values of  $f(x,y)$  into a  $4 \times 3$  called **`fvals`**, where the rows correspond to the  $x$ -values and the columns correspond to the  $y$ -values.

We will need to create two vectors holding the  $x$ -values and  $y$ -values. Then, we will fill in the matrix **`fvals`** by using two for loops.

```
fvals = zeros(4,3); %the values for f(x,y)=x+y
xvals = [1, 2, 3, 4];
yvals = [-2, -1, 0];

for i = 1:4 %for each x-value (*note: there are 4 x-values)
    x = xvals(i); %the i-th entry in xvals
    for j = 1:3 %for each y-value (*note: there are 3 y-values)
        y = yvals(j); %the j-th entry in yvals
        fvals(i,j) = x + y;
    end
end
```

```
>> fvals

fvals =

    -1     0     1
     0     1     2
     1     2     3
     2     3     4

fx >>
```

**Transposes of Vectors and Matrices.** We often work with transposes of vectors and matrices. Or rather, perhaps in a code we find that a transpose is necessary. This can be done using an apostrophe `'`. The same rule applies to transposing matrices. See below for an example.

**Example: Transpose of a Vector.**

In MATLAB<sup>®</sup> we type `vector = [1, 2, 3]` to create a *row* vector. If we then type `vector'` we get the following output:

$$\mathbf{vector}' = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Notice that the apostrophe `'` has transposed the *row* vector into a *column* vector.

**Pulling out elements of a vector or matrix.** Let's say we define a  $5 \times 1$  column vector called **vec** in MATLAB<sup>®</sup>, and we want to pull out the third entry. We say **vec(3)** to pull out that value.

Similarly, say we define a  $5 \times 4$  matrix called **mat** in MATLAB<sup>®</sup> and want to pull out the entry in the third row and second column. Then we say **mat(3,2)** to pull out that value.

Let's say we want to pull out the  $i$ th row of a matrix called **mat**. Then we say **mat(i,:)**. Similarly, if we want to pull out the  $j$ th column of a matrix called **mat**, then we say **mat(:,j)**.

See figure 13 for an example.

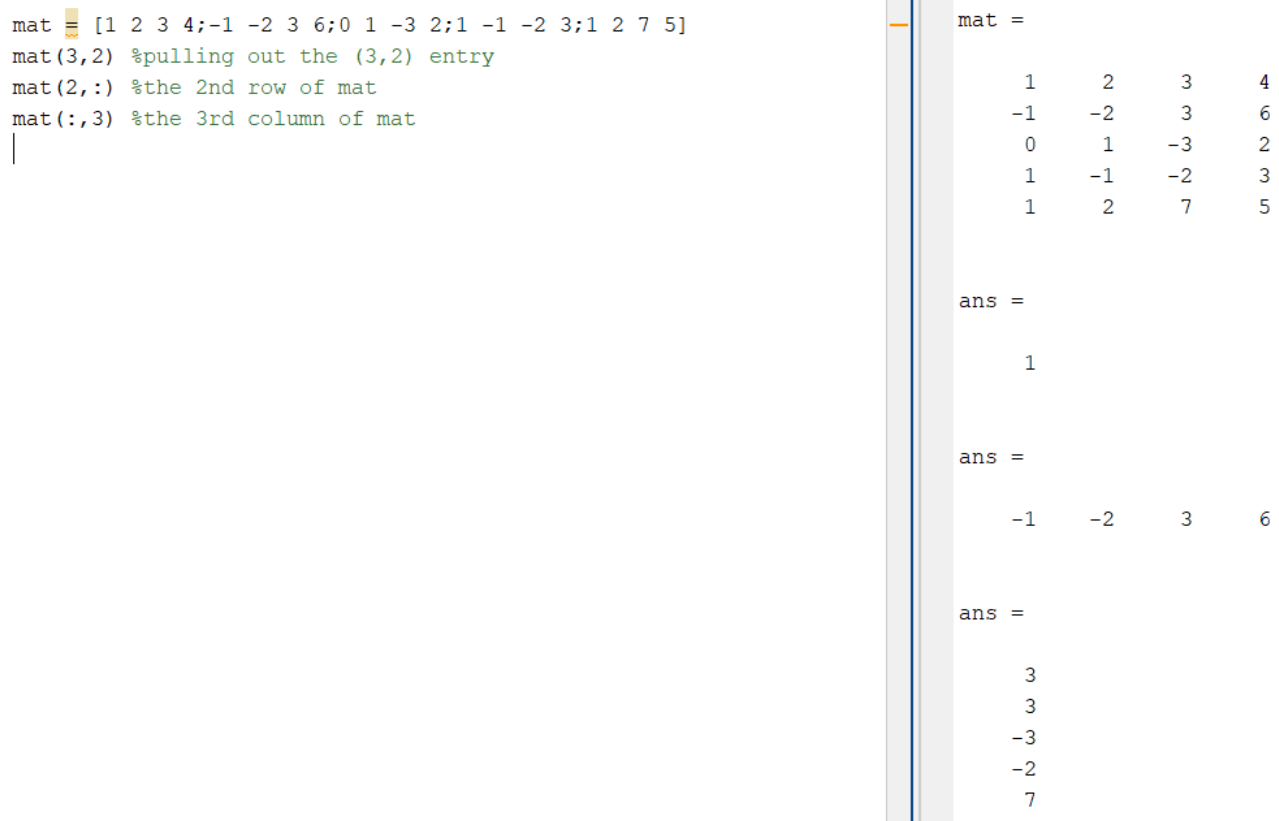


Figure 13: Pulling out elements of a matrix.

**Higher Dimensional Arrays.** What if we want to work with “matrices” of higher dimension? These are called **arrays** (not to be confused with **cell arrays**, which are described in the next section). A rank 1 array is a vector; these have one dimension, i.e.,  $a_i = a(i)$ . A rank 2 array is a matrix; these have two dimensions, i.e.,  $a_{i,j} = a(i,j)$ . A rank 3 array is geometrically like a box, i.e.,  $a_{i,j,k} = a(i,j,k)$  where there are three components. Higher rank arrays follow similarly.

These can be constructed the same way we made a matrix. If you want a  $p \times q \times r$  sized array (i.e., a “box” with dimensions  $p$ -by- $q$ -by- $r$ ), then we can use **zeros(p,q,r)** or **ones(p,q,r)**.

**Two Types of Multiplication.** In MATLAB<sup>®</sup> there are two types of multiplication between vectors/matrices/arrays. If we want to use **normal vector/matrix multiplication**, then we simply use an asterisk `*` for the multiplication. Note that the matrix dimensions must make sense.

For example, we can perform matrix multiplication on  $\mathbf{A} \in \mathbb{R}^{3 \times 5}$  and  $\mathbf{B} \in \mathbb{R}^{5 \times 4}$  because the number of columns of  $\mathbf{A}$  matches the number of rows of  $\mathbf{B}$ . However, this would not work if  $\mathbf{B} \in \mathbb{R}^{4 \times 4}$ . To see this, notice below that the dimensions don't match.

MATRIX DIMENSIONS DO NOT AGREE!

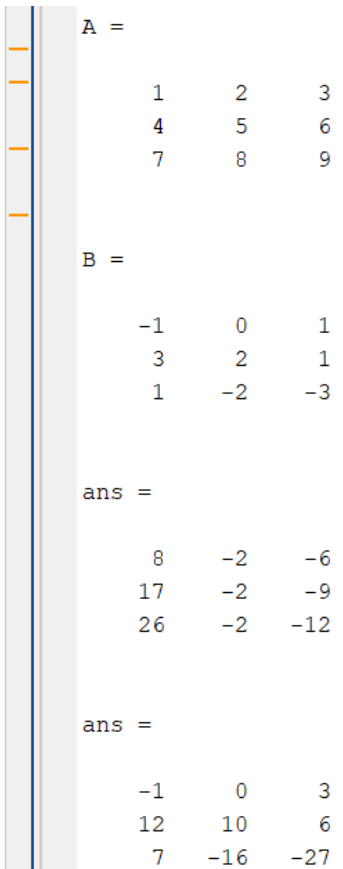
$$\begin{bmatrix} 1 & -1 & 2 & 5 & 0 \\ -4 & 6 & 7 & 2 & 5 \\ -3 & 3 & 2 & 1 & 1 \end{bmatrix}_{3 \times 5} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 3 & 2 & 0 \\ -1 & -2 & 4 & 3 \\ 1 & 1 & 2 & 1 \end{bmatrix}_{4 \times 4}$$

The other type of multiplication (and division) is **component-wise multiplication/division**. If we have two matrices of the *same size* and want to multiply like-entries (i.e., imagine laying these matrices on top of each other and performing multiplication on the overlapping entries), then we must use `.*` where a period goes right before the asterisk. Similarly, we use `./` for component-wise division. We do an example next.

**Example: Matrix Multiplication vs. Component-wise Multiplication.**

```
A = [1 2 3;4 5 6;7 8 9]
B = [-1 0 1;3 2 1;1 -2 -3]

A*B %matrix multiplication (only need *)
A.*B %component-wise multiplication (need .*)
```



The figure shows the MATLAB command window output for the example. It displays the definition of matrices A and B, followed by the results of two multiplication operations. Matrix A is a 3x3 matrix with values [1, 2, 3; 4, 5, 6; 7, 8, 9]. Matrix B is a 3x3 matrix with values [-1, 0, 1; 3, 2, 1; 1, -2, -3]. The first operation, A\*B, performs standard matrix multiplication, resulting in a 3x3 matrix with values [8, -2, -6; 17, -2, -9; 26, -2, -12]. The second operation, A.\*B, performs component-wise multiplication, resulting in a 3x3 matrix with values [-1, 0, 3; 12, 10, 6; 7, -16, -27].

Figure 14: Matrix multiplication vs. component-wise multiplication.

---

## EXERCISES

**Exercise 1.** Refer to the example where we filled in a matrix with values. We will do a very similar problem. Be sure to do parts (a) and (b) in the same file.

(a) Using `linspace(a,b,n)`, create two vectors called `xvals` and `yvals` such that: the  $x$ -values consist of 5 equally spaced points from 0 to  $2\pi$ ; and the  $y$ -values consist of 3 equally spaced points from 0 to  $\pi$ .

(b) Consider the function  $f(x, y) = \sin(x + y)$ . Create a  $5 \times 3$  matrix called `fxnvals`. Fill in the matrix such that the  $(i, j)$  entry of `fxnvals` is  $f(x, y)$  evaluated at the  $i$ th entry of `xvals` and the  $j$ th entry of `yvals`.

*\*NOTE: Type in `pi` for the number  $\pi$ . Type `sin( )` for  $\sin( )$  in MATLAB<sup>®</sup>.*

*Answer: The output for `fxnvals` should be*

$$\mathbf{fxnvals} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

**Exercise 2.** Define the *column* vectors  $\mathbf{a} = \langle 1, 2, 3 \rangle^T$  and  $\mathbf{b} = \langle -1, 0, 2 \rangle^T$ . Using transposes, vector multiplication, and/or component-wise multiplication, compute the following:

- (a) The dot product  $\mathbf{a} \cdot \mathbf{b}$ .
- (b) The matrix  $\mathbf{ab}^T$ . What is the second column of this matrix?
- (c) The *row* vector  $\mathbf{c}$  whose entries are  $c_i = a_i b_i$  for  $i = 1, 2, 3$ .

**Exercise 3.** Define the vectors `xvals = linspace(0,1,11)` and `yvals = linspace(1,2,11)`. Consider the function  $f(x, y) = xy - 0.5y$ . Write a MATLAB<sup>®</sup> code that calculates the number of positive values when evaluating  $f(x, y)$  over all points `(xvals(i),yvals(j))`. There are many ways to do this.

*Answer: 55*

**Exercise 4.** Consider the function  $f(x, y, z) = x + y + z$ . Create three vectors `xvals`, `yvals`, and `zvals` all containing 11 equally spaced points from 0 to 1. Create a rank 3 array called `fxnvals` such that the `(i,j,k)` entry is  $f(x_i, y_j, z_k)$ .

(a) Print the matrix of values of `fxnvals` for which  $z = 0$ . In other words, print the matrix of values corresponding to  $f(x_i, y_j, z_1 = 0)$ .

*\*hint: This corresponds to fixing the  $z$ -value at the first element of `zvals`.*

(b) Print the vector of values of `fxnvals` for which  $y = 0$  and  $z = 0$ . In other words, print

the vector of values corresponding to  $f(x_i, y_1 = 0, z_1 = 0)$ .

*\*hint: This corresponds to fixing the  $y$ -value at the first element of **yvals** and fixing the  $z$ -value at the first element of **zvals**.*

## Cell Arrays

When writing either long or complex scripts in MATLAB<sup>®</sup> it can be easy to find yourself defining a large amount of variables to save various things such as matrices, vectors, etc. While this will not directly cause an issue with the computation, it can be quite easy to lose track of what is defined. This can also create issues of ease of implementation in loops.

For example, lets say in a script you are writing you need to save 10 separate matrices  $\mathbf{A}_i$  with 10 separate corresponding vectors  $\mathbf{v}_i$  as well as the computation  $\mathbf{v}_i^T \mathbf{A}_i \mathbf{v}_i$ . Naively you may just create 10 separate matrix variables, 10 separate vector variables, and 10 separate product variables. The first issue you may see is that now you have 30 variables saved that you have to worry about. A big issue you may or may not see is what if you need to loop through all the matrices and vectors? Since they all have a different variable name, this becomes a problem. Lucky for us, there is a way to store all this information in a single place that we can easily call from like we do with matrices.

**Cell Arrays:** In MATLAB<sup>®</sup> a cell array allows us to store various types of data in an array. That is, we can store matrices of different sizes, vectors, strings, etc. all in one place under one variable name. A cell array functions like a matrix however the way you add entries and pull entries is different. **You must use `{}` when working with cell arrays.**

For example, if you have three matrices  $\mathbf{A}_1$ ,  $\mathbf{A}_2$ , and  $\mathbf{A}_3$  of different sizes and you want to put them all in a cell array named **Matrices** you would enter

$$\mathbf{Matrices} = \{\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3\}$$

and if you want to pull the second matrix out you would use the command

$$\mathbf{Matrices}\{2\}$$

In this example, only matrices were used but cell arrays can store many other types of variables together.

**Example 1.** If we have a matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and we wish to store the matrix, the determinant of  $\mathbf{A}$ , as well as a string of the matrix name  $\mathbf{A}$  in one variable named **Ainfo** we can use the cell array to do this. All we need to do is construct the cell array as you would a normal array but use `{}` as opposed to `[]`.

$$\mathbf{Ainfo} = \{\mathbf{A}, -2, 'A'\}$$



In MATLAB<sup>®</sup> the output of this command is given below. We can then store as well as access

```
>> Ainfo = {A,-2,'A'}

Ainfo =

1x3 cell array

    {2x2 double}    {[-2]}    {'A'}
```

Figure 15: Example 1. Saving the variable

all the information saved by calling on **Ainfo**, e.g. if we need to get the matrix **A** then we can simply type **Ainfo{1}**.

```
>> Ainfo{1}

ans =

     1     2
     3     4
```

Figure 16: Example 1. Calling the Matrix

---

## EXERCISES

**Exercise 1.** Consider the two vectors

$$\mathbf{u} = \begin{bmatrix} 4 \\ 3 \\ -2 \\ 7 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 1 \\ -4 \\ 0 \\ 3 \end{bmatrix}$$

Create a cell array named **P** which stores the information **u**, **v**, **uv<sup>T</sup>**, and **u<sup>T</sup>v**.

**Exercise 2.** Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 1/2 & -1/4 \\ 1 & 1/3 \end{bmatrix}$$

(a) Compute the determinant of **A** and create a cell array named **D** containing **A** as well as the determinant of **A**.

(b) Write a for loop over *i* ranging from 2 to 5 that will calculate **A<sup>i</sup>** and the determinant of **A<sup>i</sup>** as well as store this information in **D** such that **D{*i*, 1}** = **A<sup>i</sup>** and **D{*i*, 2}** = **det(A<sup>i</sup>)**.

# Debugging and Break Points

As your scripts become longer and more complicated, chances are at some point you will make an error or a typo. This can either result in your script not running at all and giving you an error, or the more frustrating situation of your script running and outputting nonsense. In the first situation, MATLAB<sup>®</sup> will usually tell you where your script failed with the reason for failure. However, in the second situation, you will be able to use Break Points to aid you in the search for your error.

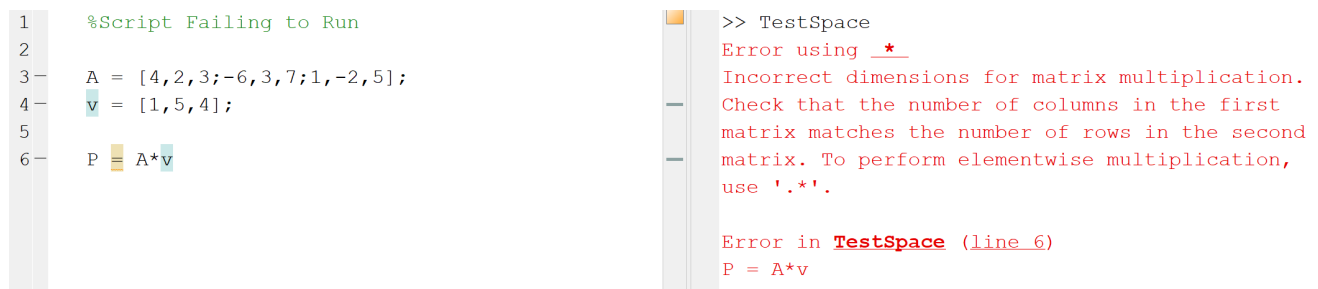
**Scripts Failing to Run** Generally speaking, if your script is failing to run to completion, there is more likely than not either a mathematical error or a syntax error. The way MATLAB<sup>®</sup> will inform you of this is by an audible "ding" followed by a red error message in the command window. The format that these errors will be output will give you a path to search.

The most recent output in the error message will give the location in the script where the script failed to run. Error messages prior to this will either be further locations where the error could be located or a message stating what the error was that stopped the script.

**Example 1.** If we have the matrix and vector

$$\mathbf{A} = \begin{bmatrix} 4 & 2 & 3 \\ -6 & 3 & 7 \\ 1 & -2 & 5 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 1 \\ 5 \\ 4 \end{bmatrix}$$

and we wish to compute  $\mathbf{A}\mathbf{v}$  but we accidentally enter  $\mathbf{v}$  as a row vector then the output is seen below. As we can see, the last output message tells us the error occurred on line 6 which is



```
1 %Script Failing to Run
2
3 A = [4,2,3;-6,3,7;1,-2,5];
4 v = [1,5,4];
5
6 P = A*v

>> TestSpace
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 TestSpace (line 6)
P = A*v
```

Figure 17: Example 1, Error Output.

where we attempted to compute  $\mathbf{A}\mathbf{v}$ . Above this error we see the explanation for the error being incorrect dimensions. This tells us the either the matrix or the vector has an error in how it was input which in this case,  $\mathbf{v}$  should be a column vector.

Now that we know what the errors that MATLAB<sup>®</sup> will generally look like we can start to talk about the main tool of this section.

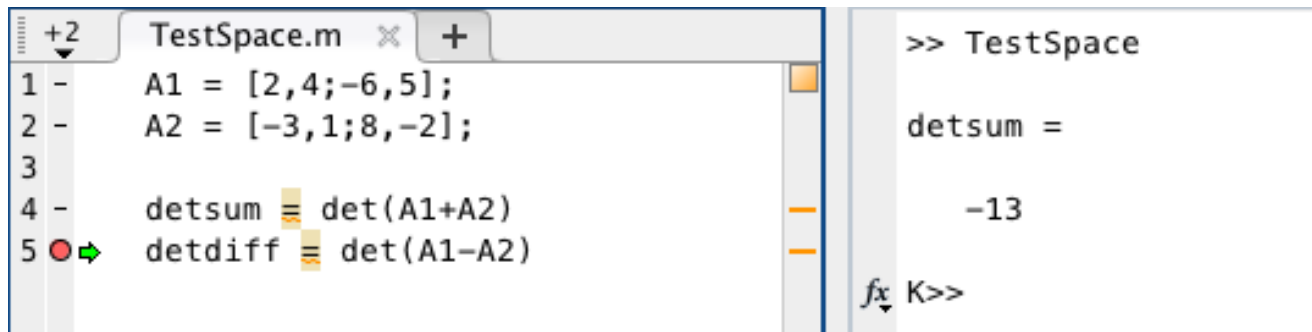
**Non-Conditional Break Points** In longer scripts it may be beneficial to add in break points even if there is no error but for the time being we will use them to fix broken scripts.

A break point is effectively a “road block” in your script that will halt the computation and allow you to go in a look at computed quantities so far. There are a couple ways to enable these break points. The easiest way is to determine where you would like your break point in the script and click the little dash next to the line number. When you click this, a little red dot will appear where the dash was and this will signify that there is now a break point at this line.

**Example 1.** Let’s say we have a basic script that takes two matrices

$$\mathbf{A}_1 = \begin{bmatrix} 2 & 4 \\ -6 & 5 \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} -3 & 1 \\ 8 & -2 \end{bmatrix},$$

and computes the determinants of  $\mathbf{A}_1 + \mathbf{A}_2$  and  $\mathbf{A}_1 - \mathbf{A}_2$ . To demonstrate how a break point works we will stop the script after  $\det(\mathbf{A}_1 + \mathbf{A}_2)$  is computed but before  $\det(\mathbf{A}_1 - \mathbf{A}_2)$  is computed. The script that will perform this break is shown below.



The screenshot shows the MATLAB script editor for a file named 'TestSpace.m'. The script contains the following code:

```
1 - A1 = [2,4;-6,5];
2 - A2 = [-3,1;8,-2];
3
4 - detsum = det(A1+A2)
5 ● → detdiff = det(A1-A2)
```

The command window on the right shows the output of the script execution:

```
>> TestSpace
detsum =
-13
fx K>>
```

The break point is indicated by a red dot and a green arrow next to line 5 in the script editor.

Figure 18: Example 1, Break Point.

Here we can see that the break point is set to stop the script at line 5 which is signified by the red dot. When the script is run it will do all computations up to the break point. Once it has reached it and stopped a green arrow will appear next to the break point. As we can see in the command window, the quantity “detsum” was computed but the difference was not. We can then continue the script running by either hitting the F5 key or click the “Continue” button that will replace the “Run” button used to run scripts. This will resume the script as if there were no break point to start.

**Conditional Break Points** We can use break points to aid in debugging scripts that we write. To do this we will use the command **dbstop**. How this command works in terms of finding errors is that it will add a break point to the script in the event that some condition is met. For example, if we want the script to stop where an error occurred we can add “dbstop if error” at the beginning of the script.

Other types of conditions can be added using dbstop. One example is that you can add a break point into a loop at a specific number of iterations done. This may be beneficial in the event that you are debugging a script that will run with out errors but produces a result you know is incorrect.

For example, let’s say you have a script called TestSpace.m that contains some code. If you have a loop that begins on line 12 and will iterate 100 times and you want it to stop at the 50<sup>th</sup> iteration, then you can add “dbstop in TestSpace at 12 if n=50”. Here we have assumed that the variable for the iteration is “n” but if you use “i” you can just change the condition in the command.

**Example 2.** We will use the conditional break point to detect an error. Consider the following problem. We are given a matrix **A** and a vector **v**

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 3 \\ 9 & -2 & 4 \\ 1 & -1 & 1 \end{bmatrix}, \quad \mathbf{v} = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

and we need to compute a cell array “MatrixInfo” that holds some information. MatrixInfo{1, i} =  $\mathbf{A}^i$ , MatrixInfo{2, i} =  $\det(\mathbf{A}^i)$ , and MatrixInfo{3, i} =  $(\mathbf{v}^T \mathbf{A}^i \mathbf{v})^{1/i}$  for  $i$  ranging from 1 to 10. Below is a script to complete this problem that is run with the “dbstop if error” command included.

```

1- dbstop if error
2-
3- A = [4,1,3;9,-2,4;1,-1,1];
4- v = (1/sqrt(3))*[1,1,1];
5-
6- for i = 1:10
7-     MatrixInfo{1,i} = A^i;
8-     MatrixInfo{2,i} = det(A^i);
9-     MatrixInfo{3,i} = (v'*(A^i)*v)^(1/i);
10- end

```

```

>> TestSpace
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 TestSpace (line 9)
    MatrixInfo{3,i} = (v'*(A^i)*v)^(1/i);

```

Figure 19: Example 2, Error Break Point.

As we can see in Figure 21, when the script was run, a break was added at a line where an error occurred. Here the issue was in line 9 doing the computation of  $(\mathbf{v}^T \mathbf{A}^i \mathbf{v})^{1/i}$ . Checking this will show that we entered the formula correctly so the error must come from one of the quantities of line 9. That is, there must be a problem with either **v** or **A**. Looking into the error we see that **v** was entered as a row vector and not a column vector. Making this fix in line 4 and running the script again will result in a successful output.

**Warning:** The “dbstop if error” command will stop the script at the first line where there is an error. This however does not mean that the line stopped on contains the error that needs to be fixed. You may need to backtrack the quantities to find the error as seen in example 2.

**Example 3.** You are given a summation definition for  $\pi$

$$\pi = \sum_{n=1}^{\infty} \frac{2^n ((n-1)!)^2}{(2n-1)!}$$

and need to write a script that will determine the number of terms  $k$  needed to approximate  $\pi$  so that  $|\pi_k - \pi| < 10^{-3}$ . Here  $\pi_k$  is the truncated summation above where  $n$  will range from 1 to  $k$ . Below is a script that was written to do this that will have an error.

```

1- approx = 0;
2- n = 1;
3
4- while abs(approx - pi) > 1e-3
5-     approx = approx + (2^n*factorial((n-1)^2))/factorial(2*n-1);
6-     n = n+1;
7- end

```

Figure 20: Example 3, Incorrect Formula

Here what this script is doing is using a while loop to keep going until the approximation is below the allowed error. However, if this script is run it will never stop. This is an issue of there being an error in the script but still having it run with out a physical error. This can be investigated by inserting the command “dbstop in TestSpace at (line number) if (some condition)”. Here one option is to stop the while loop when  $n \geq 5$  where 5 was picked arbitrarily as we could use  $n \geq 1$  or  $n \geq 10$ . The altered script is seen below.

<pre> 1- dbstop in TestSpace at 8 if n&gt;=5 2 3- approx = 0; 4- n = 1; 5 6- while abs(approx - pi) &gt; 1e-3 7-     approx = approx + (2^n*factorial((n-1)^2))/factorial(2*n-1); 8-     n = n+1; 9- end </pre>	<pre> K&gt;&gt; n n =     5 K&gt;&gt; approx approx =     1.8450e+09 </pre>
---	---

Figure 21: Example 2, Conditional Break Point.

As we can see that the script stopped when  $n = 5$  and the approximation is no where near  $\pi$  which tells us that we input the formula incorrectly. In fact, in this example the error is that for the numerator,  $((n-1)!)^2$  was entered instead of  $(n-1)!$ . Fixing this will yield that  $k = 12$  if you decide to check.

## EXERCISES

**Exercise 1.** Write a script that constructs a vector  $\mathbf{v}$  via a for loop of length 10 such that the  $i^{\text{th}}$  entry of  $\mathbf{v}$  is given by

$$\mathbf{v}(i) = \sum_{n=1}^i n$$

Include a conditional break point that halts the script when  $v$  is length 5.

**Exercise 2.** Below is a script as well as the output with the "dbstop if error" command that takes the vector

$$\mathbf{v} = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10]$$

and creates a new vector **prodv** by the following procedure.

$$\mathbf{prodv}(1) = \mathbf{v}(1)$$

$$\mathbf{prodv}(i) = \mathbf{v}(i)\mathbf{v}(i-1)$$

```
1- dbstop if error
2-
3- v = [1,2,3,4,5,6,7,8,9,10];
4- prodv(1) = v(1);
5-
6- for i = 1:10
7-     prodv(i) = v(i)*v(i-1);
8- end
```

Array indices must be positive integers or logical values.

Error in TestSpace (line 7)  
prodv(i) = v(i)\*v(i-1);

fx K>>

Figure 22: Exercise 2

Determine what the error that occurred is and provide a fix for this error.

## Function Handles

There are several mathematical functions already built into MATLAB<sup>®</sup> that we can conveniently use, such as  $\sin(x)$ ,  $\cos(x)$ , and  $\exp(x)$ . But, what if we need to use a function that does not have such a nice form, such as  $f(x) = (\sqrt[3]{x} + 2x^x) \sin(2x^2 + 1)$ ? This is where **function handles** come in handy. For the sake of the typical engineering/science/math major, we shall think of function handles as ways to implement an arbitrary function. The general layout is as follows:

$$\text{FUNCTION NAME} = @(x_1, x_2, \dots, x_n) f(x_1, x_2, \dots, x_n)$$

where FUNCTION NAME is the name that you choose to call the function,  $(x_1, x_2, \dots, x_n)$  are the variables/inputs for the function, and  $f(x_1, x_2, \dots, x_n)$  is the function itself. If you've ever used the online calculator Wolfram Alpha (or Mathematica), the syntax for coding the function is very similar.

\*NOTE: We must be careful with the type of multiplication/division (see previous sections).

If the inputs are all scalars, then regular multiplication and division (i.e., `*` and `/`) is fine. However, if your inputs are vectors and matrices, then you must decide whether to use matrix/vector multiplication or component-wise multiplication/division. We show a few examples.

**Example 1.** Consider the function

$$f(x) = (\sqrt[3]{x} + 1) \sin(x^2 + x + 1).$$

(a) Compute  $f(\pi/3)$ . As seen in figure 23, we have called this function  $f$ .

```
>> f = @(x) (x^(1/3) + 1)*sin(x^2+x+1)

f =

    function\_handle with value:

    @(x) (x^(1/3)+1)*sin(x^2+x+1)

>> f(pi/3)

ans =

    -0.0045
```

Figure 23: Example 1a.

(b) Compute  $f(x)$  for the values  $x = 0, \pi/4, \pi/2, 3\pi/4, \pi$  using a for loop. We just include the code in figure 24, not the output.

```
f = @(x) (x^(1/3) + 1)*sin(x^2+x+1);
xvals = 0:pi/4:pi;
for i = 1:numel(xvals)    %numel gives you the number of elements
    f(xvals(i))          %in an array, which in this case xvals
end                       %has 5 elements.
```

Figure 24: Example 1b.

(c) Repeat part (b), but this time use a vector. Notice that now we try putting **xvals** into **f**. We end up getting an error. Why? Looking at where we typed the actual mathematical function, MATLAB<sup>®</sup> is reading the multiplication as vector multiplication, *not* component-wise multiplication. This about it – what would it even mean to take a 1/3 power of a vector?

Figure 25 shows the INCORRECT version. Figure 26 shows the CORRECT version. To make it clearer what the component-wise multiplication is doing in figure 26, we hand-wavily think

of it like this:

$$\left( \begin{bmatrix} 0^{1/3} \\ (\pi/4)^{1/3} \\ (\pi/2)^{1/3} \\ (3\pi/4)^{1/3} \\ \pi^{1/3} \end{bmatrix} + 1 \right) \cdot \sin \left( \begin{bmatrix} 0^2 \\ (\pi/4)^2 \\ (\pi/2)^2 \\ (3\pi/4)^2 \\ \pi^2 \end{bmatrix} + \begin{bmatrix} 0 \\ \pi/4 \\ \pi/2 \\ 3\pi/4 \\ \pi \end{bmatrix} + 1 \right) = \begin{bmatrix} 0^{1/3} + 1 + \sin(0^2 + 0 + 1) \\ (\pi/4)^{1/3} + 1 + \sin((\pi/4)^2 + \pi/4 + 1) \\ (\pi/2)^{1/3} + 1 + \sin((\pi/2)^2 + \pi/2 + 1) \\ (3\pi/4)^{1/3} + 1 + \sin((3\pi/4)^2 + 3\pi/4 + 1) \\ \pi^{1/3} + 1 + \sin(\pi^2 + \pi + 1) \end{bmatrix}$$

```

1 - f = @(x) (x^(1/3) + 1)*sin(x^2+x+1);
2 - xvals = 0:pi/4:pi;
3 - f(xvals)
4

```

```

>> test
Error using ^ (line 51)
Incorrect dimensions for raising a matrix to a power.
Check that the matrix is square and the power is a
scalar. To perform elementwise matrix powers, use
'.'.

Error in test>@(x) (x^(1/3)+1)*sin(x^2+x+1) (line 1)
f = @(x) (x^(1/3) + 1)*sin(x^2+x+1);

Error in test (line 3)
f(xvals)

```

Figure 25: Example 1c, incorrect version.

```

1 - f = @(x) (x.^(1/3) + 1).*sin(x.^2+x+1);
2 - xvals = 0:pi/4:pi;
3 - f(xvals)
4

```

```

>> test

ans =

    0.8415    1.2955   -2.0487    1.1519    2.4451

fx >> |

```

Figure 26: Example 1c, correct version.

**Example 2.** Consider the function

$$f(x, y) = e^{-(x^2+y^2)}.$$

(a) Compute  $f(3, 3)$ . This is shown in figure 27.

(b) Compute  $f(x, y)$  for all  $x = 0, 1/3, 2/3, 1$  and  $y = 0, 1/2, 1$  using arrays; we could also have done this using for loops. As seen in figure 28, the columns correspond to the  $x$ -values and the rows correspond to the  $y$ -values. For example, the second row and third column is  $f(1/2, 2/3)$ .  
*\*Note: This problem requires using a function called **meshgrid**. We discuss **meshgrid** more in-depth in the section “Plotting Functions of One and Two Variables.” Please quickly go to this section and read what **meshgrid** does. Sorry for this slight inconvenience!*



```

>> f = @(x,y) exp(-(x^2+y^2))

f =

function_handle with value:

@(x,y) exp(-(x^2+y^2))

>> f(3,3)

ans =

1.5230e-08

```

Figure 27: Example 2a.

```

f = @(x,y) exp(-(x.^2+y.^2));
xvals = linspace(0,1,4);
yvals = linspace(0,1,3);
[X,Y] = meshgrid(xvals,yvals);
f(X,Y)

```

ans =	1.0000	0.8948	0.6412	0.3679
	0.7788	0.6969	0.4994	0.2865
	0.3679	0.3292	0.2359	0.1353

Figure 28: Example 2b, using `meshgrid`.

**Example 3.** Consider the function  $f(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \mathbf{y}^T \mathbf{x}$ , where  $\mathbf{x}$  and  $\mathbf{y}$  are column vectors. Compute  $f(\mathbf{x}, \mathbf{y})$  for the column vectors  $\mathbf{x} = \langle 1, 2, 3 \rangle^T$  and  $\mathbf{y} = \langle -1, 2, -3 \rangle^T$ . It is important that we have column vectors as the inputs. Otherwise, the matrix/array dimensions will not agree.

```

f = @(x,y) y'*x; %x,y are column vectors
xvec = [1 2 3]'; %make column vector
yvec = [-1 2 -3]'; %make column vector
f(xvec,yvec)

```

ans =	-6
-------	----

Figure 29: Example 3.

---

## EXERCISES

**Exercise 1.** Create a function handle called `fxn` for the function

$$f(x) = x^2 + x + 1.$$

- (a) Compute  $f(2)$ .
- (b) Compute  $f(x)$  for  $x = 0, 0.2, 0.4, 0.6, 0.8, 1$  using for loops.
- (c) Repeat part (b) using component-wise multiplication by modifying your function handle.

**Exercise 2.** Recall the scalar projection of a vector  $\mathbf{b}$  onto a vector  $\mathbf{a}$  is defined by

$$\text{comp}_{\mathbf{a}}\mathbf{b} = \frac{\mathbf{b} \cdot \mathbf{a}}{\|\mathbf{a}\|}$$

Create a function handle called **compba** that takes column vectors  $\mathbf{a}$  and  $\mathbf{b}$  as the inputs and outputs the scalar projection of  $\mathbf{b}$  onto  $\mathbf{a}$ . Using this function handle, compute the scalar projection of  $\mathbf{b} = \langle 3, -5, 2 \rangle^T$  onto  $\mathbf{a} = \langle -1, 2, -1 \rangle^T$ .

*\*Note:  $\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$ . However, you can also use the MATLAB<sup>®</sup> function **norm**( ) if you want. We discuss this function in a later section.*

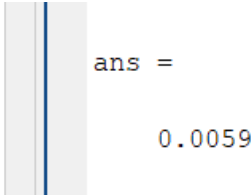
**Exercise 3.** Referring to the figure below, something is off. We want to compute  $f(x)$  for  $x = 0, 1/9, 2/9, \dots, 8/9, 1$ , given the function

$$f(x) = \frac{x^2 + x - 1}{x^2 + 1}.$$

Although the code runs and gives an output without an error, something is clearly not right. We expect a vector of values. But for some reason, MATLAB<sup>®</sup> is spitting out just a single number. Investigate this and figure out what the issue is. Then, fix the code. The correct output is

$$\left[ -1.0000 \quad -0.8659 \quad -0.6941 \quad -0.5000 \quad -0.2990 \quad -0.1038 \quad 0.0769 \quad 0.2385 \quad 0.3793 \quad 0.5000 \right]$$

```
f = @(x) (x.^2 + x - 1)/(x.^2 + 1);
xvec = linspace(0,1,10);
f(xvec)
```



ans =  
0.0059

## Plotting Functions of One and Two Variables

Before starting this section, the reader should read the section on function handles. Although we can use for loops to store function values, sometimes evaluate vectors/matrices/arrays in function handles saves computational time.

**Plotting Functions of One Variable.** For simplicity, consider plotting a function  $f(x)$  for some  $x$ -values. Say the  $x$ -values are stored in a vector called **xvals** and the *corresponding* functions values are stored in a vector called **fvals**. Note that you can call these vectors whatever you want; these names are just for the sake of this demonstration. Then, we can plot this function using the (basic) command

**plot(xvals,fvals)**

Similarly, you can do a scatter plot of the data points using the command

**scatter(xvals,fvals)**

However, what if you want to label the axes, fix a range of values, color the plot, change the thickness, etc.? You will need to modify these commands. We offer a few *basic* plotting attributes that are nice to know. More specific attributes can be found online or in Stormy Attaway's book. Using the attributes listed here, please refer to the next example and figure 30. Other attributes are shown in figure 30.

- **Colors.** The possible colors are: **b** for blue; **c** for cyan; **g** for green; **k** for black; **m** for magenta; **r** for red; **w** for white; and **y** for yellow.
- **Line Types (for plots).** The possible line types are: **-** for solid lines; **--** for dashed lines; **-.** for dash dot lines; and **:** for dotted lines.
- **Plot Markers (for scatter plots).** The possible scatter plot symbols/markers are: **o** for circles; **d** for diamonds; **h** for hexagons; **p** for pentagons; **+** for plus; **.** for points; **s** for squares; **\*** for stars; **v** for down triangles; **<** for left triangles; **>** for right triangles; **^** for up triangles; and **x** for x-marks.

**Example: 1D Plot with Various Attributes.** In this example, we plot the function  $f(x) = \sin(x)$  for  $x$ -values ranging from  $-\pi$  to  $\pi$ . We also showcase several different attributes. A few important notes about this code:

- The start of the code has **clear all** and **close all**. As we already know, **clear all** clears the variables/workspace. But, now that we have figures for the plots, we must also clear the figures (if we choose). It's a good habit to do this at the start of your code, unless you do not want to clear your figures.
- We label each plot with a figure number by typing **figure(#)**; If you don't specify a figure number by typing **figure**; then MATLAB<sup>®</sup> will automatically assign this plot with the next available number.
- Referring to the last figure (with multiple plots), we can plot another function on the same figure by using **hold on**; on the specified figure.
- Again referring to the last figure (with multiple plots), the location of the legend on the figure can be specified by adding **'location','northwest'** to the legend command. Note that there are MANY possible locations: northwest, northeast, southwest, southeast, north, south, etc... You can easily look these up online.

```

close all;
f = @(x) sin(x);
xvals = linspace(-pi,pi,100);
fvals = f(xvals);

% FIGURE 1: A PLAIN PLOT WITHOUT ATTRIBUTES.
figure(1);plot(xvals,fvals);

% FIGURE 2: COLOR, LINE TYPE.
figure(2);plot(xvals,fvals,'b-.');

% FIGURE 3: COLOR, LINE TYPE, LINE THICKNESS.
figure(3);plot(xvals,fvals,'b-.','linewidth',1.5);

% FIGURE 4: COLOR, LINE TYPE, LINE THICKNESS, AXES LABELS, TITLE.
figure(4);plot(xvals,fvals,'b-.','linewidth',1.5);
xlabel('x values'); %x axis label as a char
ylabel('y values'); %y axis label as a char
title('sin(x)'); %title as a char

% FIGURE 5: COLOR, LINE TYPE, LINE THICKNESS, AXES LABELS, TITLE,
% FIXED DOMAIN/RANGE.
figure(5);plot(xvals,fvals,'b-.','linewidth',1.5);
xlabel('x values');ylabel('y values');title('sin(x)');
axis([-pi,pi,-1,1]); %-pi<x<pi and -1<y<1.

% FIGURE 6: PLOTTING MORE THAN ONE FUNCTION, WITH LEGEND.
figure(6);plot(xvals,fvals,'b-.','linewidth',1.5);
hold on;plot(xvals,cos(xvals),'r-.','linewidth',1.5);
legend('sin(x)','cos(x)','location','northwest'); %legend as chars

```

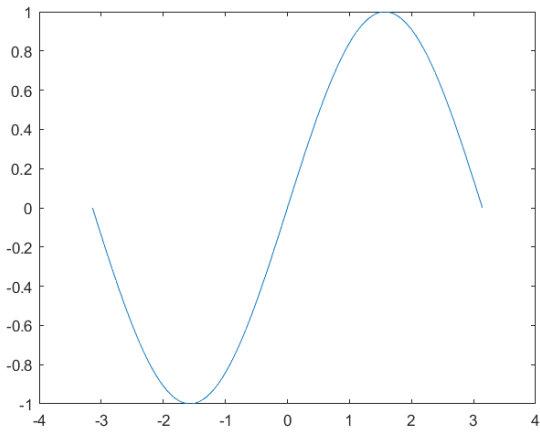
Figure 30: Attributes to plots. Same applies to scatter plots.

**Meshgrid.** The section on function handles provides an example of inputting matrices into a function handle. By now, hopefully it's more or less straightforward to input a vector into a function handle. For instance, if we want to plot  $f(x)$  for a vector of values called **xvec**, then we simply put this vector into the function handle (assuming we have component-wise multiplication/division). But what if we want to plot a function of two variables  $f(x, y)$ ?

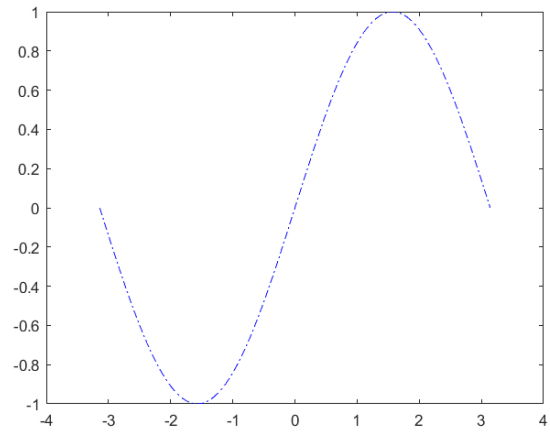
The set-up is as follows:

We have a *vector* of  $x$ -values, say it's called **xvec**. Similarly, we have a *vector* of  $y$ -values, say it's called **yvec**.

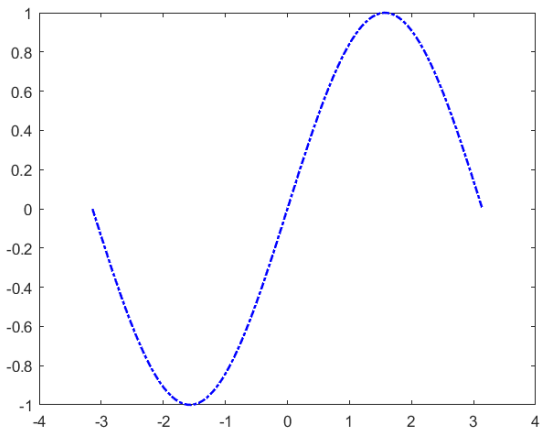
We need all possible combinations of the  $x$  and  $y$  values. For example, if **xvec** = **linspace(0,2,100)** and **yvec** = **linspace(0,1,30)** then there are  $100 \times 30$  possible combinations. We use the built-in MATLAB<sup>®</sup> function **meshgrid** to create two matrices that will be inputs for our function handle. Look at the next example.



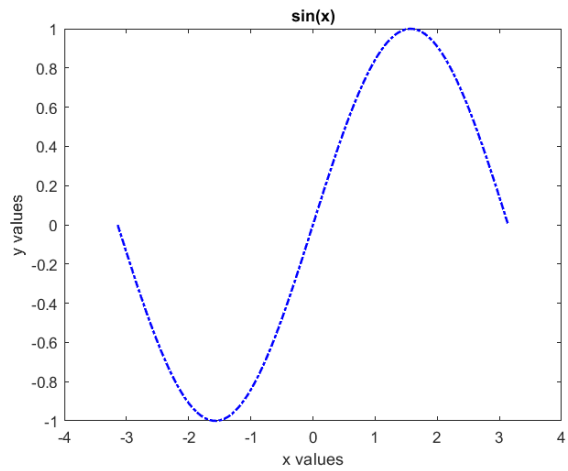
MATLAB<sup>®</sup> Figure 1



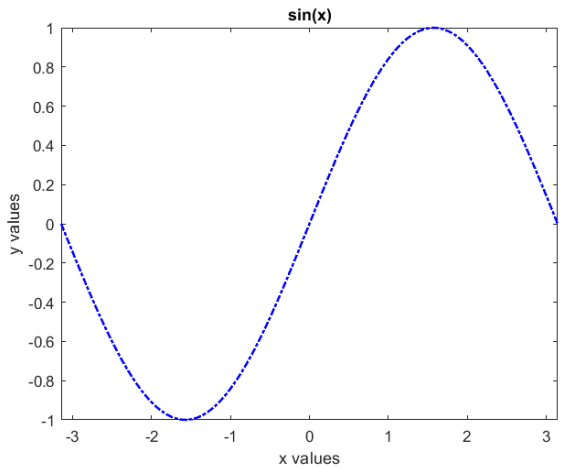
MATLAB<sup>®</sup> Figure 2



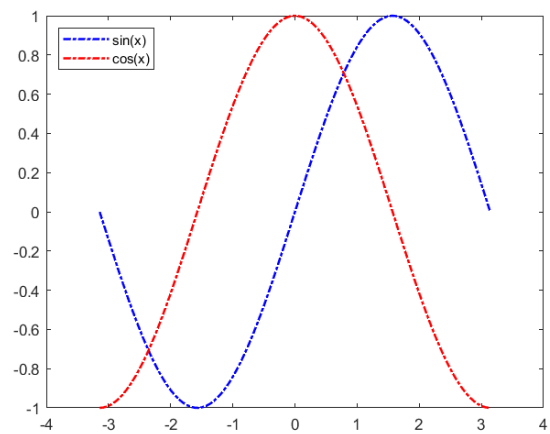
MATLAB<sup>®</sup> Figure 3



MATLAB<sup>®</sup> Figure 4



MATLAB<sup>®</sup> Figure 5



MATLAB<sup>®</sup> Figure 6

**Example: Using meshgrid.** As seen in figure 31, the format of `meshgrid` is as follows:

$$[\mathbf{matrix1}, \mathbf{matrix2}] = \mathbf{meshgrid}(\mathbf{vec1}, \mathbf{vec2})$$

where `matrix1` and `matrix2` are the names we give the matrices for our  $x$  and  $y$  values, respectively. Similarly, `vec1` and `vec2` are the vectors of our  $x$  and  $y$  values, respectively. In this example, we name these matrices `X` and `Y`.

Notice that each *column* of `X` is a different  $x$ -value. And, each *row* of `Y` is a different  $y$ -value. We can see that if we “overlap” these two matrices, then we have all possible combinations and essentially form an  $xy$ -grid.

*\*Note: If you want the  $x$ -values to vary row-by-row in `X` and the  $y$ -values to vary column-by-column in `Y`, then you simply take their transposes. In this case, you’ll need to type*

$$\mathbf{X} = \mathbf{X}' \quad \text{and} \quad \mathbf{Y} = \mathbf{Y}'$$

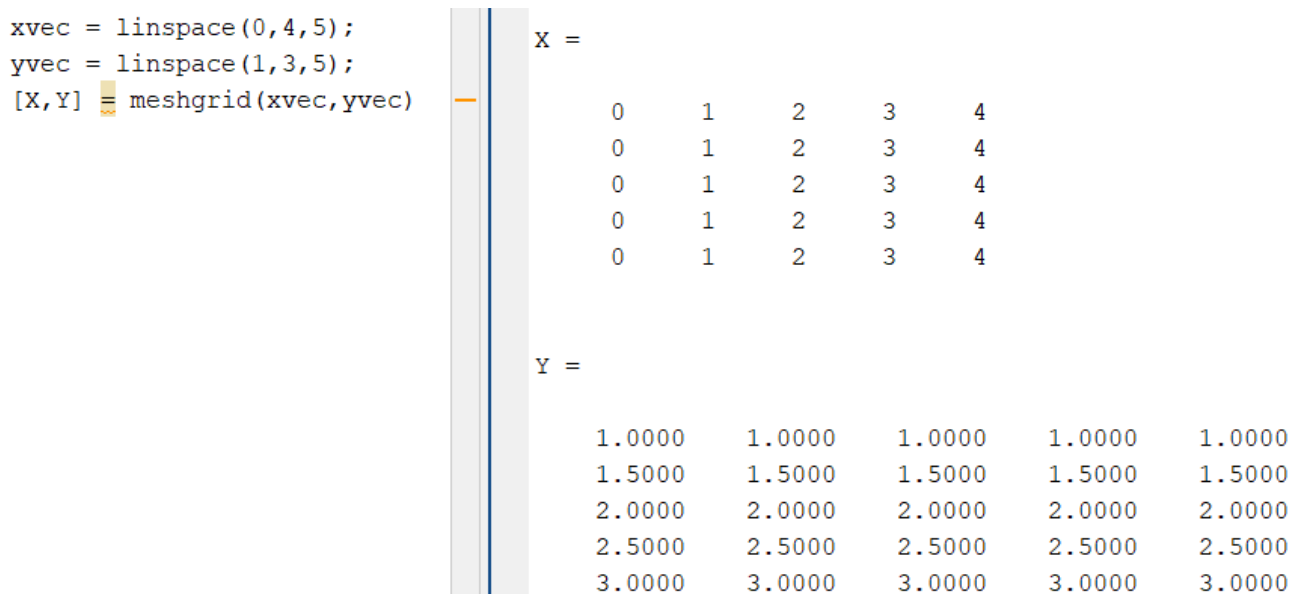


Figure 31: Using meshgrid.

**Example: 2D Plot.** Plotting a function of two variables does not use `plot( )`. Instead, we use

$$\mathbf{surf}(\mathbf{X}, \mathbf{Y}, \mathbf{fvals})$$

where `fvals` is the matrix that holds the function values and `X, Y` are from `meshgrid`. Consider plotting  $\text{Exp}(-(x/2)^2 - y^2)$  for  $-2 \leq x \leq 2$  and  $-1 \leq y \leq 1$ . As in the 1D plot example, we show various attributes in figure 32.

As seen in the last plot (color change), we change the `color(map)` to `winter`. There are many options that can easily be looked up online. We offer a few others: `autumn`, `winter`, `spring`, `summer`, `parula`, `pink`.

*\*Note: Again, there are MANY more options that you have a lot of control over. However, this is somewhat complicated matter and is not a focus of this guide. The colormap options provided here should suffice for the reader's purposes. A detailed elaboration of more colormap options can be found in Stormy Attaway's book or online.*

```
close all;

f = @(x,y) exp(-(x/2).^2 - y.^2);
xvec = linspace(-2,2,200); yvec = linspace(-1,1,100);
[X,Y] = meshgrid(xvec,yvec);

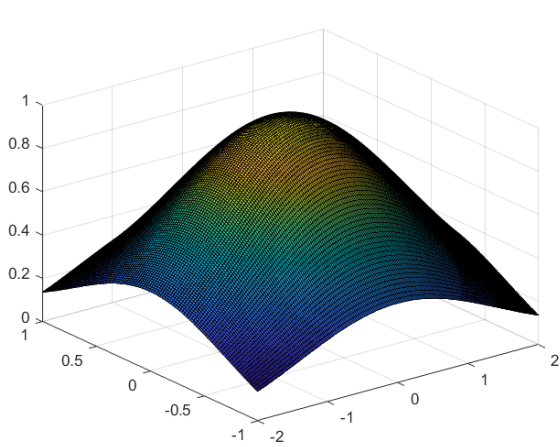
% FIGURE 1: A PLAIN PLOT WITHOUT ATTRIBUTES.
figure(1);surf(X,Y,f(X,Y));

% FIGURE 2: AXES LABELS, TITLE, SPECIFIED VALUES.
figure(2);surf(X,Y,f(X,Y));
xlabel('x');ylabel('y');title('sin(x+y)');
axis([-2,2,-2,2,0,1]); %x values, y values, z values

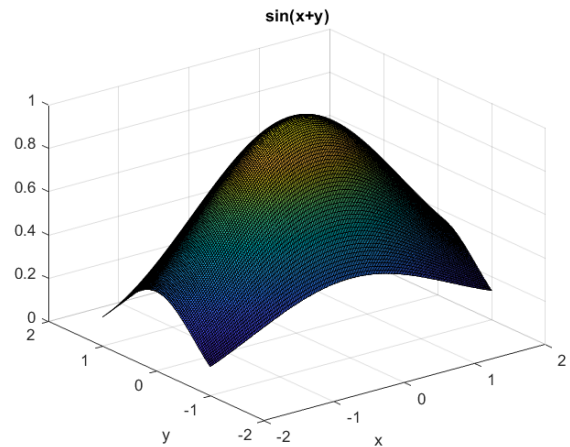
% FIGURE 3: AXES LABELS, TITLE, NO GRID LINES.
figure(3);surf(X,Y,f(X,Y));
xlabel('x');ylabel('y');title('sin(x+y)');
shading interp; %no grid lines

% FIGURE 4: AXES LABELS, TITLE, NO GRID LINES, COLOR CHANGE, SHOW RANGE.
figure(4);surf(X,Y,f(X,Y));
xlabel('x');ylabel('y');title('sin(x+y)');
shading interp;colormap winter; %change the colormap
colorbar; %show a bar of color values
```

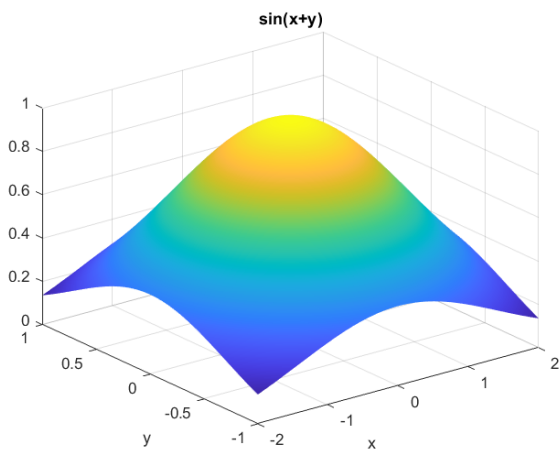
Figure 32: Attributes to surf.



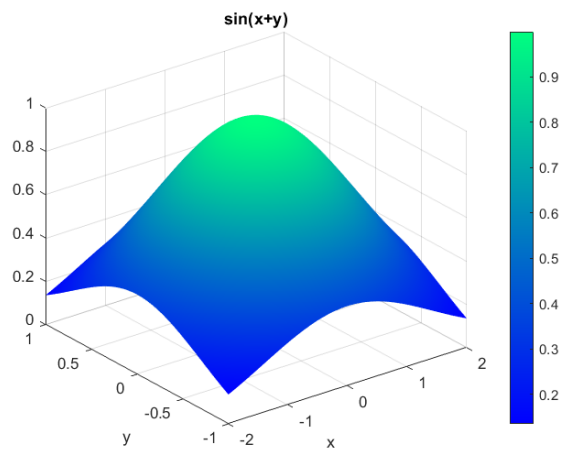
MATLAB<sup>®</sup> Figure 1



MATLAB<sup>®</sup> Figure 2



MATLAB<sup>®</sup> Figure 3



MATLAB<sup>®</sup> Figure 4

---

## EXERCISES

**Exercise 1.** Reproduce figure 33 (with the linewidth set to 1.5). The function shown is known as the zeroth order Bessel function. This can be implemented in MATLAB<sup>®</sup> using function `besselj(0,xvec)` where `xvec` is your vector of  $x$ -values.

**Exercise 2.** Reproduce figure 34 (with the linewidth set to 1.5).

**Exercise 3.** Reproduce figure 35 (with the colormap set to winter).



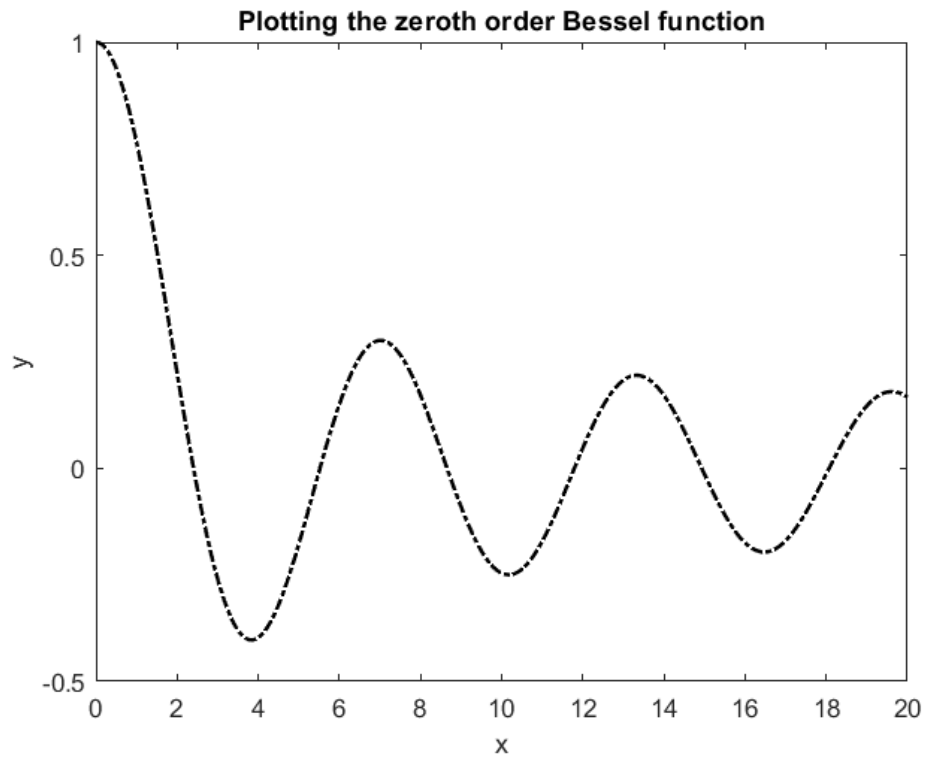


Figure 33: Exercise 1.

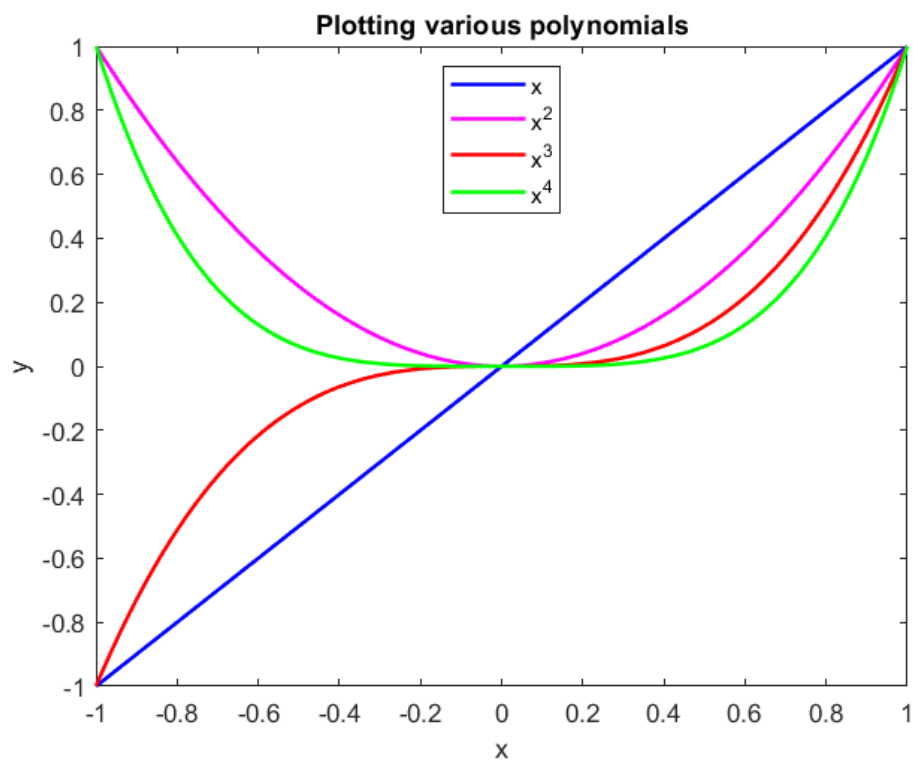


Figure 34: Exercise 2.

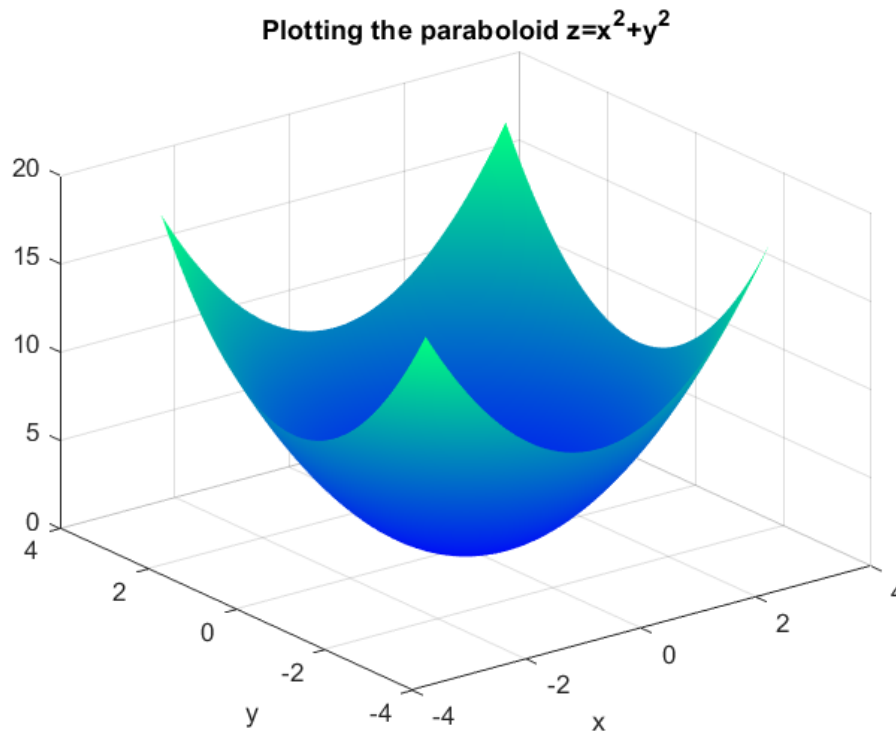


Figure 35: Exercise 3.

## Creating Your Own Functions

There will be times when your code requires several steps. Rather than having one *long* code in a single file, we typically split up the work into several smaller codes that we put together. These smaller codes are often referred as *functions* or *subroutines*. In MATLAB<sup>®</sup> these are called **functions**.

- The smaller codes are called **functions** in MATLAB<sup>®</sup>.
- We typically call the main (script) file that uses these smaller codes/functions the **main file**.
- The **main script file** and all **functions** use in the main script file must be in the same directory/folder!

Splitting up a long code into smaller parts not only simplifies the main script file, but it also allows you to possibly use those smaller parts in other codes. For instance, if you build a smaller code/function that computes the definite integral  $\int_a^b f(x)dx$ , then you can use it whenever you need it! Building functions in MATLAB<sup>®</sup> is best taught through examples, so we will jump straight into them.

**Example 1: Compute the dot product and cross product of two vectors.** Clearly, we could just code this up in a single file. In easier/shorter examples a single file is not too bad. However, as you start coding longer and longer programs, using functions will become important.

Before starting to code, we should first work out the math so that we know what to code. We consider any two length 3 (column) vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \text{and} \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

We know that their dot product is

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v}^T \mathbf{u} = u_1 v_1 + u_2 v_2 + u_3 v_3$$

and their cross product (by definition) is

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$$

When building a MATLAB<sup>®</sup> function we open a completely new file. The first line of the file should read

**function [OUTPUTS] = filename(INPUTS)**

and the last line of the file should read

**end**

Note that **[OUTPUTS]** is simply the outputted variables from this function; **filename** is the name of this MATLAB<sup>®</sup> function; and **INPUTS** are the inputted variables.

*\*Note: This MATLAB<sup>®</sup> function file name must be the same as **filename**.*

As seen in figure 36, we have built a MATLAB<sup>®</sup> function whose filename we've called *dotcross*. The inputs are two column vectors  $\mathbf{u}$  and  $\mathbf{v}$ . The outputs are the dot product and the cross product. When you have more than one output we must put them between brackets. A few notes:

- We DO NOT run MATLAB<sup>®</sup> functions. Remember, these are to be used in a main script file. We run the main script file. Hopefully by now you see why we call these MATLAB<sup>®</sup> *functions*; they are simply codes that take in inputs and spit out outputs...like a *function*.
- Notice that we included a few comments that describe the INPUTS and OUTPUTS. This is simply good manners. Sometimes you'll end up sharing these functions with others so that they can use them. It's polite to tell other users exactly what everything is and how to use your MATLAB<sup>®</sup> function.

Figure 37 shows the **main script file** that uses our MATLAB<sup>®</sup> file. In this main script file, we are asked to do the following: Compute the dot and cross products of the following pairs of vectors:

$$\langle 3, 2, 1 \rangle^T \quad \text{and} \quad \langle 1, 2, 3 \rangle^T$$

$$\begin{aligned} \langle 10, -2, 2 \rangle^T & \quad \text{and} \quad \langle -3, 1, -7 \rangle^T \\ \langle -4, 2, 2 \rangle^T & \quad \text{and} \quad \langle 5, 6, -3 \rangle^T \end{aligned}$$

Notice that we could compute the dot and cross products for each pair all in a single file. But think about how long that code would be! Instead, we can save A LOT of space if we simply use our MATLAB<sup>®</sup> function *dotcross*.

**An important note:** When calling/using a MATLAB<sup>®</sup> function you must follow the same format. In other words, as seen in figures 36 and 37 the format for using the function *dotcross* is

$$[\mathbf{dot}, \mathbf{cross}] = \mathbf{dotcross}(\mathbf{u}, \mathbf{v})$$

where **dot** and **cross** are whatever you want to call the dot and cross products; as seen in the main script file we call them **dot1**, **cross1**, and so forth.

```

1  function [dot,cross] = dotcross(u,v)
2  % INPUTS:
3  %   u = 3x1 column vector
4  %   v = 3x1 column vector
5  % OUTPUTS:
6  %   dot = dot product of u and v
7  %   cross = cross product of u and v as a column vector
8
9  dot = v'*u;
10
11 cross = zeros(3,1); %cross product needs to be a (column) vector
12 cross(1) = u(2)*v(3) - u(3)*v(2);
13 cross(2) = u(3)*v(1) - u(1)*v(3);
14 cross(3) = u(1)*v(2) - u(2)*v(1);
15 end

```

Figure 36: A MATLAB<sup>®</sup> function for computing the dot and cross products of two vectors.

**Example 2: Riemann Sums (Left Rectangles).** Recall from calculus that Riemann sums can be used to approximate definite integrals. In general, we can partition the interval  $[a, b]$  into  $N$  sub-intervals by discretizing in space,

$$a = x_0 < x_1 < x_2 < \dots < x_{N-1} < x_N = b,$$

where the  $i$ -th sub-interval has width  $\Delta x_i = x_i - x_{i-1}$  for  $i = 1, 2, \dots, N$ . By choosing any arbitrary point  $x_{i-1} \leq x_i^* \leq x_i$ , the general Riemann sum states

$$\int_a^b f(x)dx \approx \sum_{i=1}^N f(x_i^*)\Delta x_i.$$

```

main.m x dotcross.m x +
1 - clear all;
2
3 - u1 = [3,2,1]'; v1 = [1,2,3]';
4 - u2 = [10,-2,2]'; v2 = [-3,1,-7]';
5 - u3 = [-4,2,2]'; v3 = [5,6,-3]';
6
7 - [dot1,cross1] = dotcross(u1,v1);
8 - [dot2,cross2] = dotcross(u2,v2);
9 - [dot3,cross3] = dotcross(u3,v3);
10
11 - dot1
12 - cross1'
13 - dot2
14 - cross2'
15 - dot3
16 - cross3'

>> main
dot1 =
    10
ans =
     4     -8     4
dot2 =
   -46
ans =
    12    64     4
dot3 =
   -14
ans =
   -18    -2   -34

```

Figure 37: Main script file for dot and cross products of two vectors.

In particular, we can partition  $[a, b]$  into  $N$  equal sub-intervals. Using **left rectangles** (i.e., choose the left endpoint of each sub-interval by letting  $x_i^* = x_{i-1}$  for  $i = 1, 2, \dots, N$ ) so that

$$\int_a^b f(x)dx \approx \Delta x \sum_{i=1}^N f(x_{i-1}).$$

Create a MATLAB<sup>®</sup> function called *riemannleft* that computes the Riemann sum (for left rectangles). The inputs should be: a function handle for  $f(x)$ ; the number of equally spaced sub-intervals  $N$ ; and the endpoints  $a$  and  $b$ . The output should be the approximate definite integral. Write a main script file that approximates the integral  $\int_0^3 x^2 dx$  using  $N = 20$  sub-intervals. Compare your approximate answer against the exact answer 9. The code is shown in figure 38. *\*Note: The MATLAB<sup>®</sup> function **sum**( ) can be used to add up all of the entries of a vector.*

```

main.m
1 - clear all;
2
3 - f = @(x) x.^2;
4 - a = 0; b = 3;           %endpoints
5 - N = 20;                %number of sub-intervals
6 -                        %i.e., N+1 points in the partition
7 - approx = riemannleft(a,b,N,f)

riemannleft.m
1 - function approxintegral = riemannleft(a,b,N,f)
2 - % INPUTS:
3 - %   a,b = left and right endpoints
4 - %   N = number of equally spaced sub-intervals
5 - %   f = function to integrate
6 - % OUTPUT:
7 - %   approxintegral = Riemann sum with left rectangles
8
9 - xvals = linspace(a,b,N+1);
10 - dx = xvals(2)-xvals(1); %width of each sub-interval
11 - approxintegral = dx*sum(f(xvals(1:N)));
12 - % NOTE: We use xvals(1:N) because we are using left rectangles.
13 - end

Command Window
>> main
approx =
    8.3362
fx >>

```

Figure 38: Riemann sums (left rectangles).

---

## EXERCISES

**Exercise 1.** Modify the code in figure 38 to create a code that computes the Riemann sum using: (a) *right rectangles*; and (b) *midpoint rectangles*. Name these two MATLAB<sup>®</sup> functions *riemannright* and *riemannmid*.

**Exercise 2.** Using exercise 1 we now have MATLAB<sup>®</sup> functions for Riemann sums using left, right, and midpoint rectangles. Combine these three MATLAB<sup>®</sup> functions into a single MATLAB<sup>®</sup> function that outputs all three approximations.

**Exercise 3.**

(a) Repeat exercise 2 from the section “Vectors, Matrices, Arrays, and ‘Two’ Types of Multiplication/Division”. However, code parts (a) through (c) in a single MATLAB<sup>®</sup> function such that it outputs:  $\mathbf{a} \cdot \mathbf{b}$ ,  $\mathbf{a}\mathbf{b}^T$ , and  $\mathbf{c}$ .

*\*Note: Write your code so that you can input any two  $3 \times 1$  vectors.*

(b) Modify part (a) so that you can accommodate *any* two  $n \times 1$  vectors.

**Exercise 4.** The `minmod` function is defined as

$$\text{minmod}(a, b) = \begin{cases} \text{sgn}(a)\min(|a|, |b|), & ab \geq 0, \\ 0, & ab < 0. \end{cases}$$

Write a MATLAB<sup>®</sup> function for the `minmod` function and call it `minmod`. Test this MATLAB<sup>®</sup> function on a some values and see if it gives you the correct answers.

## Saving and Importing Data

As your usage of MATLAB<sup>®</sup> increases, you may find yourself trying to cut run time of scripts that you put together. Also, you might eventually want to transfer data over from another platform. Learning to save and import data to MATLAB<sup>®</sup> can be rather beneficial in multiple aspects.

Starting with saving data, this really has far more options than those stated in this section. However, a few important options will be covered. First, what if we just want to save some type of array to a file that we can then later load into MATLAB<sup>®</sup>?

When saving an array in MATLAB<sup>®</sup> you need to specify what type of file you want it to be saved as. For this guide, we will stick with `.mat` file types, however this is not the only option.

To use the `save` command you will first need to make sure the object that you wish to save is in the workspace. To save the object you will use the command

`save('filename.mat', 'variable')`

Here, `filename.mat` is whatever you wish to have the saved file named and `variable` is the name of the physical object you have in the workspace. When this is run, a file with the name you picked will be saved to the folder in which you are working in.

**Note:** This command will work with an array of different types. That is, you can save a numerical array, string array, and even a cell structure as a `.mat` file with this command.

**Example 1. Saving Arrays.** We have both a numerical array and a string array that we wish to save.

$$\mathbf{A} = [1 \ 2 \ 3 \ 4], \quad \mathbf{B} = ["This" \ "Is" \ "A" \ "String" \ "Array"]$$

In order to save these two arrays we can use the `save` command. We will save array **A** as `ANumerical.mat` and we will save **B** as `BString.mat`. The corresponding save commands as well as the file folders are seen in figure 39.

As we can see, in the current folder there are now two `.mat` files corresponding to the two arrays that were saved. So now that these have been saved, we need to know how we would be able to

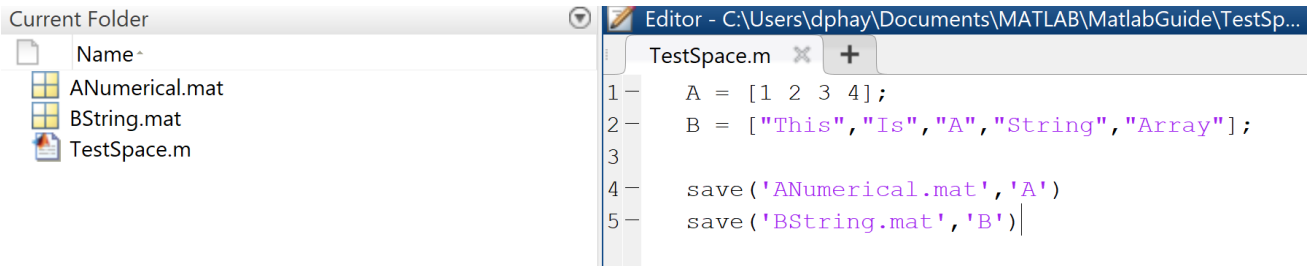


Figure 39: Saving Arrays.

load them in at a later date.

Loading in arrays is just as simple as saving arrays. You will need to use the load command which is at the basic level, easier to use than the save command.

**load('filename.mat')**

This will load in the file “filename.mat” from the current folder.

**Note:** If this file was a saved array, the loaded in file will retain the same variable name that it was saved as. So the following two commands produce very different results.

**load('filename.mat')**

**“variablename” = load('filename.mat')**

The first option will load in the file and assign the loaded file the original variable name it had when it was saved. The second option will create a cell structure “variablename” that has the loaded file name as one of the objects in the structure. There are benefits to both, but at the end of the day if you remain consistent with variable names, the first option should be fine.

**Example 2. Loading in .mat files.** We need to load in the saved files from example 1. For this example we start with a clear workspace, i.e. there are no saved variables. If we need to load in the files, we can use the commands

load('ANumerical.mat'),    load('BString.mat')

Further, to demonstrate the difference, the command

C = load('ANumerical.mat')

is also used. We can see in figure 40 that since the files “ANumerical.mat” and “BString.mat” saved the variables “A” and “B” respectively, then they are automatically assigned these variable names. To this end, a special amount of attention should be taken in order to not try and double assign variable names. Further, we can see that for the third load command, a structure “C” is created and the variable “A” is an object of the structure. In the event that you wish to use the third option, you would need to type “C.A” in order to use the loaded in array.



```

TestSpace.m x +
1 %Load in A and B
2
3 load('ANumerical.mat')
4 load('BString.mat')
5
6 %Demonstrate the Cell structure
7
8 C = load('ANumerical.mat');

>> TestSpace
>> A
A =
    1    2    3    4
>> B
B =
1x5 string array
    "This"    "Is"    "A"    "String"    "Array"
>> C
C =
struct with fields:
    A: [1 2 3 4]

```

Figure 40: Loading Arrays.

Lastly, you may need to transfer data/arrays from MATLAB<sup>®</sup> to some other platform or vice versa. A rather important and commonly used filetype is a .xls file (Excel/Google Sheets). In order to do this, we need two new commands that are specific to .xls files.

In order to save an array in MATLAB<sup>®</sup> to a .xls file we use the following command.

**xlswrite('filename.xls',variable)**

This will create a .xls file with the information saved in the variable transferred to a table. This can be specified further to add a specific sheet as well as the location in the sheet. To do this you need only add two more inputs to the command.

**xlswrite('filename.xls',variable,'Sheet','Location')**

This second command will save the variable information to the sheet that you specify and to the cells that you list in the location.

**Example 3. Exporting to .xls** You run an experiment and input the data in MATLAB<sup>®</sup> only to find out you need it in a .xls file. You have two types of data, time and position of some object. The measurements collected are given below where  $\mathbf{P}(i)$  is the position at time  $\mathbf{T}(i)$ .

$$\mathbf{T} = \begin{bmatrix} 0 \\ 0.5 \\ 1 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 1 \\ 4 \\ 6 \end{bmatrix}$$

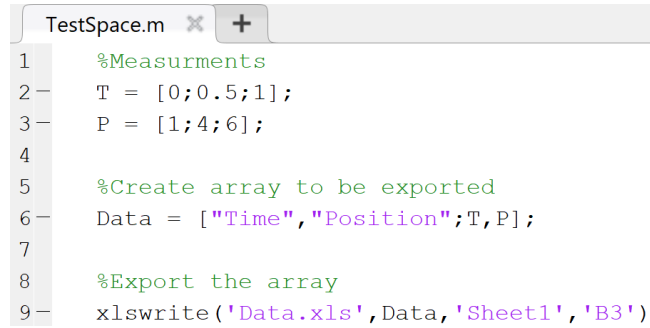
To transfer this to a .xls file we want it so that there are titles “Time” and “Position” above the two columns of data and it must start in position B3. In order to do this in a clean command we will want to create a string array which we will call “Data”.

```
Data = ["Time", "Position"; T, P]
```

Then we use the command

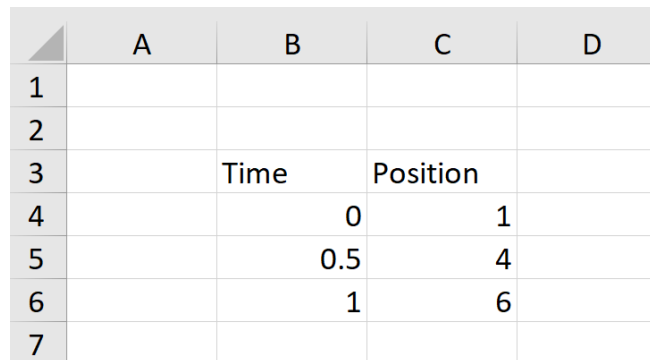
```
xlswrite('Data.xls',Data,'Sheet1','B3')
```

This is shown in figure 41. Figure 42 shows the .xls file that was saved when it is opened. As we can see the data was transferred and positioned so that the top left part of the array is in cell B3.



```
TestSpace.m x +
1 %Measurements
2 T = [0;0.5;1];
3 P = [1;4;6];
4
5 %Create array to be exported
6 Data = ["Time", "Position";T,P];
7
8 %Export the array
9 xlswrite('Data.xls',Data,'Sheet1','B3')
```

Figure 41: Exporting to .xls.



	A	B	C	D
1				
2				
3		Time	Position	
4		0	1	
5		0.5	4	
6		1	6	
7				

Figure 42: Exporting to .xls.

To import a .xls file into MATLAB®, there are a couple options. This will only cover the option that allows for direct computation with transferred data. The command is quite simple and very similar to the export case.

```
[numeric,text] = xlsread('filename.xls')
```

Notice that in this case we have two outputs. The first output “numeric” will hold all of the numerical information from the .xls file whereas the “text” output holds the non numeric information in a cell structure. This can also be further specified for a range of the .xls file using the following command.

```
[numeric,text] = xlsread('filename.xls','Location')
```

This will only import the portion of the file in the specified location.

**Example 4. Importing .xls files** You go for runs and record your times into the file RunTimes.xls with the time it took to complete the run. You wish to import this data to MATLAB®. To do this we can use the command

$$[\text{num}, \text{text}] = \text{xlsread}(\text{'RunTimes.xls'})$$

The file RunTimes.xls is shown in figure 43 output of this command is shown in figure 44.

	A	B
1	Time(min)	Miles
2	13	2
3	17	2.5
4	14	2.1
5	19	3
6	18	2.8

Figure 43: Importing to MATLAB®.

```

TestSpace.m x +
1 %Import
2 [num,text]=xlsread('RunTimes.xls')
3

>> TestSpace
num =
    13.0000    2.0000
    17.0000    2.5000
    14.0000    2.1000
    19.0000    3.0000
    18.0000    2.8000

text =
1x2 cell array
    {'Time (min)'}    {'Miles'}
  
```

Figure 44: Importing to MATLAB®.

As we can see, the output “num” holds the numerical data, and “text” hold the text data from the .xls file. If you only wished to include the first three data points you would then change the command to the following.

$$[\text{num}, \text{text}] = \text{xlsread}(\text{'RunTimes.xls'}, \text{'A1:B4'})$$

The output would however follow in the same fashion as before.

## EXERCISES

**Exercise 1.** Write a script that will create a  $5 \times 3$  matrix  $\mathbf{A}$  such that the entry  $\mathbf{A}(i, j) = \sin(i) \cos(j)$ . Save this matrix in a .mat file named “TrigA”. Clear your workspace and command window using the command “clear, clc” and load this matrix back into the workspace.

**Exercise 2.** Create a .xls file named “HeatData” that mimics figure 45. Import this to file to MATLAB<sup>®</sup>, plot the imported data and determine the average temperature over the 10 hour period.

	A	B
1	Time(hours)	Temperature
2	0	180
3	1	136.7
4	2	110.5
5	3	94.5
6	4	84.9
7	5	79
8	6	75.5
9	7	73.3
10	8	72
11	9	71.2
12	10	70.7

Figure 45: Exercise 2.

## Appendix (Other Useful Functions)

MATLAB<sup>®</sup> is very well documented. That is, there are very clear instructions on how to use the numerous built-in functions/commands. To look up the instructions on how to use a command in MATLAB<sup>®</sup>, simply type “help” into the command window followed by the command of interest. For instance, to view the instructions for the built-in function **norm**, type the following into the command window,

**help norm**

Here we list a few very important functions in MATLAB<sup>®</sup> that might come in handy. We opt to not show an example because we would pretty much be regurgitating the documentation. Simply look up the commands in the command window. Oftentimes an example or two will be included. This list is not exhaustive and doesn’t even scratch the surface of all the useful functions available in MATLAB<sup>®</sup>. We hope the reader makes develops the good habit of googling for other commands they might need for different situations.

**input** is used to ask the user to define a value in the command window. For instance, perhaps you want the code to ask the user “How many data points do you want?” The **input** command allows you to pose this question mid-code and have the user type in/declare the value in the command window.

**rref** computes the reduced row echelon form of a matrix.

**semilogx**, **semilogy**, **loglog** produces a plot (just like the command **plot**), but with a log scale on the  $x$  axis,  $y$  axis, and both axes, respectively.

**eig** computes the eigenvalue decomposition (matrix of eigenvectors, and diagonal matrix of

eigenvalues) of a matrix.

**svd** computes the full singular value decomposition of a matrix.

**qr** computes the full QR factorization of a matrix.

**ode45** solves non-stiff ordinary differential equations (or system thereof). We recommend google searching an example of this in addition to checking out the documentation. There are several other ordinary differential equation solvers in MATLAB<sup>®</sup> (e.g., **ode23** for stiff problems).

**norm** computes the norm of a vector or matrix. It compute the 2-norm by default.

**array2table** turns an array into a table. Same you have a matrix for which each column holds data for different variables (e.g., first column is height, second column is weight). **array2table** will turn this into a table for which you can also label the rows and columns.

**fft** and **ifft** compute the discrete Fourier transform and inverse Fourier transform of a signal.

## Lab 1 – Newton’s Method

A common root-finding algorithm is Newton’s method,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Note that in certain circumstances the sequence  $\{x_n\}_n$  may not converge. One such case is when for our initial point  $x_1$  we have  $f'(x_1) \approx 0$ .

Write a MATLAB<sup>®</sup> function called *newton* that takes the following inputs:  $x_1$ ,  $f(x)$ ,  $f'(x)$ , *tol*. Here, *tol* is a given tolerance. Be sure to include a loop that terminates the code after a pre-determined number of iterations; you can set the max number of iterations to 100 for now.

Test your code on the following problems:

(i)  $f(x) = \sin(x) - x - 1$ ,  $x_1 = 1$ ,  $tol = 0.00001$ .

(ii)  $f(x) = x^3 - 3x + 6$ ,  $x_1 = 1$ ,  $tol = 0.001$ . This is a case where Newton’s method doesn’t work because  $f'(x_1) = 0$ .

## Lab 2a – Trapezoid Rule

Recall the trapezoid rule for approximating definite integrals,

$$\int_a^b f(x)dx \approx \Delta x \sum_{i=1}^N \frac{f(x_{i-1}) + f(x_i)}{2},$$

where we have divided  $[a, b]$  into  $N$  *equally* spaced sub-intervals. Let the points be denoted by

$a = x_0 < x_1 < x_2 < \dots < x_{N-1} < x_N = b$ . Write a MATLAB<sup>®</sup> function called *trapezoid* that takes the following inputs:  $a, b, N, f$ . Test your code on

$$\int_0^\pi \sin(x) dx = 2$$

using  $N = 16$  sub-intervals.

## Lab 2b – Simpson’s Rule

Recall the Simpson’s rule for approximating definite integrals,

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} [f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{N-2}) + 4f(x_{N-1}) + f(x_N)],$$

where we have divided  $[a, b]$  into  $N$  *equally* spaced sub-intervals with  $N$  an even positive integer. Let the points be denoted by  $a = x_0 < x_1 < x_2 < \dots < x_{N-1} < x_N = b$ . Write a MATLAB<sup>®</sup> function called *simpson* that takes the following inputs:  $a, b, N, f$ . Test your code on

$$\int_0^\pi \sin(x) dx = 2$$

using  $N = 8$  sub-intervals. Notice that (for this problem) Simpson’s rule performs comparable to the trapezoid rule with half as many sub-intervals!

## Lab 2c – Gauss-Legendre Quadrature

An advanced integration technique is Gaussian quadrature. In short, a continuously differentiable function on a finite interval  $[a, b]$ , say  $f \in C^1[a, b]$  can be approximated by evaluating  $f(x)$  at very special points/nodes and taking a weighted sum of these point evaluations. The point of this lab is not to review the derivations of Gaussian quadratures *nor* their properties and convergence results; for that we refer the reader to: *Numerical Mathematics* by Alfio Quarteroni, or *A First Course in Numerical Analysis* by Anthony Ralston and Philip Rabinowitz. Instead, we shall simply state the Gauss-Legendre quadrature rule and leave it up to the reader to implement.

Consider a function  $f \in C^1[a, b]$ . The function can be purely in  $C^0[a, b]$ , but *MANY* more nodes/weights are required for a reasonable approximation; even the nice function  $f(x) = |x|$  on  $[-1, 1]$  has an error  $\mathcal{O}(10^{-3})$  for 75 nodes/weights! Moreover, convergence results are meaningless for functions purely in  $C^0[a, b]$ . If dealing with a piecewise  $C^1$  function, then it is best to use Gauss-Legendre quadrature on each  $C^1$  function.

By the change of variables  $x = a + \frac{b-a}{2}(\xi + 1)$  with  $x \in [a, b]$  and  $\xi \in [-1, 1]$  (from the linear interpolant),

$$\int_a^b f(x) dx = \frac{b-a}{2} \int_{-1}^1 f\left(a + \frac{b-a}{2}(\xi + 1)\right) d\xi \approx \frac{b-a}{2} \sum_{k=1}^n w_k f\left(a + \frac{b-a}{2}(x_k + 1)\right),$$

where  $\{w_1, w_2, \dots, w_n\}$  are the weights, and  $\{x_1, x_2, \dots, x_n\}$  are the (Gauss-Legendre) nodes on  $[-1, 1]$ . Here,  $n$  is the number of nodes/weights used. In short, the more nodes taken results in a better approximation. But in most applications three to five nodes/weights suffice. A table of nodes and weights up to  $n = 7$  are given below. The nodes and weights for larger  $n$  can easily be found online. Create a MATLAB<sup>®</sup> function called *GLquad* that takes the following inputs:  $a, b, n, f$ . Test your code on

$$\int_0^\pi \sin(x) dx = 2$$

for  $n = 2, 3, \dots, 7$  and observe how the approximations get better (they should!) as  $n$  increases. *\*note – you will need to write **format long** at the start of your code, along with **clear all**. This will allow you to see more digits.*

$n$	$x_k$	$w_k$
2	$\pm 0.577350269189625765$	1
3	$\pm 0.774596669241483377$ 0	0.555555555555555556 0.888888888888888889
4	$\pm 0.861136311594052575$ $\pm 0.339981043584856265$	0.347854845137453857 0.652145154862546143
5	$\pm 0.906179845938663993$ $\pm 0.538469310105683091$ 0	0.236926885056189088 0.478628670499366468 0.568888888888888889
6	$\pm 0.932469514203152028$ $\pm 0.66120938646626451$ $\pm 0.238619186083196909$	0.171324492379170345 0.360761573048138608 0.46791393457269105
7	$\pm 0.949107912342758525$ $\pm 0.74153118559939444$ $\pm 0.405845151377397167$ 0	0.129484966168869693 0.279705391489276668 0.38183005050511894 0.417959183673469388

Nodes and weights for Gauss-Legendre quadrature on  $[-1, 1]$ .

## Lab 3a – Forward Euler Method

Consider the ordinary differential equation (ODE)  $\dot{y} = f(t, y)$ ,  $t > 0$  with an initial condition  $y(0) = y_0$ . Say we want to numerically solve this ODE up to time  $t = T_f$ . We can discretize in time by splitting up the time interval  $[0, T_f]$  into  $N + 1$  *equally* spaced points,

$$0 = t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N = T_f.$$

Letting  $y^n \approx y(t_n)$  for  $n = 0, 1, 2, \dots, N$  and  $\Delta t = (T_f - 0)/N$ , the **forward Euler method** states:

$$y^{n+1} = y^n + \Delta t f(t_n, y^n).$$

Write a MATLAB<sup>®</sup> function called *feuler* that takes the following inputs:  $t_n, y^n, \Delta t, f(t, y)$ . This MATLAB<sup>®</sup> function should output the approximate solution at the next time step.

Test your code with  $f(t, y) = 4ty^{1/2}$ ,  $y_0 = 1$ ,  $T_f = 3$ , and  $N = 10$ . Plot the solution and compare it to the exact solution,  $y(t) = (y_0^{1/2} + t^2)^2$ .

*Optional: show that the forward Euler method has first order convergence (in the  $L^1$  norm).*

## Lab 3b – Heun’s Method

Consider the ordinary differential equation (ODE)  $\dot{y} = f(t, y)$ ,  $t > 0$  with an initial condition  $y(0) = y_0$ . Say we want to numerically solve this ODE up to time  $t = T_f$ . We can discretize in time by splitting up the time interval  $[0, T_f]$  into  $N + 1$  *equally* spaced points,

$$0 = t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N = T_f.$$

Letting  $y^n \approx y(t_n)$  for  $n = 0, 1, 2, \dots, N$  and  $\Delta t = (T_f - 0)/N$ , **Heun’s method** states:

$$\begin{aligned} K_1 &= f(t_n, y^n) \\ K_2 &= f(t_n + \Delta t, y^n + \Delta t K_1) \\ y^{n+1} &= y^n + \frac{\Delta t}{2}(K_1 + K_2) \end{aligned}$$

Write a MATLAB<sup>®</sup> function called *heun* that takes the following inputs:  $t_n$ ,  $y^n$ ,  $\Delta t$ ,  $f(t, y)$ . This MATLAB<sup>®</sup> function should output the approximate solution at the next time step.

Test your code with  $f(t, y) = 4ty^{1/2}$ ,  $y_0 = 1$ ,  $T_f = 3$ , and  $N = 10$ . Plot the solution and compare it to the exact solution,  $y(t) = (y_0^{1/2} + t^2)^2$ .

*Optional: show that the Heun’s method has second order convergence (in the  $L^1$  norm).*

## Lab 3c – 4th Order Runge-Kutta (RK4)

Consider the ordinary differential equation (ODE)  $\dot{y} = f(t, y)$ ,  $t > 0$  with an initial condition  $y(0) = y_0$ . Say we want to numerically solve this ODE up to time  $t = T_f$ . We can discretize in time by splitting up the time interval  $[0, T_f]$  into  $N + 1$  *equally* spaced points,

$$0 = t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N = T_f.$$

Letting  $y^n \approx y(t_n)$  for  $n = 0, 1, 2, \dots, N$  and  $\Delta t = (T_f - 0)/N$ , the **fourth order Runge-Kutta (RK4) method** states:

$$\begin{aligned} K_1 &= f(t_n, y^n) \\ K_2 &= f\left(t_n + \frac{\Delta t}{2}, y^n + \frac{\Delta t}{2}K_1\right) \\ K_3 &= f\left(t_n + \frac{\Delta t}{2}, y^n + \frac{\Delta t}{2}K_2\right) \\ K_4 &= f(t_n + \Delta t, y^n + \Delta t K_3) \\ y^{n+1} &= y^n + \frac{\Delta t}{6}(K_1 + 2K_2 + 2K_3 + K_4) \end{aligned}$$



Write a MATLAB<sup>®</sup> function called *RK4* that takes the following inputs:  $t_n$ ,  $y^n$ ,  $\Delta t$ ,  $f(t, y)$ . This MATLAB<sup>®</sup> function should output the approximate solution at the next time step.

Test your code with  $f(t, y) = 4ty^{1/2}$ ,  $y_0 = 1$ ,  $T_f = 3$ , and  $N = 10$ . Plot the solution and compare it to the exact solution,  $y(t) = (y_0^{1/2} + t^2)^2$ .

On the same figure, plot the exact solution, as well as the results from forward Euler, Heun's method, and RK4.

*Optional: show that the RK4 has fourth order convergence (in the  $L^1$  norm).*

## Lab 4 – Linear Regression

In many applications we often desire relationships within data. Say we are looking at a single dependent variable,  $Y$ , that is dependent on  $m$  independent variables  $X_1, X_2, \dots, X_m$ . The goal of regression is to predict  $Y$  given  $X_1, X_2, \dots, X_m$ . There are many different types of functions that can describe these relationships, for instance a linear vs nonlinear relationship. With these come many different types of regression models (linear, quadratic, logarithmic, inverse, exponential, Poisson, logistic, etc...). Lab 4 will investigate the simplest model – (multiple) linear regression. The purpose of this guide and these labs is not to teach you an entire course, so we opt not to cover more complicated models. We care more about the you learning how to *use* MATLAB<sup>®</sup> for basic problems and labs.

Assuming a *very simple* linear relationship between the dependent variable and independent variables, one can suppose

$$\mathbf{Y} = \beta_0 + \beta_1 \mathbf{X}_1 + \beta_2 \mathbf{X}_2 + \dots + \beta_m \mathbf{X}_m,$$

or rather,

$$Y_i = \beta_0 + \beta_1 X_{1,i} + \beta_2 X_{2,i} + \dots + \beta_m X_{m,i},$$

where the index  $i$  represents the  $i$ -th observation (i.e., a data point) with  $i = 1, 2, \dots, n$ . We desire the coefficients  $\beta_0, \beta_1, \dots, \beta_m$ . This is known as **multiple linear regression**. If the data is given in matrix form, that is,  $\mathbf{Y}$ ,  $\mathbf{X}_1$ ,  $\mathbf{X}_2$ , ...,  $\mathbf{X}_m$  are all vectors, then the (ordinary) least-squares estimation leads to

$$\beta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y},$$

where

$$\mathbf{X} = \begin{bmatrix} 1 & X_{1,1} & X_{2,1} & \dots & X_{m,1} \\ 1 & X_{1,2} & X_{2,2} & \dots & X_{m,2} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & X_{1,n-1} & X_{2,n-1} & \dots & X_{m,n-1} \\ 1 & X_{1,n} & X_{2,n} & \dots & X_{m,n} \end{bmatrix} = \begin{bmatrix} | & | & | & & | \\ \mathbf{1} & \mathbf{X}_1 & \mathbf{X}_2 & \dots & \mathbf{X}_m \\ | & | & | & & | \end{bmatrix}$$

For further details, we refer the reader to the Wikipedia page on linear regression. We consider the case where there is one dependent variable and one independent variable (known as **simple linear regression**). A common first relationship taught in classes is relating a person's height

and weight. Consider the sample simulated data given below (taken from SOCR Data on UCLA statistics' website).

[http://wiki.stat.ucla.edu/socr/index.php/SOCR\\_Data\\_Dinov\\_020108\\_HeightsWeights](http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_Dinov_020108_HeightsWeights)

70.18 – 147.89	67.01 – 128.82	69.92 – 140.40
70.41 – 155.90	70.81 – 135.32	68.25 – 128.52
66.36 – 119.37	69.06 – 142.47	66.36 – 120.30
67.54 – 133.81	67.73 – 132.75	68.36 – 138.60
66.50 – 128.73	67.37 – 124.73	65.48 – 132.96
69.00 – 137.55	65.27 – 129.31	67.73 – 122.52
68.30 – 129.76	70.84 – 134.02	

height (in) – weight (lbs)

Using MATLAB<sup>®</sup> compute and plot the simple linear regression model (on the same figure as the data points).

## Lab 5 – Least Squares Problems: Solving a Basic Linear System

As our graduate school professor once said, “One does not simply *solve*  $\mathbf{Ax} = \mathbf{b}$ .” Indeed, this is true! By junior year most engineering students have solved some basic linear systems, e.g.,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 1 \\ -2 & 3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

In the example above, the matrix  $\mathbf{A}$  is square and invertible. Hence, finding the solution is as easy as computing the inverse (whether it be by hand or with a computer) and multiplying on the left by  $\mathbf{A}^{-1}$ . But what if the matrix  $\mathbf{A}$  is not square? Clearly we cannot find the inverse of a non-square matrix (if one were to even exist)! In particular, we assume the matrix has more rows than columns, i.e., a “tall” matrix. For example, say we want to *solve*

$$\begin{bmatrix} 1 & -1 \\ 0 & 0.5e - 7 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.5e - 7 \\ 1 \end{bmatrix}.$$

Here,  $\mathbf{A} \in \mathbb{R}^{3 \times 2}$ ,  $\mathbf{x} \in \mathbb{R}^{2 \times 1}$ , and  $\mathbf{b} \in \mathbb{R}^{3 \times 1}$ . This system is called **overdetermined** because there are more equations than unknowns; there are three equations from the three rows of  $\mathbf{A}$ , and two unknowns  $x_1, x_2$ . Generally speaking, such problems do not have an exact solution. The rough idea for *solving* this system is to find the vector  $\mathbf{x}$  that produces the *smallest* residual (which for simplicity you can think of as producing the smallest error). The residual is the vector  $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ , and it is *measured* by taking its magnitude/2-norm. In other words, which vector  $\mathbf{x}$  *minimizes*

$$\|\mathbf{b} - \mathbf{Ax}\|_2?$$

We will refrain from going into the theory since again, that is not the purpose of this guide nor these labs. Ultimately, there are three classical methods for finding the desired solution  $\mathbf{x}$ : normal equations, QR factorization, and the singular value decomposition (SVD). Each method has pros and cons. We do not concern ourselves with the nitty gritty details comparing the three methods; see *Numerical Linear Algebra* by Trefethen and Bau, or *Matrix Computations* by Golub and Van Loan.

*\*We feel the need to mention the normal equations are generally unstable in the presence of rounding errors, and as such the QR factorization and SVD are more commonly used.*

Instead, we outline the three methods below and want you to implement them in MATLAB<sup>®</sup> to solve the least squares problem (i.e., the overdetermined system) that we used above as an example. The algorithms included are taken and summarized from *Numerical Linear Algebra* by Trefethen and Bau.

### Normal equations

You will need to use the **Cholesky factorization** which simply takes a Hermitian positive definite matrix  $\mathbf{B}$  and writes it as  $\mathbf{B} = \mathbf{R}^T\mathbf{R}$ , where  $\mathbf{R}$  is an upper-triangular matrix. The MATLAB<sup>®</sup> command for this is

$$\mathbf{R} = \text{chol}(\mathbf{B})$$

The algorithm for solving the least squares problem  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is:

1. Compute the Cholesky factorization (i.e., find the upper-triangular matrix  $\mathbf{R}$ ) for the matrix  $\mathbf{A}^T\mathbf{A}$ .
2. Solve the lower-triangular system  $\mathbf{R}^T\mathbf{w} = \mathbf{A}^T\mathbf{b}$  for  $\mathbf{w}$ . You can do this using the backslash command.
3. Solve the upper-triangular system  $\mathbf{R}\mathbf{x} = \mathbf{w}$  for  $\mathbf{x}$ . You can do this using the backslash command.

### QR Factorization

You will need to use the **QR factorization** which simply takes an  $m \times n$  (assume  $m > n$ ) matrix  $\mathbf{A}$  and writes it as  $\mathbf{A} = \mathbf{Q}\mathbf{R}$ , where  $\mathbf{Q} \in \mathbb{R}^{m \times m}$  is a square matrix with orthonormal columns, and  $\mathbf{R} \in \mathbb{R}^{m \times n}$  is a rectangular matrix whose first  $n$  rows form an upper-triangular matrix (denoted as  $\hat{\mathbf{R}}$ ) and whose last  $m - n$  rows are zeros. In particular, the first  $n$  columns of  $\mathbf{Q}$  (whose matrix we denote  $\hat{\mathbf{Q}}$ ) span the column space of  $\mathbf{A}$ . Hence,  $\mathbf{A} = \mathbf{Q}\mathbf{R} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$ . The MATLAB<sup>®</sup> command for the full QR factorization is

$$[\mathbf{Q},\mathbf{R}] = \text{qr}(\mathbf{A})$$

The algorithm for solving the least squares problem  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is:

1. Compute the reduced QR factorization  $\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$ . Note that you'll need to compute the full QR factorization using `qr` and then keep the first  $n$  columns of  $\mathbf{Q}$  and the first  $n$  rows of  $\mathbf{R}$ .
2. Solve the upper-triangular system  $\hat{\mathbf{R}}\mathbf{x} = \hat{\mathbf{Q}}^T\mathbf{b}$ . You can do this using the backslash command.

### Singular Value Decomposition

You will need to use the SVD which simply takes an  $m \times n$  (assume  $m > n$ ) matrix  $\mathbf{A}$  and

writes it as  $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ , where  $\mathbf{U} \in \mathbb{R}^{m \times m}$  is a square matrix with orthonormal columns,  $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$  is a rectangular matrix whose first  $n$  rows form a diagonal matrix and whose last  $m - n$  rows are zeros, and  $\mathbf{V} \in \mathbb{R}^{n \times n}$  is a square matrix with orthonormal columns. Similar to the reduced QR factorization, we can also compute the reduced SVD  $\mathbf{A} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\mathbf{V}^T$  by taking the first  $n$  columns of  $\mathbf{U}$  and the first  $n$  rows of  $\mathbf{\Sigma}$ . The MATLAB<sup>®</sup> command for the full SVD is

$$[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{A})$$

The algorithm for solving the least squares problem  $\mathbf{A}\mathbf{x} = \mathbf{b}$  is:

1. Compute the reduced SVD  $\mathbf{A} = \hat{\mathbf{U}}\hat{\mathbf{\Sigma}}\mathbf{V}^T$ . Note that you'll need to compute the full SVD using `svd` and then keep the first  $n$  columns of  $\mathbf{U}$  and the first  $n$  rows of  $\mathbf{\Sigma}$ .
2. Solve the diagonal system  $\hat{\mathbf{\Sigma}}\mathbf{w} = \hat{\mathbf{U}}^T\mathbf{b}$  for  $\mathbf{w}$ .
3. Set  $\mathbf{x} = \mathbf{V}\mathbf{w}$ .

Compute the solution  $\mathbf{x}$  for all three algorithms. Observe that the answer obtained using the normal equations is a bit different than the answers obtained using the QR factorization and SVD.

## Lab 6a – Lagrange Polynomials

When given several data points it is often desired to fit a curve to said data, with the desired curve going through every data point. In other words, we want to *interpolate* the data. This gives rise to the idea of an *interpolating polynomial*. In labs 7a-c the reader will implement three elementary ways to interpolate data: Lagrange interpolating polynomials, Hermite interpolating polynomials, and cubic splines.

Given  $n$  points  $(x_i, y_i)$  with  $i = 1, 2, \dots, n$  with no two  $x_i$  being the same, the **Lagrange interpolating polynomial** (a.k.a. the Lagrange interpolant) is the polynomial  $P(x)$  of degree  $\leq n - 1$  that passes through all  $n$  points and is given by

$$P(x) = \sum_{i=1}^n \left( y_i \prod_{\substack{k=1 \\ k \neq i}}^n \frac{x - x_k}{x_i - x_k} \right).$$

(i) Consider the following data points:  $(-3, -5)$ ,  $(-1, 2)$ ,  $(0, 4)$ ,  $(1, 5)$ , and  $(3, -1)$ . Construct the Lagrange interpolant for this data and code it in MATLAB<sup>®</sup>. Plot the data points and interpolant on the same plot.

(ii) In some situations (details omitted), Lagrange interpolation yields a highly oscillatory polynomial that diverges as the number of points increases. In other words, the Lagrange interpolant is not always a good choice. This well-known observation is called Runge's phenomena. Consider the function (known as Runge's function)

$$f(x) = \frac{1}{1 + 25x^2}.$$

Consider sixteen equally spaced nodes  $x_i, i = 1, 2, \dots, 16$  over the interval  $[-1, 1]$ . Then, let the sixteen points be  $(x_i, f(x_i)), i = 1, 2, \dots, 16$ . Construct the Lagrange interpolant for this data and code it in MATLAB<sup>®</sup>. Plot the data points and interpolant on the same plot. Observe Runge's phenomena.

*\*Note 1 – Try doing part (ii) using for loops, rather than typing in the Lagrange interpolant explicitly. As you will see, trying to type this Lagrange interpolant involving sixteen nodes is quite long.*

*\*Note 2– There are several ways to mitigate Runge's phenomena. One common strategy is to use Chebyshev nodes instead of equally spaced nodes. We omit the details.*

## Lab 6b – Hermite Polynomials

As we saw in part (i), the Lagrange interpolating polynomial can be used to interpolate a function  $f(x)$  at several nodes, i.e.,  $x_i, i = 1, 2, \dots, n$ . Similar to the Lagrange interpolating polynomial, the **Hermite interpolating polynomial** (a.k.a. the Hermite interpolant) can interpolate a function  $f(x)$  at several nodes. The main difference is the Hermite interpolant agrees with the function  $f(x)$  and  $f'(x)$  at the  $n$  nodes  $x_i, i = 1, 2, \dots, n$ . In other words, the Hermite interpolant  $P(x)$  has the same point values and derivatives as  $f(x)$  at the nodes. Note that the original function  $f(x) \in C^1[a, b]$  (continuously differentiable) with  $x_1 = a$  and  $x_n = b$ .

Given a function  $f(x) \in C^1[a, b]$  and  $n$  nodes  $a = x_1 < x_2 < \dots < x_n = b$ , the Hermite interpolant is the polynomial  $H(x)$  of degree  $\leq 2n - 1$  that agrees with  $f(x)$  and  $f'(x)$  at the  $n$  nodes and is given by

$$H(x) = \sum_{i=1}^n f(x_i)H_i(x) + \sum_{i=1}^n f'(x_i)\hat{H}_i(x),$$

where

$$H_i(x) = [1 - 2(x - x_i)L'_i(x_i)](L_i(x))^2 \quad \text{and} \quad \hat{H}_i(x) = (x - x_i)(L_i(x))^2.$$

Here,

$$L_i(x) = \prod_{\substack{k=1 \\ k \neq i}}^n \frac{x - x_k}{x_i - x_k}.$$

Consider a function  $f(x) \in C^1[-2, 1]$  and the three nodes  $x_1 = -2, x_2 = 0$  and  $x_3 = 1$ , such that  $f(-2) = f(1) = 1, f(0) = -1$ , and  $f'(-2) = 0, f'(0) = 1, f'(1) = 2$ . Using pen and paper, verify that the Hermite interpolant is

$$H(x) = -\frac{1}{36}x^5 - \frac{1}{3}x^4 - \frac{1}{4}x^3 + \frac{29}{18}x^2 + x - 1.$$

Then, use MATLAB<sup>®</sup> to plot the three nodes and  $H(x)$  on the same figure.

*\*Note – The Hermite interpolant is typically computed using divided differences. Since this lab is not meant to be a crash course in computational mathematics, we omit the details. The method of divided differences can be found in most upper undergraduate/graduate level textbooks on numerical mathematics.*

## Lab 6c – Cubic Splines

Roughly speaking, **cubic splines** are made up of piecewise third-order polynomials (cubic functions) that pass through  $n$  points  $(x_i, y_i)$ ,  $i = 1, 2, \dots, n$ . If we have  $n$  nodes, then we must have  $n - 1$  subintervals  $[x_i, x_{i+1}]$ ,  $i = 1, 2, \dots, n - 1$ . Each subinterval gets its own cubic interpolant  $f_i(x)$ ,  $x \in [x_i, x_{i+1}]$ . We can consider “stitching” all these cubic interpolants to compose the overall interpolation function  $S(x)$ ,  $x \in [x_1, x_n]$ . The resulting function is

$$S(x) = \begin{cases} A_1x^3 + B_1x^2 + C_1x + D_1, & x \in [x_1, x_2] \\ A_2x^3 + B_2x^2 + C_2x + D_2, & x \in (x_2, x_3] \\ \vdots & \vdots \\ A_{n-1}x^3 + B_{n-1}x^2 + C_{n-1}x + D_{n-1}, & x \in (x_{n-1}, x_n] \end{cases}$$

We want to set up a system of equations that solves for all the coefficients  $A_i, B_i, C_i, D_i$ ,  $i = 1, 2, \dots, n - 1$ . For each subinterval (fixed  $i = 1, 2, \dots, n - 1$ ),

$$f_i(x_i) = y_i \quad \text{and} \quad f_i(x_{i+1}) = y_{i+1}$$

since we want each cubic polynomial to go through the two endpoints  $(x_i, y_i)$  and  $(x_{i+1}, y_{i+1})$ . Moreover, we want the cubic polynomials to have agreeing derivatives at the interior nodes. For instance, we want  $f_1$  (defined on  $[x_1, x_2]$ ) and  $f_2$  (defined on  $[x_2, x_3]$ ) to have the same derivative at the shared point  $x_2$ .

$$f'_i(x_{i+1}) = f'_{i+1}(x_{i+1}), \quad i = 1, 2, \dots, n - 2$$

Similar to the first derivatives, we also want the cubic polynomials to have agreeing second derivatives at the interior nodes.

$$f''_i(x_{i+1}) = f''_{i+1}(x_{i+1}), \quad i = 1, 2, \dots, n - 2$$

It is time to take a step back. Note that we need to solve for a total  $4(n - 1)$  coefficients; and hence need  $4(n - 1)$  equations. We have  $2(n - 1) + (n - 2) + (n - 2) = 4(n - 1) - 2$  equations. We need two more equations! These last two equations come from the boundary conditions that we impose at  $x_1$  and  $x_n$ . There are several boundary conditions to choose from: natural spline, not-a-knot spline, periodic spline, quadratic spline. Each gives a slightly different spline, but we omit the details. For simplicity, we shall assume a *natural spline* for which we assume

$$f''_1(x_1) = 0 \quad \text{and} \quad f''_{n-1}(x_n) = 0.$$

We now have  $4(n - 1)$  equations. Written cleanly for implementation ease, the  $4(n - 1)$  equations are:

$$\begin{aligned} A_ix_i^3 + B_ix_i^2 + C_ix_i + D_i &= y_i, & i = 1, 2, \dots, n - 1 \\ A_ix_{i+1}^3 + B_ix_{i+1}^2 + C_ix_{i+1} + D_i &= y_{i+1}, & i = 1, 2, \dots, n - 1 \\ 3A_ix_{i+1}^2 + 2B_ix_{i+1} + C_i - 3A_{i+1}x_{i+1}^2 + 2B_{i+1}x_{i+1} + C_{i+1} &= 0, & i = 1, 2, \dots, n - 2 \\ 6A_ix_{i+1} + 2B_i - 6A_{i+1}x_{i+1} + 2B_{i+1} &= 0, & i = 1, 2, \dots, n - 2 \end{aligned}$$

$$6A_1x_1 + 2B_1 = 0$$

$$6A_{n-1}x_n + 2B_{n-1} = 0$$

Consider Runge's function and the same sixteen equally spaced nodes from Lab 7a(ii). Solve for the coefficients  $A_i, B_i, C_i, D_i$  by setting up a system of the form  $\mathbf{Ax} = \mathbf{b}$ . Then, plot the points and cubic spline on the same figure. Notice that we no longer observe Runge's phenomena.

*\*Note - A linear system can be solved using the backslash command. See the Appendix.*

## Lab 7 – The 1D Heat Equation (Separation of Variables)

Consider the 1D heat equation,

$$\begin{cases} u_t = u_{xx}, & 0 < x < 1, t > 0, \\ u_x(x = 0, t) = u_x(x = 1, t) = 0, & t > 0, \\ u(x, t = 0) = (2x - 1)^2, & 0 < x < 1. \end{cases}$$

(a) Use separation of variables to solve the 1D heat equation defined above. The answer is

$$u(x, t) = \frac{1}{3} + \sum_{n=1}^{\infty} \frac{8(1 + \cos(n\pi))}{n^2\pi^2} e^{-n^2\pi^2 t} \cos(n\pi x).$$

If you haven't taken a course in partial differential equations, then you can skip this step. You will still need the solution for part (b).

(b) Write a MATLAB<sup>®</sup> script that plots the solution (for  $n$  up to 50) in part (a). Plot the solution at times  $t = 0, t = 0.01, t = 0.02, t = 0.03, t = 0.04,$  and  $t = 0.05$ .

*\*Note 1 - There is no need to write this as a MATLAB<sup>®</sup> function. Simply code this up in a main script file.*

*\*Note 2 - The solution provided uses meshgrid in order to simply sum over all  $n = 1, 2, 3, \dots, 50$ . However, the same result can be obtained using for loops (looping over each  $n = 1, 2, 3, \dots, 50$ ). It should be in the reader's interest to understand the solution provided.*

## Lab 8 – The 2D Laplace Equation (Separation of Variables)

Consider the 2D Laplace equation,

$$\begin{cases} u_{xx} + u_{yy} = 0, & 0 < x, y < 1, \\ u(x, y = 0) = u(x, y = 1) = 0, & 0 < x < 1, \\ u(x = 0, y) = 1, & 0 < x < 1, \\ u(x = 1, y) = 0, & 0 < x < 1. \end{cases}$$

(a) Use separation of variables to solve the 2D Laplace equation defined above. The answer is

$$u(x, t) = \sum_{n=1}^{\infty} \frac{2(1 - \cos(n\pi))}{n\pi} \sin(n\pi y) (\cosh(n\pi x) - \coth(n\pi) \sinh(n\pi x)).$$

If you haven't taken a course in partial differential equations, then you can skip this step. You will still need the solution for part (b).

(b) Write a MATLAB<sup>®</sup> script that plots the solution (for  $n$  up to 50).

*\*Note 1 – There is no need to write this as a MATLAB<sup>®</sup> function. Simply code this up in a main script file.*

*\*Note 2 – The solution provided uses meshgrid in order to simply sum over all  $n = 1, 2, 3, \dots, 50$ . However, the same result can be obtained using for loops (looping over each  $n = 1, 2, 3, \dots, 50$ ). It should be in the reader's interest to understand the solution provided.*

## Lab 9 – The 1D Linear Advection Equation (Finite Differences)

Consider the 1D linear advection equation (with periodic boundary conditions),

$$\begin{cases} u_t + 2u_x = 0, & 0 < x < 2\pi, t > 0, \\ u(x = 0, t) = u(x = 2\pi, t), & t > 0, \\ u(x, t = 0) = \sin(x), & 0 < x < 2\pi. \end{cases}$$

Consider dividing the interval  $[0, 2\pi]$  into  $N_x$  equally spaced cells/sub-intervals (i.e.,  $N_x + 1$  equally spaced points). Let  $x_j$  denote the center of cell  $I_j$ , for  $j = 1, 2, \dots, N_x$ . Let  $\Delta x$  be the width of each cell.

We want to numerically solve this linear advection equation up to the final time  $T_f = 3\pi$ . Let  $\Delta t = 0.3\Delta x$  be the time step size, and let  $t^n$  denote the  $n$ -th time step.

Let  $u_j^n$  denote the numerical solution at point  $x_j$  and time  $t^n$ . A stable first order finite difference scheme (**upwind scheme**) states

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + 2\frac{u_j^n - u_{j-1}^n}{\Delta x} = 0,$$

or rather,

$$u_j^{n+1} = (1 - 2\lambda)u_j^n + 2\lambda u_{j-1}^n,$$

where  $\lambda = \Delta t/\Delta x$ . Expressing this as a system (noting the periodic boundary conditions),

$$\begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \vdots \\ u_{N_x-1}^{n+1} \\ u_{N_x}^{n+1} \end{bmatrix} = \begin{bmatrix} 1 - 2\lambda & & & & 2\lambda \\ & 2\lambda & 1 - 2\lambda & & \\ & & \ddots & \ddots & \\ & & & & 2\lambda & 1 - 2\lambda \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{N_x-1}^n \\ u_{N_x}^n \end{bmatrix}$$

(a) Write a MATLAB<sup>®</sup> script that plots the numerical solution at the final time  $T_f = 3\pi$  using  $N_x = 400$  cells, as well as the exact solution  $u(x, T_f) = \sin(x - 2T_f)$ . Carefully note that the



cell centers are shifted by  $\Delta x/2$  (e.g.,  $x_1 = \Delta x/2$ ,  $x_2 = 3\Delta x/2$ ,  $x_3 = 5\Delta x/2$ , ...). Also, you will need to append  $T_f$  to the end of your time vector to ensure that  $T_f$  is included.

*\*Note 1 – An  $n \times n$  tridiagonal matrix  $\text{tridiag}(a,b,c)$  can be coded in MATLAB<sup>®</sup> using the following command:*

**full(gallery('tridiag',n,a,b,c))**

*\*Note 2 – A linear system can be solved using the backslash command. See the Appendix.*

(b) The  $L^1$  error of your numerical solution (at time  $T_f$ ) can be computed,

$$\|u_{approx} - u_{exact}\|_{L^1} = \Delta x \sum_{j=1}^N |u_{approx}(x_j, T_f) - u_{exact}(x_j, T_f)|$$

Compute the  $L^1$  error for  $N_x = 100, 200, 400, 800$ . Store your  $L^1$  error values in a vector called *L1*. What do you observe? You should see the  $L^1$  error cut in half whenever you double the number of points (i.e., cut  $\Delta x$  in half).

Next, compute what's called the **order of convergence** by typing the command

**log2(L1(1:end-1)./L1(2:end))**

You should see the order converge to 1.

## Lab Solutions

**DISCLAIMER:** These solutions are not necessarily optimal. For the sake of these labs, efficiency doesn't pose much of a concern. Although these solutions are not as efficient as they could be, it is because we want to explicitly show the reader what everything is doing. Oftentimes, optimal code is quite challenging to read.

---

```
% Lab 1 Solutions
clear all;clc;format long;
```

## Lab 1(i) -- Newton's Method

```
x1 = 1;
f = @(x) sin(x) - x - 1;
fp = @(x) cos(x) - 1;
tol = 0.00001;
root = newton(x1,f,fp,tol);
disp(['(i) Desired root = ' num2str(root)])

(i) Desired root = -1.9346
```

## Lab 1(ii) -- Newton's Method

```
x1 = 1;
f = @(x) x^3 - 3*x + 6;
fp = @(x) 3*x^2 - 3;
tol = 0.001;
root = newton(x1,f,fp,tol);
disp(['(ii) Desired root = ' num2str(root)])

(ii) Desired root = NaN
```

*Published with MATLAB® R2019b*

---

```
function root = newton(x1,f,fp,tol)
% INPUTS:
%   x1 = initial point
%   f = function we desire a root from
%   tol = tolerance
% OUTPUT:
%   root = desired root of f(x)=0

diff = 1; %initiate difference between consecutive terms
xn = x1;
xnn = 1;
count = 0;
maxiter = 100; %terminate after maxiter iterations
while diff > tol
    count = count + 1;
    if count == maxiter
        disp(['Terminated because ran ' num2str(maxiter) '
iterations'])
        return
    end
    xn = xnn;
    xnn = xn - f(xn)/fp(xn);
    diff = abs(xnn-xn);
end
root = xnn;
end
```

*Not enough input arguments.*

*Error in newton (line 10)*  
*xn = x1;*

*Published with MATLAB® R2019b*

---

## Table of Contents

.....	1
Lab 2a -- Trapezoid Rule .....	1
Lab 2b -- Simpson's Rule .....	1
Lab 2c -- Gauss-Legendre Quadrature .....	1

```
% Lab 2 Solutions
clear all;clc;format long;
```

## Lab 2a -- Trapezoid Rule

```
a = 0; b = pi; %endpoints
N = 16; %number of sub-intervals
f = @(x) sin(x);
trapapprox = trapezoid(a,b,N,f);
disp(['Trapezoid rule approximation: ' num2str(trapapprox)])

Trapezoid rule approximation: 1.9936
```

## Lab 2b -- Simpson's Rule

```
a = 0; b = pi; %endpoints
N = 8; %number of sub-intervals
f = @(x) sin(x);
simpapprox = simpson(a,b,N,f);
disp(['Simpsons rule approximation: ' num2str(simpapprox)])

Simpsons rule approximation: 2.0003
```

## Lab 2c -- Gauss-Legendre Quadrature

```
a = 0; b = pi; %endpoints
f = @(x) sin(x);
for n = 2:7 %number of nodes/weights
    quadapprox = GLquad(a,b,n,f);
    disp(['Gauss-Legendre quadrature with ' num2str(n) ' nodes/
weights: ' num2str(quadapprox,13)])
end

Gauss-Legendre quadrature with 2 nodes/weights: 1.935819574651
Gauss-Legendre quadrature with 3 nodes/weights: 2.001388913608
Gauss-Legendre quadrature with 4 nodes/weights: 1.999984228458
Gauss-Legendre quadrature with 5 nodes/weights: 2.000000110284
Gauss-Legendre quadrature with 6 nodes/weights: 1.999999999477
Gauss-Legendre quadrature with 7 nodes/weights: 2.000000000002
```

Published with MATLAB® R2019b

---

```
function trapapprox = trapezoid(a,b,N,f)
% INPUTS:
%   a,b = endpoints
%   N = number of sub-intervals, i.e., N+1 points
%   f = function to be integrated
% OUTPUT:
%   trapapprox = trapezoid rule on  $\int_a^b \{f(x)dx\}$ 

x = linspace(a,b,N+1);
dx = x(2)-x(1);
trapapprox = dx*sum((f(x(1:end-1)) + f(x(2:end)))/2);
end
```

*Not enough input arguments.*

*Error in trapezoid (line 9)*  
*x = linspace(a,b,N+1);*

*Published with MATLAB® R2019b*

---

```

function simpapprox = simpson(a,b,N,f)
% INPUTS:
%   a,b = endpoints
%   N = (even) number of sub-intervals, i.e., N+1 points
%   f = function to be integrated
% OUTPUT:
%   simpapprox = Simpson's rule on  $\int_a^b f(x)dx$ 

x = linspace(a,b,N+1); %*note: row vector
dx = x(2)-x(1);
coeffs = ones(N+1,1); %coefficients in Simpson's rule
                    %*note: column vector
for i = 1:N/2
    coeffs(2*i) = 4;
    coeffs(2*i + 1) = 2;
end
coeffs(end) = 1;
simpapprox = (dx/3)*f(x)*coeffs; %*note: row vector*column vector
end

```

*Not enough input arguments.*

*Error in simpson (line 9)*

```
x = linspace(a,b,N+1); %*note: row vector
```

*Published with MATLAB® R2019b*

---

```

function quadapprox = GLquad(a,b,n,f)
% INPUTS:
%   a,b = endpoints
%   n = number of nodes/weights
%   f = function to be integrated
% OUTPUT:
%   quadapprox = Gauss-Legendre quadrature for int_a^b{f(x)dx}

if n == 2
    nodes = [-0.577350269189625765,...
             0.577350269189625765];
    weights = [1,...
              1];
elseif n == 3
    nodes = [-0.774596669241483377,...
             0,...
             0.774596669241483377];
    weights = [0.55555555555555556,...
              0.88888888888888889,...
              0.55555555555555556];
elseif n == 4
    nodes = [-0.861136311594052575,...
             -0.339981043584856265,...
             0.339981043584856265,...
             0.861136311594052575];
    weights = [0.347854845137453857,...
              0.652145154862546143,...
              0.652145154862546143,...
              0.347854845137453857];
elseif n == 5
    nodes = [-0.906179845938663993,...
             -0.538469310105683091,...
             0,...
             0.538469310105683091,...
             0.906179845938663993];
    weights = [0.236926885056189088,...
              0.478628670499366468,...
              0.56888888888888889,...
              0.478628670499366468,...
              0.236926885056189088];
elseif n == 6
    nodes = [-0.932469514203152028,...
             -0.66120938646626451,...
             -0.238619186083196909,...
             0.238619186083196909,...
             0.66120938646626451,...
             0.932469514203152028];
    weights = [0.171324492379170345,...
              0.360761573048138608,...
              0.46791393457269105,...
              0.46791393457269105,...
              0.360761573048138608,...
              0.171324492379170345];

```

---

---

```
        0.171324492379170345];
elseif n == 7
    nodes = [-0.949107912342758525,...
            -0.74153118559939444,...
            -0.405845151377397167,...
            0,...
            0.405845151377397167,...
            0.74153118559939444,...
            0.949107912342758525];
    weights = [0.129484966168869693,...
              0.279705391489276668,...
              0.38183005050511894,...
              0.417959183673469388,...
              0.38183005050511894,...
              0.279705391489276668,...
              0.129484966168869693];
end
quadapprox = ((b-a)/2)*sum(weights.*f(a + ((b-a)/2)*(nodes+1)));
end
```

*Not enough input arguments.*

*Error in GLquad (line 9)  
if n == 2*

*Published with MATLAB® R2019b*



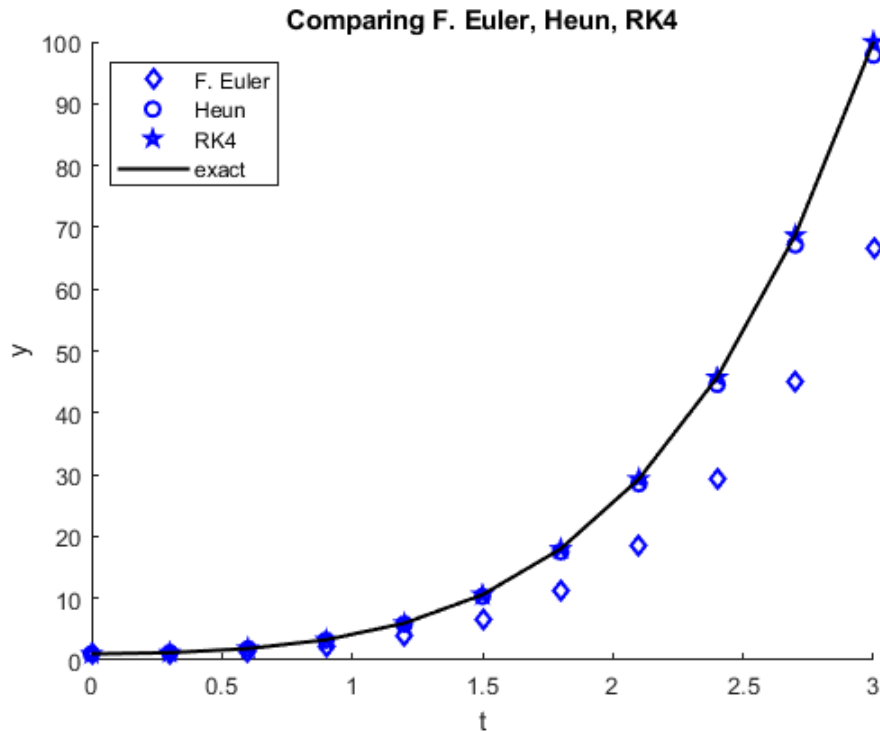
---

```
% Lab 3 Solutions
clear all;close all;clc;format long;
```

## Labs 3a-3c -- Forward Euler, Heun, RK4

```
f = @(t,y) 4*t*sqrt(y);
Tf = 3; %final time
y0 = 1; %initial condition
N = 10; %number of time steps
t = linspace(0,Tf,N+1);
dt = t(2) - t(1);
y_feuler = zeros(1,N+1); y_feuler(1) = y0; %forward Euler
solution
y_heun = zeros(1,N+1); y_heun(1) = y0; %Heun solution
y_rk4 = zeros(1,N+1); y_rk4(1) = y0; %RK4 solution
for n = 2:N+1
    tn = t(n-1);
    yn_feuler = y_feuler(n-1); %forward Euler
    yn_heun = y_heun(n-1); %Heun
    yn_rk4 = y_rk4(n-1); %RK4
    y_feuler(n) = feuler(tn,yn_feuler,dt,f);
    y_heun(n) = heun(tn,yn_heun,dt,f);
    y_rk4(n) = RK4(tn,yn_rk4,dt,f);
end

yexact = (sqrt(y0) + t.^2).^2; %exact solution
figure(1);scatter(t,y_feuler,'bd','linewidth',1.5);
hold on;scatter(t,y_heun,'bo','linewidth',1.5);
hold on;scatter(t,y_rk4,'bp','linewidth',1.5);
hold on;plot(t,yexact,'k-','linewidth',1.5);
title('Comparing F. Euler, Heun, RK4');
xlabel('t');ylabel('y');
legend('F. Euler','Heun','RK4','exact','location','northwest');
```



## Optional: showing the order of convergence

```

f = @(t,y) 4*t*sqrt(y);
Tf = 3; %final time
y0 = 1; %initial condition
Nvals = [10,20,40,80,160]; %number of time steps
Llfeuler = zeros(1,numel(Nvals)); %L1 error, forward Euler
Llheun = zeros(1,numel(Nvals)); %L1 error, Heun
Llrk4 = zeros(1,numel(Nvals)); %L1 error, RK4
for l = 1:numel(Nvals)
    N = Nvals(l);
    t = linspace(0,Tf,N+1);
    dt = t(2) - t(1);
    y_feuler = zeros(1,N+1); y_feuler(1) = y0; %forward Euler
    solution
    y_heun = zeros(1,N+1); y_heun(1) = y0; %Heun solution
    y_rk4 = zeros(1,N+1); y_rk4(1) = y0; %RK4 solution
    for n = 2:N+1
        tn = t(n-1);
        yn_feuler = y_feuler(n-1); %forward Euler
        yn_heun = y_heun(n-1); %Heun
        yn_rk4 = y_rk4(n-1); %RK4
        y_feuler(n) = feuler(tn,yn_feuler,dt,f);
        y_heun(n) = heun(tn,yn_heun,dt,f);
        y_rk4(n) = RK4(tn,yn_rk4,dt,f);
    end
end

```

---

```

end
yexact = (sqrt(y0) + t.^2).^2; %exact solution
Llfeuler(1) = dt*sum(abs(y_feuler-yexact));
Llheun(1) = dt*sum(abs(y_heun-yexact));
Llrk4(1) = dt*sum(abs(y_rk4-yexact));
end
disp(['F. Euler order of convergence: '
      num2str(log2(Llfeuler(1:end-1)./Llfeuler(2:end)),4)])
disp(['Heun order of convergence: ' num2str(log2(Llheun(1:end-1)./
Llheun(2:end)),4)])
disp(['RK4 order of convergence: ' num2str(log2(Llrk4(1:end-1)./
Llrk4(2:end)),4)])

F. Euler order of convergence: 1.009      1.005      1.003      1.002
Heun order of convergence: 2.048      2.016      2.005      2.002
RK4 order of convergence: 3.928      3.964      3.982      3.991

```

*Published with MATLAB® R2019b*

---

```
function ynn = feuler(tn,yn,dt,f)
% INPUTS:
%   tn = previous time
%   yn = approximate solution y(tn)
%   dt = time step
%   f = RHS of the ODE y'=f(t,y)
% OUTPUT:
%   ynn = approximate solution y(tn+dt)
```

```
ynn = yn + dt*f(tn,yn);
end
```

*Not enough input arguments.*

*Error in feuler (line 10)*  
*ynn = yn + dt\*f(tn,yn);*

*Published with MATLAB® R2019b*

---

```
function ynn = heun(tn,yn,dt,f)
% INPUTS:
%   tn = previous time
%   yn = approximate solution y(tn)
%   dt = time step
%   f = RHS of the ODE y'=f(t,y)
% OUTPUT:
%   ynn = approximate solution y(tn+dt)

K1 = f(tn,yn);
K2 = f(tn+dt,yn+dt*K1);
ynn = yn + (dt/2)*(K1 + K2);
end
```

*Not enough input arguments.*

*Error in heun (line 10)*

*K1 = f(tn,yn);*

*Published with MATLAB® R2019b*

---

```
function ynn = RK4(tn,yn,dt,f)
% INPUTS:
%   tn = previous time
%   yn = approximate solution y(tn)
%   dt = time step
%   f = RHS of the ODE y'=f(t,y)
% OUTPUT:
%   ynn = approximate solution y(tn+dt)

K1 = f(tn,yn);
K2 = f(tn+dt/2,yn+(dt/2)*K1);
K3 = f(tn+dt/2,yn+(dt/2)*K2);
K4 = f(tn+dt,yn+dt*K3);
ynn = yn + (dt/6)*(K1 + 2*K2 + 2*K3 + K4);
end
```

*Not enough input arguments.*

*Error in RK4 (line 10)*  
*K1 = f(tn,yn);*

*Published with MATLAB® R2019b*

---

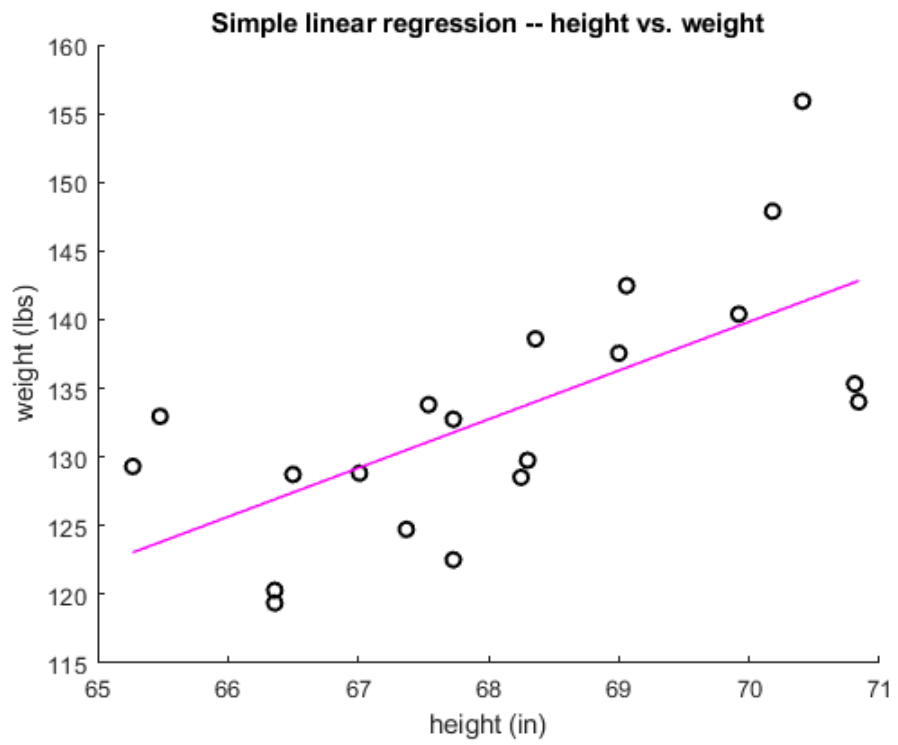
```
% Lab 4 Solutions
clear all;close all;clc;format long;
```

## Linear Regression

```
data = [
70.18 147.89;
70.41 155.90;
66.36 119.37;
67.54 133.81;
66.50 128.73;
69.00 137.55;
68.30 129.76;
67.01 128.82;
70.81 135.32;
69.06 142.47;
67.73 132.75;
67.37 124.73;
65.27 129.31;
70.84 134.02;
69.92 140.40;
68.25 128.52;
66.36 120.30;
68.36 138.60;
65.48 132.96;
67.73 122.52
];

X = ones(numel(data(:,1)),2);
X(:,2) = data(:,1);
Y = data(:,2);
beta = inv(X'*X)*X'*Y;
x = linspace(min(data(:,1)),max(data(:,1)),100);

figure(1);scatter(data(:,1),data(:,2),'blacko','linewidth',1.5);
figure(1);hold on;plot(x,beta(1)+beta(2)*x,'m-','linewidth',1);
xlabel('height (in)');ylabel('weight (lbs)');
title('Simple linear regression -- height vs. weight');
```



*Published with MATLAB® R2019b*



---

## Table of Contents

.....	1
Least Squares Problems: Solving a Linear System .....	1
Normal Equations .....	1
QR Factorization .....	1
Singular Value Decomposition .....	2

```
% Lab 5 Solutions
clear all;close all;clc;format long;
```

## Least Squares Problems: Solving a Linear System

```
A = [1 -1;0 0.5e-7;0 0];
b = [0;0.5e-7;1];
```

## Normal Equations

```
R = chol(A'*A);
w = (R')\ (A'*b);
disp('Solution via the normal equations:')
x = R\w
```

*Solution via the normal equations:*

```
x =
    1.023545369856931
    1.023545369856931
```

## QR Factorization

```
[Q,R] = qr(A);
Qhat = Q(:,1:2);
Rhat = R(1:2,:);
disp('Solution via QR factorization:')
x = Rhat\ (Qhat'*b)
```

*Solution via QR factorization:*

```
x =
    1
    1
```

---

# Singular Value Decomposition

```
[U,S,V] = svd(A);
Uhat = U(:,1:2);
Shat = S(1:2,:);
w = Shat\uhat'*b;
disp('Solution via the SVD:')
x = V*w
```

*Solution via the SVD:*

*x =*

```
1.0000000000000000
1.0000000000000000
```

*Published with MATLAB® R2019b*

---

## Table of Contents

.....	1
Lab 6a -- Lagrange Interpolating Polynomial .....	1
Lab 6b -- Hermite Interpolating Polynomial .....	3
Lab 6c -- Cubic Splines .....	3

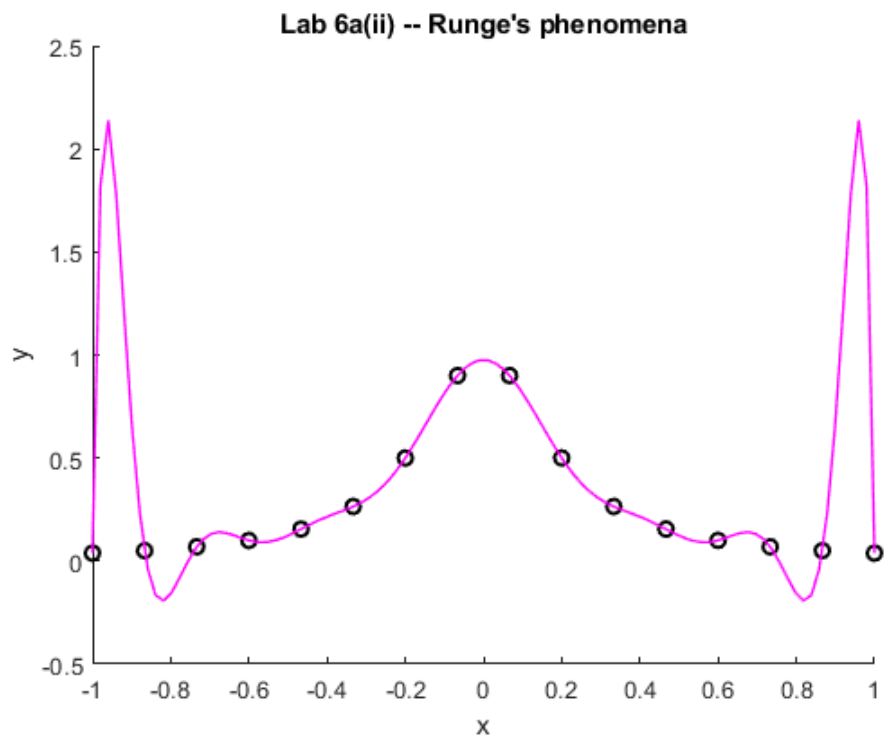
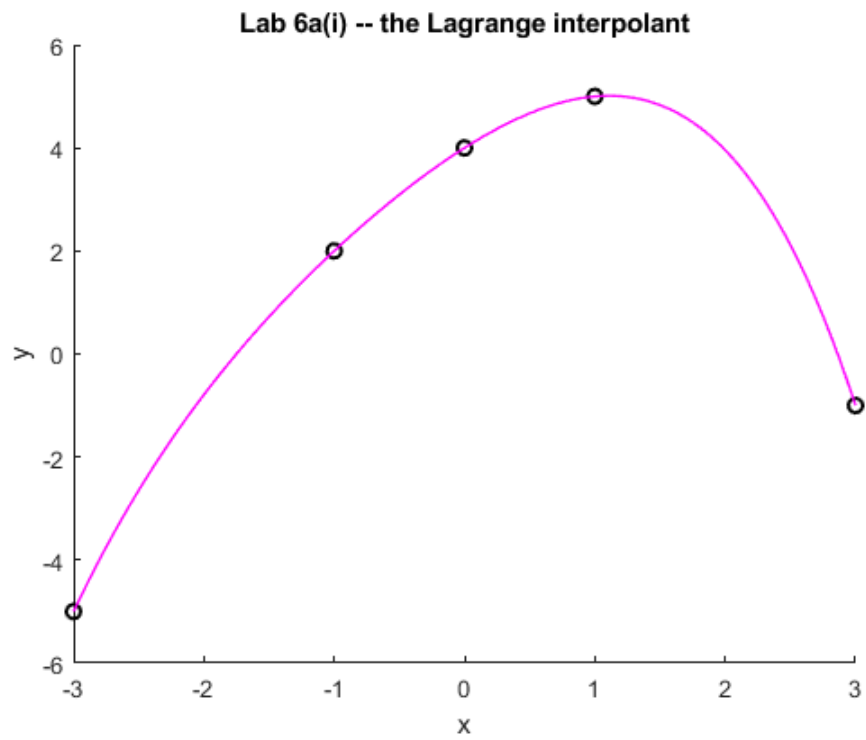
```
% Lab 6 Solutions
clear all;close all;clc;format long;
```

## Lab 6a -- Lagrange Interpolating Polynomial

Part (i)

```
P = @(x) -5*((x+1).*(x-0).*(x-1).*(x-3))/
((-3+1)*(-3-0)*(-3-1)*(-3-3))...
+ 2*((x+3).*(x-0).*(x-1).*(x-3))/((-1+3)*(-1-0)*(-1-1)*(-1-3))...
+ 4*((x+3).*(x+1).*(x-1).*(x-3))/((0+3)*(0+1)*(0-1)*(0-3))...
+ 5*((x+3).*(x+1).*(x-0).*(x-3))/((1+3)*(1+1)*(1-0)*(1-3))...
- 1*((x+3).*(x+1).*(x-0).*(x-1))/((3+3)*(3+1)*(3-0)*(3-1));
x = linspace(-3,3,100);
figure(1);scatter([-3,-1,0,1,3],
[-5,2,4,5,-1], 'blacko', 'linewidth',1.5);
figure(1);hold on;plot(x,P(x), 'm-', 'linewidth',1);
xlabel('x');ylabel('y');title('Lab 6a(i) -- the Lagrange
interpolant');

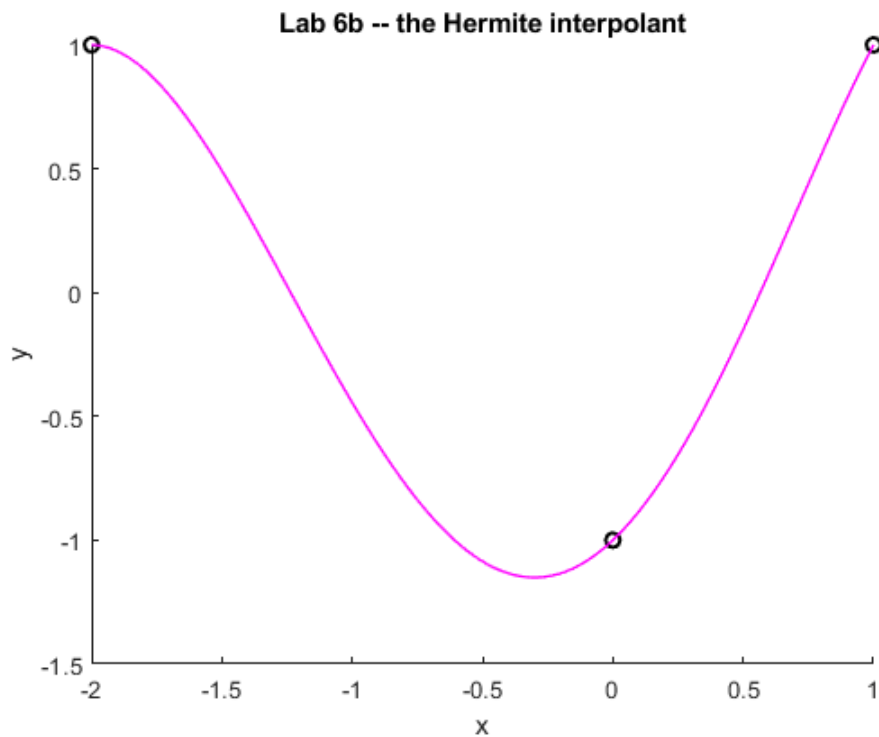
% Part (ii)
f = @(x) 1./(1+25*x.^2);
nodes = linspace(-1,1,16);
x = linspace(-1,1,100);
P = zeros(1,100);
for i = 1:16
    temp = 1;
    for j = 1:16
        if j ~= i
            temp = temp.*(x-nodes(j))/(nodes(i)-nodes(j));
        end
    end
    P = P + f(nodes(i))*temp;
end
figure(2);scatter(nodes,f(nodes), 'blacko', 'linewidth',1.5);
figure(2);hold on;plot(x,P, 'm-', 'linewidth',1);
xlabel('x');ylabel('y');title('Lab 6a(ii) -- Runge's phenomena');
```



---

## Lab 6b -- Hermite Interpolating Polynomial

```
H = @(x) (-1/36)*x.^5 - (1/3)*x.^4 - (1/4)*x.^3 + (29/18)*x.^2 + x - 1;  
x = linspace(-2,1,100);  
figure(3);scatter([-2,0,1],[1,-1,1],'blacko','linewidth',1.5);  
figure(3);hold on;plot(x,H(x),'m-','linewidth',1);  
xlabel('x');ylabel('y');title('Lab 6b -- the Hermite interpolant');
```



## Lab 6c -- Cubic Splines

```
f = @(x) 1./(1+25*x.^2);  
nodes = linspace(-1,1,16);  
  
A = zeros(4*15,4*15);  
b = zeros(4*15,1);  
for i = 1:15  
    xleft = nodes(i);  
    xright = nodes(i+1);  
    A(i,4*(i-1)+1) = xleft^3;  
    A(i,4*(i-1)+2) = xleft^2;  
    A(i,4*(i-1)+3) = xleft;  
    A(i,4*(i-1)+4) = 1;  
    b(i) = f(xleft);  
end
```

---

```

    A(15+i,4*(i-1)+1) = xright^3;
    A(15+i,4*(i-1)+2) = xright^2;
    A(15+i,4*(i-1)+3) = xright;
    A(15+i,4*(i-1)+4) = 1;
    b(15+i) = f(xright);
end

for i = 1:14
    xright = nodes(i+1); %interior nodes
    A(30+i,4*(i-1)+1) = 3*xright^2;
    A(30+i,4*(i-1)+2) = 2*xright;
    A(30+i,4*(i-1)+3) = 1;
    A(30+i,4*i+1) = -3*xright^2;
    A(30+i,4*i+2) = -2*xright;
    A(30+i,4*i+3) = -1;
end

for i = 1:14
    xright = nodes(i+1); %interior nodes
    A(44+i,4*(i-1)+1) = 6*xright;
    A(44+i,4*(i-1)+2) = 2;
    A(44+i,4*i+1) = -6*xright;
    A(44+i,4*i+2) = -2;
end

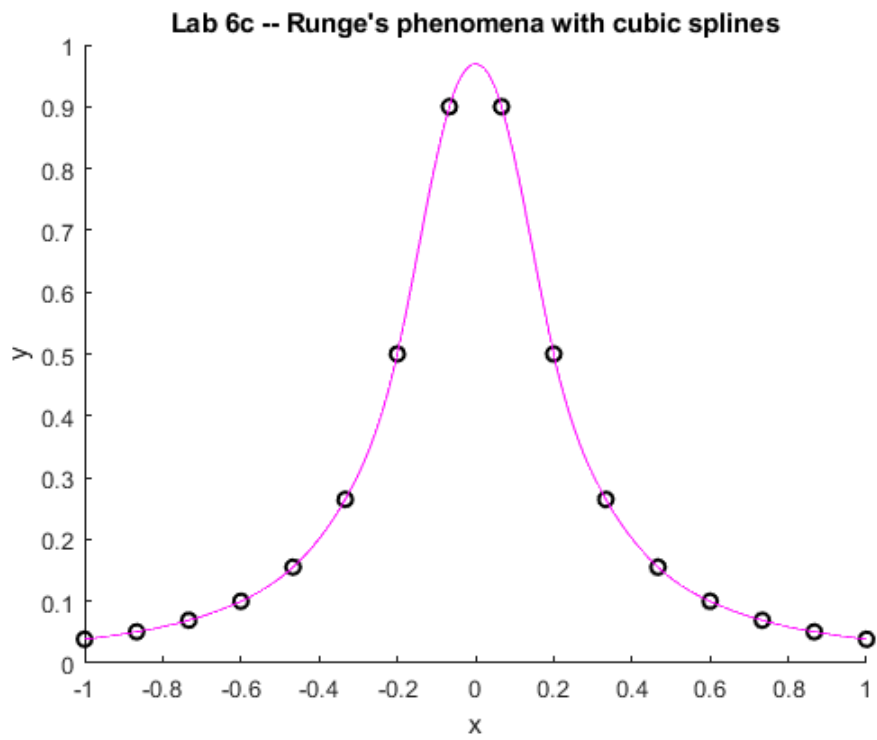
A(59,1) = 6*nodes(1);
A(59,2) = 2;
A(60,4*14+1) = 6*nodes(16);
A(60,4*14+2) = 2;

coef = A\b; %coefficients
x = linspace(-1,1,15*30); %each subinterval gets 30 points
S = zeros(1,15*30); %values for the cubic spline
for i = 1:15
    coef1 = coef(4*(i-1)+1);
    coef2 = coef(4*(i-1)+2);
    coef3 = coef(4*(i-1)+3);
    coef4 = coef(4*(i-1)+4);
    tempx = x(30*(i-1)+1:30*i);
    S(30*(i-1)+1:30*i) = coef1*tempx.^3 + coef2*tempx.^2 + coef3*tempx
    + coef4;
end

figure(4);scatter(nodes,f(nodes),'blacko','linewidth',1.5);
figure(4);hold on;plot(x,S,'m-','linewidth',1);
xlabel('x');ylabel('y');title('Lab 6c -- Runge''s phenomena with cubic
splines');

```

---



*Published with MATLAB® R2019b*

---

```

% Lab 7 Solutions
clear all;clc;close all;

```

## Solution to 1D heat equation

$u_t = u_{xx}$ ,  $0 < x < 1$ ,  $t > 0$   $u_x(0,t) = u_x(1,t) = 0$   $u(x,0) = (2x-1)^2$

```

N = 50; %number of terms, n=1,2,3,...,N
Nx = 100;
x = linspace(0,1,Nx);

```

```

[xmat,nmat] = meshgrid(x,1:N);

```

```

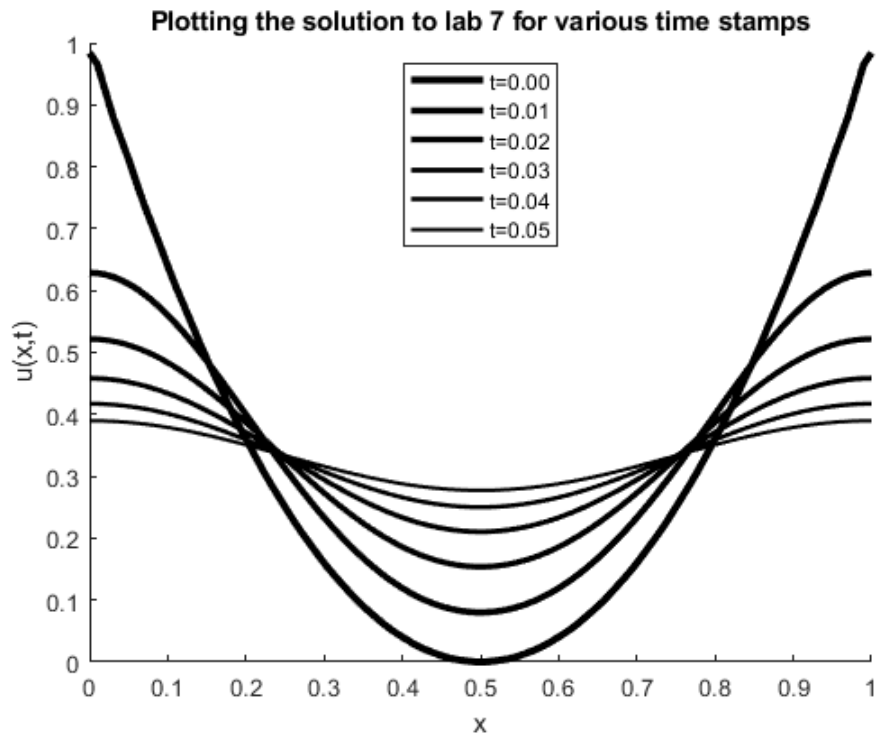
for t = [0,0.01,0.02,0.03,0.04,0.05]
    u = 1/3 + sum((8*(1+cos(nmat*pi))./(pi^2*nmat.^2)).*...
        cos(pi*nmat.*xmat).*exp(-pi^2*nmat.^2*t),1);
    %note -- sum(X,m) sums array X over the m dimension.
    figure(1);hold on;plot(x,u,'k-','linewidth',3*(1-10*t));
end

```

```

figure(1);title('Plotting the solution to lab 7 for various time
stamps');
xlabel('x');ylabel('u(x,t)');
legend('t=0.00','t=0.01','t=0.02','t=0.03','t=0.04','t=0.05','location','north');

```





---

*Published with MATLAB® R2019b*

---

```
% Lab 8 Solutions
clear all;clc;close all;
```

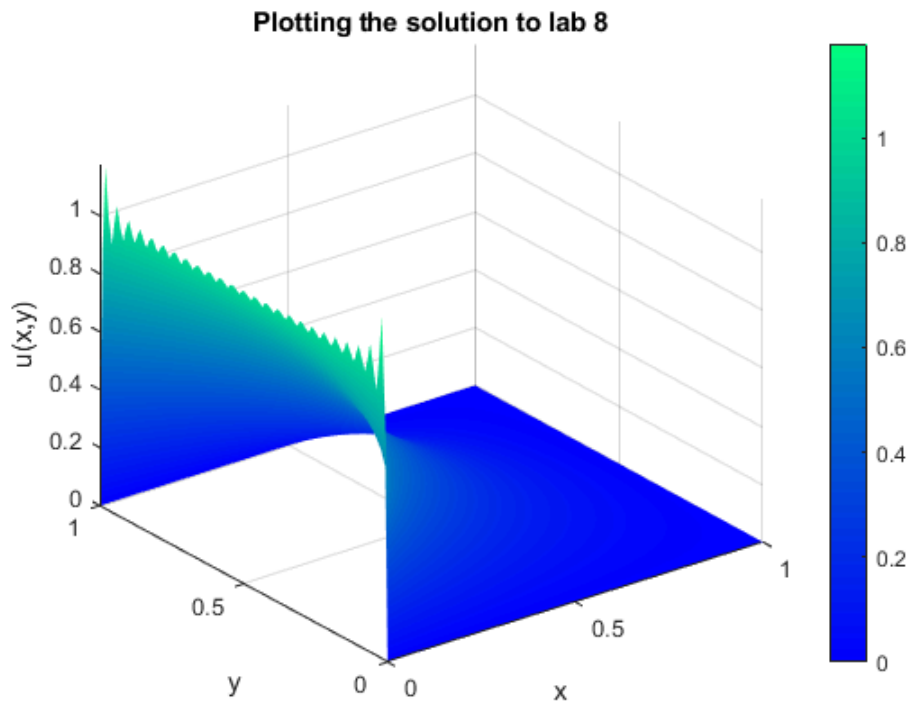
## Solution to 1D heat equation

```
ut = uxx + uyy, 0 < x,y < 1 u(x,0) = u(x,1) = u(1,y) = 0 u(0,y) = 1

N = 50; %number of terms, n=1,2,3,...,N
Nx = 100; Ny = 100;
x = linspace(0,1,Nx); y = linspace(0,1,Ny);
[X,Y] = meshgrid(x,y);
[xmat,ymat,nmat] = meshgrid(x,y,1:N);

u = sum((2*(1-cos(nmat*pi))./(nmat*pi)).*sin(pi*nmat.*ymat).*...
        (cosh(pi*nmat.*xmat)-coth(pi*nmat).*sinh(pi*nmat.*xmat)),3);
%note -- sum(X,m) sums array X over the m dimension.

figure(1);surf(X,Y,u);
shading interp;colormap winter;colorbar;
figure(1);title('Plotting the solution to lab 8');
xlabel('x');ylabel('y');zlabel('u(x,y)');
```



*Published with MATLAB® R2019b*

---

```
% Lab 9 Solutions
clear all;clc;close all;
```

## Solution to 1D linear advection equation via upwind finite difference method

$u_t + 2u_x = 0, 0 < x < 2\pi, t > 0$   $u(0,t) = u(2\pi,t)$   $u(x,0) = \sin(x)$

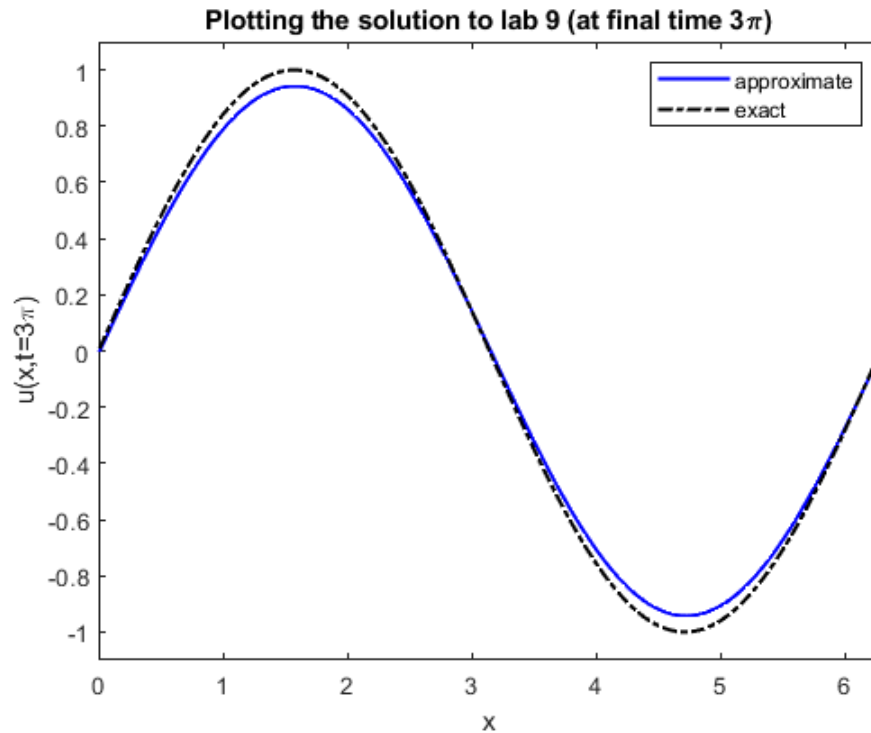
```
Nx = 400; %number of cells
x = linspace(0,2*pi,Nx+1);
dx = x(2) - x(1);
x = x(1:Nx) + dx/2; %cell centers

T = 3*pi; %final time
dt = 0.3*dx;
t = 0:dt:T; t = [t,T]; %force T to be the final time

lambda = dt/dx;
A = full(gallery('tridiag',Nx,2*lambda,1-2*lambda,0));
A(1,Nx) = 2*lambda;

u = sin(x'); %initial condition (as column vector)
for n = 2:numel(t)
    u = A*u;
end

figure(1);plot(x,u,'b-', 'linewidth',1.5);
title('Plotting the solution to lab 9 (at final time 3\pi)');
xlabel('x');ylabel('u(x,t=3\pi)');
axis([0,2*pi,-1.1,1.1]);
figure(1);hold on;plot(x,sin(x-2*T),'k-.', 'linewidth',1.5);
legend('approximate','exact');
```



## (b) Computing the L1 error

```

L1 = zeros(1,4);
Nx = 50;
for k = 1:4
    Nx = 2*Nx; %so we will have Nx = 100,200,400,800
    x = linspace(0,2*pi,Nx+1);
    dx = x(2) - x(1);
    x = x(1:Nx) + dx/2; %cell centers

    T = 3*pi; %final time
    dt = 0.3*dx;
    t = 0:dt:T; t = [t,T]; %force T to be the final time

    lambda = dt/dx;
    A = full(gallery('tridiag',Nx,2*lambda,1-2*lambda,0));
    A(1,Nx) = 2*lambda;

    u = sin(x'); %initial condition (as column vector)
    for n = 2:numel(t)
        u = A*u;
    end

    uexact = sin(x' - 2*T); %exact solution at time T
    L1(k) = dx*sum(abs(u - uexact));

```

---

```
end
disp(['Nx: ' num2str([100,200,400,800])])
disp(['L1 error: ' num2str(L1)])
disp(['Order: ' num2str(log2(L1(1:end-1)./L1(2:end)))]])
```

```
Nx: 100  200  400  800
L1 error: 0.85634    0.45297    0.23304    0.11821
Order: 0.91879    0.95883    0.97926
```

*Published with MATLAB® R2019b*